Interpretations of Classical Logic Using λ-calculus

Freddie Agestam

Abstract

Lambda calculus was introduced in the 1930s as a computation model. It was later shown that the simply typed λ-calculus has a strong connection to intuitionistic logic, via the Curry-Howard correspondence. When the type of a λ-term is seen as a proposition, the term itself corresponds to a proof, or construction, of the object the proposition describes.

In the 1990s, it was discovered that the correspondence can be extended to classical logic, by enriching the λ-calculus with control operators found in some functional programming languages such as Scheme. These control operators operate on abstractions of the evaluation context, so called continuations. These extensions of the λ-calculus allow us to find computational content in non-constructive proofs.

Most studied is the $\lambda\mu$-calculus, which we will focus on, having several interesting properties. Here it is possible to define `catch` and `throw` operators. The type constructors $\wedge$ and $\vee$ are definable from only $\rightarrow$ and $\bot$. Terms in $\lambda\mu$-calculus can be translated to λ-calculus using continuations.

In addition to presenting the main results, we go to depth in understanding control operators and continuations and how they set limitations on the evaluation strategies. We look at the control operator $\mathcal{C}$, as well as `call/cc` from Scheme. We find $\lambda\mu$-calculus and $\mathcal{C}$ equivalents in Racket, a Scheme implementation, and implement the operators $\wedge$ and $\vee$ in Racket. Finally, we find and discuss terms for some classical propositional tautologies.

Contents

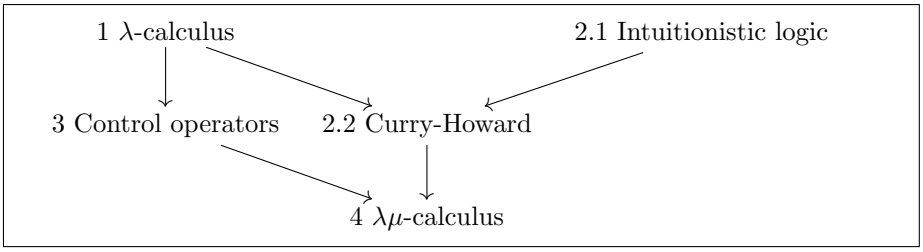


Figure 1: Chapter dependency graph — a directed acyclic graph (partial order) showing in what order sections can be read. The reader may do his/her own topological sorting of the content.

1 Lambda Calculus

The lambda calculus was introduced in the 1930s by Alonzo Church, as an attempt to construct a foundation for mathematics using functions instead of sets. We assume the reader has some basic familiarity with λ-calculus and go quickly through the basics. For a complete introduction, see Hindley and Seldin [HS08], considered the standard book. For a short and gentle introduction, see [Roj97].

We start by introducing the untyped lambda calculus.

1.1 The Calculus

Definition 1.1 (Lambda-terms)**.** Lambda terms are recursively defined, where x ranges over an infinite set of variables, as [1]

$$M, N ::= x \mid \lambda x.M \mid MN$$

where x is called a *variable*, $\lambda x.M$ an *abstraction* and MN is called *application*.

Variables are written with lower case letters. Upper case letters denote meta variables. In addition to variables, λ and ., our language also consists of parentheses to delimit expressions where necessary and specify the order of computation.

The λ (lambda) is a binder, so all free occurrences of x in M become *bound* in the expression $\lambda x.M$. A variable is *free* if it is not bound. The set of free variables of a term M is denoted $\mathrm{FV}(M)$.

A variable that does not occur at all in a term is said to be *fresh* for that term.

A term with no free variables is said to be *closed*. Closed terms are also called *combinators*.

Terms that are syntactically identical are denoted as equivalent with the relation $\equiv$.

Remark 1.1 (Priority rules)**.** To not have our terms cluttered with parentheses, we use the standard conventions.

- Application is left-associative: NMP means $(NM)P$.
- The lambda abstractor binds weaker than application: $\lambda x.MN$ means $\lambda x.(MN)$.

1.1.1 Substitution

Definition 1.2 (Substitution of terms)**.** Substitution of a variable x in a term M with a term N, denoted $M[x := N]$, recursively replaces all free occurrences of x with N.

[1] If the reader is unfamiliar with the recursive definition syntax, it is called Backus-Naur Form (BNF).

We assume $x \neq y$. The variable z is assumed to be fresh for both M and N.

$$\begin{aligned}
x[x := N] &= N \\
(\lambda x.M)[x := N] &= \lambda x.M \\
(\lambda y.M)[x := N] &= \lambda y.M[x := N] \quad \text{if } y \notin \mathrm{FV}(N) \\
(\lambda y.M)[x := N] &= \lambda z.M[y := z][x := N] \quad \text{if } y \in \mathrm{FV}(N) \\
(M_1 M_2)[x := N] &= M_1[x := N] M_2[x := N]
\end{aligned}$$

When substituting $(\lambda y.M)[x := N]$, y is not allowed to occur freely in N, because that y would become bound by the binder and the term would change its meaning. A substitution that prevents variables to become bound in this way is called *capture-avoiding*.

1.1.2 Alpha-conversion

Renaming bound variables in a term, using capture-avoiding substitution, is called α-conversion. We identify terms that can be α-reduced to each other, or say that they are equal modulo α-conversion.

$$\lambda x.M \equiv_\alpha \lambda y.M[x := y] \quad \text{if } y \notin \mathrm{FV}(M)$$

We will further on mean $\equiv_\alpha$ when we write $\equiv$, no longer requiring that bound variables have the same name.

We want this property since the name of a bound variable does not matter semantically — it is only a placeholder.

1.1.3 Beta-reduction

The only computational rule of the lambda calculus is β-reduction, which is interpreted as function application.

$$(\lambda x.M)N \to_\beta M[x := N]$$

Sometimes is also η-reduction mentioned.

$$\lambda x.Mx \to_\eta M$$

The notation $\twoheadrightarrow_\beta$ is used to denote several (0 or more) steps of β-reduction. It is a reflexive and transitive relation on lambda terms.

1.2 Properties

1.2.1 Normal Forms

Definition 1.3 (Redex and Normal Form)**.** A term on the form $(\lambda x.M)N$ is called a *redex* (reducible expression). A term containing no redices is said to be *normal*, or in *normal form*.

A term in normal form have no further possible reductions, so the computation of the term is complete. We can think of a term in normal form to be a fully computed value.

The following theorem states that we may do reductions in any order we want.

Theorem 1.1 (Church-Rosser Theorem)**.** *If $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then there exists a term D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.*

For a proof, see [SU06].

A calculus with a reduction rule $\to$ satisfying this property is called *confluent*. A relation $\to$ having this property is said to have the *Church-Rosser property*.

Theorem 1.2. *The normal form of a term, if it exists, is unique.*

Proof. Follows from Theorem 1.1. □

Theorem 1.3 (Normal Order Theorem[2])**.** *The normal form of a term, if it exists, can be reached by repeatedly evaluating the leftmost-outermost[3] redex.*

The Normal Order Theorem presents an evaluation strategy for reaching the normal form. For evaluation strategies, see further Chapter 3.2, which also presents an informal argument to why the theorem holds.

1.2.2 Computability

Normal forms do not always exist. Here is a well-known example:

Example 1.1 (The Ω-combinator)**.** The term $\lambda x.xx$ takes an argument and applies it to itself. Applying this term on itself, self-application applied on self-application, we get the following expression, often denoted by Ω:

$$\Omega \equiv (\lambda x.xx)(\lambda x.xx)$$

This combinator is a redex that reduces to itself and can thus never reach a normal form. △

Since this combinator can regenerate itself, we suspect that the lambda calculus is powerful enough to accomplish recursion. To accomplish recursion, a function must have access to a copy of itself, i.e. be applied to itself. A recursion operator was presented by Haskell Curry, known as the Y-combinator:

$$\mathtt{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

It has the reduction rule

$$\mathtt{Y}f \twoheadrightarrow_\beta f(\mathtt{Y}f)$$

when applied to any function f. The function f is passed $\mathtt{Y}f$ as its first argument and can use this to call itself.

With the accomplishment of recursion, it is not surprising that the lambda calculus is Turing complete.

Functions of multiple arguments can be achieved through partial application. Note that the function spaces $A \times B \to C$ and $A \to (B \to C)$ have a bijection. A function taking two arguments a and b, giving c, can be replaced by a function taking the first argument a and evaluating to a new function taking b as an argument and giving c. This partial application of functions is called *currying*. In λ-calculus, the function $(a, b) \mapsto c$ is then

$$\lambda a.\lambda b.c$$

[2]Sometimes called the *second Church-Rosser theorem*

[3]Reducing leftmost-innermost will also do.

1.3 Typed Lambda Calculus

Lambda calculus can be regarded as a minimalistic programming language: it has terms and a computation rule. As in a programming language, it is meaningful to add *types* to restrict the possible applications of a term. For example, if we have a term representing addition, it is reasonable to restrict its arguments to being something numeric.

The study of types is called *type theory*.

1.3.1 Types

The types are constructed from a set of *base types*.

- If $A \in \texttt{BaseTypes}$, then A is a type.
- If A_1, A_2 are types, then $A_1 \to A_2$ is a type.

The $\to$ is an example of a *type constructor* since it constructs new types out of existing types. Other types constructors will appear later (Section 2.2.1). A type on the form $A_1 \to A_2$ is a function type: it represents a function that takes an argument term of type A_1 and produces a result term of type A_2.

These types should not be thought of as domain and codomain, because the functions we are using in lambda calculus are not functions between sets. The types should instead be thought of as a predicate that a term must satisfy.

The set of base types could be for example some types used in programming languages: {`int, real, bool`}. We will not be using any specific base types, just metatypes such as A_1, A_2, and will thus not be concerned with which the base types actually are.

Remark 1.2 (Precedence rule)**.** $A \to B \to C$ means $A \to (B \to C)$.

1.3.2 Simply Typed Lambda Calculus

We will add *simple types*[4] to lambda calculus, using only the type constructor $\to$.

To denote that a term M is of type A we write $M : A$. We often add the type to bound variables, for example $\lambda x : A.M$.

Definition 1.4 (Typing judgement)**.** Let Γ be a set of statements on the form $x : A$. Γ is then called a *typing environment*.

A statement on the form $\Gamma \vdash M : A$, meaning that from Γ, we can derive $M : A$, is called a *typing judgement*.

The type of a term can be derived using the rules of Figure 2. The following theorem asserts that these rules work as we expect them to, so that the type of a term is not ambiguous.

Theorem 1.4 (Subject reduction)**.** *If $M : A$ and $M \twoheadrightarrow_\beta M'$, then $M' : A$.*

Proof. Induction on the typing rules. □

[4]Simple types are called simple as opposed to *dependent types*, where types may take parameters. We will only consider simple types.

$$\Gamma, x : A \vdash x : A \quad (ax)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \lambda\ abs \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \lambda\ app$$

Figure 2: Typing judgements for λ-calculus.

Note that our type restriction restricts what terms we may work with — what terms we may compose (apply) — but not the computation (the reduction) of these terms. Thus we still have the confluence property.

The next result is a quite powerful one: A term that can be typed (its type can be derived) can also be normalised (a normal form can be computed).

Theorem 1.5 (Weak normalisation)**.** *If $\Gamma \vdash M : A$ for some A, then M has a normal form. In other words, there is a finite chain of reductions $M \to_\beta M_1 \to_\beta M_2 \ldots \to_\beta M_n$ such that M_n is in normal form.*

For a proof, see [SU06]. The proof requires some work, but the idea used in [SU06] is to look at the complexity, or informally, the size, of the type. With a certain strategy for choosing the next redex to reduce, the complexity of the the type will decrease. Note that this is not trivial, because some reductions can increase the length of the type (counted in the number of characters), because the function body contains something complex.

In fact, we have a stronger result.

Theorem 1.6 (Strong normalisation)**.** *Every chain of reductions $M \to_\beta M_1 \to_\beta M_2 \ldots$ must reach a normal form M_n after a finite number of reductions.*

Again, for a proof, see [SU06].

As a result, the Ω-combinator cannot be given a type, since it has no normal form. Also a recursion operator, such as the Y-combinator, cannot be typed. If a function can be self-applied, it must have its own type D as the first part of its type, which would look something like

$$D = D \to A$$

for some A. Trying to make a such recursive definition of D will expand to something infinite. Attempts at typing other terms which involve some sort of self application will also result in an "infinite type".

Since the simply typed lambda calculus satisfies strong normalisation, all computations are guaranteed to halt and the calculus is not Turing complete.

As well as there being terms that cannot be typed, there are types without closed terms of that type.

Definition 1.5 (Type inhabitation)**.** A type A is said to be *inhabited* if there exists some term M such that $M : A$. M is then called an *inhabitant* of the type A.
A type is said to be *empty* if it is not inhabited.

The *type inhabitation problem* is to decide whether a type is inhabited.

2 The Curry-Howard Isomorphism

The Curry-Howard isomorphism, or the Curry-Howard correspondence, is a correspondence between intuitionistic logic and simply typed lambda calculus. This correspondence will be central in our main topic. We first introduce intuitionistic logic.

2.1 Intuitionistic Logic

The reader is assumed to have seen the "standard logic" before, i.e. *classical logic*. Central in classical logic is that all propositions are either true or false. Intuitionistic logic was introduced in the early 1900s, when mathematicians began to doubt the consistency of the foundations of mathematics. The central idea is that a mathematical object can only be shown to exist by giving a construction of the object.

Some principles taken for granted in classical logic are no longer valid. One of these is the principle of the excluded middle, "for any proposition P, we must have $P \vee \neg P$", which is immediate in classical logic. Likewise, the double negation elimination (also known as *reduction ad absurdum*) "$\neg\neg P \to P$ for any formula P" is not valid. Intuitionistic logic rejects proofs that rely on these and other equivalent principles.

The probably most famous example demonstrating the difference is the following.

> There exist irrational numbers a, b such that a^b is rational.
> *Proof:* $\sqrt{2}$ is known to be irrational. Consider $\sqrt{2}^{\sqrt{2}}$. By the principle of excluded middle, it is either rational or irrational. If it is rational, we are done. If it is irrational, take $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$ and we have $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$ which is rational.

This proof is *non-constructive*, since it does not does not tell us which the numbers a and b are. There exist other proofs showing which the numbers are (it can be shown that $\sqrt{2}^{\sqrt{2}}$ is irrational). However, to show that, the proof becomes much longer.

In fact, by requiring that all proofs shall contain constructions, less theorems can be proved. On the other hand, a proof with a construction will provide a richer understanding. Often, we do not want to merely show the existence of an object, but to construct the object itself.

2.1.1 Proofs as Constructions

In intuitionistic logic, a proof is a construction. The most known interpretation of intuitionistic proofs is the BHK-interpretation, which follows. A *construction* is defined inductively, where P and Q are propositions:

- A construction of $P \wedge Q$ is a construction of P and a construction of Q.
- A construction of $P \vee Q$ is a construction of P or a construction of Q, as well as an indicator of whether it is P or Q.

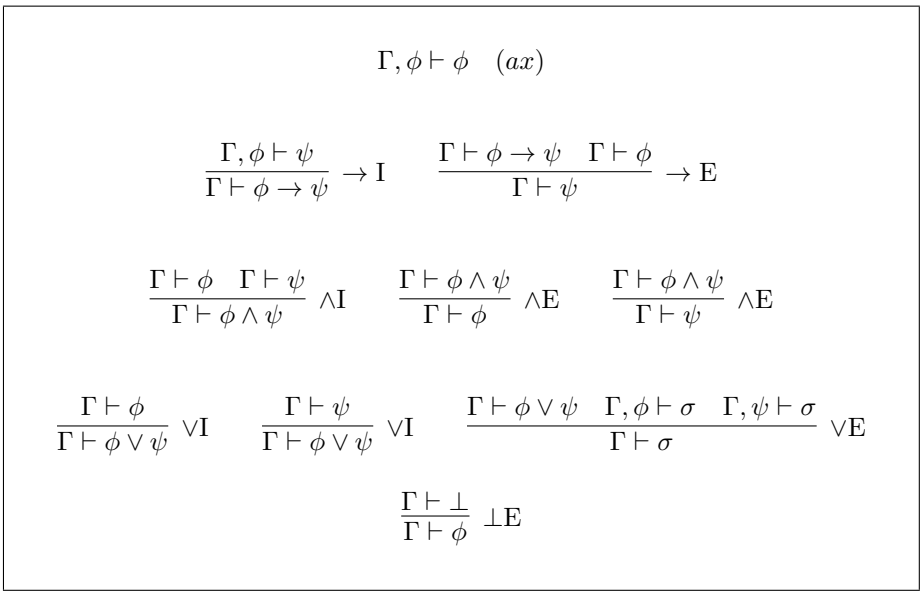


Figure 3: Natural deduction for intuitionistic propositional logic.

- A construction of $P \to Q$ is a function that from a construction of P constructs a construction of Q.
- A construction of $\neg P$ (or $P \to \bot$) is a function that produces a contradiction from every construction of P.
- There is no construction of $\bot$.

Such a proof is also called a proof or a *witness*.

The classical axiom $\neg\neg P \to P$ does not make any sense under a constructive interpretation of proofs. To prove a proposition P we need a proof of P. But given a proof of $\neg\neg P$, we cannot extract a construction of P.

2.1.2 Natural Deduction

The notation $\Gamma \vdash P$, where Γ is a set of propositions, means that from Γ, we can derive P. The set Γ is called the *assumption set* and the statement $\Gamma \vdash P$ is called a *proof judgement*. The derivation rules for natural deduction in intuitionistic logic is shown in Figure 3.

This notation is called the *sequent notation*, where all free assumptions are written left of the $\vdash$ (turnstile). The reader may be more familiar with the tree style notation, where assumptions are written in the leaves of a tree and marked as discharged during derivation. The sequent style notation is much easier to reason about formally, whereas the tree notation is easier to read.

2.1.3 Classical and Intuitionistic Interpretations

Natural deduction formalises our concept of a proof. But we also need to introduce the concept of *(truth) value of a formula*, which corresponds to our

notion of a formula being "true" or "false" when the atomic propositions it consists of are given certain *values*. When we use classical logic, our propositions are assigned values from *Boolean algebras*. The most well-known such is $\mathbb{B}$, which consists of only two elements, $\{0, 1\}$. When we study intuitionistic logic, our equivalent of Boolean algebras are called *Heyting algebras*. The Boolean algebras and the Heyting algebras specify how the constants and connectives $\bot, \top, \vee, \wedge, \neg, \to$ are interpreted. We do not define these algebras here; the interested reader can read about it in for example [SU06] or a simpler presentation in [Car13], covering only classical logic. We point out that all Boolean algebras are Heyting algebras.

This is a quite technical area, but we go quickly through it, a bit informally, and state the main results without proofs.

Definition 2.1. A *valuation* is any function $v : \Gamma \to \mathcal{H}$, sending every atomic proposition p in Γ to an element $v(p)$ in the Heyting algebra $\mathcal{H}$. The value $[\![P]\!]_v$ of a proposition P is defined inductively:

- $[\![p]\!]_v = v(p)$
- $[\![\bot]\!]_v = \bot$
- $[\![\top]\!]_v = \top$
- $[\![P \vee Q]\!]_v = [\![P]\!]_v \vee [\![Q]\!]_v$
- $[\![P \wedge Q]\!]_v = [\![P]\!]_v \wedge [\![Q]\!]_v$
- $[\![P \to Q]\!]_v = [\![P]\!]_v \to [\![Q]\!]_v$
- $[\![\neg P]\!]_v = \neg [\![P]\!]_v$

where the constants and connectives $\bot, \top, \vee, \wedge, \neg, \to$ on the right side are interpreted in $\mathcal{H}$.

We introduce some notation:

- The notation $v, \Gamma \models P$ means that under the valuation v of Γ, P is true.
- The notation $\mathcal{H}, \Gamma \models P$ means that $v, \Gamma \models P$ for all $v : \Gamma \to \mathcal{H}$.
- The notation $\Gamma \models P$ means that $\mathcal{H}, \Gamma \models P$ for all Heyting algebras $\mathcal{H}$.
- The notation $\models P$ means that $\Gamma \models P$ with $\Gamma = \emptyset$.

For classical logic, we do the corresponding definitions, replacing the Heyting algebra $\mathcal{H}$ with the Boolean algebra $\mathcal{B}$.

Definition 2.2 (tautology)**.** A tautology is a formula P that is true under all valuations. We write $\models P$.

A well known example of a Boolean algebra is the following. Note that there exist other Boolean algebras.

Example 2.1. The Boolean algebra $\mathbb{B}$ is the set $\{0, 1\}$, with

- $a \vee b$ as $\max(a, b)$
- $a \wedge b$ as $\min(a, b) = a \times b$
- $\neg a$ as $1 - a$.

- $a \to b$ as $\neg a \vee b$
- $\top$ as 1
- $\bot$ as 0

$\triangle$

Theorem 2.1. *A formula in classical logic is true under all valuations to Boolean algebras iff it is true under all valuations to* $\mathbb{B}$*. Or more compact:*

$$\mathbb{B}, \Gamma \models P \iff \Gamma \models P$$

This theorem states that for verifying if $\Gamma \models P$, it is sufficient to test all assignments of atomic propositions to 0 or 1, i.e. to draw "truth tables".

Example 2.2. Consider subsets of $\mathbb{R}$, where $\texttt{int}(S)$ (the *interior*) is the largest open subset contained in the subset $S \subseteq \mathbb{R}$. An example of an Heyting algebra are all open subsets in $\mathbb{R}$, with

- $A \vee B$ as set union $A \cup B$
- $A \wedge B$ as set intersection $A \cap B$
- $\neg A$ as the interior of the set complement $\texttt{int}(A^C)$
- $A \to B$ as $\texttt{int}(A^C \cup B)$
- $\top$ as the entire set $\mathbb{R}$
- $\bot$ as the empty set $\emptyset$

Note that all these sets are open sets, as required (given that A and B are open). We will denote this Heyting algebra by $\mathcal{H}_{\mathbb{R}}$. $\triangle$

We can immediately find a valuation where the Principle of the Excluded Middle, $P \vee \neg P$ is not valid. Let $v : \{P\} \to \mathcal{H}_{\mathbb{R}}$, with $v(P) = (0, \infty)$. Then $[\![\neg P]\!]_v = \texttt{int}(\mathbb{R} \setminus (0, \infty)) = \texttt{int}((-\infty, 0]) = (-\infty, 0)$, so $[\![P \vee \neg P]\!]_v = (0, \infty) \cup (-\infty, 0) = \mathbb{R} - \{0\} \neq \mathbb{R} = [\![\top]\!]_v$.

Theorem 2.2. *A formula in intuitionistic logic is true under all valuations to Heyting algebras iff it is true in all valuations to* $\mathcal{H}_{\mathbb{R}}$*. Or more compact:*

$$\mathcal{H}_{\mathbb{R}}, \Gamma \models P \iff \Gamma \models P$$

This theorem states that verifying if $\Gamma \models P$, intuitionistically, comes down to searching for counter examples in open sets on the real line.

We have two central results:

Theorem 2.3 (Soundness Theorem)**.**

$$\Gamma \vdash P \implies \Gamma \models P$$

Stated in words: "what can be proved, is also true under all valuations".

Theorem 2.4 (Completeness Theorem)**.**

$$\Gamma \models P \implies \Gamma \vdash P$$

Stated in words: "what is true under all valuations, can also be proved".

These two theorems are true for both classical and intuitionistic logic, using Boolean and Heyting algebras respectively. These two theorems assert that our notions of "provable" and "true" coincide. A tautology is then a formula that can be derived with all assumptions discharged, i.e. what we usually call a *theorem*.

Example 2.3 (Classical, but not intuitionistical, tautologies)**.**

Double negation elimination, or Reduction ad absurdum (RAA), $\neg\neg P \to P$

The principle of excluded middle (PEM), $P \vee \neg P$

Peirce's law, $((P \to Q) \to P) \to P$

Proof by contradiction, $(\neg Q \to \neg P) \to (P \to Q)$

Negation of conjunction (de Morgan's laws), $\neg(P \wedge Q) \to (\neg P \vee \neg Q)$

Proof by cases, $(P \to Q) \to (\neg P \to Q) \to Q$ [5]

RAA implies PEM, $(\neg\neg P \to P) \to P \vee \neg P$ △

To prove this, one firstly concludes using truth tables that these are classical tautologies. To show that they are not intuitionistic tautologies, one finds a valuation into a Heyting algebra, e.g. $\mathcal{H}_{\mathbb{R}}$, where they are not true. The reader is suggested to try to construct such counter examples in $\mathcal{H}_{\mathbb{R}}$ for some of these propositions.

2.2 The Curry-Howard Isomorphism

Typed lambda calculus can be introduced with another motivation. Under the BHK-interpretation of proofs, proofs are constructions. If p is a construction and P is a formula, we let $p : P$ denote that p is a construction of P.

We now see a similarity to lambda calculus: If $q(p) : Q$ can be constructed given a $p : P$, then the function $\lambda p.q(p)$ is a construction of the formula $P \to Q$.

We get the following correspondence:

proof	term
proposition	type
assumption set	typing environment
implication	function
unprovable formula	empty type
undischarged assumption	free variable
normal derivation	normal form
tautology	combinator

The list can be extended, for example in order to find a correspondence for the other logical connectives. That is what we will do now.

2.2.1 Additional Type Constructors

To get a correspondence for $\vee$, $\wedge$ and $\bot$, we extend the simply typed lambda calculus with new constructs.

[5] or $(P \to Q) \wedge (\neg P \to Q) \to Q$ by currying

We add a new type $\bot$, which is empty. This type is hard to interpret. It can be interpreted as contradiction. A term of this type can be interpreted as abortion of computation. Another interpretation that a term of type $\bot$ is an "oracle", that can produce anything. It has a destruction rule: given an oracle, we can produce a variable of any type. Since such an oracle is impossible, the type $\bot$ is an empty type.

For $\wedge$ and $\vee$, we introduce two new type constructors alongside our old ($\to$). They also have *destructors*, or elimination rules, as well as reduction rules. The type $\bot$ only has a destructor.

$$\Gamma, x : A \vdash x : A \quad \text{(ax)}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \lambda\ abs \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \lambda\ app$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \texttt{pair}(M, N) : A \wedge B}\ pair$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \texttt{fst}(M) : A}\ fst \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \texttt{snd}(M) : B}\ snd$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{inl}(M) : A \vee B}\ inl \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \texttt{inr}(M) : A \vee B}\ inr$$

$$\frac{\Gamma \vdash M : A \vee B \quad \Gamma, a : A \vdash N_1 : C \quad \Gamma, b : B \vdash N_2 : C}{\Gamma \vdash \texttt{case}(M, \lambda a.N_1, \lambda b.N_2) : C}\ case$$

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \texttt{any}(M) : A}\ any \qquad \text{where } A \text{ is any type}$$

Figure 4: Typing judgements for λ-calculus.

Function If A_1, A_2 are types, then $A_1 \to A_2$ is a type. It is interpreted as a function.

The data constructor for $\to$ is λ. It has the destructor $\texttt{apply}(M, N)$, but we typically write just MN. The reduction rule is

$$\begin{aligned} & \texttt{apply}(\lambda x.M, N) && \to_\beta M[x := N] \\ \text{or simply} \quad & (\lambda x.M)N && \to_\beta M[x := N] \end{aligned}$$

Pair If A_1, A_2 are types, then $A_1 \wedge A_2$ is a type. It is interpreted as a pair (or Cartesian product).

The data constructor for $\wedge$ is $\texttt{pair}(,)$. It has the destructors $\texttt{fst}(M)$ and $\texttt{snd}(M)$ (*projection*) and the reduction rules

$$\texttt{fst}(\texttt{pair}(M, N)) \to_\wedge M$$
$$\texttt{snd}(\texttt{pair}(M, N)) \to_\wedge N$$

Injection If A_1, A_2 are types, then $A_1 \vee A_2$ is a type. It is interpreted as a union type, which can be analysed by cases.

The data constructors for $\vee$ are $\texttt{inl}()$ and $\texttt{inr}()$. It has the destructor $\texttt{case}(,,)$. and the reduction rules

$$\texttt{case}(\texttt{inl}(M), \lambda a.N_1, \lambda b.N_2) \to_\vee (\lambda a.N_1)M$$
$$\texttt{case}(\texttt{inr}(M), \lambda a.N_1, \lambda b.N_2) \to_\vee (\lambda b.N_2)M$$

Empty Type We add the base type $\bot$, which does not have a constructor. It has a destructor $\texttt{any}()$, without an reduction rule. The type of $\texttt{any}(M)$ is any type we want.

Example 2.4 (Injection type)**.** The type $A \vee B$ can be interpreted as a type where either A or B works for the purpose. An example from programming can be the operator "less than", $<$. It can compare terms of both `int` and `real`. The type for such a term $<$ is then $(\texttt{int} \vee \texttt{real}) \wedge (\texttt{int} \vee \texttt{real}) \to \texttt{bool}$.

The term $<$ itself does a case analysis on the type of its two arguments, to determine how they should be compared. △

Our new typing judgements are shown in Figure 4. Note that if we strip our typing statements of their variables and just keep the types, we get exactly the rules in Figure 3. We extend our "glossary" from previously:

logical connective	type constructor
introduction	constructor
elimination	destructor
implication	function
conjunction	pair
disjunction	injection
absurdity	the empty type

Theorem 2.5 (The Curry-Howard Isomorphism (for intuitionistic logic))**.**

Let $\texttt{types}(\Gamma)$ *denote* $\{A \mid x : A \in \Gamma\}$.

(i) Let Γ *be a typing environment. If* $\Gamma \vdash M : A$ *then* $\texttt{types}(\Gamma) \vdash A$.

(ii) Let Γ *be an assumption set. If* $\Gamma \vdash A$, *then there are* Γ', M *such that* $\Gamma' \vdash M : A$ *and* $\texttt{types}(\Gamma') = \Gamma$.

Proof. Induction on the derivation. The set Γ' can be constructed as $\Gamma' = \{x_A : A \mid A \in \Gamma\}$, where x_A denotes a fresh variable for every A. □

This is a quite remarkable result. To show that a formula P is intuitionistically valid, it is sufficient to construct a term M of type P, i.e. an inhabitant. As a bonus, we also get a proof, since the derivation of the typing judgement

$M : P$ also contains the proof of the formula P, which is retrieved by removing all variables in each typing judgement.

Conversely, to construct a term of a type A, one can construct one given a proof that A is a valid intuitionistic formula.

Remark 2.1. Note that the word "isomorphism" is not a perfect description of the correspondence. Some types can have several inhabitants, still different after α-conversion and normalisation, if there are several variables of the same type. The type derivation thus contains more information than an intuitionistic deduction (in sequent notation), because the variables reveal *which* construction is used in which situation. A solution is to use the tree notation, with annotations of which assumption is discharged at which step in the deduction.

2.2.2 Proofs as Programs

Under the Curry-Howard correspondence, proofs are terms and propositions are types. Since the λ-calculus is a Turing complete functional programming language, a proof can be seen as a program. When we find an intuitionistic proof for a proposition, we have also an algorithm for constructing an object having that property. Conversely, if we have an algorithm for constructing an object having a property, given that the algorithm is correct, we can also construct a proof from it showing the correctness of the algorithm. This is the content of the *proofs-as-programs* principle.

It was discovered in the 1990's that the correspondence can be extended to classical logic, if lambda calculus is extended to $\lambda\mu$-calculus. This will also be our main topic further on. Although we have argued for the non-constructiveness of classical proofs, where algorithms cannot be constructed, we will try to find some computational content of the classical proofs.

3 Lambda Calculus with Control Operators

We will introduce *control operators* into lambda calculus. Control operators allows abstraction of the execution state of the program, enabling program jumps, like in imperative languages. We will look at examples from the functional programming language Scheme, which is based on lambda calculus. We will work with untyped lambda calculus most of this chapter and therefore Scheme is suitable since it is dynamically typed.

To understand control operators, we will first introduce the concepts of *contexts*, *evaluation strategies* and *continuations*.

With the notion of control operators, we will be able to extend the Curry-Howard isomorphism to classical logic in the next chapter.

3.1 Contexts

Definition 3.1. A *context* is a term with a hole in it, i.e. a subterm has been replaced with a "hole", denoted by $\Box$.

Example 3.1. For example, in the term

$$(\lambda x.\lambda y.x)(\lambda z.z)(\lambda x.\lambda y.y)$$

the subterm $\lambda z.z$ can be replaced by a hole $\Box$, to get the context

$$(\lambda x.\lambda y.x)\Box(\lambda x.\lambda y.y)$$

If we call this context E, then substituting a term M for the hole in E, which is denoted $E[M]$, will give us the term

$$(\lambda x.\lambda y.x)M(\lambda x.\lambda y.y)$$

$\triangle$

In lambda calculus, if we allow any subterm to be replaced by a hole, we can define contexts as

$$E ::= \ \Box \mid EM \mid ME \mid \lambda x.E \tag{1}$$

where M is a lambda-term and x is a lambda-variable.

We will make several more limited definitions of contexts for different purposes.

3.2 Evaluation Strategies

One use of contexts is to define an evaluation strategy. An evaluation strategy specifies the order of evaluation of terms. In untyped and simply typed lambda calculus, we have the confluence property which guarantees that the normal form is unique. That is, two computations of the same term, using different sequences of β-reduction, cannot yield two different results (if we by "result" mean a term in normal form).

In both these calculi, all evaluation strategies are thus allowed. Using the notion of contexts, we can express the β-reduction rules as

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]} \qquad \frac{M \to_\beta N}{E[M] \to_\beta E[N]}$$

which captures the idea that redices in subterms can also be reduced. Writing these rules explicit, using the definition of contexts in equation (1), we get

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]}\ \beta \qquad \frac{M \to_\beta N}{PM \to_\beta PN}\ \lambda_{\text{right}}$$

$$\frac{M \to_\beta N}{MP \to_\beta NP}\ \lambda_{\text{left}} \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N}\ \lambda_{\text{body}}$$

In other calculi, one may choose to restrict the possible evaluation strategies. It might be required to get a confluent calculus. It can also be of interest when implementing an programming language, in order to get computational efficiency or a specific behaviour. For example, the rule λ_{body} is typically not used in programming languages — the body M of a function $\lambda x.M$ is not evaluated before application.

3.2.1 Common Evaluation Strategies

The following defines an evaluation strategy where arguments of a function are evaluated before the function is applied, a so called *call-by-value* strategy. It is commonly used in programming languages, for example in Scheme.

Definition 3.2 (Call-by-value lambda calculus)**.** Let x range over variables, M over lambda terms, and define values as variables or λ-abstractions:

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

and define contexts as follows:

$$E ::= \square \mid VE \mid EM$$

Then define β-reduction as

$$\frac{}{(\lambda x.M)V \to_\beta M[V/x]}\ \beta_{cbv} \qquad \frac{M \to_\beta N}{E[M] \to_\beta E[N]}\ C_{cbv}$$

The left side rule (β_{cbv}) specifies that arguments to a function are evaluated before function application. The context definition specifies that arguments are evaluated left to right. [6]

To better understand why the contexts define a certain evaluation strategy, the reader is suggested to construct an arbitrary lambda term, use the Backus-Naur-grammar definition of contexts to derive a syntax tree and use this syntax tree to derive the order of evaluation of the term.

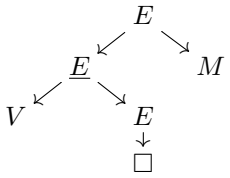
[6] Most call-by-value programming languages specify the order of argument evaluation. Scheme and C do not. In Scheme and other functional languages it does not matter since (the pure part of) the language has no side-effects. For details on Scheme evaluation order, see R7RS section 4.1.3 [R7R13].

In C it is undefined and varies between compilers.

Example 3.2. The term below, a non-value function applied to a non-value argument, separated by brackets for legibility,

$$[(\lambda x.(\lambda z.z)x)(\lambda y.y)][(\lambda a.\lambda b.a)(\lambda c.c)]$$

has, in a left-to-right call-by-value calculus, the context derivation



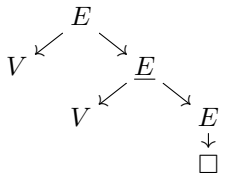
The underlined context marks the redex closest to the hole, which is the next redex to reduce. After one step of reduction, the term looks like

$$[(\lambda z.z)(\lambda y.y)][(\lambda a.\lambda b.a)(\lambda c.c)]$$

and happens to have the same context derivation. After another step of reduction, the function is now to a value and the evaluation moves to the argument. The term looks like

$$[(\lambda y.y)][(\lambda a.\lambda b.a)(\lambda c.c)]$$

and the context looks like



△

If instead right-to-left call-by-value evaluation is wanted, one defines contexts as

$$E ::= \square \mid EV \mid ME$$

We have already encountered an evaluation strategy: In untyped lambda calculus, the Normal order theorem (Theorem 1.3) states that normal order reduction will always reach the normal form if it exists.

Definition 3.3 (Normal order evaluation)**.** Define contexts by

$$E ::= \square \mid EM \mid \lambda x.E$$

and use the usual rule for β-reduction. This defines a strategy where the leftmost redex is reduced first and the insides of functions are reduced before application. This is also called the leftmost-innermost reduction. [7]

[7] As commented on in Theorem 1.3, both leftmost-innermost and leftmost-outermost strategies will reach the normal form if it exists. Normal order reduction is typically defined to be leftmost-outermost, but leftmost-innermost is easier to define with contexts, which is why leftmost-innermost was chosen in this example.

Another well-known evaluation strategy is call-by-name, which we will use later.

Definition 3.4 (Call-by-name evaluation)**.** Define contexts by

$$E ::= \Box \mid EM$$

and use the usual rule for β-reduction. This defines a strategy where the leftmost redex is reduced first, but the insides of functions are *not* reduced.

The memoized version of call-by-name is called *call-by-need* and is also known as *lazy evaluation* in programming.

3.2.2 Comparison of Evaluation Strategies

Below is an informal argument that the Normal Order Theorem (Theorem 1.3) holds.

Definition 3.5. Given an evaluation strategy, we say that an evaluation of a term has *terminated* if the term is a value and the evaluation strategy cannot reduce the term further.[8]

The criterion that an evaluation of a term has terminated is weaker than the evaluation having reached normal form.

If the evaluation of term never terminates, it is because a subterm loops or reproduces itself in some way, for example the Ω-combinator (see Example 1.1).

In certain cases, some evaluation strategies can fail to terminate while another may succeed. Another evaluation strategy may choose another way to evaluate an expression, so that the problematic term never has to be evaluated, because it is never used. For example, a constant function applied to the Ω-combinator:

$$(\lambda x.a)\Omega$$

A call-by-value strategy will try to reduce the argument before application, but end up in an infinite reduction of the Ω-combinator. A normal order reduction or the call-by-name reduction will not evaluate the term before application, and the term evaluates to a. In general, the call-by-name strategy never evaluates a term until it is used, and is therefore the evaluation strategy most likely to terminate.

Still, call-by-name leaves the body of a function unevaluated, so if the function body contains redices, it is not in normal form. Normal order reduction is like call-by-name, except that it also evaluates function bodies (first or last), which is why it will find the normal form if it exists. However, if the function body contains something that loops, normal order reduction will become caught in the loop, while call-by-name terminates.

[8] We only consider evaluation strategies where the term after termination is a value (a variable or an abstraction), like those mentioned above. Such evaluation strategies are the only meaningful ones, but it is technically possible to construct one that terminates without producing a value.

3.3 Continuations

Definition 3.6. A *continuation* is a procedural abstraction of a context. That is, if E is a context, then $\lambda z.E[z]$ is a continuation. The variable z is assumed to be fresh for the context.

What we mean by a continuation slightly depend on what definition of contexts we are using. The idea is the same though — it is a function $\lambda z.M$, where z occurs exactly once in M. If M is to be evaluated, evaluation will start at the point where z occurs.

3.3.1 Continuations in Scheme

From a programming perspective, a different description of a continuation can be given: *A continuation is a procedure waiting for the result (of the current computation).* Consider the following Scheme code

```
(define divide
  (lambda (a b)
    (/ a b)))

(display (divide 5 3))
```

The expression `(divide 5 3)` is evaluated in the context `(display [])`. A procedural abstraction of that context has the form `(lambda (z) (display z))`. Now let us rewrite the code, so that we instead send this continuation to the current computation, `divide`, and let `divide` apply this function to the result once it is done. This continuation is a function waiting for the result of the computation of the division.

```
(define divide
  (lambda (a b k)
    (k (/ a b))))

(divide 5 3 (lambda (z)
              (display z)))
```

Now `divide` is passed its continuation as the final parameter `k`. This is the idea of *continuation-passing-style*.

3.3.2 Continuation Passing Style

In continuation-passing-style programming (CPS), one embraces this idea:

> Every function is passed its continuation as a final parameter.

All functions are then rewritten to follow this idea. The resulting expressions will look like the old expressions written inside out.

Example 3.3 (Pythagoras' formula for hypotenuse)**.** Using Pythagoras' theorem, we have the formula for the hypotenuse

```
(define pythagoras
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))
```

which is rewritten into CPS as

```
(define pythagoras
  (lambda (a b k)
    (mul a a (lambda (a2)
               (mul b b (lambda (b2)
                          (add a2 b2 (lambda (a2b2)
                                       (sqrt a2b2 k)))))))))
```

assuming that `mul`, `add` and `sqrt` are also rewritten into CPS version. △

An effect of writing a program in CPS are that all middle step terms, such as `a2`, i.e. a^2, will be explicitly named. Another effect is that the evaluation order becomes explicit — a^2 is calculated before b^2 in the `pythagoras`-example.

With CPS, all functions have access to their continuations. That means that functions potentially have access to *other* continuations, since they can be passed around. In the below example, `safe-divide` is a helper function that divides numbers. What is interesting is that it is given another continuation `fail` in addition to its own, `success`. The parent procedure that calls `safe-divide` can here send another continuation, maybe a top level procedure, to be called if an error occurs (`b` is zero). Then `safe-divide` can abort computation by ignoring the parent procedure and instead call the top level procedure `fail`. Now we have a functional program that can jump!

Example 3.4 (CPS with multiple continuations)**.** For an example program, see Appendix A.1.

```
(define safe-divide
  (lambda (a b fail success)
    (if (= b 0)
        (fail "Division by zero!")
        (success (/ a b)))))
```

△

Since continuations can be passed around, they can also be invoked (called) multiple times, producing other interesting results. That is left to the reader to explore.

3.4 Control Operators

The term *control flow* refers to the order in which statements of a program are executed. We are usually not too concerned with the control flow when working with functional programming, because there are no side effects. The addition of continuations to a language will however add more imperative features.

We established that the hole in a context represents the current point of evaluation. It is therefore also the current position of the control flow. An abstraction of a context — a continuation — is therefore a function that allows

to return to a certain point of control flow when we invoke it. In a functional language, calling another function is typically the only way to jump to another part of the program, but with access to continuations, the program may jump anywhere.

To use these ideas, our program must have access to the context. Either we can write the entire program in continuation passing style. Or we can extend the language with a *control operator* operating on the context. Such operators gives terms full access to the context where they are applied. An abstraction of the current context is called the *current continuation*.

Since we are operating on contexts, and therefore on the state of the program, we may suspect that the order of evaluation of terms — our evaluation strategy — can affect the outcome of the program. Not only will the choice of an evaluation strategy affect whether a program halts — we may also lose confluence.

3.4.1 Operators $\mathcal{A}$ and $\mathcal{C}$

Following Griffin 1990 [Gri90] we introduce two control operators for manipulating the contexts. They were originally introduced by Felleisen 1988 [FWFD88]. It can be assumed below that we use either the call-by-value or call-by-name lambda calculus — which one does not matter, unless otherwise is specified.

The operator $\mathcal{A}$, for *abort*, abandons the current context. It has the reduction rule

$$E[\mathcal{A}(M)] \to_{\mathcal{A}} M$$

The operator $\mathcal{C}$, for *control*, abstracts on the current context and applies its argument to that abstraction. [9]

$$E[\mathcal{C}(M)] \to_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z])$$

The term $\lambda z.\mathcal{A}(E[z])$ is an abstraction of a context, a continuation, and furthermore the same context where $\mathcal{C}$ was applied — the *current continuation*. Expressing the reduction rule in words, we say that $\mathcal{C}$ applies its argument to the current continuation.

What does the application do? The continuation becomes bound to a variable in M, let us call it k. If k never is invoked (applied), then M will return normally. However if k is invoked, then it is applied to some subterm N of M in some context E_1:

$$\begin{aligned} E[\mathcal{C}(M)] &\to_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]) \\ &\dots \\ &\to E_1[kN] \\ &\to E_1[(\lambda z.\mathcal{A}(E[z]))N] \\ &\to E_1[\mathcal{A}(E[N])] \\ &\to E[N] \end{aligned}$$

[9] Note that, since we excluded the rule $E ::= \dots \mid \lambda x.E$, the term $M\lambda z.\mathcal{A}(E[z])$ will not exit the abstraction and reduce to $E[z]$. The insides of functions are not evaluated before application.

Why is not the continuation simply $\lambda z.E[z]$? The operator $\mathcal{A}$ is also included to ensure that control exits the context where the continuation is applied. When we are done evaluating $E[N]$, we do not want to return to E_1. Compare this to Example 3.4 with the `safe-div` procedure, where application of the `fail`-continuation exits the parent procedure of `safe-div`, so that control never returns to the parent procedure.

Note that $\mathcal{A}$ can be defined from $\mathcal{C}$ by taking a dummy argument d:

$$E[\mathcal{A}(M)] \to_{\mathcal{A}} E[\mathcal{C}(\lambda d.M)]$$

3.4.2 Operators $\mathcal{A}$ and $\mathcal{C}$ in Racket

Felleisen started in 1990 the team behind the programming language Racket, a variant of Scheme. Racket has several control operators, including the operators `abort` and `control` with behaviour corresponding to $\mathcal{A}$ and $\mathcal{C}$.[10]

The syntax looks like

$$(\texttt{abort}\ expr)$$
$$(\texttt{control}\ k\ expr)$$

where k is the name of the variable to which the continuation is bound. If $M = \lambda k.N$, then the Racket term corresponding to $\mathcal{C}(M)$ is `(control k N)`. The operator `control` is evaluated like

$$E[(\texttt{control}\ k\ expr)] \to_{\mathcal{C}} ((\texttt{lambda}\ (k)\ expr)\ (\texttt{lambda}\ (z)\ E[z]))$$

Example 3.5 (Operators `control` and `abort` in Racket)**.** From the Racket shell:

```
> (control k (* 3 2)) ;; (* 3 2)
6
> (abort (* 3 2)) ;; (* 3 2)    (same as previous)
6
> (+ 4 (control k (* 3 2))) ;; (* 3 2)
6
> (+ 4 (abort (* 3 2))) ;; (* 3 2)    (same as previous)
6
> (+ 4 (control k (k (* 3 2)))) ;; (+ 4 (* 3 2))
10
> (+ 4 (control k (* 3 (k 2)))) ;; (* 3 (+ 4 2))
18
> (+ 4 (control k (* 3 (control k1 (k 2))))) ;; (+ 4 2)
6
> (+ 4 (control k (* 3 (abort (k 2))))) ;; (+ 4 2)    (same as previous)
6
```

△

[10] The operators are available in `racket/control`-module. Put `(require racket/control)` at the top of your file.

3.4.3 `call/cc`

Scheme provides the operator `call-with-current-continuation`, or `call/cc` for short, for manipulating the context. It has a behaviour similar to $\mathcal{C}$, except that it always returns to the context where it was called (which $\mathcal{C}$ will not do if the continuation never is invoked). Adding such an operator $\mathcal{K}$ would have the following reduction rule

$$E[\mathcal{K}(M)] \rightarrow_{\mathcal{K}} E[M\lambda z.\mathcal{A}(E[z])]$$

If the evaluation exits normally, it will return to the outer E. If the continuation is invoked, it will return to the inner E.

Example 3.6 (Comparison between `control` and `call/cc`)**.** If the continuation `k` is never invoked, then `control` and `call/cc` act differently. Operator `control` will abort, i.e. skip the outer context `(+ 4 [])`, whereas `call/cc` will return to that context.

```
> (+ 4 (control k (* 3 2))) ;; (* 3 2)
6
> (+ 4 (call/cc (lambda (k) (* 3 2)))) ;; (+ 4 (* 3 2))
10
```

△

Example 3.7 (Loss of confluence)**.** The result of a computation can depend on the evaluation order. The same expression can have different results in a call-by-value and call-by-name calculus. This example can even have different results in (different implementations of) Scheme, since the order of evaluation of arguments is not specified in the standard.

In a call-by-name calculus, it would depend on the (probably hidden) implementation of the function `+`.

```
(call/cc
 (lambda (k)
   (call/cc
    (lambda (l)
      (+ (k 10) (l 20))))))
```

If k is evaluated first, the result is 10, but if l is evaluated first, it is 20. △

3.4.4 Use of Control Operators

As pointed out earlier, control operators allow us to jump to any part of the program. Is is basically the equivalent of the imperative `GOTO`. Abstracting on the current context corresponds to setting a label, and invoking the continuation jumps to that label.

For this reason, Scheme's `call/cc` has received similar criticism as `GOTO`. Excessive use of `call/cc` can lead to a program where the control flow is very hard to follow — so called "spaghetti code". However, there are some useful high-level applications of `call/cc` such as exceptions, threads and generators, to mention a few.

A `throw`/`catch`-mechanism — exceptions — is particularly easy to simulate, because that is exactly how `call/cc` works. We repeat that `call/cc` has the reduction rule

$$E[\mathcal{K}(M)] \to_{\mathcal{K}} E[M\lambda z.\mathcal{A}(E[z])]$$

During the evaluation of $E[\mathcal{K}(M)]$, the continuation is bound to some variable that we again refer to as k. We then have $M = \lambda k.T$ for some T.

$$\begin{array}{rcl} E[\mathcal{K}(M)] & \to_{\mathcal{K}} & E[M\lambda z.\mathcal{A}(E[z])] \\ & \to & E[T[k := \lambda z.\mathcal{A}(E[z])]] \end{array}$$

If k is never invoked, we have a normal return to the context E. If, during the evaluation of T, k is applied to some argument N, the rest of the term T is ignored and N is returned. We have an exceptional return where N is *thrown* to the context E.

3.5 Typed Control Operators

So far we have only considered untyped lambda calculus with our control operators. Griffin 1990 [Gri90] attempts to give the operator $\mathcal{C}$ a type.

We have the reduction rule

$$E[\mathcal{C}(M)] \to_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z])$$

and we want that evaluation preserves typing (subject reduction), so both sides should have the same type.

Let $E[z]$ be of type B if $z : A$. The continuation should thus be of type $A \to B$. The term M takes this as an argument, and returns $E[z] : B$, so it has type $(A \to B) \to B$. So what about preservation of types? Well, if $E[\mathcal{C}(M)]$ should be a valid substitution, and the hole in E has type A while M has type $(A \to B) \to B$, then $\mathcal{C}(M) : A$, so $\mathcal{C} : ((A \to B) \to B) \to A$.

This derivation has an important consequence. We noted before that $\mathcal{A}(M) = \mathcal{C}(\lambda d.M)$ for some dummy variable d. Then if M is a closed term of type B, then we can deduce any type A for d. That is, from the typing judgement $M : B$ we can derive any typing judgement $d : A$. To get a logically consistent typing, we must then have that B is the proposition which has no proof, $B = \bot$. Then operator $\mathcal{C}$ has then the type of double negation elimination, $\neg\neg A \to A$. [11]

This conclusion suggests a connection between control operators and classical logic. We will now turn to an extension of the lambda calculus entirely based on this idea, the $\lambda\mu$-calculus.

[11] The reader may point out that if $B = \bot$, then $E[] : \bot$, so the rule can never be applied because $\bot$ is not inhabited. Griffin suggests a solution to this problem, see further [Gri90].

4 $\lambda\mu$-calculus

After Griffin's discovery of the connection between control operators and classical logic, several calculi have been constructed. The most known and studied is the $\lambda\mu$-calculus, originally introduced by Parigot in [Par92], which will allow us to extend the Curry-Howard isomorphism to classical logic.

The $\lambda\mu$-calculus is the simply typed λ-calculus enriched with a binder μ for abstracting on the current continuation. These continuations will be referred to as *addresses* and have separate variables. This calculus is typed and satisfies subject reduction, confluence and strong normalisation.

4.1 Syntax and Semantics

We extend the λ-calculus with a binder μ for abstracting on continuations. We will initially just consider the simply typed lambda calculus with the type constructor $\to$. The typing judgements for our new terms will correspond to the derivation rules in classical logic, considering only the connective $\to$ (see Figure 5). An extension with pairs and injections will be discussed in Section 4.5.

4.1.1 Terms

Definition 4.1 (Raw $\lambda\mu$-terms)**.** The raw terms of the $\lambda\mu$-calculus are defined as follows:

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

where x is a λ-variable and α is a μ-variable. The μ-variables are also referred to as *addresses*.

Remark 4.1. Convention: Roman letters will be used for λ-variables and Greek letters for μ-variables.

We have extended the lambda-calculus with two new constructs, address application $[\cdot]$ on terms and address abstraction $\mu\_.$ that binds free addresses. These two constructs are control operators.

- *Address abstraction* abstracts on the current context and the current continuation is bound to that address. Evaluation of $\mu\alpha.M$ evaluates M with the addition that α is a reference to the current context.
- *Address application* $[\alpha]M$ applies an address α (a continuation) to a term M.

That is, if we have an expression

$$E[\mu\alpha.\cdots[\alpha]M\cdots]$$

where α is applied to the term M, the evaluation of M will exit the current context and resume where α is bound, i.e. in the context E. Thus the resulting expression of this jump will be $E[M]$.

Note that this is somewhat informal since we have neither defined the contexts we will use, nor what reduction rules we will allow.

Addresses are thus a special type of variables used for continuations. Addresses can only be invoked and abstracted, not passed as arguments to functions like normal λ-terms are.

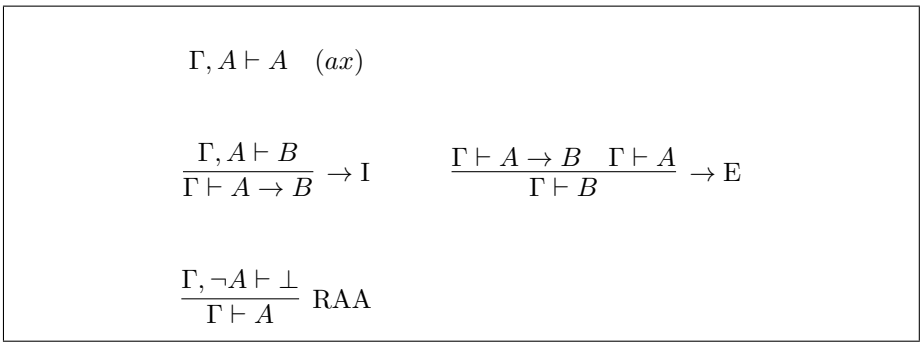


Figure 5: Derivation rules for our subset of classical logic.

We introduce some additional notation. By writing $\mu\_.M$ we will mean $\mu\gamma.M$ for some γ not free in M. By $\mathrm{FV}_\mu(M)$ we denote the set of free addresses in M.

Remark 4.2 (First class continuations)**.** Since addresses cannot be passed directly to functions, in programming terms, we would say that addresses are not "first-class objects". This means that they cannot appear anywhere where a normal λ-term may appear — single addresses α are not listed as a term in our inductive definition of $\lambda\mu$-terms. However an address α can be passed to a function by the term $\lambda x.[\alpha]x$. Anywhere we want a first class address α, we can use the term $\lambda x.[\alpha]x$ instead. So in practise, we have first class addresses/continuations.

Remark 4.3. The use of square brackets for address application has nothing to do with contexts or substitution. This notation for address application follows the convention, which happens to be similar.

Remark 4.4 (Precedence rules)**.** Convention: The control operators have lowest precedence. $[\alpha]MN$ means $[\alpha](MN)$.

4.1.2 Typing and Curry-Howard Correspondence

In this chapter we consider a subset of classical logic, with the connective $\to$, the constant $\bot$, and $\neg$, where $\neg A$ abbreviates $A \to \bot$. The derivation rules for natural deduction are shown in Figure 5. Note that the rule RAA replaces the rule $\bot E$ as a stronger version; $\bot E$ is given by RAA without discharging any assumptions.

We will now add types to the $\lambda\mu$-calculus. Consistent with the conclusions of Griffin, we will let addresses (continuations) have the type $A \to \bot$, if A is the type of the term M that the address is applied to. We abbreviate $A \to \bot$ as $\neg A$. An address application $[\alpha]M$ thus has type $\bot$. It can be thought of as an aborted computation. Abstraction of addresses restores the type σ that the address was applied to.

The typing judgements for $\lambda\mu$-calculus are shown in figure 6. Comparing the typing judgements with the rules for natural deduction, we see what our extended Curry-Howard correspondence will consist of: address abstraction (μ *abs*) corresponds to the rule (RAA). Address application (μ *app*) has no

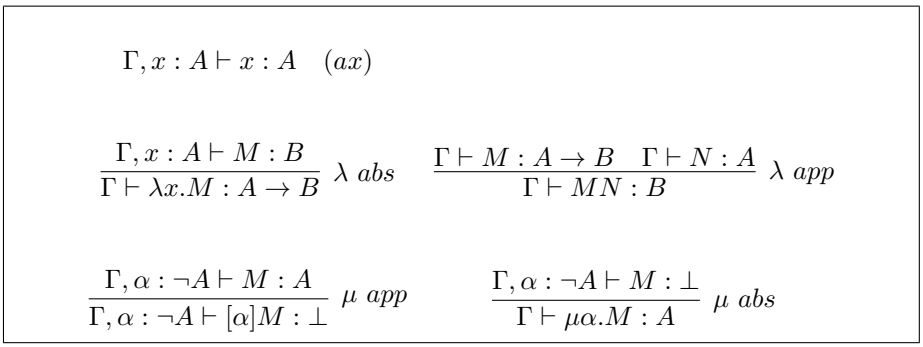


Figure 6: Typing judgements for $\lambda\mu$-calculus.

corresponding rule, but it corresponds to deriving $\bot$, using rule $(\to E)$ with $\neg A$ and A as premises. [12]

Note particularly when assumptions are discharged.

We can now construct a Curry-Howard isomorphism for classical logic. The construction is taken from [SU06].

Theorem 4.1 (The Curry-Howard Isomorphism (for classical logic and the type constructor $\to$.))**.**

Let $\texttt{types}(\Gamma)$ *denote* $\{A \mid x : A \in \Gamma\}$.

(i) Let Γ *be a typing environment. If* $\Gamma \vdash x : A$ *then* $\texttt{types}(\Gamma) \vdash A$.

(ii) Let Γ *be an assumption set. If* $\Gamma \vdash A$, *then there are* Γ', M *such that* $\Gamma' \vdash M : A$ *and* $\texttt{types}(\Gamma') = \Gamma$.

Proof. Induction on the derivation. The set Γ' can be constructed as $\Gamma' = \{x_A : A \mid A \in \Gamma\}$, where x_A denotes a fresh variable for every proposition A. The construction in (ii) is not quite trivial. Addresses are introduced with (ax) and applied using $(\lambda\ app)$, where the latter is only valid for $\lambda\mu$-terms (single addresses are not $\lambda\mu$-terms). This makes no distinction between addresses and variables. It is only when addresses become bound that there is a distinction. Therefore the translation from the rule RAA, binding addresses, must be done with some care. We consider two cases when we apply the rule RAA (see again Figure 5). The premise is $\Gamma, \neg A \vdash \bot$ and the conclusion is $\Gamma \vdash A$. Is $\neg A \in \Gamma$?

- $\neg A \in \Gamma$ (the assumption $\neg A$ is not discharged).

  Introduce a new variable α (i.e. not on the form $x_{\neg A}$, so $\alpha \notin \Gamma'$). By the induction hypothesis, there is $\Gamma' \vdash M : \bot$. Take the typing judgement to be $\Gamma' \vdash (\mu\alpha : \neg A.M) : A$

- $\neg A \notin \Gamma$ (the assumption $\neg A$ is discharged).

  By the induction hypothesis, there is $\Gamma', x_{\neg A} : \neg A \vdash M : \bot$, with $x_{\neg A} : \neg A \notin \Gamma'$ (there is a variable $x_{\neg A}$ in the typing environment from previously). That means that $x_{\neg A}$ can be free in M, but it is not on the form $\ldots[x_{\neg A}]\ldots$ as required for addresses. We replace all occurrences

[12] Every derivation of $\bot$ does not correspond to an address application though. It might as well be a normal application $(\lambda\ app)$, as variables, not only addresses, may have type $\neg A$.

of $x_{\neg A}$ with $\lambda z.[x_{\neg A}]z$ which gives us a valid term, and then bind the address. That is, we take

$$\Gamma', x_{\neg A} : \neg A \vdash (\mu x_{\neg A} : \neg A.M[x_{\neg A} := \lambda z.[x_{\neg A}]z]) : A$$

as the typing judgement.

□

4.1.3 Restricted Terms

In practise we will add a requirement on the $\lambda\mu$-terms: address application and abstraction may only appear in pairs. Then an address invocation will still have a normal return type.

Definition 4.2 (Restricted $\lambda\mu$-terms)**.** The restricted[13] terms of the $\lambda\mu$-calculus are defined as follows:

$$M, N ::= x \mid \lambda x.M \mid MN \mid \mu\alpha.C$$

and the commands

$$C ::= [\alpha]M$$

where x is a λ-variable and α is a μ-variable.

If we ignore typing, any raw term can be transformed to a restricted term, by replacing

- all single $[\alpha]M$, having no accompanying abstraction, to $\mu\_.[\alpha]M$
- all single abstractions $\mu\alpha.M$, having no accompanying application, to $\mu\alpha.[\alpha]M$.

However, this does not work well with typing, because we then require that α is applicable to M (i.e. $\mu\alpha.[\alpha]M$ has the same type as M), which is not always the case.

Most results work well with both raw and restricted terms, but where one of them will be required, this will be stated.

4.2 Computation Rules

The next step is to define reduction rules, to be able to compute values of terms.

4.2.1 Substitution

To define the reduction rules for our extension, we first need to define contexts as well as substitution for addresses, so called *structural substitution*. The $\lambda\mu$-calculus is a call-by-name calculus, so we will use call-by-name contexts (compare Definition 3.4).

Definition 4.3 ($\lambda\mu$-contexts)**.** The $\lambda\mu$-contexts are defined recursively

$$E ::= \Box \mid EN$$

where N is a $\lambda\mu$-term.

[13] called "restricted" as in [SU06]

Substitution of an address and a context for an address in a term, denoted $M[\alpha := \beta E]$, recursively replaces all commands $[\alpha]N$ with $[\beta]E[N']$, where $N' := N[\alpha := \beta E]$. For example if $E = \Box$, this will just replace all free occurrences of α with β.

Definition 4.4 (Structural substitution)**.** Structural substitution is recursively defined:

$$\begin{array}{rcll} x[\alpha := \beta E] & := & x & \\ (\lambda x.M)[\alpha := \beta E] & := & \lambda x.M[\alpha := \beta E] & \\ (MN)[\alpha := \beta E] & := & M[\alpha := \beta E]N[\alpha := \beta E] & \\ ([\alpha]M)[\alpha := \beta E] & := & [\beta]E[M[\alpha := \beta E]] & \\ ([\gamma]M)[\alpha := \beta E] & := & [\gamma]M[\alpha := \beta E] & \text{if } \gamma \neq \alpha \\ (\mu\gamma.C)[\alpha := \beta E] & := & \mu\gamma.C[\alpha := \beta E] & \end{array}$$

As with usual substitution of λ-variables, we want the structural substitution to be capture avoiding, both for λ- and μ-variables.

4.2.2 Reduction

Reduction in $\lambda\mu$-calculus is given by the following rules:

$$\begin{array}{rcll} (\lambda x.M)N & \to_\beta & M[x := N] & \\ \mu\alpha.[\alpha]M & \to_{\mu\eta} & M & \text{if } \alpha \notin \mathrm{FV}_\mu(M) \\ [\beta]\mu\alpha.C & \to_{\mu R} & C[\alpha := \beta\Box] & \\ (\mu\alpha.C)M & \to_{\mu C} & \mu\alpha.C[\alpha := \alpha(\Box M)] & \end{array}$$

The closure of these rules are denoted by $(\to_\mu)$.

The first rule $(\to_\beta)$ is the usual β-reduction. The second rule $(\to_{\mu\eta})$ is the address equivalence to η-reduction, hence its name. It corresponds to having a proof Σ of P, assuming $\neg P$, deriving $\bot$ and then applying the RAA-rule, giving P again. If $\neg P$ is not discharged in Σ, we could just as well use P directly. We use tree notation for clarity:

$$\dfrac{\dfrac{\dfrac{\Sigma}{P} \quad [\neg P]^1}{\bot}}{P}\, RAA_1 \qquad \Longrightarrow \qquad \frac{\Sigma}{P}$$

If the last step does not discharge $\neg P$ anywhere in Σ, this is a detour which can be eliminated.

The third rule $(\to_{\mu R})$ is renaming of addresses. Instead of jumping to the address α, and from there jump to the address β, one can directly jump to the address β.

The fourth rule $(\to_{\mu C})$ states that instead of jumping to α, and there apply the result on a term M, we may instead first do the application, and then jump to address α.

Iterated application of the fourth rule can be written as

$$E[\mu\alpha.C] \quad \twoheadrightarrow \quad \mu\alpha.C[\alpha := \alpha E]$$

More generally stated: instead of jumping to address α, and there evaluate the term in a context, we may instead evaluate a term in this context, and then jump to address α. Thus the jump is pushed towards the end of the evaluation. Compare this to natural deduction, where one often tries to push the use of the RAA-rule towards the end, as to have the rest of the proof intuitionistically valid.

We add types to the bound variables to clarify the type:

$$\begin{array}{rll} (\lambda x : A.M)N & \to_\beta & M[x := N] \\ \mu\alpha : \neg A.[\alpha]M & \to_{\mu\eta} & M \qquad \text{if } \alpha \notin \mathrm{FV}_\mu(M) \\ [\beta]\mu\alpha : \neg A.C & \to_{\mu R} & C[\alpha := \beta\Box] \\ (\mu\alpha : \neg(A \to B).C)M & \to_{\mu C} & \mu\alpha : \neg A.C[\alpha := \alpha(\Box M)] \end{array}$$

Note that the rule $(\to_{\mu C})$ changes the type of α. The overall type of the term is unchanged.

4.2.3 Properties

As with the simply typed λ-calculus, we have the following three properties for $\lambda\mu$-calculus, which we again state without proof. We refer to [SU06] for a proof.

Theorem 4.2 (Subject reduction)**.** *If $M : A$ and $M \twoheadrightarrow_\beta M'$, then $M' : A$.*

Proof. Induction on the derivation of the type. □

Theorem 4.3 (Strong normalisation)**.** *Every chain of reductions $M \to_\beta M_1 \to_\beta M_2 \ldots$ must reach a normal form M_n after a finite number of reductions.*

Theorem 4.4 (Confluence)**.** *If $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then there exists a term D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.*

4.3 Interpretation

As in the previous chapter about control operators, we present operators from the `racket/control` module in Racket which correspond to our operators. As mentioned before, control operators can be used to implement `catch` and `throw` operators. We will shortly introduce such operators in $\lambda\mu$-calculus. We will also find the a $\lambda\mu$-term for the operator `call/cc`, and its type.

4.3.1 Implementation in Racket

In Racket, *prompt-tags* can be used as μ-variables. The functions `call-with-continuation-prompt` (abbreviated as `call/prompt`) and `abort-current-continuation` (abbreviated as `abort/cc`) then corresponds to μ-abstraction and μ-variable application, respectively.

The syntax for these control operators looks like

μ-abstraction:	(`call/prompt` *proc* [*prompttag handler*] *args* ...)
μ-application:	(`abort/cc` *proc* [*prompttag*])

To simply the syntax for our purpose, we can redefine

```
(define (mu address proc)
  (call/prompt proc address identity))
```

The argument `proc` must be a procedure that can be called when the corresponding expression should evaluate. We should thus take the corresponding expression `expr` and wrap it in an argument-less lambda expression, `(lambda () expr)`, to prevent it from evaluating before being called. [14]

The programs one can write in Racket using prompt-tags are of course a bit different from what can be achieved in $\lambda\mu$-calculus — for example, Racket uses call-by-value evaluation and has dynamic typing. Nevertheless, the idea of setting labels and jumping to them is the same. However, substitution is *not* capture-avoiding for prompt-tags in Racket.

Example 4.1. Primality checking of n can be done by testing divisibility with all numbers $[2, n)$. This procedure uses a jump to exit the testing once a divisor is found.

```
(define (prime? n)
  (define (divisible-check i)
    (if (= (modulo n i) 0)
        (abort/cc alpha #f)
        #t))
  ;;; define mu-variable
  (define alpha (make-continuation-prompt-tag 'alpha))

  (if (> n 1)
      (mu alpha (lambda ()
                  (andmap divisible-check (range 2 n))))
          #f))
```

△

4.3.2 Operators `throw` and `catch`

The address application and abstraction can be interpreted as `catch` and `throw` operators in functional languages. An expression of the type $\mu\alpha.M$ catches any terms labelled with α in M. The expression $[\alpha]N$ throws (labels) N to the address α. We define two new operators capturing this idea, but for restricted terms, as done in [SU06] and [GKM13].

$$\texttt{catch}_\alpha M := \mu\alpha.[\alpha]M$$
$$\texttt{throw}_\alpha M := \mu\_.[\alpha]M$$

Since substitution of μ-variables is capture-avoiding, we get *statically bound* exceptions: a term will not catch any exceptions thrown by its argument, only those thrown by itself. This differs from most programming languages where exceptions conversely are *dynamically bound*.

[14] Such a wrapping is called a *thunk*. It can be used to simulate call-by-name in a call-by-value language.

We can derive reduction rules for `throw` and `catch` operators.

$$\begin{aligned}
\texttt{catch}_\alpha M &\to_{\mu\eta} M \qquad \text{if } \alpha \notin \mathrm{FV}_\mu(M) \\
\texttt{throw}_\alpha \texttt{catch}_\beta &\to_{\mu R} \texttt{throw}_\alpha M[\beta := \alpha\Box] \\
\texttt{catch}_\alpha \texttt{catch}_\beta M &\to_{\mu R} \texttt{catch}_\alpha M[\beta := \alpha\Box] \\
\texttt{throw}_\alpha \texttt{throw}_\beta &\to_{\mu R} \texttt{throw}_\beta M \\
\texttt{catch}_\alpha \texttt{throw}_\alpha M &\to_{\mu R} \texttt{catch}_\alpha M \\
E[\texttt{catch}_\alpha M] &\twoheadrightarrow_{\mu C} \texttt{catch}_\alpha M[\alpha := \alpha E] \\
E[\texttt{throw}_\alpha M] &\twoheadrightarrow_{\mu C} \texttt{throw}_\alpha M
\end{aligned}$$

Example 4.2. Consider a variant of the $\lambda\mu$-calculus with natural numbers. Here we can formulate the term

$$\texttt{catch}_\alpha\ (K\ 0)\ \texttt{throw}_\alpha\ 2$$

where K is the K-combinator $K \equiv \lambda x.\lambda y.x$.

A programming language with call-by-value evaluation will evaluate the arguments to K before the application. Thus the expression $\texttt{throw}_\alpha\ 2$ will be evaluated and 2 will be thrown, which is the overall result of the expression.

A programming language with call-by-name evaluation only evaluates terms where it is needed. The arguments to K will not be evaluated before application. The result of the application $K\ 0\ \texttt{throw}_\alpha\ 2$ is therefore 0 which is the overall result of the application. △

Which result is correct? We expect two functional (and therefore side-effect-free) programming languages to produce the same result after evaluation of the same expression, given that they halt. This example demonstrates that *operations on the control flow depend on the order of evaluation.* According to the reduction rules we have defined, a correct evaluation of the term would be

$$\begin{aligned}
\texttt{catch}_\alpha\ ((\lambda x.\lambda y.x)\ 0)\ \texttt{throw}_\alpha\ 2 &\to_\beta \\
\texttt{catch}_\alpha\ (\lambda y.0)\ \texttt{throw}_\alpha\ 2 &\to_\beta \\
\texttt{catch}_\alpha\ 0 &\to_{\mu\eta} \\
0 &
\end{aligned}$$

so the "correct" result is 0 which is not surprising, since $\lambda\mu$-calculus is a call-by-name calculus. If we were to extend our contexts with

$$E ::= \cdots \mid NE$$

and add a reduction rule

$$M\mu\alpha.C \to_* \mu\alpha.C[\beta := \beta(M\Box)]$$

then also the call-by-value evaluation of this term would be correct, but then we would lose confluence, as demonstrated by this example.

4.3.3 Type of `call/cc`

The formula $((A \to B) \to A) \to A$, known as Peirce's law, is a classical tautology that is not intuitionistically valid. The type is therefore not inhabited in the

usual lambda calculus, but in the $\lambda\mu$-calculus it is inhabited by the following term.

$$\lambda x : (P \to Q) \to P.\mu\alpha : \neg P.[\alpha]x(\lambda z : P.\mu\beta : \neg Q.[\alpha]z)$$

Using the `catch` and `throw`-operators, as well as omitting the type annotations, we get a more compact term.

$$\lambda x.\texttt{catch}_\alpha x(\lambda z.\texttt{throw}_\alpha z)$$

If we call this term $\mathcal{P}$, we get the following reduction rule for $\mathcal{P}$. It is obtained by first applying $\to_\beta$ and then $\twoheadrightarrow_{\mu C}$.

$$E[\mathcal{P}M] \quad \twoheadrightarrow \quad \texttt{catch}_\alpha E[M(\lambda z.\texttt{throw}_\alpha E[z])]$$

As noted by Parigot in the original paper on $\lambda\mu$-calculus [Par92], the behaviour of this term is the same as that of `call/cc` . Compare this reduction rule with the reduction rule for the term $\mathcal{K}$ in Section 3.4.3. The throwing to address α exits the context just like the operator $\mathcal{A}$.

4.3.4 A Definition of $\mathcal{C}$

Parigot [Par92] also presents a term that acts as operator $\mathcal{C}$.

If we in $\mathcal{P}$ change the application of $\alpha : \neg P$ to an application of the free variable $\gamma : \neg Q$, we instead get the type $((P \to Q) \to Q) \to P$. If we furthermore take $\bot$ as Q, we have an inhabitant of the type $\neg\neg P \to P$. Note that this is not a combinator (!), since γ is free.

$$\lambda x : (P \to \bot) \to \bot.\mu\alpha : \neg P.[\gamma]x(\lambda z : P.\mu\beta : \neg\bot.[\alpha]z)$$

We call this term $\mathcal{N}$. The somewhat different reduction rule is

$$E[\mathcal{N}M] \quad \twoheadrightarrow \quad \mu\alpha.[\gamma]M(\lambda z.\texttt{throw}_\alpha E[z])$$

The difference is that when the continuation $\lambda z.\texttt{throw}_\alpha E[z]$ never is invoked in M, the term will not return to the context E and instead be thrown to the free address γ. Compare with the operator $\mathcal{C}$ in Section 3.4.1.

We will later (see Section 4.6) find a combinator inhabiting $\neg\neg P \to P$.

4.4 Logical Embeddings

A logical embedding is a translation of theorems in classical logic to counterparts in intuitionistic logic.

Definition 4.5. The *Kolmogorov negative translation* translates a formula in classical propositional logic to a formula in intuitionistic propositional logic. It is defined as follows:

$$\begin{aligned} k(\alpha) &:= \neg\neg\alpha \qquad \text{if } \alpha \text{ is an atomic proposition} \\ k(\phi \to \psi) &:= \neg\neg(k(\phi) \to k(\psi)) \end{aligned}$$

Theorem 4.5. *$\vdash \psi$ in classical propositional logic, if and only if $\vdash k(\psi)$ in intuitionistic propositional logic.*

For a proof, see [SU06].

4.4.1 CPS-translation of $\lambda\mu$-calculus

As discussed in Section 3.4, an alternative way to write a program with control operators is to write the program in continuation passing style, CPS.

We will now explore this idea and find an inductive CPS-translation for our terms. The translation is a standard one, found in for example [SU06]. A program in $\lambda\mu$-calculus will be transformed into a CPS-program in λ-calculus. The translation of a term M will be denoted by $\underline{M}$. Since all functions are single-parameter functions, we modify our previous description of CPS (where the continuation is the final parameter): *All terms must take a continuation as parameter, usually denoted by k.*

Below follows an informal motivation on how $\lambda\mu$-terms should be translated to CPS. We initially ignore types.

Functions are values, so they should be "returned" with the continuation. The function body must be transformed though, so the translation of $\lambda x.M$ is $\lambda k.k(\lambda x.\underline{M})$.

Variables are placeholders for other terms, so they should be given the continuation. The terms that will substitute them will take a continuation as first parameter. The translation of x is $\lambda k.xk$.

Applications MN are trickier. The terms M and N must be translated, but then we want to extract M to be able to apply it to N. M should therefore be given a continuation $\lambda m.\cdots m\underline{N}\cdots$ that applies it to the argument $\underline{N}$. The result of the application $m\underline{N}$ should then be passed the continuation k. The complete translation of MN is thus $\lambda k.\underline{M}(\lambda m.m\underline{N}k)$.

How do we deal with addresses? Addresses are similar in behaviour to continuations. Address abstraction means that we ignore the given continuation k and supply our own. The continuation we supply is the identity function $\lambda d.d$ that will return to this point. We also abstract on a continuation α (the address) that will be visible to inner functions, so they can exit or jump to this point. The translation of $\mu\alpha.M$ could thus be $\lambda k.(\lambda\alpha.\underline{M}(\lambda d.d))k$. But we realise the outermost k is unnecessary, since α also is a continuation, so the criterion "all terms take a continuation as parameter" is met by simplifying to $\lambda\alpha.\underline{M}(\lambda d.d)$, which is our translation.

Address application also means supplying another continuation than the given one, but the difference is that no new continuation is created — instead we use one from the scope. The address application $[\alpha]M$ is simply $\lambda k.\underline{M}\alpha$, where k is ignored and α from the scope is provided instead.

Definition 4.6 (CPS-translation of $\lambda\mu$-terms)**.** The CPS-translation of a (raw) $\lambda\mu$-term M, denoted $\underline{M}$ is defined inductively:

$$\begin{aligned}
\underline{x} &= \lambda k.xk \\
\underline{\lambda x.M} &= \lambda k.k(\lambda x.\underline{M}) \\
\underline{MN} &= \lambda k.\underline{M}(\lambda m.m\underline{N}k) \\
\underline{[\alpha]M} &= \lambda k.\underline{M}\alpha \\
\underline{\mu\alpha.M} &= \lambda\alpha.\underline{M}(\lambda d.d)
\end{aligned}$$

where $k \notin \mathrm{FV}(M, N)$ and α is a fresh variable.

Is this translation well-typed, and how? What is the type of a continuation? Particularly, what is the type of the "top-continuation", the continuation that

will be applied to the final result? Assume applying this top-continuation to the final result yields a term of some special result type $\mathcal{R}$. Then a continuation k taking a term x of type A and producing the final result must have term $A \to \mathcal{R}$. That is, the CPS-translation $\lambda k.xk$ of x must have type $(A \to \mathcal{R}) \to \mathcal{R}$.

Since addresses are translated directly to continuations, and they have type $A \to \bot$, we see that $\bot$ is an appropriate choice as $\mathcal{R}$.

The typed version of the terms looks like, assuming that A, A_1, A_2 are base types:

$$\begin{array}{rcl}
\underline{x : A} & = & \lambda k : \neg A.xk \\
 & & \text{where } x : \neg\neg A \\
\underline{\lambda x.M : A_1 \to A_2} & = & \lambda k : \neg(\neg\neg A_1 \to \neg\neg A_2).k(\lambda x : \neg\neg A_1.\underline{M}) \\
 & & \text{where } \underline{M} : \neg\neg A_2 \\
M : A_1 \to A_2 \ \underline{MN : A_2} & = & \lambda k : \neg A_2.\underline{M}(\lambda m : A_1 \to \neg\neg A_2.m\underline{N}k) \\
 & & \text{where } \underline{M} : \neg\neg A_2 \\
M : A,\ \alpha : \neg A \quad \underline{[\alpha]M : \bot} & = & \lambda k : \neg\bot.\underline{M}\alpha \\
 & & \text{where } \underline{M} : \neg\neg A, \quad \alpha : \neg A \\
C : \bot \qquad \underline{\mu\alpha : \neg A.C : A} & = & \lambda\alpha : \neg A.\underline{C}(\lambda d : \neg\bot.d) \\
 & & \text{where } \underline{C} : \neg\neg\bot
\end{array}$$

If A, A_1, A_2 not are base types, we need to translate them recursively as well. What does the general case look like? We get the following result:

Theorem 4.6 (CPS-translation corresponds to Kolmogorov translation)**.** *If $\Gamma \vdash M : A$ in $\lambda\mu$-calculus, and $\underline{\Gamma} = \{\underline{M} : k(A) \mid M : A \in \Gamma\}$, then $\underline{\Gamma} \vdash \underline{M} : k(A)$ in simply typed λ-calculus.*

Proof. Induction on the typing derivation of M. □

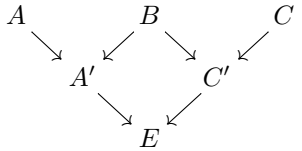
We also want "equality" on terms to be preserved. Terms should not only have the same type, but they should be equal in some sense: up to reduction.

Lemma 4.7 (Term equality)**.** *Let $\to_*$ be a relation with the Church-Rosser property. Define a relation*

$$A \sim B \quad \overset{\text{def}}{\Leftrightarrow} \quad \exists C : A \twoheadrightarrow_* C \text{ and } B \twoheadrightarrow_* C$$

The relation $\sim$ is an equivalence relation.

Proof. The relation is obviously symmetric and reflexive. For transitivity, assume $A \sim B$ and $A \sim C$, that is, for some A' and C' we have $A \twoheadrightarrow_* A'$, $B \twoheadrightarrow_* A'$, $B \twoheadrightarrow_* C'$, $C \twoheadrightarrow_* C'$. Then by the Church-Rosser, considering B, A' and C', there is an E such that $A' \twoheadrightarrow_* E$ and $C' \twoheadrightarrow_* E$.



□

We will denote this equivalence relation by $=_\beta$ and $=_\mu$ for $\to_\beta$ and $\to_\mu$ respectively.

For our preservation of equivalence, we will need restricted terms. If we use restricted terms, we can simplify the translation of address application and abstraction.

$$\begin{array}{rcl}
\underline{x} & = & \lambda k.xk \\
\underline{\lambda x.M} & = & \lambda k.k(\lambda x.\underline{M}) \\
\underline{MN} & = & \lambda k.\underline{M}(\lambda m.m\underline{N}k) \\
\underline{[\alpha]M} & = & \underline{M}\alpha \\
\underline{\mu\alpha.C} & = & \lambda\alpha.\underline{C}
\end{array}$$

When only restricted terms are considered, this CPS-translation is sometimes presented instead, as in [GKM13].

We will now show that the CPS-translation preserves equality. The proof is a detailed variant of the proof in [SU06].

Theorem 4.8 (Equality preservance of CPS-translation)**.** *Let M and N be restricted $\lambda\mu$-terms. Then $\underline{M} =_\beta \underline{N}$ if and only if $M =_\mu N$.*

Proof. It is sufficient to show that $A \to_\mu B$ implies $\underline{A} =_\beta \underline{B}$. We have to study our reduction rules and see what happens with substitution under CPS-translations.

$$\begin{array}{lrcl}
(\to_\beta) & \underline{M[x := N]} & =_\beta & \underline{M}[x := \underline{N}] \\
(\to_{\mu R}) & \underline{C[\alpha := \beta\Box]} & =_\beta & \underline{C}[\alpha := \beta] \\
(\to_{\mu C}) & \underline{C[\alpha := \alpha(\Box M)]} & =_\beta & \underline{C}[\alpha := \lambda m.m\underline{N}\alpha]
\end{array}$$

Then also note that, since all CPS-translations of terms are abstractions, with a continuation as parameter, we have $\lambda k.\underline{P}k =_\beta \underline{P}$, where $k \notin \mathrm{FV}(P)$.

Now we need to show that

$$\begin{array}{lrcl}
(\to_\beta) & \underline{(\lambda x.M)N} & =_\beta & \underline{M[x := N]} \\
(\to_{\mu\eta}) & \underline{\mu\alpha.[\alpha]M} & =_\beta & \underline{M} \qquad \text{if } \alpha \notin \mathrm{FV}_\mu(M) \\
(\to_{\mu R}) & \underline{[\beta]\mu\alpha.C} & =_\beta & \underline{C[\alpha := \beta\Box]} \\
(\to_{\mu C}) & \underline{(\mu\alpha.C)M} & =_\beta & \underline{\mu\alpha.C[\alpha := \alpha(\Box M)]}
\end{array}$$

We take each of the rules in order.

$(\to_\beta)$

$$\begin{array}{rcl}
\underline{(\lambda x.M)N} & = & (\lambda k.(\lambda k.k(\lambda x.\underline{M}))(\lambda f.f\underline{N}k)) \\
& \twoheadrightarrow_\beta & (\lambda k.(\lambda x.\underline{M})\underline{N}k) \\
& \to_\beta & \lambda k.\underline{M}[x := \underline{N}]k \\
& =_\beta & \underline{M}[x := \underline{N}] \\
& =_\beta & \underline{M[x := \underline{N}]}
\end{array}$$

$(\to_{\mu\eta})$ We assume $\alpha \notin \mathrm{FV}_\mu(M)$.

$$\begin{array}{rcl}
\underline{\mu\alpha.[\alpha]M} & = & \lambda\alpha.(\lambda k.\underline{M}\alpha)(\lambda d.d) \\
& \twoheadrightarrow_\beta & \lambda\alpha.\underline{M}\alpha \\
& =_\beta & \underline{M}
\end{array}$$

($\to_{\mu R}$) Assuming raw terms, we show a weaker statement, which implies the stronger statement if we restrict ourselves to raw terms.

$$\begin{array}{rcl} \underline{[\beta]\mu\alpha.[\gamma]M} & = & \lambda k.(\lambda\alpha.(\lambda k.\underline{M}\gamma)(\lambda d.d))\beta \\ & \twoheadrightarrow_\beta & \lambda k.\underline{M}\gamma \\ & = & \underline{[\gamma]M[\alpha := \beta\Box]} \end{array}$$

The restriction we have made is that C must be on the form $[\gamma]M$.

($\to_{\mu C}$)

$$\begin{array}{rcl} \underline{(\mu\alpha.C)M} & = & \lambda k.(\lambda\alpha.\underline{C}(\lambda d.d))(\lambda c.c\underline{M}k) \\ & \to_\beta & \lambda k.\underline{C}[\alpha := (\lambda c.c\underline{M}k)](\lambda d.d) \\ & \equiv_\alpha & \lambda\alpha.\underline{C}[\alpha := (\lambda c.c\underline{M}\alpha)](\lambda d.d) \\ & = & \underline{\mu\alpha.C[\alpha := \alpha(\Box M)]} \end{array}$$

□

Example 4.3. As pointed out in the proof, the rule ($\to_{\mu R}$) is exactly where equality fails for raw terms. For example

$$\begin{array}{lrcl} & [\beta]\mu\alpha.x & \to_{\mu R} & x \\ \text{but} & \underline{[\beta]\mu\alpha.x} & = & \lambda k.\lambda\alpha.(\lambda k.xk)(\lambda d.d)\beta \\ \text{and} & \underline{x} & = & \lambda k.xk \\ \text{when} & \lambda k.xk & \neq & \lambda k.\lambda\alpha.(\lambda k.xk)(\lambda d.d)\beta \end{array}$$

The restriction of the rule ($\to_{\mu R}$) to a new rule ($\to_{\mu R*}$) is sufficient to get equality also for raw terms.

$$[\beta]\mu\alpha.[\gamma]M \to_{\mu R*} [\gamma]M[\alpha := \beta\Box]$$

△

Are the raw terms too unrestricted? Will restricted terms be sufficient? No, many of our classical tautologies require raw terms to be inhabited in $\lambda\mu$-calculus (see Section 4.6). But as a system for computation, the restricted terms are sufficient, because all terms can be rewritten to restricted terms if we allow the type of the term to change.

4.5 Extension with Conjunction and Disjunction

So far we have only considered classical logic with the connective $\to$, corresponding to $\lambda\mu$-calculus with the type constructor $\to$. The $\lambda\mu$-calculus can be extended with type constructors for $\wedge$ and $\vee$, as done with the λ-calculus in Section 2.2.1. This gives us a complete Curry-Howard-correspondence.

A more interesting extension is to *define* the connectives in $\lambda\mu$-calculus, as done in [SU06]. A similar construction, though not in $\lambda\mu$-calculus, was done by Griffin [Gri90]. It is essentially the same construction, but using λ-calculus with the operators $\mathcal{A}$ and $\mathcal{C}$ instead.

4.5.1 Definitions in $\lambda\mu$-calculus

The connectives $\wedge$ and $\vee$ cannot be defined through $\to$ (and $\bot$) in intuitionistic logic, but definitions can easily be found in classical logic using truth tables. We will use these definitions, which are symmetrical in A and B:

$$\begin{array}{rrcl} \text{OR:} & A \vee B & := & (A \to \bot) \to (B \to \bot) \to \bot \\ \text{AND:} & A \wedge B & := & (A \to B \to \bot) \to \bot \end{array}$$

In $\lambda\mu$-calculus we can find corresponding terms and find a computational meaning of these definitions from classical logic. That is, we can find definitions of $\texttt{inl}(), \texttt{inr}(), \texttt{case}(,,), \texttt{pair}(,), \texttt{fst}(), \texttt{snd}()$ instead of adding them as axioms.

We start with the definition of $\vee$:

$$\begin{array}{rcl} A \vee B & := & (A \to \bot) \to (B \to \bot) \to \bot \\ \texttt{inl}() : x \mapsto \texttt{inl}(x) & := & \lambda x.\lambda f_1.\lambda f_2.f_1 x \\ \texttt{inr}() : x \mapsto \texttt{inr}(x) & := & \lambda x.\lambda f_1.\lambda f_2.f_2 x \\ \texttt{case}(,,) : (e, g_1, g_2) \mapsto \texttt{case}(e, g_1, g_2) & := & \lambda e.\lambda g_1.\lambda g_2.\mu\alpha.e(\lambda x.[\alpha]g_1 x)(\lambda x.[\alpha]g_2 x) \end{array}$$

We add type annotations to bound variables, to clarify the type:

$$\begin{array}{rcl} \texttt{inl}() & := & \lambda x : A.\lambda f_1 : \neg A.\lambda f_2 : \neg B.f_1 x \\ \texttt{inr}() & := & \lambda x : B.\lambda f_1 : \neg A.\lambda f_2 : \neg B.f_2 x \\ \texttt{case}(,,) & := & \lambda e : A \vee B.\lambda g_1 : A \to C.\lambda g_2 : B \to C.\mu\alpha : \neg C.e(\lambda x : A.[\alpha]g_1 x)(\lambda x : B.[\alpha]g_2 x) \end{array}$$

Note that if we ignore typing, the terms for $\texttt{inl}()$ and $\texttt{inr}()$ already solves the task for untyped λ-calculus. They are injection terms that only need to be applied to two selection functions f_1, f_2 and use the correct one, depending on if it was a left or right injection.

The problem with the typing is then: Something of type $\bot$ must be returned, so it is impossible to return the $f_i x$ which is of type C, or in any other way extract it. The only solution is to throw the result to a level above and simulate a return of type $\bot$.

The definition of $\wedge$ follows:

$$\begin{array}{rcl} A \wedge B & := & (A \to B \to \bot) \to \bot \\ \texttt{pair}(,) : (a, b) \mapsto \texttt{pair}(a, b) & := & \lambda a.\lambda b.\lambda s.sab \\ \texttt{fst}() : p \mapsto \texttt{fst}(p) & := & \lambda p.\mu\alpha.p(\lambda a.\lambda b.[\alpha]a) \\ \texttt{snd}() : p \mapsto \texttt{snd}(p) & := & \lambda p.\mu\alpha.p(\lambda a.\lambda b.[\alpha]b) \end{array}$$

We add types to bound variables:

$$\begin{array}{rcl} \texttt{pair}(,) : (a, b) \mapsto \texttt{pair}(a, b) & := & \lambda a : A.\lambda b : B.\lambda s : A \to B \to \bot.sab \\ \texttt{fst}() : p \mapsto \texttt{fst}(p) & := & \lambda p : A \wedge B.\mu\alpha : \neg A.p(\lambda a : A.\lambda b : B.[\alpha]a) \\ \texttt{snd}() : p \mapsto \texttt{snd}(p) & := & \lambda p : A \wedge B.\mu\alpha : \neg B.p(\lambda a : A.\lambda b : B.[\alpha]b) \end{array}$$

As with injection, if types are disregarded, the pair constructor already solves the task for untyped λ-calculus. Applied to a selector, the correct component will be extracted. This is insufficient in a typed calculus, because the return type should be $\bot$, not A or B. Therefore an address abstraction and application

must be used, to simulate a return of type $\bot$, when we in fact return one of the components of the pair.

See Appendix A.3 for an implementation in Racket of the pair and injection constructors and destructors using $\lambda\mu$-calculus. This implementation uses continuation prompts, the Racket equivalence of addresses, described in Section 4.3.1. Some work has been put into simulating a typed language, since Racket is untyped. The implementation does type checks in constructors and destructors as well as in address application and abstraction.

4.5.2 A Complete Correspondence

With our extension with pairs and injections, we get a complete Curry-Howard correspondence, for classical logic with all logical connectives. The theorem statement and proof are as before.

Since we have not added any rules to our calculus, our results from previously hold. The $\lambda\mu$-calculus with pairs and injections is confluent and strongly normalising.

We have another result, quite unexpected. If we can ignore the use of addresses, except in the definition of pairs and injections, we have exactly the λ-calculus with type constructors $\to$, $\wedge$, $\vee$, corresponding to intuitionistic logic. This means that the proofs of strong normalisation and confluence for $\lambda\mu$-calculus also prove these properties for λ-calculus extended with pairs and injections.

4.6 Terms for Classical Tautologies

When we introduced the concept of a proof being a construction, we listed several tautologies (in Boolean algebras) which had no constructions (see Example 2.3). On our search for constructions for them, we introduced continuations, control operators and finally the $\lambda\mu$-calculus. This now allows us to find terms for these propositions, since all classical tautologies, when interpreted as types, are inhabited in the $\lambda\mu$-calculus, by the Curry-Howard correspondence. The reader familiar with the programming language Coq can compare the terms below with the Coq-terms presented in Appendix B.

We have already seen the term for Peirce's law, as it corresponds to `call/cc` by the Curry Howard correspondence (see Section 4.3.3). We include it here again in our list.

Example 4.4. Peirce's law, $((P \to Q) \to P) \to P$, is inhabited by the term,

$$\lambda x : (P \to Q) \to P.\mu\alpha : \neg P.[\alpha]x(\lambda p : P.\mu\beta : \neg Q.[\alpha]p)$$

We have also written it more compactly as

$$\lambda x.\mathtt{catch}_\alpha x(\lambda z.\mathtt{throw}_\alpha z)$$

$\triangle$

We now present terms of other classical tautologies. If we change Peirce's law to $((P \to Q) \to Q) \to P$, we get another classical tautology, best known when we take Q as $\bot$ and get $(\neg\neg P \to P)$. Despite the similarities, this term cannot be written as a restricted term, only as a raw term.

Example 4.5. Reduction ad absurdum, $(\neg\neg P \to P)$, is inhabited by the term

$$\lambda x.\mu\pi.x(\lambda p.[\pi]p)$$

and type annotated

$$\lambda x : \neg\neg P.\mu\pi : \neg P.x(\lambda p : P.[\pi]p)$$

△

The commonly used proof technique *proof by contradiction* is, not surprisingly, also not valid intuitionistically. Here is a construction in $\lambda\mu$-calculus.

Example 4.6. Proof by contradiction, $(\neg Q \to \neg P) \to (P \to Q)$, is inhabited by the term

$$\lambda x.\lambda p.\mu\alpha.x(\lambda q.[\alpha]q)p$$

$$\lambda x : (\neg Q \to \neg P).\lambda p : P.\mu\alpha.x(\lambda q : Q.[\alpha]q)p$$

△

Another interesting combinator is one of de Morgan's laws. Three of de Morgan's laws are inhabited in λ-calculus; the fourth is not. Here are the three first.

$\neg(P \vee Q) \to (\neg P \wedge \neg Q)$	$\lambda c.\mathtt{pair}(\lambda p.c\ \mathtt{inl}(p), \lambda q.c\ \mathtt{inr}(q))$
$(\neg P \wedge \neg Q) \to \neg(P \vee Q)$	$\lambda c.\lambda d.\mathtt{case}(d, \mathtt{fst}(c), \mathtt{snd}(c))$
$(\neg P \vee \neg Q) \to \neg(P \wedge Q)$	$\lambda d.\lambda c.\mathtt{case}(d, \lambda p.p\ \mathtt{fst}(c), \lambda q.q\ \mathtt{snd}(c))d$

The fourth follows here in $\lambda\mu$-calculus.

Example 4.7. Negation of conjunction (de Morgan's laws), $\neg(P \wedge Q) \to (\neg P \vee \neg Q)$, is inhabited by the term

$$\lambda x.\mu\gamma.x(\mathtt{pair}(\mu\alpha.[\gamma]\mathtt{inl}(\lambda p.[\alpha]p), \mu\beta.[\gamma]\mathtt{inr}(\lambda q.[\beta]q)))$$

Annotated with types, the term looks like

$$\lambda x : \neg(P \wedge Q).\mu\gamma : \neg\neg(P \wedge Q).x(\mathtt{pair}(\mu\alpha : \neg P.[\gamma]\mathtt{inl}(\lambda p : P.[\alpha]p), \mu\beta : \neg Q.[\gamma]\mathtt{inr}(\lambda q : Q.[\beta]q)))$$

△

Another interesting proposition is that RAA implies the Principle of Excluded Middle (PEM). Neither RAA nor PEM are derivable in intuitionistic logic, but when we add the derivation rule double negation elimination (RAA) to natural deduction and get classical logic, both are derivable. Nonetheless, "RAA $\to$ PEM" is *not* derivable in intuitionistic logic. This is initially quite confusing, which is what makes this proposition interesting.

The reason that there is a difference between *adding a derivation rule* RAA and *showing an implication* from RAA, is that the derivation rule really looks like $\forall P.(\neg\neg P \to P)$. That is, we may eliminate double negations in any formula, while the implication from RAA only allows us to eliminate double negations in front of P. As we see in the inhabitant below, we eliminate double negation not in front of P, but in front of $P \vee \neg P$.

Example 4.8. RAA implies PEM, $(\neg\neg P \to P) \to (P \vee \neg P)$, is inhabited by the term

$$\lambda r.\mu\alpha.[\alpha]\mathtt{inl}(r(\lambda n.[\alpha]\mathtt{inr}(n)))$$

or type annotated

$$\lambda r : \neg\neg P \to P.\mu\alpha : \neg(P \vee \neg P).[\alpha]\mathtt{inl}(r(\lambda n : \neg P.[\alpha]\mathtt{inr}(n)))$$

△

Understanding the term for the principle of the excluded middle itself, is possibly the hardest, since it is not a function.

Example 4.9 (Principle of Excluded Middle)**.** The following term inhabits the principle of excluded middle, $P \vee \neg P$:

$$\mu\alpha : \neg(A \vee \neg A).[\alpha]\mathtt{inr}(\lambda p : P.\mu\beta : \neg\bot.[\alpha]\mathtt{inl}(p))$$

Skipping the μ-abstraction of the address β is also an inhabitant, but this way we get a restricted term. We can write it with `catch` and `throw`-operators:

$$\mathtt{catch}_\alpha\mathtt{inr}(\lambda p : P.\mathtt{throw}_\alpha\mathtt{inl}(p))$$

If we denote this term by $\mathcal{M}$, we have the following reduction rule

$$E[\mathcal{M}] \twoheadrightarrow_\mu \mathtt{catch}_\alpha E[\mathtt{inr}(\lambda p : P.\mathtt{throw}_\alpha E[\mathtt{inl}(p)])]$$

The "result" of this computation are that both computations, for terms of P and $\neg P$ respectively, are done in parallel. △

Like with the combinator $\mathcal{P}$, which acts like `call/cc`, two computations are done in parallel. But $\mathcal{P}$ differs in that that one of the these computations will eventually become the final result: either the continuation is ignored, or it is used. If the continuation is used, that computation becomes the final result. Else if it is ignored, the outer computation becomes the final result. However, with the combinator $\mathcal{M}$, none of the two computations can be eliminated (else we would know if we have P or not $\neg P$).

Example 4.10 (Eliminating the Principle of Excluding Middle)**.** If we try to eliminate $\mathcal{M}$, with the `case`$(,,)$-destructor, maybe we can simplify it further?

We first observe what happens if we rewrite $\mathcal{M}$ using the classical definitions of disjunction [15]

$$\mathtt{catch}_\alpha\lambda f_1.\lambda f_2.f_2(\lambda p.\mathtt{throw}_\alpha(\lambda f_1.\lambda f_2.f_1 p))$$

Now if this term is applied to two selector functions $F_1 : \neg P$ and $F_2 : \neg\neg P$, we get

$$\begin{aligned}(\mathtt{catch}_\alpha\lambda f_1.\lambda f_2.f_2(\lambda p.\mathtt{throw}_\alpha(\lambda f_1.\lambda f_2.f_1 p)))F_1F_2 &\quad \to_{\mu C}\\ (\mathtt{catch}_\alpha\lambda f_1.\lambda f_2.f_2(\lambda p.\mathtt{throw}_\alpha((\lambda f_1.\lambda f_2.f_1 p)F_1F_2))F_1F_2) &\quad \to_\beta\\ (\mathtt{catch}_\alpha F_2(\lambda p.\mathtt{throw}_\alpha F_1 p))\end{aligned}$$

[15] Type annotated, the term looks like

$$\mu\alpha : \neg(P \vee \neg P).[\alpha]\lambda f_1 : \neg P.\lambda f_2 : \neg\neg P.f_2(\lambda p : P.\mu\beta : \neg\bot.[\alpha](\lambda f_1 : \neg P.\lambda f_2 : \neg\neg P.f_1 p))$$

Remember that the type of α usually changes when we apply the rule $(\to_{\mu C})$. Here the new type becomes $\neg\bot$.

We now try to use the $\texttt{case}(,,)$-destructor, with two functions $G_1 : P \to Q$ and $G_2 : \neg P \to Q$. We expect to get a term of type Q.

$$\begin{aligned} \texttt{case}(\mathcal{M}, G_1, G_2) &\to_\beta \\ \mu\beta.(\texttt{catch}_\alpha \texttt{inr}(\lambda p : P.\texttt{throw}_\alpha \texttt{inl}(p)))(\lambda x.[\beta]G_1 x)(\lambda x.[\beta]G_2 x) &\twoheadrightarrow_\mu \\ \mu\beta.(\texttt{catch}_\alpha[\beta]G_2(\lambda p.\texttt{throw}_\alpha[\beta]G_1 p)) \end{aligned}$$

If we call this term $\hat{\mathcal{M}}$, we again get a similar result. (Note that since $\beta : \neg Q$, we have $\hat{\mathcal{M}} : Q$ as expected.)

$$E[\hat{\mathcal{M}}] \twoheadrightarrow_\mu \mu\beta.(\texttt{catch}_\alpha[\beta]E[G_2(\lambda p.\texttt{throw}_\alpha[\beta]E[G_1 p])])$$

So we did not advance much further — we skipped the $\texttt{inr}()$ and $\texttt{inl}()$ constructors, but introduced another address β. △

The next classical tautology is *proof by cases*. We do not know whether we have P or $\neg P$, but if we can show that either of them implies Q, then we must have Q. This is written $(P \to Q) \wedge (\neg P \to Q) \to Q$, or using currying, for a less complex term, we have $(P \to Q) \to (\neg P \to Q) \to Q$. The reasoning behind this term builds on the principle of excluded middle, why it cannot be constructive.

Example 4.11. Proof by cases, $(P \to Q) \to (\neg P \to Q) \to Q$, is inhabited by the term

$$\lambda p : P \to Q.\lambda n : \neg P \to Q.\mu\alpha : \neg Q.[\alpha]n(\lambda x : P.[\alpha]px)$$

or omitting type annotations

$$\lambda p.\lambda n.\mu\alpha.[\alpha]n(\lambda x.[\alpha]px)$$

Note that we can also write it as

$$\lambda p : P \to Q.\lambda n : \neg P \to Q.\texttt{case}(\mathcal{M}, p, n)$$

using the combinator $\mathcal{M}$ for the principle of excluded middle. △

5 References

References

[Car13] Jesper Carlström. *Logic.* Matematiska institutionen, Stockholms Universitet, 2013.

[FWFD88] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 52–62, New York, NY, USA, 1988. ACM.

[GKM13] Herman Geuvers, Robbert Krebbers, and James McKinna. The $\lambda\mu$T-calculus. volume 164 of *Annals of Pure and Applied Logic*, pages 676–701. Elsevier B.V., 2013.

[Gri90] Timothy G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, New York, NY, USA, 2 edition, 2008.

[Par92] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 190–201, London, UK, UK, 1992. Springer-Verlag.

[R7R13] Revised^7 Report on the algorithmic language Scheme (small language). `http://trac.sacrideo.us/wg/wiki/R7RSHomePage`, 2013.

[Roj97] Raúl Rojas. A tutorial introduction to the lambda calculus. `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`, 1997.

[SU06] Morten Heine Sørensen and Paweł Urzyczyin. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, 2006.

A Code in Scheme/Racket

Code examples in Scheme and Racket.

A.1 Continuations with CPS

This program uses the procedure `safe-div` mentioned in Example 3.4. The main program uses `elemwise-divide` to divide two lists element-wise. The entire program is written in CPS. If the division fails, `safe-div` aborts, by invoking the `top-continuation` with the message "Division by zero!". Then control never returns to `elemwise-divide`.

Programmers might notice that `elemwise-divide` uses tail recursion. This is essential for the construction to work, ensuring that the recursive call only runs if `continue-here` is invoked. It is also much more memory effective when working with continuations, since using tail recursion, the parent continuation is always reused in the recursion. CPS-programs might consume a lot of memory.

```
#lang racket

(define safe-divide
  (lambda (a b fail success)
    (if (= b 0)
        (fail "Division by zero!")
        (success (/ a b)))))

(define elemwise-divide
  (lambda (list1 list2 k)

    ;; define tail recursive helper function
    (define iter
      (lambda (list1 list2 result k)
        ;; base case -- empty list
        (if (or (empty? list1) (empty? list2))
            (k result)

            ;; else -- construct continuation at this point
            (let [(continue-here
                   (lambda (the-quotient)
                     ;; make recursive call
                     (iter (rest list1) (rest list2)
                           (append result (list the-quotient)) k)))]

              ;; and call safe-divide with that continuation
              (safe-divide (first list1) (first list2) k continue-here)))))

    (iter list1 list2 empty k)))

(define top-continuation
  (lambda (result)
    (display result)
```

```
    (newline)))

;; run program
(elemwise-divide (list 1 2 3 6) (list 5 0 3 4) top-continuation)
```

A.2 Continuations with `call/cc`

A variant of the previous program (`safe-div` example), using `call/cc` instead of CPS, for comparison. Tail recursion is no longer required, but it has been kept, so that this program will be similar to the previous one.

```
#lang racket

(define safe-divide
  (lambda (a b fail success)
    (if (= b 0)
        (fail "Division by zero!")
        (success (/ a b)))))

(define elemwise-divide
  (lambda (list1 list2 k)

    ;; define tail recursive helper function
    (define iter
      (lambda (list1 list2 result k)
        ;; base case -- empty list
        (if (or (empty? list1) (empty? list2))
            (k result)

            ;; else -- construct the current continuation
            (let [(the-quotient
                   (call/cc
                    (lambda (continue-here)
                      ;; call safe-divide with that continuation
                      (safe-divide (first list1) (first list2) k continue-here))))]
              ;; and make recursive-call
              (iter (rest list1) (rest list2)
                    (append result (list the-quotient)) k)))))

    (iter list1 list2 empty k)))

;; run program
(display
 (call/cc
  (lambda (top-continuation)
    (elemwise-divide (list 1 2 3 6) (list 5 0 3 4) top-continuation))))
(newline)
```

A.3 Classical Definitions of Pairing and Injection

The classical definitions of pairing and projection. Scheme is an untyped language and does few type checks. The tricky part of defining pairing and projection is to get the type correct, so we need to do type checks at every application. Therefore, Racket *contracts* have been used to check the typing. Since contracts cannot take parameters, we need to first create the constructors and destructors for the specific combination of types that we wants to use. I.e. to use a pairing constructor for types A and B, we need to call `((pair A B) arg1 arg2)`.

There is a custom type for $\bot$, `bot`, which consists of the value `'bot`.

A.3.1 Base Definitions

```
#lang racket
;; mu.rkt -- address abstraction & application operators
(require racket/control)
(provide (all-defined-out))

;;;;; Some type aliases
(define bot 'bot)
(define bot/c bot)
(define type/c (-> any/c boolean?))
(define address/c pair?)

;;;;;; Control operators

(define/contract (make-address type name)
  (-> type/c symbol? address/c)
  (cons type (make-continuation-prompt-tag name)))

(define/contract (mu address expr-thunk)
  (-> address/c (-> any/c) any/c)
  (define/contract (func addr expr-th)
    (-> continuation-prompt-tag? (-> bot/c) (car address))
    (call/prompt expr-th addr identity))
  (func (cdr address) expr-thunk))

(define/contract (ab address expr-value)
  (-> address/c any/c bot/c)
  (define/contract (func addr expr-val)
    (-> continuation-prompt-tag? (car address) bot/c)
    (abort/cc addr expr-val)
    bot)
  (func (cdr address) expr-value))

(define address-abstract mu)
(define address-apply ab)
```

A.3.2 Disjunction

```
#lang racket
;; disjunction.rkt -- disjunction operators
(require "mu.rkt")
(provide inl inr or-case)

;; Some type aliases
(define (left/c t)
  (-> t bot/c))
(define (right/c t)
  (-> t bot/c))
(define (injection/c t0 t1)
  (-> (left/c t0) (right/c t1) bot/c))

;;;; Left constructor
(define/contract (inl t0 t1)
  (-> type/c type/c any/c)
  (define/contract (constructor-left x)
    (-> t0 (injection/c t0 t1))
    (lambda (left right)
      (left x)))
  constructor-left)

;;;; Right constructor
(define/contract (inr t0 t1)
  (-> type/c type/c any/c)
  (define/contract (constructor-right x)
    (-> t1 (injection/c t0 t1))
    (lambda (left right)
      (right x)))
  constructor-right)

;;;; Destructor
(define/contract (or-case t0 t1 t2)
  (-> type/c type/c type/c any/c)
  (define/contract (case-apply inj left right)
    (-> (injection/c t0 t1) (-> t0 t2) (-> t1 t2) t2)
    (define alpha (make-address t2 'alpha))
    (define/contract (retrieve-left x)
      (left/c t0)
      (ab alpha (left x)))
    (define/contract (retrieve-right x)
      (right/c t1)
      (ab alpha (right x)))
    (mu alpha (lambda ()
                (inj retrieve-left retrieve-right))))
  case-apply)
```

A.3.3 Conjunction

```
#lang racket
;; conjunction.rkt -- conjunction operators
(provide pair fst snd)
(require "mu.rkt")

;; Some type aliases
(define (select/c t0 t1)
  (-> t0 t1 bot/c))

(define (pair/c t0 t1)
  (-> (select/c t0 t1) bot/c))

;;;;;; Constructor
(define/contract (pair t0 t1)
  (-> type/c type/c any/c)
  (define/contract (pair-constructor a b)
    (-> t0 t1 (pair/c t0 t1))
        (lambda (selector)
          (selector a b)))
  pair-constructor)

;;;;; Left destructor
(define/contract (fst t0 t1)
  (-> type/c type/c any/c)
  (define/contract (fst-selector pair)
    (-> (pair/c t0 t1) t0)
    (define alpha (make-address t0 'alpha))
    (define/contract (fst-retrieve a b)
      (select/c t0 t1)
      (ab alpha a))
    (mu alpha (lambda ()
                (pair fst-retrieve))))
  fst-selector)

;;;;; Right destructor
(define/contract (snd t0 t1)
  (-> type/c type/c any/c)
  (define/contract (snd-selector pair)
    (-> (pair/c t0 t1) t1)
    (define alpha (make-address t1 'alpha))
    (define/contract (snd-retrieve a b)
      (select/c t0 t1)
      (ab alpha b))
    (mu alpha (lambda ()
                (pair snd-retrieve))))
  snd-selector)
```

A.3.4 Examples

Some examples of the constructions in use.

```
#lang racket

;; examples.rkt -- Examples of conjunction and disjunction

(require "conjunction.rkt")
(require "disjunction.rkt")

;;;;;;; Conjunction examples

> (define my-pair ((pair number? symbol?) 10 'hi))
> ((fst number? symbol?) my-pair)
10
> ((snd number? symbol?) my-pair)
'hi

;;;;;;; Disjunction examples

> (define foo ((inl number? symbol?) 20))
> (define bar ((inr number? symbol?) 'hello))
>
> ((or-case number? symbol? string?) foo number->string symbol->string)
"20"
> ((or-case number? symbol? string?) bar number->string symbol->string)
"hello"
```

B Code in Coq

Below are the derivations of the classical tautologies, which were given terms in Section 4.6, as well as the first three of de Morgan's laws, also presented there. RAA has been added as an axiom, in order to prove these theorems. Care has been taken to name the variables the same as in the terms. The reader can run the Coq-code and compare the terms — they are indeed very similar. The only difference is how RAA is handled. In Coq, abstracting on the address α will be done in two steps, `apply RAA.` and `intro alpha.`.

Another difference, although not visible here, is that the addresses are first class objects here, so we may write simply `alpha` instead of `(fun p: P => (alpha p))`. I.e. we need not write $\lambda x.[\alpha]x$ instead of α. Nevertheless, we have used the more verbose syntax $\lambda x.[\alpha]x$ below for sake of similarity.

```
Theorem RAA (P : Prop):
  ~(~P) -> P.
Proof.
  admit.
Qed.

Definition peirce (P Q:Prop) :
  ((P->Q)->P)->P.
Proof.
intro x.
apply RAA.
intro alpha.
apply alpha.
apply x.
intro p.
apply RAA.
intro _beta.
exact (alpha p).
Qed.

Definition raa (P : Prop) :
  ~(~P) -> P.
Proof.
intro x.
apply RAA.
intro pi.
apply x.
exact (fun p: P => (pi p)).
Qed.

Definition proof_by_contradiction (P Q:Prop):
  (~Q->~P) -> (P->Q).
Proof.
intro x.
intro p.
apply RAA.
```

```
intro alpha.
apply x.
exact (fun q:Q => (alpha q)).
exact p.
Qed.

Definition morgan1 (P Q: Prop):
  ~(P \/ Q) -> (~P /\ ~Q).
Proof.
intro c.
split.
exact (fun p: P => (c (or_introl p))).
exact (fun q: Q => (c (or_intror q))).
Qed.

Definition morgan2 (P Q: Prop) :
  (~P /\ ~Q) -> ~(P \/ Q).
Proof.
intros c d.
case d.
exact (proj1 c).
exact (proj2 c).
Qed.

Definition morgan3 (P Q: Prop):
  (~P \/ ~Q) -> ~(P /\ Q).
Proof.
intros d c.
case d.
exact (fun p: ~P => (p (proj1 c))).
exact (fun q: ~Q => (q (proj2 c))).
Qed.

Definition morgan4 (P Q: Prop):
  ~(P /\ Q) -> (~P \/ ~Q).
Proof.
intro x.
apply RAA.
intro gamma.
apply x.
split.
apply RAA.
intro alpha.
apply gamma.
exact (or_introl (fun p: P => alpha p)).
apply RAA.
intro _beta.
apply gamma.
exact (or_intror (fun q: Q => _beta q)).
Qed.
```

```
Definition RAA_implies_PEM (P: Prop) :
  (~~P -> P) -> (P \/ ~P).
Proof.
intro r.
apply RAA.
intro alpha.
apply alpha.
apply or_introl.
apply r.
exact (fun n:~P => (alpha (or_intror n))).
Qed.

Definition PEM (P: Prop):
  P \/ ~P.
Proof.
apply RAA.
intro alpha.
apply alpha.
apply or_intror.
intro p.
apply RAA.
intro _beta.
apply (alpha (or_introl p)).
Qed.

Definition proof_by_cases (P Q: Prop) :
  (P -> Q) -> (~P -> Q) -> Q.
Proof.
intros p n.
apply RAA.
intro alpha.
apply alpha.
apply n.
exact (fun x:P => (alpha (p x))).
Qed.

Definition proof_by_cases_using_pem (P Q: Prop) :
  (P -> Q) -> (~P -> Q) -> Q.
Proof.
intros p n.
assert (pem : P \/ ~P).
apply PEM.
case pem.
exact p.
exact n.
Qed.
```