The Curry-Howard isomorphism

A study of the computational content of intuitionistic and classical logic

Jasmijn van Harskamp
6067956

Supervisor:
Dr. Matthijs Vákár

Second reader:
Prof. dr. Rosalie Iemhoff

Utrecht University
Faculty of Humanities

Bachelor Thesis
Bachelor Artificial Intelligence
15 ECTS

26 June 2020

Abstract

The Curry-Howard isomorphism relates systems of formal logic to models of computation. It broadly states that proofs correspond to programs and formulae to types. For a long time it was believed the correspondence merely applied to intuitionistic logic. This changed when the axioms of classical logic were found to correspond to control operators. The correspondence is an important theoretical result connecting logic and computer science, but also has practical implications in e.g. program verification.

This thesis will discuss the isomorphism with respect to both intuitionistic and classical logic. First, a language for the intuitionistic part of the correspondence will be constructed and assigned meaning by an operational semantics. These constructs will be used for exploring various aspects of the isomorphism, such as continuations and the relation between normalization and reduction. Subsequently, the language will be extended for a correspondence with classical logic by adding a control operator based on Peirce's law. The computational effects of this operator will be analysed and compared with other possible control operators, which arise from different classical axioms —i.e. a generalised version of Peirce's law, double negation elimination and the law of excluded middle. The correspondence will be shown to relate double negation embeddings of classical logic into intuitionistic logic to CPS translations. Finally, the complete language and semantics will be proven to have the properties of determinism, progress, preservation and termination.

Contents

Chapter 1

Introduction

The *Curry-Howard isomorphism* states the correspondence between proof systems of formal logic and models of computation. In general, it states that proofs correspond to programs and that the proven formulae correspond to the types of the obtained programs [13]. Additionally, the isomorphism relates certain problems from both formalisms, such as provability to type inhabitation and proof normalization to program evaluation.

For many years, it was believed the correspondence only applied to intuitionistic logic. This changed, however, in 1990, when Griffin related classical proofs to typed programs by using control operators [14]. His paper initiated research into many aspects of classical logic and the corresponding programming with control operators. One notable result is the correspondence between double negation translation and continuation-passing-style transformation [14, 23].

The results from the Curry-Howard isomorphism are, first of all, of great theoretical interest. The conclusion that the systems of formal logic and computation are in some sense equivalent, is fundamentally remarkable. Also for artificial intelligence, this is an important theoretical insight, since the scientific areas of logic and computer science have been intertwined with its development from the start. For example, the most influential programming language in the early days of artificial intelligence, Lisp [26], is based on Church's λ-calculus.

More practically, the Curry-Howard isomorphism has contributed to the development of proof assistants, such as Coq and Agda. Proof assistants have many useful applications, one of which is the verification of system designs and proving the correctness of programs [12].

The extension of the correspondence into classical logic adds control operators to the system of computation, which allows for programming with continuations. While these should be used with caution, they can be very useful. One recent example is the use of —a more general class of— continuations for implementing deep learning [28].

The objective of this thesis is to present a clear overview of the Curry-Howard isomorphism, for both intuitionistic and classical logic. Rather than presenting new results, we will study the isomorphism in a coherent way, by constructing a simple language and using that language as a base for analysing various principles. Throughout this thesis, we will discuss questions such as 'What is the dynamic aspect of logical proofs?', 'What is a computationally interesting definition of classical logic?' and 'What is the computational content of classical logic?'.

This thesis will be structured as follows.

Chapter 2 will start with introducing intuitionistic propositional logic, the simply-typed λ-calculus and the correspondence between both systems. Subsequently, the language which we will have constructed in this study, will be implemented and assigned meaning by an operational semantics. We will conclude the chapter by examining the correspondence between proof and type checking, formula and type inference, provability and type inhabitation, and normalization and reduction.

Chapter 3 starts by discussing classical logic, the $\lambda_{\mathcal{C}}$-calculus and the extension of the Curry-Howard isomorphism into classical logic. Then, we will extend our language with a notion of control. Again, we will discuss the implementation and assign meaning in terms of an operational semantics. Furthermore, we will present the correspondence between double negation translation and continuation-passing style transformation.

Chapter 4 will deal with the meta-theoretic properties of the language and operational semantics discussed in chapters 1 and 2. We will prove the properties of determinism, progress, preservation and termination.

Chapter 5 will conclude our study. We will provide an overview of the main observations and give suggestions for further research.

An implementation of the presented language and operational semantics can be found at `https://github.com/JasmijnvH/BSc-Thesis.git`.

Chapter 2

Intuitionistic Logic

When we think about logic, we often think in terms of truth. We reason about statements as if every proposition is either true or false. This principle, known as the *law of excluded middle*, is what is rejected in intuitionistic logic. Instead of reasoning about a statement based on its truth value, we are now interested in the proof or *construction* of that statement.

At the time the *Curry-Howard isomorphism* was first introduced, it only considered proofs in intuitionistic logic. Curry established a correspondence between combinators of combinatory logic and axioms of intuitionistic logic [7], while Howard elaborated on the correspondence between natural deduction and λ-calculus [16]. In this chapter we will discuss the correspondence by introducing the principles of intuitionistic propositional logic, the simply typed λ-calculus and their correspondence. We will continue by providing a computational interpretation and elaborating on the operational semantics.

2.1 The Curry-Howard isomorphism

In this section, we will shortly review the language of intuitionistic propositional logic and the simply typed λ-calculus. Subsequently, we will specify the correspondence between those systems. Most of the following notation is taken from Chapter 2 from Sørensen and Urzyczyn [27].

2.1.1 Intuitionistic propositional logic

To define the language of intuitionistic propositional logic, we assume an infinite set PV of propositional variables and inductively define the set of formulas Φ as

$$\Phi ::= \top \mid \bot \mid PV \mid (\Phi \to \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi)$$

Here, $\top$ and $\bot$ denote two constants interpreted as 'true' and 'false', respectively. Or rather, since in intuitionistic logic we reason in terms of proofs, $\top$ is the proposition that has exactly one canonical proof, whereas $\bot$ is the proposition

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ Ax$$

$$\frac{}{\Gamma \vdash \top}\ \top I \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot E$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \to I \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \to E$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge I \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge E \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \wedge E$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \vee I \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \vee I$$

$$\frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \rho \quad \Gamma, \psi \vdash \rho}{\Gamma \vdash \rho} \vee E$$

Figure 2.1: Intuitionistic propositional logic

that can never be proved. We have three binary operators: implication '$\to$', where we read $\varphi \to \psi$ as φ implies ψ; conjunction '$\wedge$', where $\varphi \wedge \psi$ denotes φ and ψ; and disjunction '$\vee$', where $\varphi \vee \psi$ means φ or ψ. Following convention, we often use the abbreviation $\neg\phi$ for the negation $\phi \to \bot$ and omit the outermost parentheses.

We now define the proof system, by specifying the axiom schemes and rules for each connective. We let uppercase Greek letters Γ, Δ, etc. stand for a finite subset of Φ, which is called a *context*. The notation Γ, Δ is short for $\Gamma \cup \Delta$ and we write Γ, φ instead of $\Gamma, \{\varphi\}$. The rules that define the relation $\Gamma \vdash \varphi$ are presented in Figure 2.1. Given this system, a formal *proof* of $\Gamma \vdash \varphi$ is a finite tree in which the root is labelled $\Gamma \vdash \varphi$; each internal node has label $\Gamma' \vdash \varphi'$; all leaves are labelled by axioms; and the label of each parent node is derived by applying one of the rules on the labels of the children. If $\vdash \varphi$, i.e. $\{\} \vdash \varphi$, then φ is called a *theorem*. Intuitively, $\Gamma \vdash \varphi$ means that from the set of assumptions Γ we can conclude φ.

2.1.2 Simply typed λ-calculus

In the 1930s, Alonzo Church introduced a formal model of computation, called the λ-calculus [5]. His work was based on the development of combinatory logic by Schönfinkel and Curry [4]. A few years later, the λ-calculus was revised and *types* were added, thus introducing the simply typed λ-calculus. There are two

different approaches to the typing, one introduced by Church [6], and the other by Curry. In Curry-style systems, abstractions have no domain. They are not annotated with types and can therefore be assigned meaning regardless of the typing. In Church-style systems, on the contrary, abstractions do have domains. Consequently, each term in a system à la Church has a unique type. We will study the system as introduced by Church.

We assume an infinite set U of type variables and define the set Π of simple types by the following grammar.

$$\Pi ::= U \mid (\Pi \to \Pi)$$

Note that there is only one type constructor, $\to$, which creates function types. In order to give a full correspondence with intuitionistic propositional logic we will need additional constructors. These will be introduced in the next subsection; for now we will continue with the definitions as proposed by Church.

Let V be an infinite set of variables. We inductively define the set Λ_Π of simply typed terms.

$$\Lambda_\Pi ::= V \mid (\lambda x{:}\Pi\ \Lambda_\Pi) \mid (\Lambda_\Pi\ \Lambda_\Pi)$$

The typing of these terms happens in accordance with the following rules. We read $\Gamma \vdash M : \sigma$ as *'M has type σ in Γ'*.

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

A term $M \in \Lambda_\Pi$ is called *typable* if there exist Γ and σ such that $\Gamma \vdash M : \sigma$.

The simply typed λ-calculus is a model of computation. It can be seen as a simple programming language in which the terms are programs. The typing of the programs expresses certain properties and helps avoid programming errors. In the coming sections we will extend this programming language and discuss its logical interpretation.

2.1.3 The Curry-Howard isomorphism

The Curry-Howard isomorphism states that for any derivation in intuitionistic logic there exists a corresponding typable λ-term à la Church, and the other way around. However, since the simply typed λ-calculus as defined above only contains the function type constructor, the correspondence would so far merely apply to the implicational fragment of intuitionistic logic, i.e. the subsystem in which we solely define the connective $\to$ and its rules ($\to I$ and $\to E$). To extend the correspondence to the whole system of intuitionistic logic, we need to extend the simply typed λ-calculus. First, we add type constructors for the constants $\top$ and $\bot$ and for product and sum types.

$$\Pi ::= \top \mid \bot \mid U \mid (\Pi \to \Pi) \mid (\Pi \wedge \Pi) \mid (\Pi \vee \Pi)$$

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi}\ Ax$$

$$\frac{}{\Gamma \vdash \mathbf{taut} : \top}\ \top I \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathbf{abort}^{\varphi}\ M : \varphi}\ \bot E$$

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x{:}\varphi\ .\ M : \varphi \to \psi}\ \to I \qquad \frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash M\ N : \psi}\ \to E$$

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N\rangle : \varphi \wedge \psi}\ \wedge I \qquad \frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathbf{fst}\ M : \varphi}\ \wedge E \qquad \frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathbf{snd}\ M : \psi}\ \wedge E$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \mathbf{inl}^{\varphi\vee\psi}\ M : \varphi \vee \psi}\ \vee I \qquad \frac{\Gamma \vdash M : \psi}{\Gamma \vdash \mathbf{inr}^{\varphi\vee\psi}\ M : \varphi \vee \psi}\ \vee I$$

$$\frac{\Gamma \vdash L : \varphi \vee \psi \quad \Gamma, x : \varphi \vdash M : \rho \quad \Gamma, y : \psi \vdash N : \rho}{\Gamma \vdash \mathbf{vcase}(L; x.M; y.N) : \rho}\ \vee E$$

Figure 2.2: Typed terms

Secondly, we expand the language Λ_Π.

$$\begin{aligned}\Lambda_\Pi ::= {} & V \mid \mathbf{taut} \mid \mathbf{abort}^{\varphi}\Lambda_\Pi \mid (\lambda x{:}\Pi\ \Lambda_\Pi) \mid (\Lambda_\Pi\ \Lambda_\Pi) \\ & \mid \langle \Lambda_\Pi, \Lambda_\Pi\rangle \mid \mathbf{fst}\ \Lambda_\Pi \mid \mathbf{snd}\ \Lambda_\Pi \\ & \mid \mathbf{inl}^{\varphi\vee\psi}\ \Lambda_\Pi \mid \mathbf{inr}^{\varphi\vee\psi}\ \Lambda_\Pi \mid \mathbf{vcase}(\Lambda_\Pi; V.\Lambda_\Pi; V.\Lambda_\Pi)\end{aligned}$$

This expansion allows us to define the Curry-Howard isomorphism. If we let PV equal U, then Φ equals Π. We specify the correspondence as in [27].

Theorem 2.1. ***(Curry-Howard isomorphism)***

i. If $\Gamma \vdash M : \varphi$ *then* $|\Gamma| \vdash \varphi$*, where* $|\Gamma| = \{\sigma \in \Pi \mid (x : \sigma) \in \Gamma$*, for some* $x\}$*.*

ii. If $\Gamma \vdash \varphi$ *then there exists* $M \in \Lambda_\Pi$ *such that* $\Delta \vdash M : \varphi$*, where* $\Delta = \{(x_\varphi : \varphi) \mid \varphi \in \Gamma\}$*.*

Thus, we can combine the systems of intuitionistic logic and simply typed λ-calculus, yielding the rules presented in Figure 2.2. We will discuss some intuitive interpretations of this correspondence.

An **abort**-expression is the term that corresponds to the $\bot E$-rule, also *ex falso quodlibet*.

The $\to$-introduction rule corresponds to a logical abstraction. The $\to$-elimination rule is the application of two constructions.

A pair $\langle M, N\rangle$ corresponds to a product type $\varphi \wedge \psi$. The **fst** M and **snd** M are the first and second projection of a product type, respectively. In logical terms, we can say the $\wedge$-introduction rule transforms two constructions M and N of formulae φ and ψ into a construction $\langle M, N\rangle$ of a formula $\varphi \wedge \psi$. The two $\wedge$-elimination rules use a construction M of a formula $\varphi \wedge \psi$ to build constructions for the formulae φ and ψ.

The $\mathbf{inl}^{\varphi\vee\psi}\ M$ and $\mathbf{inr}^{\varphi\vee\psi}\ M$ are the left and right injections into a sum type. Thus, taking a construction M and transforming it into a construction of a disjunction. The $\vee$-elimination rule is a case-expression; it evaluates whether L is of the form $\mathbf{inl}^{\varphi\vee\psi}\ L'$, in which case it returns M with the substitution of L' for x; or if L is of the form $\mathbf{inr}^{\varphi\vee\psi}\ L'$, in which case it returns N with the substitution of L' for y. Computationally, this is interesting, since it resembles pattern matching, which is often used in programming. In logic, the rule corresponds to having a construction L of a formula $\varphi \vee \psi$ and two constructions M and N of the same formula ρ, one of which relies on the assumption φ and the other on ψ. We then create a construction of the formula ρ.

In general, we can conclude that if we regard the type as a proposition, we can interpret the term as a representation of its proof tree. This is the core of the Curry-Howard isomorphism.

2.1.4 β-reduction

The simply typed λ-calculus comes with a natural notion of term —or program— evaluation: *β-reduction*. This presents the following reduction-rules.

$$
\begin{aligned}
(\lambda x{:}\varphi\ .\ M)\ N &\to_\beta M[N/x] \\
\mathbf{fst}\ \langle M, N\rangle &\to_\beta M \\
\mathbf{snd}\ \langle M, N\rangle &\to_\beta N \\
\mathbf{vcase}(\mathbf{inl}^{\varphi\vee\psi}\ L; x.M; y.N) &\to_\beta M[L/x] \\
\mathbf{vcase}(\mathbf{inr}^{\varphi\vee\psi}\ L; x.M; y.N) &\to_\beta N[L/y]
\end{aligned}
$$

Here $M[N/x]$ is the *capture-avoiding substitution* of N for x in M. To explain its effect, we need the notions of free-variables and α-equivalence.

Free-variables are variables that are not bound by an operator. Any variable that is not free is called a *bound-variable*. In our expressions we encounter the binding of variables in λ- and **vcase**-terms. An expression with no free-variables is called *closed*. We define the free-variables FV recursively.

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\mathbf{taut}) &= \emptyset \\
FV(\mathbf{abort}^{\varphi}\ M) &= FV(M) \\
FV(\lambda x{:}\varphi\ .\ M) &= FV(M) \setminus \{x\} \\
FV(M\ N) &= FV(M) \cup FV(N) \\
FV(\langle M, N\rangle) &= FV(M) \cup FV(N)
\end{aligned}
$$

$$
\begin{aligned}
FV(\mathbf{fst}\ M) &= FV(M) \\
FV(\mathbf{snd}\ M) &= FV(M) \\
FV(\mathbf{inl}^{\varphi\vee\psi}\ M) &= FV(M) \\
FV(\mathbf{inr}^{\varphi\vee\psi}\ M) &= FV(M) \\
FV(\mathbf{vcase}(L; x.M; y.N)) &= FV(L) \cup (FV(M) \setminus \{x\}) \cup (FV(N) \setminus \{y\})
\end{aligned}
$$

α-equivalence defines a form of equivalence between two expressions. It states that the naming of the bound variables does not influence the evaluation, i.e. two terms are α-equivalent if the only difference is the names of the bound variables. For example, $\lambda x{:}\varphi\ .\ x$ and $\lambda y{:}\varphi\ .\ y$ are α-equivalent, whereas $\lambda x{:}\varphi\ .\ y$ and $\lambda x{:}\varphi\ .\ z$ are not. The principle of α-equivalence can be used in α-renaming. This is the process in which the bound variable of an expression is renamed, in order to create an α-equivalent expression. It is desirable to use a *fresh* variable for each new λ-term.

Now capture-avoiding substitution ensures that no previously free variable gets bound —captured— as a result of the substitution. If, however, such a capturing impends, α-renaming is used and the substitution is performed on the α-equivalent expression.

The process of β-reducing a term —and hence evaluating a program— corresponds to normalizing a proof. In subsection 2.5.4 we will elaborate on this correspondence.

2.2 Type checking and inference

The previous section established the correspondence between proofs and programs. This result enables us to implement a simply typed λ-calculus and use the programs it produces to reason about intuitionistic proofs. We will implement our simply typed λ-calculus in an existing programming language, a meta- or hostlanguage. We chose to use Haskell. In this section we will describe the syntax and type checking and inference. The next section will concern the operational semantics.

2.2.1 Data types

To implement the simply typed λ-calculus we need two kinds of data types; one to represent the simple types (Π), and one to construct simply typed terms, or expressions (Λ_Π).

We store the simple types in a data type `Type`.

```
data Type = TVar    String
          | Unit
          | Error
          | Func    Type Type
          | Product Type Type
          | Sum     Type Type
```

```
            deriving (Eq)
```

The constructor `TVar` allows us to define any type variable, mirroring our infinte set of variables U. The parameterless constructors `Unit` and `Error` denote our basic types $\top$ and $\bot$, respectively. The final three constructors implement the binary operations for building function, product and sum types.

Next, we define the data type `Expr` for representing —possibly ill-typed— terms.

```
data Expr = Var   String
          | Taut
          | Abort Expr Type
          | Lam   String Type Expr
          | App   Expr Expr
          | Pair  Expr Expr
          | Fst   Expr
          | Snd   Expr
          | Inl   Expr Type
          | Inr   Expr Type
          | VCase Expr String Expr String Expr
          deriving (Eq)
```

An expression could be any variable from the infinite set V, which is represented by the constructor `Var`. The constant $\top$ only has an introduction rule, implemented by the parameterless constructor `Taut`. The constructor `Abort` simulates the result of the $\bot E$-rule. Since the '*ex falso sequitur quodlibet*'-rule allows this expression to be of any type, we need to explicitly provide the type we want to infer. This is done by the parameter of data type `Type`.

`Lam` and `App` are the constructors for the $\to I$- and $\to E$-rule, respectively. Following Church's convention for λ-terms, we need to annotate our variable in `Lam` (contained in the `String`-parameter) with a type (provided in the `Type`-parameter).

The constructor `Pair` simulates the result of the $\wedge I$-rule; `Fst` and `Snd` provide the first and second projection, obeying the $\wedge E$-rules.

`Inl` and `Inr` construct the injections, following the $\vee I$-rules. Note again that we need to provide the desired result type. Finally, `VCase` implements term obtained by the $\vee E$-rule.

2.2.2 Type checking and type inference

The defined data structures allow us to build terms. However, since no restrictions on typing have been provided, we are still able to build poorly typed terms. For example, we should allow the projection constructors `Fst` and `Snd` to merely take expressions of a product type as an argument. To guarantee only well-typed expressions are used for computation, we need to distinguish the correctly typed terms from the incorrect ones. This is where we will use type checking and type inference.

Assuming we have a function `inferType` which given an expression infers its type, type checking is quite simple. We implement the function `checkType` by providing it an expression and the expected type, and testing whether the inferred type is equal to the provided type.

However, the implementation of this assumed function `inferType` is less obvious. A complicating factor are the variables and their bindings in `Lam` and `VCase`. We need to manage the correct typing within their scope. So, while recursively traversing our expressions, we have to maintain a context of variable typings. This will be achieved by introducing an additional function `inferTypeS` which handles this context by using the State-monad. As the result type, we want `Type`; the type of our state will be a `MultiMap String Type`, which will store the type(s) of our variables. Now, our function `inferType` will initiate the recursive inference in `inferTypeS` with the empty context and subsequently extract the result.

The function `inferTypeS` will provide a case-by-case type inference, resulting in a state containing the current result and context. It is defined by pattern matching on all possible expressions.

In the first case, the pattern match is on `Var x`. To infer the type of a variable `x`, we need to retrieve the assigned type from the current context. If the variable is not contained in this context, it means it has not been declared —that is, at least not in the current scope. We should not be able to assign a type to such a variable, so the function will throw an error.

If the expression is a `Taut`, we should simply return its type `Unit`.

An expression of the form `Abort t a` requires us to check whether its given expression `t` is of type `Error`. Otherwise, it would not be permitted to apply the *'ex falso sequitur quodlibet'*-rule and derive any type. If it is, we infer the provided type `a`; if it is not, we generate an error. Note that we indeed need the decoration of `Abort` with the type information `a` to be able to perform type inference, as mentioned before.

If a `Lam x a t` expression is encountered, the variable `x` is added to the context with its type `a`. Subsequently, the enclosed expression `t` is evaluated. When this inference is done, we have evaluated everything within the scope of the current λ-term. The variable, therefore, should be removed from the context. The type we return in the end is the function type consisting of type `a` and the inferred expression type.

For `App t1 t2`, we first infer the type of both expressions `t1` and `t2`. The type of `t1` is supposed to be a function type $\varphi \to \psi$; the type of `t2` then has to be φ. This will result in the inference of the type ψ.

To infer the type of `Pair t1 t2`, we simply infer the types of `t1` and `t2` and combine those in a product type.

Inferring the type of `Fst t` and `Snd t`, requires the type inference of `t`. This is supposed to result in a product type; otherwise an error will be thrown. The result will either be the first or second type of the product, respectively.

In the case of the injections, `Inl t a` and `Inr t a`, we first have to ensure the intended result type `a` is a sum type. If it is, we infer the type of the expression `t`. We proceed by comparing this to the right or left disjunct of our

type `a` and returning `a` if it is a match. Note again that the type annotations are necessary to perform the type inference of the entire expression.

The last pattern match is for `VCase t x1 s1 x2 s2`. First, we will have to infer the type of `t`, which should be a sum type. The variable `x1` and the left disjunct of this type are added to the context and the type of `s1` is inferred. When this is done, `x1` is deleted from the context. We now insert `x2` with the right disjunct and infer the type of `s2`. Lastly, we compare both results and if they are equal, we return this type.

This concludes the definition of `inferTypeS` and thus of `inferType`. Both type checking and type inference can now be used to confirm that an expression is well-typed before proceeding with the computation.

2.3 Operational Semantics

With the syntax of our implementation discussed, we now arrive at its semantics. We define the *operational semantics*, in which programs are assigned meaning based on the computational steps of their execution. These steps will culminate in returning the value of the program.

The two main approaches to operational semantics are *big-step semantics* and *small-step semantics*. Big-step semantics associate a term with its final value. Therefore, it is often simpler to define, but it fails to describe the process of evaluation. Small-step semantics, on the other hand, provide a more dynamic interpretation by giving an inductive definition. Both approaches will be implemented. However, our main focus will be on the small-step semantics.

Because we will be considering an impure language when studying classical logic in the next chapter, we cannot work with full $\beta\eta$- or β-reduction. Namely, these strategies are non-deterministic, which would, undesirably, lead to non-deterministic programs. We therefore have to decide upon a different, deterministic evaluation strategy. Mainly, this comes down to choosing between *call-by-value* or *call-by-name* evaluation. In the case of call-by-value evaluation, we only replace identifiers by values; in call-by-name evaluation, we replace identifiers only by unevaluated terms [19]. We will concern ourselves with call-by-value evaluation. Furthermore, if there are multiple expressions to evaluate, we always start by evaluating the leftmost expression.

2.3.1 Values

In order to derive the *value* of a program, we first need to specify this notion. Values are terms which require no further evaluation. They are formally defined by the grammar

$$V ::= \mathbf{taut} \mid x \mid \lambda x{:}\varphi \,.\, M \mid \langle V, V \rangle \mid \mathbf{inl}^{\varphi \vee \psi}\ V \mid \mathbf{inr}^{\varphi \vee \psi}\ V$$

2.3.2 Stacks

The small-step semantics should, at any point in the computation, indicate the next step. Hence, we somehow need to keep track of the current point of evaluation. For this purpose, we use a *stack*. While evaluating a specific part of an expression, we store the remaining part, which is yet to be evaluated, on the stack. Thus, the stack will contain expressions with one missing part —the part currently being evaluated. This missing piece of the expression will be denoted by a special variable $\_$, called a *hole*. To explicitly distinguish expressions from the 'incomplete' expressions on the stack, we will introduce the set F of frames. A frame simply is an expression with exactly one occurrence of a hole. A stack thus is a list of frames.

$$\begin{aligned} F := \mathbf{abort}^{\varphi}\ \_ \mid \_\ N \mid M\ \_ \\ \mid \langle \_, N \rangle \mid \langle M, \_ \rangle \mid \mathbf{fst}\ \_ \mid \mathbf{snd}\ \_ \\ \mid \mathbf{inl}^{\varphi \vee \psi}\ \_ \mid \mathbf{inr}^{\varphi \vee \psi}\ \_ \mid \mathbf{vcase}(\_; x.M; y.N) \end{aligned}$$

A stack at a certain point in evaluation is also called a *continuation*, for it marks the position to continue the evaluation of the expression. We will use the words 'stack' and 'continuation' interchangeably. We denote continuations by the variables k, l, k', etc.

For the considered expressions, a computation would be completed once the stack is empty. However, to ease the correspondence with logic, we work with 'named' empty stacks. That is, we represent empty stacks by continuation variables such as **halt**. A computation is done once this variable is on top of the stack. The initial stack contains only **halt**.

To reason about well-typed computations later on, we first need to introduce a typing system for stacks. This requires a specification of the relation $\vdash$. The relation $\vdash$ as encountered before could be interpreted as follows: a statement of the form $\Gamma \vdash M : \varphi$ means that from a context Γ we can derive term M of type φ. We could denote the same relation by $\vdash^{\text{true}}$, i.e. $\Gamma \vdash^{\text{true}} M : \varphi$ means that from context Γ we can derive a program M as a proof establishing the truth of φ.

We have seen that continuations hold the computational steps which are yet to be performed. Therefore, a continuation always expects a value of a certain type φ to fill the hole. However, it will not return anything. This implies that we can regard a stack which takes a value of type φ, as a proof of the falsehood of φ. To axiomatize this notion, we introduce the relation $\vdash^{\text{false}}$. A statement $\Gamma \vdash^{\text{false}} k : \varphi$ now means that from context Γ we can derive a continuation k which takes a value of type φ and thus is a proof establishing the falsehood of φ.

Moreover, before presenting the axioms and rules that define these relations, we need to specify the definition of a context. We introduced a relation that could prove that φ is false, so we need a way to propagate this in our proofs. This is done by splitting a context into two parts: $\Gamma_{\text{true}}; \Gamma_{\text{false}}$. The former set contains assumptions for which we assume to have a witness of their truth, while

$$\frac{}{\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \varphi \vdash^{\text{false}} \mathbf{halt} : \varphi}\ Ax^1$$

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \mathbf{abort}^{\varphi}\ \_ :: k : \bot}\ \bot I$$

$$\frac{\Gamma \vdash^{\text{true}} N : \varphi \quad \Gamma \vdash^{\text{false}} k : \psi}{\Gamma \vdash^{\text{false}} \_\ N :: k : \varphi \to \psi}\ \to E \qquad \frac{\Gamma \vdash^{\text{true}} M : \varphi \to \psi \quad \Gamma \vdash^{\text{false}} k : \psi}{\Gamma \vdash^{\text{false}} M\ \_ :: k : \varphi}\ \to E$$

$$\frac{\Gamma \vdash^{\text{true}} N : \psi \quad \Gamma \vdash^{\text{false}} k : \varphi \wedge \psi}{\Gamma \vdash^{\text{false}} \langle \_, N \rangle :: k : \varphi}\ \wedge E \qquad \frac{\Gamma \vdash^{\text{true}} M : \varphi \quad \Gamma \vdash^{\text{false}} k : \varphi \wedge \psi}{\Gamma \vdash^{\text{false}} \langle M, \_ \rangle :: k : \psi}\ \wedge E$$

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \mathbf{fst}\ \_ :: k : \varphi \wedge \psi}\ \wedge I \qquad \frac{\Gamma \vdash^{\text{false}} k : \psi}{\Gamma \vdash^{\text{false}} \mathbf{snd}\ \_ :: k : \varphi \wedge \psi}\ \wedge I$$

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi \vee \psi}{\Gamma \vdash^{\text{false}} \mathbf{inl}^{\varphi \vee \psi}\ \_ :: k : \varphi}\ \vee E \qquad \frac{\Gamma \vdash^{\text{false}} k : \varphi \vee \psi}{\Gamma \vdash^{\text{false}} \mathbf{inr}^{\varphi \vee \psi}\ \_ :: k : \psi}\ \vee E$$

$$\frac{\Gamma, x : \varphi \vdash^{\text{true}} M : \rho \quad \Gamma, y : \psi \vdash^{\text{true}} N : \rho \quad \Gamma \vdash^{\text{false}} k : \rho}{\Gamma \vdash^{\text{false}} \mathbf{vcase}(\_; x.M; y.N) :: k : \varphi \vee \psi}\ \vee I$$

Figure 2.3: Well-typed stacks

the latter set contains assumptions for which we assume to have a witness of their falsehood. In particular, we assume $\mathbf{halt} : \varphi \in \Gamma_{\text{false}}$. We now write Γ to denote $\Gamma_{\text{true}}; \Gamma_{\text{false}}$.

The axioms and rules of $\vdash^{\text{true}}$ are easily obtained by assuming the new definition of Γ and replacing all occurrences of $\vdash$ in Figure 2.2 with $\vdash^{\text{true}}$. The rules for $\vdash^{\text{false}}$ are presented in Figure 2.3. The notation $f :: k$ means we place frame f on top of our existing stack k.

For example in the left $\to E$-rule, if we have a proven expression N of type φ and a stack k expecting an expression of type ψ, we can build a new stack $\_\ N :: k$. This expects the first argument of an application to fill the hole, which has to be of function type $\varphi \to \psi$.

We could also approach these rules logically. Take, for example, the leftmost $\wedge E$-rule. Our assumptions are that ψ and the negation of $\varphi \wedge \psi$ are provable. This implies that the negation of φ has to provable as well.

[1] Here we use the name **halt** for a continuation variable to be suggestive. We emphasise it is merely a name however.

Note that while this definition of continuations allows type checking on stacks, it does not allow type inference. For instance, the type ψ in the left $\wedge I$-rule is not derivable without additional information. Similar problems arise in the other $\wedge I$- and $\vee I$-rules. However, stacks are tools which get constructed in the operational semantics and which the programmer does not typically write. All continuations we will encounter, are well-typed. In section 4.4 we prove that we are indeed justified in assuming this.

2.3.3 Throw

Before continuing with the operational semantics, we introduce a new constructor which we add to Λ_Π. The previous subsection provided a way to prove the negation of formulae. This implies that we could now encounter a situation in which $\Gamma \vdash^{\text{true}} M : \varphi$ and $\Gamma \vdash^{\text{false}} k : \varphi$. This clearly is contradictory. We introduce a constructor **throw** to deal with these contradictions. Let S be a stack.

$$\Lambda_\Pi := ... \mid \textbf{throw}\ S\ \Lambda_\Pi$$

We add the following rule to the proof system.[2]

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi \quad \Gamma \vdash^{\text{true}} M : \varphi}{\Gamma \vdash^{\text{true}} \textbf{throw}\ k\ M : \bot}$$

In the next chapter we will see that **throw** has an interesting computational interpretation in classical logic.[3] Note, however, that adding **throw** to intuitionistic logic does not yield classical logic. We can construct a translation $(\ )^t$ from $\Gamma \vdash^{\text{false}} k : \varphi$ to $\Gamma \vdash^{\text{true}} \lambda\_{:}\varphi\ .\ k^t : \neg\varphi$. This translation is defined by $\textbf{halt}^t = \textbf{halt}\ \_$, for all stack variables, and $(f :: k)^t = k^t[f/\_]$. This means that whenever $\Gamma \vdash^{\text{false}} k : \varphi$ is provable, $\Gamma \vdash^{\text{true}} \lambda\_{:}\varphi\ .\ k^t : \neg\varphi$ is also provable. Therefore, this rule does not change the notion of truth in intuitionistic logic.

2.3.4 Small-step semantics

We now have the tools needed to describe the small-step semantics. A *configuration* is a pair (M, k) consisting of an expression M and a stack k. Given a configuration, the small-step semantics define which computational step should be performed. The evaluation of an expression M starts with the configuration $(M, \textbf{halt})$, where **halt** is a stack variable. Subsequently, we keep applying the rules, until no further rule applies. If we then have obtained a configuration

[2] Note that we could also add

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi \quad \Gamma \vdash^{\text{true}} M : \varphi}{\Gamma \vdash^{\text{false}} \textbf{contradiction}\ M :: k : \bot}$$

which would prove the falsehood of $\bot$. This however does not contribute anything, since we already know by definition that $\bot$ is false.

[3] Because of this interpretation some authors opt to handle **throw** not as an expression, but as a new object: a jump [2, 20]. To keep our presentation as simple as possible, we have chosen to consider **throw** as an expression.

(N, k'), the expression N is the result of the computation. In subsection 4.3 we will prove that in that case N will always be a value and $k' = \mathbf{halt}$.

We discuss the various rules of the small-step semantics. We use the symbol $\leadsto$ to indicate one computational step; the capture-avoiding substitution of v for x in M is denoted as $M[v/x]$.

We start with **abort**. If we encounter an expression **abort** M, we want to evaluate M.

$$\mathbf{abort}^{\varphi}\ M,\ k \quad \leadsto \quad M,\ \mathbf{abort}^{\varphi}\ \_ :: k$$

M is embedded in the **abort**-expression and should therefore be of type $\bot$. So, if the context is empty, M will contain an expression **throw** $k'\ M'$. Such an expression installs k' as the new stack and proceeds by evaluating M'. Note that the **throw**-expression does not use the expression on top of the stack. The stack could therefore be any continuation.

$$\mathbf{throw}\ k'\ M',\ k \quad \leadsto \quad M',\ k'$$

Secondly, we could start with an application. If we have an expression $M\ N$, we first want to evaluate M.

$$M\ N,\ k \quad \leadsto \quad M,\ \_\ N :: k$$

Since M is used in an application, we should be able to reduce it to a lambda-term. This is a value, so we are done evaluating M. We now place this value in the hole on the stack and continue by evaluating N.

$$\lambda x.M',\ \_\ N :: k \quad \leadsto \quad N,\ \lambda x.M'\ \_ :: k$$

We reduce N to a value v and reinsert it in the frame. In the same step we perform the application, which finally results in the substitution.

$$v,\ \lambda x.M'\ \_ :: k \quad \leadsto \quad M'[v/x],\ k$$

The third set of rules starts with a pair $\langle M, N\rangle$. Again, we first want to evaluate M, so we have the rule

$$\langle M, N\rangle,\ k \quad \leadsto \quad M,\ \langle \_, N\rangle :: k$$

M is reduced to some value v, after which we should reduce N.

$$v,\ \langle \_, N\rangle :: k \quad \leadsto \quad N,\ \langle v, \_\rangle :: k$$

This results in a value w, which we insert in the pair. This constructs the final value $\langle v, w\rangle$ which is directly returned.

$$w,\ \langle v, \_\rangle :: k \quad \leadsto \quad \langle v, w\rangle,\ k$$

If we start with a **fst** or **snd** term, we only have one expression.

$$\mathbf{fst}\ M,\ k \quad \leadsto \quad M,\ \mathbf{fst}\ \_ :: k$$
$$\mathbf{snd}\ M,\ k \quad \leadsto \quad M,\ \mathbf{snd}\ \_ :: k$$

This expression is reduced to a value, which is supposed to be a pair. Subsequently, the intended projection is returned.

$$\begin{aligned} \langle v, w\rangle,\ \mathbf{fst}\ \_ :: k &\quad\leadsto\quad v,\ k \\ \langle v, w\rangle,\ \mathbf{snd}\ \_ :: k &\quad\leadsto\quad w,\ k \end{aligned}$$

Starting with an **inl** or **inr** results in the evaluation of the accompanying expression.

$$\begin{aligned} \mathbf{inl}^{\varphi\vee\psi}\ M,\ k &\quad\leadsto\quad M,\ \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k \\ \mathbf{inr}^{\varphi\vee\psi}\ M,\ k &\quad\leadsto\quad M,\ \mathbf{inr}^{\varphi\vee\psi}\ \_ :: k \end{aligned}$$

The reduced value is inserted in the hole. This will result in the final value.

$$\begin{aligned} v,\ \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k &\quad\leadsto\quad \mathbf{inl}^{\varphi\vee\psi}\ v,\ k \\ v,\ \mathbf{inr}^{\varphi\vee\psi}\ \_ :: k &\quad\leadsto\quad \mathbf{inr}^{\varphi\vee\psi}\ v,\ k \end{aligned}$$

The final set of rules is obtained by starting with a **vcase**. We begin by evaluating the expression L.

$$\mathbf{vcase}(L; x.M; y.N),\ k \quad\leadsto\quad L,\ \mathbf{vcase}(\_; x.M; y.N) :: k$$

Given the typing restrictions, L should either evaluate to an **inl**- or an **inr**-expression. Depending on this outcome, the correct term M or N should be invoked, by substituting the variable accordingly.

$$\begin{aligned} \mathbf{inl}^{\varphi\vee\psi}\ v,\ \mathbf{vcase}(\_; x.M; y.N) :: k &\quad\leadsto\quad M[v/x],\ k \\ \mathbf{inr}^{\varphi\vee\psi}\ v,\ \mathbf{vcase}(\_; x.M; y.N) :: k &\quad\leadsto\quad N[v/y],\ k \end{aligned}$$

Having seen all rules of the small-step semantics, we would like to remark two features. First, note that all rules except the **throw**-rule, only modify the stack by popping and pushing frames. For these rules, a stack therefore really is a stack in the traditional sense. Once we add the **throw**-rule —and in the next chapter the classical control operators— we are able to modify the stack by arbitrarily reading from and writing to it. A stack then no longer is a traditional stack and it might be better to refer to them purely as continuations.

Secondly, these small-step semantics rules give rise to an interesting, more philosophical interpretation of reduction. Evaluating an expression seems to resemble a Socratic dialogue between the expression and the stack. The initial stack variable **halt** can be seen as an opponent who tries to refute the expression. A stack at a certain point of computation forms a rebuttal to the current expression. This dialogue will eventually reveal the 'essence' of an expression, which will be the result of the computation. This approach to evaluating an expression is formalised in *dialogical logic*, or *game semantics* [17].

2.3.5 Big-step semantics

The big-step semantics can now easily be defined by iterating the small-step function. We denote this by $\rightsquigarrow^*$ and give the following definition. A configuration is called *terminal* if it is a configuration $(v, \mathbf{halt})$ for some value v and stack variable **halt**.

$$\begin{aligned}(M,k) \rightsquigarrow^* (M',k') := {} & (M',k') \text{ is terminal and } (M,k) = (M',k') \\ & \text{or there exists an } (M'',k'') \text{ such that } (M,k) \rightsquigarrow (M'',k'') \\ & \quad \text{and } (M'',k'') \rightsquigarrow^* (M',k').\end{aligned}$$

The initial stack only contains **halt**.

2.4 Syntactic sugar

With the language we have defined so far, we can write simple functional programs. We introduce some syntactic sugar which will simplify the examples we present.

We introduce Boolean values and operators. We define the type `bool` as follows.

$$\texttt{bool} = \top \vee \top$$

Now,

$$\begin{aligned}\texttt{true} &= \mathbf{inr}^{\texttt{bool}}\ \mathbf{taut} \\ \texttt{false} &= \mathbf{inl}^{\texttt{bool}}\ \mathbf{taut}\end{aligned}$$

We can mimic if-then-else-statements using a **vcase**-expression. Since we have defined `true` as a right-injection, we want the accompanying expression to be on the right in the **vcase**-expression. The variables x and y are fresh variables.

$$\texttt{ifthenelse}\ L\ N\ M = \mathbf{vcase}(L; x.M; y.N)$$

It is now possible to simulate the Boolean operators.

$$\begin{aligned}\texttt{not}\ p &= \texttt{ifthenelse}\ p \\ &\qquad \texttt{false} \\ &\qquad \texttt{true} \\ \texttt{and}\ p\ q &= \texttt{ifthenelse}\ p \\ &\qquad (\texttt{ifthenelse}\ q\ \texttt{true}\ \texttt{false}) \\ &\qquad (\texttt{ifthenelse}\ q\ \texttt{false}\ \texttt{false}) \\ \texttt{or}\ p\ q &= \texttt{ifthenelse}\ p \\ &\qquad (\texttt{ifthenelse}\ q\ \texttt{true}\ \texttt{true}) \\ &\qquad (\texttt{ifthenelse}\ q\ \texttt{true}\ \texttt{false})\end{aligned}$$

$$\texttt{xor}\ p\ q = \texttt{ifthenelse}\ p\ (\texttt{ifthenelse}\ q\ \texttt{false}\ \texttt{true})\ (\texttt{ifthenelse}\ q\ \texttt{true}\ \texttt{false})$$

Notice that this is not the most efficient implementation of the Boolean operators. The `and` and `or` functions perform an unnecessary evaluation of q after p has been evaluated to `false` or `true`, respectively. However, these additional computations will allow us to construct more interesting examples. In most literature, examples using integers are presented. While we could have chosen to pursue this approach, our definition of the logical operators ensures that we can mimic these examples using Boolean values instead. Also, this means that all our programming examples have an additional logical reading. Not only do we have the entire program as a proof by the Curry-Howard isomorphism, we also have a more literal reading of the logical operators.

2.5 Corresponding problems

Besides indicating structural similarities, the Curry-Howard isomorphism also relates various problems from λ-calculus and intuitionistic logic. These correspondences are the subject of this section.

2.5.1 Type checking vs. Proof checking

In subsection 2.2.2 we discussed type checking and its practical uses for our implementation, but one could wonder about the logical meaning of this procedure. As a brief reminder, in type checking we provide a term and a type and we test whether the given program indeed has the provided type. Hence, we verify a statement $\Gamma \vdash M : \varphi$. Following the Curry-Howard isomorphism, this corresponds to giving some construction and a formula, and testing whether the construction constitutes evidence for the formula. We thus check whether an expected formula is truly proved by the hypothesized proof. This is the process of proof checking. This correspondence is exploited in proof assistants such as Coq and Agda.

2.5.2 Type inference vs. Formula inference

In type inference we solve a problem of the form $\Gamma \vdash M : ?$. That is, for some term M we would like to construct its type. In logic this corresponds to providing a construction and inferring the formula it proves.

2.5.3 Type inhabitation vs. Provability

Type inhabitation is a problem we have not yet previously discussed. The problem we would like to solve is $\Gamma \vdash ? : \varphi$, i.e. given a type φ, is it possible construct a program of said type? In programming, this question is closely

related to program synthesis. While for type inhabitation we merely ask if there exists a program of a certain type, in program synthesis we try to construct such a program. Also in logic, type inhabitation has an interesting counterpart. Namely, if we have some formula φ, is it possible to construct a proof of φ? The type inhabitation problem thus corresponds to the problem of provability in logic.

A special instance of this problem is $\Gamma \vdash ? : \bot$. This is the question whether a system is consistent. By definition, $\nvdash \bot$, so there does not exist a closed term of type $\bot$.

2.5.4 Reduction vs. Normalization

In the previous section, we have seen the operational semantics of our simply typed λ-calculus. Running a program reduces a term to a value by consecutively applying the rules of the small-step semantics. The terms that are reduced all have a destructor —an instance of an elimination-rule— immediately followed by the accompanying constructor —an instance of the introduction rule. A term of this form is called a *reducible expression*, or simply a *redex* [27]. For example, following the rules of the operational semantics, the redex $(\lambda x{:}\varphi \to \varphi \;.\; x)\ (\lambda y{:}\varphi \;.\; y)$ reduces to the value $\lambda y{:}\varphi \;.\; y$. Viewing the term we reduce as a construction, we thus simplify the proof tree by removing what is called a *detour*: an elimination-rule immediately followed by the corresponding introduction-rule. This is the procedure of normalizing a proof. The value obtained by the reduction represents the normal proof of the formula. In the example, we normalize the proof tree

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : \varphi \to \varphi \vdash x : \varphi \to \varphi}\ Ax}{\Gamma \vdash \lambda x{:}\varphi \to \varphi \;.\; x : (\varphi \to \varphi) \to (\varphi \to \varphi)} \to I \quad \dfrac{\dfrac{}{\Gamma, y : \varphi \vdash y : \varphi}\ Ax}{\Gamma \vdash \lambda y{:}\varphi \;.\; y : \varphi \to \varphi} \to I}{\Gamma \vdash (\lambda x{:}\varphi \to \varphi \;.\; x)\ (\lambda y{:}\varphi \;.\; y) : \varphi \to \varphi} \to E$$

to

$$\dfrac{\dfrac{}{\Gamma, y : \varphi \vdash y : \varphi}\ Ax}{\Gamma \vdash \lambda y{:}\varphi \;.\; y : \varphi \to \varphi} \to I$$

Our system uses a call-by-value evaluation strategy. This entails that we do not evaluate beyond a value. For example, in a term $\lambda x{:}\varphi \;.\; M$, the expression M is not evaluated, even though it might be reducible. This indicates that the normalization we obtain is also of a weaker form, since an ultimate proof tree might still contain a redex, which lies embedded in a value.

The correspondence between reduction and normalization is especially interesting since it seems to provide a dynamic interpretation of logical proofs. Although most people see computational programs as dynamic entities which we can 'run', proofs are often considered to be rather static. However, the Curry-Howard isomorphism, and in particular the correspondence between reduction and normalization, shows that logical proof theory does, in fact, have a dynamic aspect.

Chapter 3

Classical Logic

In the previous chapter we established the correspondence between proofs of intuitionistic logic and the simply typed λ-calculus. It was long believed the Curry-Howard isomorphism would not hold for classical logic. However, Griffin changed this by presenting a correspondence between classical logic and a λ-calculus extended with control operators [14].

In this chapter, we will discuss the correspondence and adjust our implementation to include control operators. Again, we will elaborate on the operational semantics. We will present some examples and conclude the chapter by discussing the correspondence between double negation translation and continuation-passing style transformation.

3.1 The Curry-Howard isomorphism

In this section we will discuss classical propositional logic. Subsequently, we will introduce the $\lambda_{\mathcal{C}}$-calculus, which was the first calculus shown to be corresponding to classical logic.

3.1.1 Classical propositional logic

Classical logic can be obtained from intuitionistic logic by adding one of a number of (more or less) equivalent axioms. The ones that are most often discussed, and will be considered here, are the *law of excluded middle*, *double negation elimination* and *Peirce's law*.

The law of excluded middle (LEM) states that every proposition is either true or false, which was rejected in intuitionistic logic. LEM is formalised as $\varphi \vee \neg\varphi$. This law excludes the third option we had in intuitionistic logic, namely when neither a proposition nor its negation could be proved. Therefore, this law is also referred to as *tertium non datur*: "no third [possibility] is given".

Double negation elimination (DNE) is formalised as $\neg\neg\varphi \to \varphi$. In intuitionistic logic it is possible to prove the reverse, $\varphi \to \neg\neg\varphi$, but DNE is not provable.

This becomes evident when we think about the intuitive interpretation of intuitionistic logic: $\neg\neg\varphi$ can be interpreted as stating that there is no proof for $\neg\varphi$. This, however, does not imply that there is a proof for φ. Adding DNE to intuitionistic logic, and thus obtaining classical logic, results in a 'symmetrical' interpretation of truth: if some statement is not false, it has to be true.

A third possible axiom is Peirce's law (PL). This law is defined as $(\neg\varphi \to \varphi) \to \varphi$. We interpret PL as follows: if from the assumption that some proposition is false, we can conclude that it is true, then it must be the case that the proposition is true.

For reasons to become apparent later on, we will define classical logic as intuitionistic logic extended with Peirce's law. We still have our set of formulas Φ. The proof system for classical logic is defined by expanding Figure 2.1 with the rule

$$\frac{\Gamma, \neg\varphi \vdash \varphi}{\Gamma \vdash \varphi}\ PL$$

Since we now have a system with both Peirce's law and Ex Falso Quodlibet, we have obtained classical logic [1]. We show that we can derive the other classical axioms with the following proofs.

First, we show PL and EFQ imply DNE.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg\varphi \vdash \neg\neg\varphi}\ Ax \quad \dfrac{}{\neg\varphi \vdash \neg\varphi}\ Ax}{\neg\neg\varphi, \neg\varphi \vdash \bot}\ \to E}{\neg\neg\varphi, \neg\varphi \vdash \varphi}\ EFQ}{\neg\neg\varphi \vdash \varphi}\ PL}{\vdash \neg\neg\varphi \to \varphi}\ \to I$$

Secondly, PL implies LEM.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}\ Ax}{\varphi \vdash \varphi \vee \neg\varphi}\ \vee I \quad \dfrac{}{\neg(\varphi \vee \neg\varphi) \vdash \neg(\varphi \vee \neg\varphi)}\ Ax}{\varphi, \neg(\varphi \vee \neg\varphi) \vdash \bot}\ \to E}{\neg(\varphi \vee \neg\varphi) \vdash \neg\varphi}\ \to I}{\neg(\varphi \vee \neg\varphi) \vdash \varphi \vee \neg\varphi}\ \vee I}{\vdash \varphi \vee \neg\varphi}\ PL$$

Considering these proofs, we can infer that DNE is a stronger axiom than PL and LEM, since it requires the use of EFQ. Hence, a language with DNE as an inference rule does not need an inference rule for EFQ. Languages with rules such as PL and LEM, on the other hand, do. Our language already contained a rule for EFQ, which is why we chose to add PL instead of DNE.

A third law —which we have not yet introduced, but will turn out to be useful— is a generalised version of Peirce's law (PL^+); $((\varphi \to \psi) \to \varphi) \to \varphi$.

PL$^+$ is implied by PL and EFQ.

$$\dfrac{\dfrac{\dfrac{}{(\varphi \to \psi) \to \varphi \vdash (\varphi \to \psi) \to \varphi}\, Ax \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg\varphi \vdash \neg\varphi}\, Ax \quad \dfrac{}{\varphi \vdash \varphi}\, Ax}{\neg\varphi, \varphi \vdash \bot}\, \to E}{\neg\varphi, \varphi \vdash \psi}\, EFQ}{\neg\varphi \vdash \varphi \to \psi}\, \to I}{(\varphi \to \psi) \to \varphi, \neg\varphi \vdash \varphi}\, \to E}{(\varphi \to \psi) \to \varphi \vdash \varphi}\, PL}{\vdash ((\varphi \to \psi) \to \varphi) \to \varphi}\, \to I$$

The use of EFQ in this proof seems to suggest that PL$^+$ is as strong as DNE, though this is not true. It is possible to derive EFQ from DNE, while we cannot do so from PL$^+$. It therefore seems counterintuitive that we need EFQ in the proof for PL$^+$. This realisation has led Ariola, Herbelin and Sabry [2] to introducing a new judgement $\perp\!\!\!\perp$, which could be interpreted as a weaker version of $\bot$. A similar effort is made in [20] by introducing a judgement #. These approaches have interesting implications. However, we will not elaborate on them in this thesis.

3.1.2 $\lambda_{\mathcal{C}}$-calculus

Functional programming languages have long been equipped with functions to control their evaluation, such as `call/cc`. That is, programs with these functions are able to manipulate their own (call)stack. Felleisen, Friedman, Kohlbecker and Duba [11] proposed a formal integration of control into an untyped λ-calculus. They defined evaluation contexts E by the following grammar, where N is an expression and V a value. The notation is taken from [14].

$$E := [\,] \mid EN \mid VE$$

Here $[\,]$ represents a hole. We write $E[M]$ to denote the result of placing M in the hole of E. Note that these contexts are roughly equivalent to our stacks. We would write $E[M]$ as a configuration $(M, E[\,])$.

The grammar of the λ-calculus was expanded by adding two expressions: $\mathcal{A}(M)$ and $\mathcal{C}(M)$. An expression $\mathcal{A}(M)$ aborts the program and returns M.

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

An expression $\mathcal{C}(M)$ also leaves the current control context, but takes an abstraction of the continuation and applies it to M. This provides M the control over the rest of the evaluation.

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

This extended calculus, called the $\lambda_{\mathcal{C}}$-calculus, now supports the formalisation of control functions like `call/cc`.

$$\texttt{call/cc}\ M = \mathcal{C}(\lambda k.k\ (M\ k))$$

3.1.3 The Curry-Howard isomorphism

The motivation for introducing the $\lambda_{\mathcal{C}}$-calculus was purely computational. It was assumed the new expressions had no corresponding interpretation in logic. However, Griffin [14] proved otherwise. By typing the expressions, he showed that $\mathcal{C}^{\varphi}(M)$ corresponds to the double negation elimination rule and $\mathcal{A}^{\varphi}(M)$ to EFQ.

Just as DNE is a strong law which implies EFQ, a language with $\mathcal{C}(M)$ does not require $\mathcal{A}(M)$. If we let d be a dummy variable bound in M, we obtain

$$\mathcal{A}(M) = \mathcal{C}(\lambda d.M)$$

Notice that in our previously defined language Λ_{Π} we already have an expression mirroring $\mathcal{A}^{\varphi}(M)$, namely $\mathbf{abort}^{\varphi}\ M$. We therefore opt, as mentioned, to expand our language with an expression which represents Peirce's law, instead of double negation elimination. Together with **abort** we then obtain classical logic. We introduce the term $\mathbf{letcc}\ \alpha^{\varphi}.M$, where α is a continuation variable. We expand Λ_{Π}

$$\Lambda_{\Pi} := ... \mid \mathbf{letcc}\ \alpha^{\Pi}.\ \Lambda_{\Pi}$$

and add the following rule to the updated proof system of Figure 2.2

$$\frac{\Gamma_{\text{true}}; \Gamma_{\text{false}}, \alpha : \varphi \vdash^{\text{true}} M : \varphi}{\Gamma \vdash^{\text{true}} \mathbf{letcc}\ \alpha^{\varphi}.\ M : \varphi}\ PL$$

We read an expression $\mathbf{letcc}\ \alpha^{\varphi}.M$ as *"let the current continuation be α in M"*.

3.2 Type checking and inference

We extend our implementation with a notion of control. This enables us to write programs for classical proofs.

3.2.1 Data type and type inference

We merely have to add an expression which simulates **letcc**.

```
data Type = ...
          | LetCC String Type Expr
```

For the type inference of an expression `LetCC alpha phi m`, we insert variable `alpha` into the environment with type `phi` → `Error` and subsequently infer the type of `m`. If this type equals `phi`, we return that type; otherwise, an error should occur.

3.3 Operational Semantics

We have briefly mentioned the most important computational change that occurs when we move from intuitionistic to classical logic: non-local control flow.

In this section we will formalise the operational semantics of our implementation.

3.3.1 Small-step semantics

Peirce's law tells us that if we derive an expression M of type φ with an assumption α of type φ in Γ_{false}, then we can derive **letcc** α^{φ} . M of type φ. In the previous chapter we have seen that this assumption α can be interpreted as a continuation. So, when we encounter an expression **letcc** α^{φ} . M, we want to bind the current continuation to α in M and continue the evaluation of M. The continuation remains unchanged. This is captured in the following rule which we add to the small-step semantics.

$$\textbf{letcc}\ \alpha^{\varphi}\ .\ M,\ k \quad \rightsquigarrow \quad M[k/\alpha],\ k$$

Here, $M[k/\alpha]$ is the capture-avoiding substitution of a continuation k for a continuation variable α. This especially concerns the **throw**-statements. Any statement of the form **throw** α N will now become **throw** k N. We already had the small-step for **throw**, namely

$$\textbf{throw}\ k\ M,\ k' \quad \rightsquigarrow \quad M,\ k$$

When invoked, this will install k as the new continuation and thus change the evaluation of the program. In the language without **letcc**, we could only manually create stacks to pass as an argument to **throw**. Now, **letcc** provides a way of saving particular continuations and using them later on in the evaluation.

We illustrate these rules with an example. Consider the following program.

$$\texttt{or false}\ (\textbf{letcc}\ \alpha^{\texttt{bool}}.\ (\texttt{or true}\ (\textbf{throw}\ k\ \texttt{false})))$$

We will show that this expression evaluates to `false`. We expand the use of syntactic sugar into the small-step-semantics rules. The steps ${}^{3}\rightsquigarrow$ and ${}^{6}\rightsquigarrow$ correspond to the **letcc**- and **throw**-rules above, respectively.

$$\begin{aligned}
&\texttt{or false}\ (\textbf{letcc}\ \alpha^{\texttt{bool}}.\ (\texttt{or true}\ (\textbf{throw}\ \alpha\ \texttt{false}))),\ [\textbf{halt}]\\
{}^{1}\rightsquigarrow\ &\texttt{false},\ [\texttt{or _}\ (\textbf{letcc}\ \alpha^{\texttt{bool}}.\ (\texttt{or true}\ (\textbf{throw}\ \alpha\ \texttt{false}))),\ \textbf{halt}]\\
{}^{2}\rightsquigarrow\ &(\textbf{letcc}\ \alpha^{\texttt{bool}}.\ (\texttt{or true}\ (\textbf{throw}\ \alpha\ \texttt{false}))),\ [\texttt{or false _},\ \textbf{halt}]\\
{}^{3}\rightsquigarrow\ &\texttt{or true}\ (\textbf{throw}\ [\texttt{or false _},\ \textbf{halt}]\ \texttt{false}),\ [\texttt{or false _},\ \textbf{halt}]\\
{}^{4}\rightsquigarrow\ &\texttt{true},\ [\texttt{or _}\ (\textbf{throw}\ [\texttt{or false _},\ \textbf{halt}]\ \texttt{false}),\ \texttt{or false _},\ \textbf{halt}]\\
{}^{5}\rightsquigarrow\ &\textbf{throw}\ [\texttt{or false _},\ \textbf{halt}]\ \texttt{false},\ [\texttt{or true _},\ \texttt{or false _},\textbf{halt}]\\
{}^{6}\rightsquigarrow\ &\texttt{false},\ [\texttt{or false _},\ \textbf{halt}]\\
{}^{7}\rightsquigarrow\ &\texttt{false},\ [\textbf{halt}]
\end{aligned}$$

We see that in ${}^{3}\rightsquigarrow$, which is the evaluation of the **letcc**-statement, the current continuation is retrieved and stored in the position of the variable α. In ${}^{6}\rightsquigarrow$, this context is reinstalled to replace the current continuation. The `or true _`-frame is thus ignored and has no influence on the final result.

3.3.2 Stacks

The addition of **letcc** does not introduce a new type of stack, though we do now see the use of **halt**. We use the variable **halt** to represent the initial —or top— continuation of the program. This assumption ensures that we are able to construct stacks without making our logic inconsistent. Without **halt** this would not be possible, since we cannot derive a closed expression of type $\bot$ [20].

Furthermore, as briefly mentioned before, the name 'stack' does not truly apply any longer. A **letcc**-expression saves an entire continuation without modifying it, whereas **throw** changes the continuation completely. These operations go beyond the pop- and push-functions of a traditional stack.

3.3.3 Big-step semantics

In intuitionistic logic, the big-step semantics could be used to retrieve the final value of a program, without considering all steps in between. This approach, however, no longer works for classical logic. The presence of the control operator **letcc** together with **throw**-expressions entails the possibility of 'jumping out' of an evaluation context. This prevents us from directly axiomatizing the big step reduction. However, we can still define the big-step semantics in terms of the small-step semantics, as we did before.

3.3.4 Other axioms

We have seen that adding **letcc** to our calculus suffices to obtain classical logic. However, we now discuss the computational equivalents of the other classical axioms and their operational semantics here. Using the proof trees from section 3.1.1, these functions can be defined in terms of **letcc**.

The generalised version of Peirce's law mirrors the function `call/cc`. We derive its definition as follows.

$$\textbf{callcc}\ x^{\varphi\to\psi}.\ M = \textbf{letcc}\ \alpha^{\varphi}\ .\ M[\lambda y{:}\varphi\ .\ \textbf{abort}^{\psi}\ (\textbf{throw}\ \alpha\ y)/x]$$

The following typing rule for **callcc** is derivable.

$$\frac{\Gamma_{\text{true}}, x:\varphi\to\psi; \Gamma_{\text{false}} \vdash^{\text{true}} M:\varphi}{\Gamma_{\text{true}}; \Gamma_{\text{false}} \vdash^{\text{true}} \textbf{callcc}\ x^{\varphi\to\psi}.\ M:\varphi}$$

We see that **callcc** retrieves the current continuation and uses this to construct a function-like abstraction [10]. This representation gets bound to the variable x in M. The following small-step semantics rule for **callcc** is implied.

$$\textbf{callcc}\ x^{\varphi\to\psi}.\ M,\ k \quad\rightsquigarrow\quad M[\lambda y{:}\varphi\ .\ \textbf{abort}^{\psi}\ (\textbf{throw}\ k\ y)/x],\ k$$

If this substituted function thereafter gets applied to a value of type φ, the stored continuation k is reinstated. For example, consider the following expression, where M is some term of type $\psi \to \texttt{bool}$.

$$\texttt{and true}\ (\textbf{callcc}\ x^{\texttt{bool}\to\psi}.\ M\ (x\ \texttt{false}))$$

This expression will store the current continuation of type `bool` in a function-like abstraction of type $\texttt{bool} \to \psi$. Subsequently, it will perform the computations in M, before reinserting the context saved in x and throwing it the value `false`. The final value of this program is `false`.

Secondly, we could implement DNE using the following sugar.

$$\Delta\alpha^{\varphi}.\ M = \mathbf{letcc}\ \alpha^{\varphi}\ .\ \mathbf{abort}^{\varphi}\ M$$

This construct gets the typing rule

$$\frac{\Gamma_{\text{true}}; \Gamma_{\text{false}}, \alpha : \varphi \vdash^{\text{true}} M : \bot}{\Gamma_{\text{true}}; \Gamma_{\text{false}} \vdash^{\text{true}} \Delta\alpha^{\varphi}.\ M : \varphi}$$

The operational semantics of Δ is quite similar to that of **letcc**. The difference occurs after we have bound the current continuation to the continuation variable α; in Δ we now abandon the current context using an **abort**-expression, whereas in **letcc** this context is preserved. This process could be captured in the following small-step semantics rule.

$$\Delta\alpha^{\varphi}.\ M,\ k \quad \rightsquigarrow \quad M[k/\alpha],\ \mathbf{abort}^{\varphi}\ \_ :: k$$

We will see an example which uses the Δ-construct in the following section.

Finally, we define LEM.

$$\mathbf{lem}^{\varphi} = \mathbf{letcc}\ \alpha^{\varphi \vee \neg\varphi}\ .\ \mathbf{inr}^{\varphi \vee \neg\varphi}\ (\lambda x{:}\varphi\ .\ \mathbf{throw}\ \alpha\ (\mathbf{inl}^{\varphi \vee \neg\varphi}\ x))$$

The construct takes no arguments, so we obtain the typing rule

$$\frac{}{\Gamma \vdash^{\text{true}} \mathbf{lem}^{\varphi} : \varphi \vee \neg\varphi}$$

and we get the following small-step semantics rule.

$$\mathbf{lem}^{\varphi},\ k \quad \rightsquigarrow \quad \mathbf{inr}^{\varphi \vee \neg\varphi}\ (\lambda x{:}\varphi\ .\ \mathbf{throw}\ k\ (\mathbf{inl}^{\varphi \vee \neg\varphi}\ x)),\ k$$

Computationally, we see that **lem** provides an expression of type $\varphi \vee \neg\varphi$. The construct **lem** first invokes $\mathbf{inr}^{\varphi \vee \neg\varphi}\ (\lambda x{:}\varphi\ .\ \mathbf{throw}\ k\ (\mathbf{inl}^{\varphi \vee \neg\varphi}\ x))$. If the continuation is of such a form that $(\lambda x{:}\varphi\ .\ \mathbf{throw}\ k\ (\mathbf{inl}^{\varphi \vee \neg\varphi}\ x))$ will be evaluated, then we return to the continuation k with $\mathbf{inl}^{\varphi \vee \neg\varphi}\ x$ where x is supplied by the continuation. So, as [20] notes, the program 'time travels' between different moments in the evaluation. In [15], Harper argues that this construct nicely shows the dialogue between a program and its continuation, as discussed in this thesis before. By invoking the **inr**, the program assumes $\neg\varphi$. If the subsequent expression is evaluated, it means the continuation has refuted the program by claiming that φ. The program then behaves as if it 'changes its mind' and travels back to the original continuation, now inserting the **inl** and thus claiming that φ. Although this provides an interesting interpretation for the **lem**-construct, its practical use in programming is not immediately clear and remains to be investigated.

3.4 Examples

In programming, continuations are used to implement all kinds of control mechanisms, such as exceptions and co-routines. We will present an example program with exceptions.

We can implement the exception-function `catch` using continuations [18].

$$\texttt{catch}\ \alpha^{\varphi}\ M = \Delta\alpha^{\varphi}.\ (\textbf{throw}\ \alpha\ M)$$

With `catch` we can now write programs that are able to throw and handle exceptions. Take for example the situation in which we would like to non-lazily evaluate a list of Boolean values. This example resembles the often used example of integer multiplication [18, 20]. We are going to write a sugar-function `andS` in which the argument is a list of terms.

A first attempt at the function `andS`, which checks whether all expressions in some list evaluate to `true`, simply constructs an `and`-expression.

$$\begin{aligned}
&\texttt{andS} && :: \texttt{[Expr] -> Expr}\\
&\texttt{andS}\ [\,] && = \texttt{true}\\
&\texttt{andS}\ (x : xs) && = \texttt{and}\ x\ (\texttt{andS}\ xs)
\end{aligned}$$

While evaluation of this function results in the correct value, it is highly inefficient. Once we encounter a value `false`, we do not need to evaluate the rest, since we know the entire expression will result in the value `false`. We use exceptions to break out of the evaluation in such a situation.

$$\begin{aligned}
&\texttt{andS} && :: \texttt{[Expr] -> Expr}\\
&\texttt{andS}\ xs && = \texttt{catch}\ \alpha^{\texttt{bool}}\ (\texttt{andS'}\ xs\ \alpha)\\
&\\
&\texttt{andS'}\ [\,]\ k && = \texttt{true}\\
&\texttt{andS'}\ (y : ys)\ k && \mid y == \texttt{true}\ \ = \texttt{andS'}\ ys\ k\\
& && \mid y == \texttt{false} = \textbf{throw}\ k\ \texttt{false}
\end{aligned}$$

The use of continuations with the **throw**- and `catch`-expression in this function provide a way to raise an exception and immediately return a result. This makes the program more efficient.

3.5 Double negation translation vs. CPS

The Curry-Howard isomorphism provides an additional correspondence between classical logic and functional programming. It relates double negation translations to continuation-passing style transformations.

Double negation translation has long been used to embed classical logic into intuitionistic logic. The goal of this approach is to translate a classical formula φ which is not provable in intuitionistic logic, into a classically equivalent formula

φ' which is provable in intuitionistic logic. Over the years several translations have been proposed. For propositional logic, the first formulation is Glivenko's Theorem, which states that a propositional formula φ is classically provable if and only if $\neg\neg\varphi$ is provable intuitionistically [22].

In 1991, Murthy established the correspondence between double negation translations and continuation-passing style transformations [23]. *Continuation-passing style* (CPS) is a way to explicitly propagate continuations in a language with no specific control operators. It uses an extra argument, namely a function representing the continuation. The exact transformation strongly depends on the chosen evaluation strategy and double negation translation and hence there exist many different CPS transformations.

We present a transformation for our call-by-value language —based on [8, 9, 21]. First, we define a double negation translation on our types as follows.

$$\begin{aligned}
(\top)^* &= \top \\
(\bot)^* &= \bot \\
(p)^* &= p && \text{for } p \text{ a type variable} \\
(\varphi \to \psi)^* &= \varphi^* \to \neg\neg\psi^* \\
(\varphi \wedge \psi)^* &= \varphi^* \wedge \psi^* \\
(\varphi \vee \psi)^* &= \varphi^* \vee \psi^*
\end{aligned}$$

We now want a translation for our configurations. We denote the CPS transform of an expression M as $\overline{M}$. If $\Gamma \vdash^{\text{true}} M : \varphi$, then $\Gamma^* \vdash^{\text{true}} \overline{M} : \neg\neg\varphi^*$, where Γ^* is defined by transforming all elements of Γ. The translations for expressions M are presented in Figure 3.1.

Using CPS, we can transform configurations into expressions. A well-typed configuration (M, k) can be transformed into the expression $\overline{M}\ k^\circ$, where k° is the function representation of a stack given by

$$\begin{aligned}
k^\circ &= \lambda_.k^\bullet \\
\mathbf{halt}^\bullet &= \mathbf{halt}\ _ \\
(f :: k')^\bullet &= (k')^\bullet[f/_]
\end{aligned}$$

By inspection of the construction of stacks, it can be seen that k° will always be of the form $\lambda\_.\mathbf{halt}\ F$. The type of $\_$ will be the translated type of the expression which should have been inserted in the original stack: if k is of type φ, $\_$ will be of type φ^*. The type of the stack variable **halt** applied to F will be of type $\bot$. Hence, if k is of type φ, k° is of type $\neg\varphi^*$. The type of the final expression $\overline{M}\ k^\circ$ will thus be $\bot$. This reflects the original purpose of stacks; proving falsehood.

For example, say we would like to translate the configuration

$$(\mathbf{letcc}\ \alpha^\top\ .\ ((\lambda x{:}\bot\ .\ \mathbf{taut})\ (\mathbf{throw}\ \alpha\ \mathbf{taut})), [\langle \mathbf{taut}, \_\rangle, \mathbf{halt}])$$

To be able to distinguish between the **taut**-expressions, we annotate them as follows.

$$(\mathbf{letcc}\ \alpha^\top\ .\ ((\lambda x{:}\bot\ .\ \mathbf{taut}_1)\ (\mathbf{throw}\ \alpha\ \mathbf{taut}_2)), [\langle \mathbf{taut}, \_\rangle, \mathbf{halt}])$$

$$
\begin{aligned}
\overline{x} &= \lambda k.k\ x \\
\overline{\mathbf{taut}} &= \lambda k.k\ \mathbf{taut} \\
\overline{\mathbf{abort}\ M} &= \lambda k.\overline{M}\ (\lambda d.d) \\
\overline{\lambda x.M} &= \lambda k.k(\lambda x.\lambda k'.\overline{M}\ k') \\
\overline{M\ N} &= \lambda k.\overline{M}\ (\lambda f.\overline{N}\ (\lambda v.(f\ v)\ k)) \\
\overline{\langle M, N\rangle} &= \lambda k.\overline{M}\ (\lambda v.\overline{N}\ (\lambda w.k\ \langle v, w\rangle)) \\
\overline{\mathbf{fst}\ M} &= \lambda k.\overline{M}\ (\lambda v.k\ (\mathbf{fst}\ v)) \\
\overline{\mathbf{snd}\ M} &= \lambda k.\overline{M}\ (\lambda v.k\ (\mathbf{snd}\ v)) \\
\overline{\mathbf{inl}\ M} &= \lambda k.\overline{M}\ (\lambda v.k\ (\mathbf{inl}\ v)) \\
\overline{\mathbf{inr}\ M} &= \lambda k.\overline{M}\ (\lambda v.k\ (\mathbf{inr}\ v)) \\
\overline{\mathbf{vcase}(L; x.M; y.N)} &= \lambda k.\overline{L}(\lambda v.\mathbf{vcase}(v; x.\overline{M}\ k; y.\overline{N}\ k)) \\
\overline{\mathbf{throw}\ l\ M} &= \lambda k.\overline{M}\ l \\
\overline{\mathbf{letcc}\ \alpha.M} &= \lambda k.(\lambda\alpha.\overline{M})\ k\ k
\end{aligned}
$$

Figure 3.1: CPS transformation

First, we compute $\overline{\mathbf{letcc}\ \alpha^{\top}\ .\ ((\lambda x{:}\bot\ .\ \mathbf{taut}_1)\ (\mathbf{throw}\ \alpha\ \mathbf{taut}_2))}$.[1]

$$
\begin{aligned}
&\overline{\mathbf{letcc}\ \alpha^{\top}\ .\ ((\lambda x{:}\bot\ .\ \mathbf{taut}_1)\ (\mathbf{throw}\ \alpha\ \mathbf{taut}_2))} \\
&= \lambda k.(\lambda\alpha.\overline{((\lambda x{:}\bot\ .\ \mathbf{taut}_1)\ (\mathbf{throw}\ \alpha\ \mathbf{taut}_2))})\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.(\overline{\lambda x{:}\bot\ .\ \mathbf{taut}_1})\ (\lambda f.(\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.(f\ v)\ k'))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.(\lambda k''.k''\ (\lambda x.\lambda k'''.\overline{\mathbf{taut}_1}\ k'''))\ (\lambda f.(\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.(f\ v)\ k'))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.(\lambda k''.k''\ (\lambda x.\lambda k'''.k'''\ \mathbf{taut}_1))\ (\lambda f.(\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.(f\ v)\ k'))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.((\lambda f.(\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.(f\ v)\ k'))\ (\lambda x.\lambda k'''.k'''\ \mathbf{taut}_1))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.((\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.((\lambda x.\lambda k'''.k'''\ \mathbf{taut}_1)\ v)\ k'))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.((\overline{\mathbf{throw}\ \alpha\ \mathbf{taut}_2})\ (\lambda v.k'\ \mathbf{taut}_1))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.((\lambda k''.\overline{\mathbf{taut}_2}\ \alpha)\ (\lambda v.k'\ \mathbf{taut}_1))))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.\overline{\mathbf{taut}_2}\ \alpha))\ k\ k \\
&= \lambda k.(\lambda\alpha.(\lambda k'.\alpha\ \mathbf{taut}_2))\ k\ k \\
&= \lambda k.k\ \mathbf{taut}_2
\end{aligned}
$$

[1] Note that this example also demonstrates that our translation is call-by-value; a call-by-name evaluation and translation would have installed $\mathbf{taut}_1$ instead of $\mathbf{taut}_2$.

Secondly, we get $[\langle \mathbf{taut}, \_ \rangle, \mathbf{halt}]^\circ = \lambda\_.\mathbf{halt}\ \langle \mathbf{taut}, \_ \rangle$. So, for the transformation of our entire configuration we obtain the following.

$$\begin{aligned}
&(\lambda k.k\ \mathbf{taut}_2)\ (\lambda_.\mathbf{halt}\ \langle \mathbf{taut}, _ \rangle) \\
&= (\lambda_.\mathbf{halt}\ \langle \mathbf{taut}, _ \rangle)\ \mathbf{taut}_2 \\
&= \mathbf{halt}\ \langle \mathbf{taut}, \mathbf{taut}_2 \rangle
\end{aligned}$$

Chapter 4

Meta-theory

In the previous chapters we have constructed the language Λ_{Π} together with the typing rules and operational semantics. This now forms a full system of propositional classical logic. In this section we will prove that the operational semantics hava the meta-theoretic properties of determinism, progress, preservation and termination. It directly follows that these properties also hold for the intuitionistic subsystem presented in the first chapter.

4.1 Well-typed configurations

Before we can prove the meta-theoretic properties, we first need a notion of well-typed configurations. We have seen that a *configuration* is the combination of an expression and a stack at a certain point of evaluation. We say that a configuration consisting of an expression M and a stack k, is *well-typed* if there exist $\Gamma_{\text{true}}, \Gamma_{\text{false}}, \varphi$ and ψ, such that the following two statements hold.

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \varphi \vdash^{\text{true}} M : \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \varphi \vdash^{\text{false}} k : \psi$$

We will also write this as $\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \varphi \vdash^{\text{conf}} (M, k) : \psi$.

For every computation of $\Gamma \vdash M : \varphi$ we get the initial well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \varphi \vdash^{\text{conf}} (M, \mathbf{halt}) : \varphi$$

We call a configuration (M, k) *terminal* if M is a value and k is **halt** —i.e. a continuation variable.

4.2 Determinism

The first property of our system we would like to prove, is the property of *determinism*. Determinism is important as it tells us that our programs can be trusted to always yield the same result, if we choose to rerun them.

Theorem 4.1 (Determinism)**.** *For any well-typed configuration, at most one small-step transition rule applies.*

Proof. This follows from an inspection of our small-step semantics, which has been defined as a function. □

4.3 Progress

The property of progress says that each well-typed configuration either marks the end of the computation, or matches one of the small-step semantics rules. Hence, it tells us that a well-typed configuration never 'gets stuck'. This is one of the uses of a type system.

Theorem 4.2 (Progress)**.** *Every well-typed configuration* $(M,\ k)$ *is either terminal or there is some small-step rule* $(M, k) \rightsquigarrow (M', k')$ *that applies to it.*

Proof. We prove this by induction on the typing derivation of our configurations.

Case. We first consider the **abort**-expression. Such an expression could have any type φ. This places no restriction on the frame on top of the stack, as long as the type of the stack is also φ. Different top-frames have no influence on the step to be performed.

> $\underline{\mathbf{abort}^{\varphi}\ M : \varphi,\ k : \varphi}$. We apply the small-step semantics rule and yield the configuration $(M,\ \mathbf{abort}^{\varphi}\ \_ :: k)$.

Case. We now consider the configurations with an expression of the form $\lambda x{:}\varphi\ .\ M : \varphi \to \psi$. This expression thus is a value. By inspecting the typing rules of stacks in Figure 2.3 we can infer which frames can be on top of the stack in a well-typed configuration. For example, the type $\varphi \to \psi$ could never match the type $\sigma \vee \tau$. We therefore know the top-frame cannot be a **vcase**. Similary, we should not encounter an **abort**, **fst** or **snd**-frame. This leads us to the following well-typed configurations. For each configuration we show that there either is a small-step semantics rule to perform or that the computation is done.

> $\underline{\lambda x{:}\varphi\ .\ M : \varphi \to \psi,\ \mathbf{halt} : \varphi \to \psi}$. This is a terminal configuration.
>
> $\underline{\lambda x{:}\varphi\ .\ M : \varphi \to \psi,\ \_\ N :: k : \varphi \to \psi}$. We now have a value and an application-frame on top of our stack. We find a small-step semantics rule that yields the configuration $(N,\ \lambda x{:}\varphi\ .\ M\ \_ :: k)$.
>
> $\underline{\lambda x{:}\varphi\ .\ M : \varphi \to \psi,\ (\lambda y{:}\varphi \to \psi\ .\ N)\ \_ :: k : \varphi \to \psi}$. We again have a value and an application-frame. We now apply the corresponding rule and obtain the configuration $(N[\lambda x{:}\varphi\ .\ M/y],\ k)$.
>
> $\underline{\lambda x{:}\varphi\ .\ M : \varphi \to \psi,\ \langle \_, N\rangle :: k : \varphi \to \psi}$. With this configuration, we find the small-step rule that inserts the value into the frame. We obtain the configuration $(N,\ \langle \lambda x{:}\varphi\ .\ M, \_\rangle :: k)$.

$\underline{\lambda x{:}\varphi \;.\; M : \varphi \to \psi,\ \langle v, \_\rangle :: k : \varphi \to \psi}$. This configuration matches the small-step rule which returns a pair of values. We transform the configuration into $(\langle v, \lambda x{:}\varphi \;.\; M\rangle,\ k)$.

$\underline{\lambda x{:}\varphi \;.\; M : \varphi \to \psi,\ \mathbf{inl}^{\varphi\to\psi\vee\rho}\ \_ :: k : \varphi \to \psi}$. The value is now inserted as the left injection. We find a small-step semantics rule that yields the configuration $(\mathbf{inl}^{\varphi\to\psi\vee\rho}\ (\lambda x{:}\varphi \;.\; M),\ k)$.

$\underline{\lambda x{:}\varphi \;.\; M : \varphi \to \psi,\ \mathbf{inr}^{\rho\vee\varphi\to\psi}\ \_ :: k : \varphi \to \psi}$. We construct the right injection using a rule and find the configuration $(\mathbf{inr}^{\rho\vee\varphi\to\psi}\ (\lambda x{:}\varphi \;.\; M),\ k)$.

Case. We continue with the configurations in which the expression is an application $M\ N$. We find that an expression $M\ N$ could be of any type. Hence, the top of the stack could consist of any frame that matches the type. We can introduce one case, which deals with any stack.

$\underline{M\ N : \varphi,\ k : \varphi}$. We apply the small-step rule and obtain the new configuration $(M,\ \_\ N :: k)$.

Case. If we encounter a pair $\langle M, N\rangle$, this pair either is a value or not (yet). Either way, we should have a stack of type $\varphi \wedge \psi$. In case our pair is not yet a value, the exact frame on top of the stack is not important, since we first have to evaluate the expressions inside the pair. If the pair already is a value, on the contrary, we observe the frame on top of the stack to determine the right operational step. The type of the pair is a product type. The top-frame could therefore never be an **abort**, $\_\ N$, or **vcase**-frame. This leaves us with the following well-typed configurations.

$\underline{\langle M, N\rangle : \varphi \wedge \psi,\ k : \varphi \wedge \psi}$. Our pair $\langle M, N\rangle$ is not a value and consists of expressions which have not been evaluated yet. We apply the small-step semantics rule and obtain the configuration $(M,\ \langle \_, N\rangle :: k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \mathbf{halt} : \varphi \wedge \psi}$. This is a terminal configuration.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ (\lambda x{:}\varphi \wedge \psi \;.\; M)\ \_ :: k : \varphi \wedge \psi}$. We insert the value into the application frame. This yield the configuration $(M[\langle v, w\rangle/x],\ k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \langle \_, N\rangle :: k : \varphi \wedge \psi}$. We insert the value into the pair and get $(N,\ \langle\langle v, w\rangle, \_\rangle :: k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \langle u, \_\rangle :: k : \varphi \wedge \psi}$. As in the last case, we obtain a new pair in our configuration $(\langle u, \langle v, w\rangle\rangle,\ k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \mathbf{inl}^{\varphi\wedge\psi\vee\rho}\ \_ :: k : \varphi \wedge \psi}$. Using the value we construct a left injection. The resulting configuration is $(\mathbf{inl}^{\varphi\wedge\psi\vee\rho}\ \langle v, w\rangle,\ k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \mathbf{inr}^{\rho\vee\varphi\wedge\psi}\ \_ :: k : \varphi \wedge \psi}$. We can do the same thing for the right injection and yield $(\mathbf{inr}^{\rho\vee\varphi\wedge\psi}\ \langle v, w\rangle,\ k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \mathbf{fst}\ \_ :: k : \varphi \wedge \psi}$. We simply take the first projection and obtain the configuration $(v,\ k)$.

$\underline{\langle v, w\rangle : \varphi \wedge \psi,\ \mathbf{snd}\ \_ :: k : \varphi \wedge \psi}$. The second projection similarly leads to the configuration $(w,\ k)$.

Case. Configurations with a projection are never a value. Hence, the frame on top of the stack does not influence which rule to perform. If the type of the stack matches the type of the expression we get the following cases.

$\underline{\mathbf{fst}\ M : \varphi,\ k : \varphi}$. This yields the configuration $(M,\ \mathbf{fst}\ \_ :: k)$.

$\underline{\mathbf{snd}\ M : \varphi,\ k : \varphi}$. Similarly, we obtain the configuration $(M,\ \mathbf{snd}\ \_ :: k)$.

Case. Injections can be values. Hence, we need to distinguish the cases in which they are, from the cases in which they are not. If it is not yet a value, we first need to evaluate the embedded expression. To do this, the exact frame on top of the stack does not matter. On the other hand, if we do have a value, we handle the cases according to the top-frame. The type of such a value always is a sum-type. Stacks with an **abort**, $\_\ N$, **fst**, or **snd**-frame therefore do not construct well-typed configurations. We only demonstrate the cases for **inl**-expressions; handling the **inr**-expressions happens analogous.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ M : \varphi \vee \psi,\ k : \varphi \vee \psi}$. We perform the small-step semantics rule and construct the configuration $(M,\ \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \mathbf{halt} : \varphi \vee \psi}$. This configuration marks the end of computation.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ (\lambda x{:}\varphi \vee \psi\ .\ M)\ \_ :: k : \varphi \vee \psi}$. We insert the value and obtain a new configuration $(M[\mathbf{inl}^{\varphi\vee\psi}\ v/x],\ k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \langle \_, N\rangle :: k : \varphi \vee \psi}$. We apply the small-step semantics rule and get $(N,\ \langle \mathbf{inl}^{\varphi\vee\psi}\ v, \_\rangle :: k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \langle w, \_\rangle :: k : \varphi \vee \psi}$. We insert the value and get the configuration $(\langle w, \mathbf{inl}^{\varphi\vee\psi}\ v\rangle,\ k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \mathbf{inl}^{(\varphi\vee\psi)\vee\rho}\ \_ :: k : \varphi \vee \psi}$. Inserting the value now yields the configuration $(\mathbf{inl}^{(\varphi\vee\psi)\vee\rho}\ (\mathbf{inl}^{\varphi\vee\psi}\ v),\ k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \mathbf{inr}^{\rho\vee(\varphi\vee\psi)}\ \_ :: k : \varphi \vee \psi}$. Analogous to the previous case we obtain $(\mathbf{inr}^{\rho\vee(\varphi\vee\psi)}\ (\mathbf{inl}^{\varphi\vee\psi}\ v),\ k)$.

$\underline{\mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi,\ \mathbf{vcase}(\_; x.M; y.N) :: k : \varphi \vee \psi}$. This is a pattern match on the value. Following the small-step semantics rule, we generate the configuration $(M[v/x],\ k)$.

Case. If we have a **vcase**-expression, we first need to evaluate the embedded expression. So, the frame on top of a well-typed stack does not influence the course of evaluation. We therefore have one case of well-typed configuration.

> $\underline{\mathbf{vcase}(L; x.M; y.N) : \varphi,\ k : \varphi}$. We simply follow the small-step semantics rule and obtain $(L,\ \mathbf{vcase}(\_; x.M; y.N) :: k)$.

Case. For the **throw**-expressions we need a stack of type $\bot$. Thus, our top-frame cannot be a $\_\ N$, **fst**, **snd** or **vcase**-expression. However, the exact frame on top of the stack is not of great importance, since it has no effect on the computational step.

> $\underline{\mathbf{throw}\ k\ M : \bot,\ k' : \bot}$. This yields the new configuration $(M,\ k)$.

Case. For **letcc** a well-typed configuration consists of a stack which matches the determined type in the **letcc** expression. We therefore have one case for all configurations.

> $\underline{\mathbf{letcc}\ \alpha^{\varphi}\ .\ M : \varphi,\ k : \varphi}$. The small-step rule now generates the configuration $(M[k/\alpha],\ k)$.

All cases now show that for every well-typed configuration there exists a small-step rule to perform. It is thus shown that the operational semantics satisfy the property of progress. □

4.4 Preservation

We would like to prove that all small-step semantics rules preserve the well-typing of a configuration —i.e. if we have a well-typed configuration, the configuration after performing one of the small-step rules will still be well-typed.

Lemma 4.3 (Substitution Lemma Expressions)**.** *If* $\Gamma_{true}, x : \psi; \Gamma_{false} \vdash^{true} M : \varphi$ *and* $\Gamma_{true}; \Gamma_{false} \vdash^{true} v : \psi$*, then* $\Gamma_{true}; \Gamma_{false} \vdash^{true} M[v/x] : \varphi$.

Proof. By induction on the derivation of $\Gamma_{\text{true}}, x : \psi; \Gamma_{\text{false}} \vdash^{\text{true}} M : \varphi$. □

Lemma 4.4 (Substitution Lemma Stacks)**.** *If* $\Gamma_{true}; \Gamma_{false}, \alpha : \psi \vdash^{true} M : \varphi$ *and* $\Gamma_{true}; \Gamma_{false} \vdash^{false} k : \psi$*, then* $\Gamma_{true}; \Gamma_{false} \vdash^{true} M[k/\alpha] : \varphi$.

Proof. By induction on the derivation of $\Gamma_{\text{true}}; \Gamma_{\text{false}}, \alpha : \psi \vdash^{\text{true}} M : \varphi$. □

Theorem 4.5 (Preservation)**.** *If* $\Gamma \vdash^{conf} (M, k) : \varphi$ *and* $(M, k) \rightsquigarrow (M', k')$*, then* $\Gamma \vdash^{conf} (M', k') : \psi$*, for some type* ψ.

Proof. We prove this by case distinction on our small-step rules and induction on the typing derivation of our configurations.

Case ($\mathbf{abort}^{\varphi}\ M,\ k \quad \rightsquigarrow \quad M,\ \mathbf{abort}^{\varphi}\ \_ :: k$). We assume we start with a well-typed configuration.

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{abort}^{\varphi}\ M : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

We derive $\Gamma \vdash^{\text{true}} M : \bot$. After executing the operational step, we thus get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \bot$$

Using the typing rules for stacks we see

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \mathbf{abort}^{\varphi}\ \_ :: k : \bot}\ \bot I$$

This yields

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{abort}^{\varphi}\ \_ :: k : \bot$$

We have thus obtained the new well-typed configuration ($M,\ \mathbf{abort}^{\varphi}\ \_ :: k$), as desired.

Case ($M\ N,\ k \quad \rightsquigarrow \quad M,\ \_\ N :: k$). Assume the first part is a well-typed configuration. We thus have the configuration ($M\ N, k$) as follows.

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M\ N : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

From this we infer $\Gamma \vdash^{\text{true}} M : \psi \to \varphi$ for some ψ and $\Gamma \vdash^{\text{true}} N : \psi$. We now perform the small-step rule and show that the obtained configuration is still well-typed. From our assumption we immediately get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \psi \to \varphi$$

For our stack we follow the typing rules and acquire

$$\frac{\Gamma \vdash^{\text{true}} N : \psi \quad \Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \_\ N :: k : \psi \to \varphi} \to E$$

So

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \_\ N :: k : \psi \to \varphi$$

This leaves us with the new well-typed configuration ($M, \_\ N :: k$).

Case ($\lambda x{:}\varphi\ .\ M,\ \_\ N :: k \quad \rightsquigarrow \quad N,\ (\lambda x{:}\varphi\ .\ M)\ \_ :: k$). Assume we start with the well-typed configuration produced by

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \lambda x{:}\varphi\ .\ M : \varphi \to \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \_\ N :: k : \varphi \to \psi$$

We can infer $\Gamma \vdash^{\text{true}} N : \varphi$ and $\Gamma \vdash^{\text{false}} k : \psi$. We execute the operational step and instantly see

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} N : \varphi$$

For the new stack we infer the type from the typing rule

$$\frac{\Gamma \vdash^{\text{true}} \lambda x{:}\varphi \,.\, M : \varphi \to \psi \quad \Gamma \vdash^{\text{false}} k : \psi}{\Gamma \vdash^{\text{false}} \lambda x{:}\varphi \,.\, M \;\_ :: k : \varphi} \to E$$

We thus get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} (\lambda x{:}\varphi \,.\, M) \;\_ :: k : \varphi$$

This concludes the well-typed configuration $(N,\ (\lambda x{:}\varphi \,.\, M) \;\_ :: k)$.

Case $(v,\ (\lambda x{:}\varphi \,.\, M) \;\_ :: k \quad \rightsquigarrow \quad M[v/x],\ k)$. By assumption, we have the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} v : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} (\lambda x{:}\varphi \,.\, M) \;\_ :: k : \varphi$$

We can deduce we have $\Gamma \vdash^{\text{true}} \lambda x{:}\varphi \,.\, M : \varphi \to \psi$ en $\Gamma \vdash^{\text{false}} k : \psi$. This leads us to the following stack after the small-step rule has been performed.

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \psi$$

Since the value v is of type φ and we have proved the preservation under substitution, we infer

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M[v/x] : \psi$$

This yields the well-typed configuration $(M[v/x],\ k)$ as desired.

Case $(\langle M, N\rangle,\ k \quad \rightsquigarrow \quad M,\ \langle \_, N\rangle :: k)$. We assume the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \langle M, N\rangle : \varphi \wedge \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi \wedge \psi$$

From this we infer $\Gamma \vdash^{\text{true}} M : \varphi$ and $\Gamma \vdash^{\text{true}} N : \psi$. After performing the operational step we thus get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi$$

Working with the typing rules for stacks we attain

$$\frac{\Gamma \vdash^{\text{true}} N : \psi \quad \Gamma \vdash^{\text{false}} k : \varphi \wedge \psi}{\Gamma \vdash^{\text{false}} \langle \_, N\rangle :: k : \varphi} \wedge E$$

So, as desired, we find

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \langle \_, N\rangle :: k : \varphi$$

which concludes the well-typed configuration $(M,\ \langle \_, N\rangle :: k)$.

Case $(v,\ \langle \_, N\rangle :: k \quad \leadsto \quad N,\ \langle v, \_\rangle :: k)$. Assuming the starting configuration is well-typed yields

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} v : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \langle \_, N\rangle :: k : \varphi$$

We derive $\Gamma \vdash^{\text{true}} N : \psi$ for some type ψ, and $\Gamma \vdash^{\text{false}} k : \varphi \wedge \psi$. Then, upon performing the operational step, we immediately obtain

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} N : \psi$$

From the typing rules for stacks it follows that

$$\frac{\Gamma \vdash^{\text{true}} v : \varphi \quad \Gamma \vdash^{\text{false}} k : \varphi \wedge \psi}{\Gamma \vdash^{\text{false}} \langle v, \_\rangle :: k : \psi}\ \wedge E$$

This leads to

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \langle v, \_\rangle :: k : \psi$$

which forms the second part of our new well-typed configuration $(N,\ \langle v, \_\rangle :: k)$.

Case $(w,\ \langle v, \_\rangle :: k \quad \leadsto \quad \langle v, w\rangle,\ k)$. We assume we start with the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} w : \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \langle v, \_\rangle :: k : \psi$$

We infer $\Gamma \vdash^{\text{true}} v : \varphi$ for some φ, and $\Gamma \vdash^{\text{false}} k : \varphi \wedge \psi$. When the small-step rule is executed, the following typing rule is used.

$$\frac{\Gamma \vdash v : \varphi \quad \Gamma \vdash w : \psi}{\Gamma \vdash \langle v, w\rangle : \varphi \wedge \psi}\ \wedge I$$

This yields the, again well-typed, configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \langle v, w\rangle : \varphi \wedge \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi \wedge \psi$$

This is what we wanted to show.

Case $(\mathbf{fst}\ M,\ k \quad \leadsto \quad M,\ \mathbf{fst}\ \_ :: k)$. By assumption, we start with the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{fst}\ M : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

From the first statement we infer $\Gamma \vdash^{\text{true}} M : \varphi \wedge \psi$ for some type ψ. After we have taken the operational step, this directly results in

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi \wedge \psi$$

For the stack, we use the typing rule

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \mathbf{fst}\ \_ :: k : \varphi \wedge \psi}\ \wedge I$$

Hence, we find

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{fst}\ k : \varphi \wedge \psi$$

We now have a new well-typed configuration $(M,\ \mathbf{fst}\ \_ :: k)$.

Case $(\mathbf{snd}\ M,\ k \quad \rightsquigarrow \quad M,\ \mathbf{snd}\ \_ :: k)$. This proof is analogous to the previous case.

Case $(\langle v, w\rangle,\ \mathbf{fst}\ \_ :: k \quad \rightsquigarrow \quad v,\ k)$. We assume the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \langle v, w\rangle : \varphi \wedge \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{fst}\ \_ :: k : \varphi \wedge \psi$$

From these statements we infer $\Gamma \vdash^{\text{true}} v : \varphi$ and $\Gamma \vdash^{\text{true}} w : \psi$ and $\Gamma \vdash^{\text{false}} k : \varphi$. After performing the operational step, we immediately find the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} v : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

as wanted.

Case $(\langle v, w\rangle,\ \mathbf{snd}\ \_ :: k \quad \rightsquigarrow \quad w,\ k)$. This proof is analogous to the previous case.

Case $(\mathbf{inl}^{\varphi\vee\psi}\ M,\ k \quad \rightsquigarrow \quad M,\ \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k)$. Assuming we start with a well-typed configuration, we find

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{inl}^{\varphi\vee\psi}\ M : \varphi \vee \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi \vee \psi$$

We infer $\Gamma \vdash^{\text{true}} M : \varphi$. For the configuration after applying the small-step rule we thus get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi$$

From the typing rules for stacks, we derive

$$\frac{\Gamma \vdash^{\text{false}} k : \varphi \vee \psi}{\Gamma \vdash^{\text{false}} \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k : \varphi}\ \vee E$$

This yields

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k : \varphi$$

We have thus again obtained a well-typed configuration.

Case $(\mathbf{inr}^{\varphi\vee\psi}\ M,\ k \quad\leadsto\quad M,\ \mathbf{inr}^{\varphi\vee\psi}\ \_ :: k)$. This proof is analogous to the previous case.

Case $(v,\ \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k \quad\leadsto\quad \mathbf{inl}^{\varphi\vee\psi}\ v,\ k)$. The well-typed configuration we assume, is

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} v : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{inl}^{\varphi\vee\psi}\ \_ :: k : \varphi$$

We derive $\Gamma \vdash^{\text{false}} k : \varphi \vee \psi$. This allows us to immediately infer the following, after performing operational step.

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi \vee \psi$$

Using the typing rules for expressions, we see

$$\frac{\Gamma \vdash^{\text{true}} v : \varphi}{\Gamma \vdash^{\text{true}} \mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi}\ \vee I$$

We now get

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi$$

This completes the well-typed configuration $(\mathbf{inl}^{\varphi\vee\psi}\ v,\ k)$.

Case $(\mathbf{vcase}(L; x.M; y.N),\ k \quad\leadsto\quad L,\ \mathbf{vcase}(\_; x.M; y.N) :: k)$. By assumption, we obtain the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{vcase}(L; x.M; y.N) : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

From the first statement we derive, for some type ψ and χ, that $\Gamma \vdash^{\text{true}} L : \psi \vee \chi$ and $\Gamma, x : \psi \vdash^{\text{true}} M : \varphi$ and $\Gamma, y : \chi \vdash^{\text{true}} N : \varphi$. We perform the small-step rule and see

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} L : \psi \vee \chi$$

Regarding the stack, we apply the typing rule

$$\frac{\Gamma, x : \psi \vdash^{\text{true}} M : \varphi \quad \Gamma, y : \chi \vdash^{\text{true}} N : \varphi \quad \Gamma \vdash^{\text{false}} k : \varphi}{\Gamma \vdash^{\text{false}} \mathbf{vcase}(\_; x.M; y.N) :: k : \psi \vee \chi}\ \vee I$$

We thus find

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{vcase}(\_; x.M; y.N) :: k : \psi \vee \chi$$

We now again have a well-typed configuration $(L,\ \mathbf{vcase}(\_; x.M; y.N) :: k)$.

Case $(\mathbf{inl}^{\varphi\vee\psi}\ v,\ \mathbf{vcase}(\_; x.M; y.N) :: k \quad\leadsto\quad M[v/x],\ k)$. We assume we start with the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{inl}^{\varphi\vee\psi}\ v : \varphi \vee \psi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} \mathbf{vcase}(\_; x.M; y.N) :: k : \varphi \vee \psi$$

From the first statement we infer $\Gamma \vdash^{\text{true}} v : \varphi$. From the second, we derive $\Gamma \vdash^{\text{false}} k : \chi$, for some type χ, and $\Gamma, x : \varphi \vdash^{\text{true}} M : \chi$ and $\Gamma, y : \psi \vdash^{\text{true}} N : \chi$. Performing the operational step, thus immediately yields

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \chi$$

And by the substitution lemma, we find

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M[v/x] : \chi$$

This gives us a new well-typed configuration as desired.

Case ($\mathbf{inr}^{\varphi \vee \psi}\ v,\ \mathbf{vcase}(\_; x.M; y.N) :: k \quad \rightsquigarrow \quad N[v/y],\ k$). This proof is analogous to the previous case.

Case ($\mathbf{throw}\ k'\ M,\ k \quad \rightsquigarrow \quad M,\ k'$). By assumption, we have the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{throw}\ k'\ M : \bot$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \bot$$

This yields $\Gamma \vdash^{\text{true}} M : \varphi$ and $\Gamma \vdash^{\text{false}} k' : \varphi$, for some type φ. We now instantly obtain the new configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k' : \varphi$$

This again is well-typed.

Case ($\mathbf{letcc}\ \alpha^{\varphi}\ .\ M,\ k \quad \rightsquigarrow \quad M[k/\alpha],\ k$). We assume the well-typed configuration

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{letcc}\ \alpha^{\varphi}\ .\ M : \varphi$$
$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$$

From this we derive $\Gamma_{\text{true}}; \Gamma_{\text{false}}, \alpha : \varphi, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi$. We apply the small-step semantics rule. From the substitution lemma we obtain

$$\Gamma_{\text{true}}; \Gamma_{\text{false}}, \mathbf{halt} : \rho \vdash^{\text{true}} M[k/\alpha] : \varphi$$

The stack remains unchanged, so we find a new well-typed configuration.

□

4.5 Termination

We would now like to prove the property of termination. It states that every well-typed configuration terminates after a finite amount of reduction steps. Having this property ensures that there do not exist well-typed programs that run forever.

Unlike the previous proofs, we cannot prove termination by induction over the structure of our programs. We need a stronger induction hypothesis, which we obtain from *logical relations* [25]. The definitions of these predicates and the proof of the Fundamental Theorem are based on [3].

Definition 4.1 (Reduction Relation)**.** Suppose we are given a configuration (M, k). We write $(M, k) \rightsquigarrow (M', k')$ to indicate that (M, k) reduces to (M', k') in a single step of the small-step semantics. We define $\rightsquigarrow^*$ as:

$$\begin{aligned}(M, k) \rightsquigarrow^* (M', k') := {} & (M', k') \text{ is terminal and } (M, k) = (M', k') \\ & \text{or there exists an } (M'', k'') \text{ such that } (M, k) \rightsquigarrow (M'', k'') \\ & \quad \text{and } (M'', k'') \rightsquigarrow^* (M', k').\end{aligned}$$

That is, $(M, k) \rightsquigarrow^* (M', k')$ says that (M, k) reduces to (M', k') in a finite number of small-step transitions.

Definition 4.2 (Logical Predicate)**.** We introduce a predicate R_φ on well-typed values $. \,;\, \mathbf{halt} : \rho \vdash^{\text{true}} v : \varphi$. We define R_φ by induction on the type φ.

$$\begin{aligned}R_{var}(v) &= true \\ R_\top(v) &= true \\ R_\bot(v) &= false \\ R_{\varphi\to\psi}(v) &= \text{for all } w : \varphi \text{ such that } R_\varphi(w), \text{ we have } P_\psi(v\ w) \\ R_{\varphi\wedge\psi}(v) &= R_\varphi(\mathbf{fst}\ v) \text{ and } R_\psi(\mathbf{snd}\ v) \\ R_{\varphi\vee\psi}(v) &= v \text{ is } \mathbf{inl}^{\varphi\vee\psi}\ u \text{ and } R_\varphi(u), \text{ or} \\ &\qquad v \text{ is } \mathbf{inr}^{\varphi\vee\psi}\ u \text{ and } R_\psi(u)\end{aligned}$$

We define a predicate P_φ on well-typed expressions $. \,;\, \mathbf{halt} : \rho \vdash^{\text{true}} M : \varphi$.

$$P_\varphi(M) = \text{for all } k : \varphi \text{ such that } Q_\varphi(k), \text{ we have } \mathcal{N}_\rho(M, k)$$

We define a predicate Q_φ on well-typed continuations $. \,;\, \mathbf{halt} : \rho \vdash^{\text{false}} k : \varphi$.

$$Q_\varphi(k) = \text{for all } v : \varphi \text{ such that } R_\varphi(v), \text{ we have } \mathcal{N}_\rho(v, k)$$

Finally, we define $\mathcal{N}_\rho$ on well-typed configurations $. \,;\, \mathbf{halt} : \rho \vdash^{\text{conf}} (M, k) : \varphi$.

$$\mathcal{N}_\rho(M, k) = \text{ there exists a } v : \rho \text{ such that } (M, k) \rightsquigarrow^* (v, \mathbf{halt})$$

Lemma 4.6 (Termination is preserved by reduction)**.** *Suppose that*

$$. \,;\, \boldsymbol{halt} : \rho \vdash^{conf} (M, k) : \varphi$$

and $(M, k) \rightsquigarrow (M', k')$ where (M', k') is a configuration of some type ψ. Then $\mathcal{N}_\rho(M, k)$ if and only if $\mathcal{N}_\rho(M', k')$.

Proof. We first consider the direction in which $\mathcal{N}_\rho(M,k)$ implies $\mathcal{N}_\rho(M',k')$. We assume $\mathcal{N}_\rho(M,k)$. By definition of $\mathcal{N}_\rho$, this means that $(M,k) \rightsquigarrow^* (v, \mathbf{halt})$ for some value v. In addition we know $(M,k) \rightsquigarrow (M',k')$. Because of the property of determinism, which has already been proved, we infer that this has to be the first step of the reduction. We thus have $(M,k) \rightsquigarrow (M',k') \rightsquigarrow^* (v, \mathbf{halt})$. We now see $(M',k') \rightsquigarrow^* (v, \mathbf{halt})$, so $\mathcal{N}_\rho(M',k')$.

We now consider the other direction. We assume $\mathcal{N}_\rho(M',k')$. By definition of $\mathcal{N}_\rho$, this means that $(M',k') \rightsquigarrow^* (v, \mathbf{halt})$ for some value v. However, we also have $(M,k) \rightsquigarrow (M',k')$, which implies $(M,k) \rightsquigarrow^* (v, \mathbf{halt})$. This proves that $\mathcal{N}_\rho(M,k)$. □

Lemma 4.7 (Fundamental Lemma)**.** *Suppose that*

$$\begin{aligned} . \,;\boldsymbol{halt} : \rho &\vdash^{true} v_i : \varphi_i \qquad ,1 \leq i \leq n \\ . \,;\boldsymbol{halt} : \rho &\vdash^{false} k_j : \psi_j \qquad ,1 \leq j \leq m \end{aligned}$$

and

$$\begin{aligned} &R_{\varphi_1}(v_1), ..., R_{\varphi_n}(v_n) \\ &Q_{\psi_1}(k_1), ..., Q_{\psi_m}(k_m) \end{aligned}$$

Now suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \boldsymbol{halt} : \rho \vdash^{true} M : \chi$$

Then, for all $. \,;\boldsymbol{halt} : \rho \vdash^{false} l : \chi$ *with* $Q_\chi(l)$*, it follows that*

$$\mathcal{N}_\rho(M[v_1/x_1, ..., v_n/x_n, k_1/\alpha_1, ..., k_m/\alpha_m], l)$$

Proof. We prove this by induction on the typing derivation of

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi$$

We will abbreviate the substitution $[v_1/x_1, ..., v_n/x_n, k_1/\alpha_1, ..., k_m/\alpha_m]$ as $[...]$. Also, to enhance readability, we will often use Lemma 4.6 implicitly.

Case (Ax). Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} y : \chi$$

By assumption, y has to be one of the variables x_i and so $\chi = \varphi_i$. Then, $y[...] = v_i$ with $R_{\varphi_i}(v_i)$. Now for all continuations $l : \varphi_i$ with $Q_{\varphi_i}(l)$, we find by definition of Q that $\mathcal{N}_\rho(v_i, l)$. Hence, $\mathcal{N}_\rho(y[...], l)$.

Case $(\top I)$. Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{taut} : \top$$

We know $\mathbf{taut}[...] = \mathbf{taut}$. By definition, we have $R_\top(\mathbf{taut})$. Now for all continuations $l : \top$ with $Q_\top(l)$, we find by definition of Q that $\mathcal{N}_\rho(\mathbf{taut}, l)$, as desired.

Case ($\bot E$). Suppose we have

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{abort}^{\chi}\ M : \chi$$

We deduce

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \bot$$

We want to show that for all continuations $l : \chi$ with $Q_\chi(l)$, we get

$$\mathcal{N}_\rho((\mathbf{abort}^{\chi}\ M)[...], l)$$

First, note that $(\mathbf{abort}^{\chi}\ M)[...] = \mathbf{abort}^{\chi}\ M[...]$. Moreover, we have the reduction $(\mathbf{abort}^{\chi}\ M[...], l) \rightsquigarrow (M[...], \mathbf{abort}^{\chi}\ \_ :: l)$. Now, by the induction hypothesis, we find $\mathcal{N}_\rho(M[...], \mathbf{abort}^{\chi}\ \_ :: l)$ provided that $\mathbf{abort}^{\chi}\ \_ :: l$ is well-typed and satisfies $Q_\bot(\mathbf{abort}^{\chi}\ \_ :: l)$. The former is clear from the typing of continuations. For the latter, we have to show that for all values $v : \bot$ with $R_\bot(v)$, we get $\mathcal{N}_\rho(v, \mathbf{abort}^{\chi}\ \_ :: l)$. However, by definition of $R_\bot$, there do not exist such values. Therefore, $Q_\bot(\mathbf{abort}^{\chi}\ \_ :: l)$ vacuously holds.

Case ($\to I$). Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \lambda y{:}\chi'\ .\ M : \chi' \to \chi$$

We derive

$$x_1 : \varphi_1, ..., x_n : \varphi_n, y : \chi'; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi$$

We want to prove that for all continuations $l : \chi' \to \chi$ with $Q_{\chi' \to \chi}(l)$, we get

$$\mathcal{N}_\rho((\lambda y{:}\chi'\ .\ M)[...], l)$$

We first remark that $(\lambda y{:}\chi'\ .\ M)[...] = \lambda y{:}\chi'\ .\ M[...]$. Now, using the definition of Q, the desired $\mathcal{N}_\rho(\lambda y{:}\chi'\ .\ M[...], l)$ follows if we prove that $R_{\chi' \to \chi}(\lambda y{:}\chi'\ .\ M[...])$. Let us assume $w : \chi'$ with $R_{\chi'}(w)$. We will show $P_\chi((\lambda y{:}\chi'\ .\ M[...])\ w)$. To do so, we assume we have a context $l' : \chi$ with $Q_\chi(l')$ and show $\mathcal{N}_\rho((\lambda y{:}\chi'\ .\ M[...])\ w, l')$. By our semantics, $(\lambda y{:}\chi'\ .\ M[...])\ w, l') \rightsquigarrow^* (M[...][w/y], l')$. From the induction hypothesis, it follows that $\mathcal{N}_\rho(M[...][w/y], l')$, and so we also have $\mathcal{N}_\rho((\lambda y{:}\chi'\ .\ M)[...], l)$ as desired.

Case ($\to E$). Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M\ N : \chi$$

We then derive for some type χ'

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi' \to \chi$$
$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} N : \chi'$$

We would now like to prove that for all continuations $l : \chi$ with $Q_\chi(l)$, we have

$$\mathcal{N}_\rho((M\ N)[...], l)$$

First, note $(M\ N)[...] = M[...]\ N[...]$. We also have $(M[...]\ N[...], l) \rightsquigarrow (M[...], \_\ N[...] :: l)$. Now $\mathcal{N}_\rho(M[...], \_\ N[...] :: l)$ follows from the induction hypothesis provided that $\_\ N[...] :: l$ is a well-typed configuration with $Q_{\chi' \to \chi}(\_\ N[...] :: l)$. The former is obvious. To show the latter, consider some $w : \chi' \to \chi$ with $R_{\chi' \to \chi}(w)$. We now have to show $\mathcal{N}_\rho(w, \_\ N[...] :: l)$. We know that $(w, \_\ N[...] :: l) \rightsquigarrow (N[...], w\ \_ :: l)$. We get $\mathcal{N}_\rho(N[...], w\ \_ :: l)$ by the induction hypothesis if we show that $w\ \_ :: l$ is well-typed such that $Q_{\chi'}(w\ \_ :: l)$. Again, the former is obvious. To show the latter, now assume a $u : \chi'$ with $R_{\chi'}(u)$. We show $\mathcal{N}_\rho(u, w\ \_ :: l)$. Since $(u, w\ \_ :: l) \rightsquigarrow (w\ u, l)$, we should show that $\mathcal{N}_\rho(w\ u, l)$. But this follows from the definitions of R and P by our assumptions $R_{\chi' \to \chi}(w)$, $R_{\chi'}(u)$ and $Q_\chi(l)$.

Case $(\wedge I)$. Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\mathrm{true}} \langle M, N\rangle : \chi \wedge \chi'$$

We derive

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\mathrm{true}} M : \chi$$
$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\mathrm{true}} N : \chi'$$

We will show that for all continuations $l : \chi \wedge \chi'$ with $Q_{\chi \wedge \chi'}(l)$, we have

$$\mathcal{N}_\rho(\langle M, N\rangle[...], l)$$

We have $\langle M, N\rangle[...] = \langle M[...], N[...]\rangle$ and $(\langle M[...], N[...]\rangle, l) \rightsquigarrow (M[...], \langle \_, N[...]\rangle :: l)$. From the induction hypothesis it follows that $\mathcal{N}_\rho(M[...], \langle \_, N[...]\rangle :: l)$ if $\langle \_, N[...]\rangle :: l$ is a well-typed continuation with $Q_\chi(\langle \_, N[...]\rangle :: l)$. The well-typing is clear, so we show that $Q_\chi(\langle \_, N[...]\rangle :: l)$. Consider a $w : \chi$ with $R_\chi(w)$. We then want to show that $\mathcal{N}_\rho(w, \langle \_, N[...]\rangle :: l)$. Now, $(w, \langle \_, N[...]\rangle :: l) \rightsquigarrow (N[...], \langle w, \_\rangle :: l)$. From the induction hypothesis it follows that $\mathcal{N}_\rho(N[...], \langle w, \_\rangle :: l)$ if $\langle w, \_\rangle :: l$ is a well-typed configuration with $Q_{\chi'}(\langle w, \_\rangle :: l)$. The former is clear. To prove the latter, assume a $u : \chi'$ with $R_{\chi'}(u)$. We have to show $\mathcal{N}_\rho(u, \langle w, \_\rangle :: l)$. We know $(u, \langle w, \_\rangle :: l) \rightsquigarrow (\langle w, u\rangle, l)$. From the definition of $R_{\chi \wedge \chi'}$ and the assumptions that $R_\chi(w)$ and $R_{\chi'}(u)$, it follows that $R_{\chi \wedge \chi'}(\langle w, u\rangle)$. Now, from the definition of Q and the assumption $Q_{\chi \wedge \chi'}(l)$ we conclude $\mathcal{N}_\rho(\langle w, u\rangle, l)$.

Case $(\wedge E)$. *We will consider the **fst**-case. The **snd**-case is analogous.*
Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\mathrm{true}} \mathbf{fst}\ M : \chi$$

We derive for some type χ'

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\mathrm{true}} M : \chi \wedge \chi'$$

We will prove that for all continuations $l : \chi$ with $Q_\chi(l)$, we have

$$\mathcal{N}_\rho((\mathbf{fst}\ M)[...], l)$$

We know $(\mathbf{fst}\ M)[...] = \mathbf{fst}\ M[...]$ and $(\mathbf{fst}\ M[...], l) \rightsquigarrow (M[...], \mathbf{fst}\ \_ :: l)$. If $\mathbf{fst}\ \_ :: l$ is a well-typed stack with $Q_{\chi\wedge\chi'}(\mathbf{fst}\ \_ :: l)$, then $\mathcal{N}_\rho(M[...], \mathbf{fst}\ \_ :: l)$ from the induction hypothesis. It is obvious that $\mathbf{fst}\ \_ :: l$ is well-typed, so we show $Q_{\chi\wedge\chi'}(\mathbf{fst}\ \_ :: l)$. Assume we have $w : \chi \wedge \chi'$ with $R_{\chi\wedge\chi'}(w)$. We have to show that $\mathcal{N}_\rho(w, \mathbf{fst}\ \_ :: l)$. It is known that $(w, \mathbf{fst}\ \_ :: l) \rightsquigarrow (\mathbf{fst}\ w, l)$. From the definition of R and our assumption w it follows that $R_\chi(\mathbf{fst}\ w)$. Then, from the definition of $Q_\chi(l)$, we conclude $\mathcal{N}_\rho(\mathbf{fst}\ w, l)$.

Case $(\vee I)$. *We will consider the* ***inl****-case. The* ***inr****-case is analogous.*
Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{inl}^{\chi\vee\chi'}\ M : \chi \vee \chi'$$

We derive

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi$$

We have to show that for all continuations $l : \chi \vee \chi'$ with $Q_{\chi\vee\chi'}(l)$, we have

$$\mathcal{N}_\rho((\mathbf{inl}^{\chi\vee\chi'}\ M)[...], l)$$

We know $(\mathbf{inl}^{\chi\vee\chi'}\ M)[...] = \mathbf{inl}^{\chi\vee\chi'}\ M[...]$ and $(\mathbf{inl}^{\chi\vee\chi'}\ M[...], l) \rightsquigarrow (M[...], \mathbf{inl}^{\chi\vee\chi'}\ \_ :: l)$. By the induction hypothesis, $\mathcal{N}_\rho(M[...], \mathbf{inl}^{\chi\vee\chi'}\ \_ :: l)$ holds, provided that $\mathbf{inl}^{\chi\vee\chi'}\ \_ :: l$ is a well-typed continuation with $Q_\chi(\mathbf{inl}^{\chi\vee\chi'}\ \_ :: l)$. Since the former is clear, we will now show the latter. Consider some $w : \chi$ with $R_\chi(w)$. We will show that $\mathcal{N}_\rho(w, \mathbf{inl}^{\chi\vee\chi'}\ \_ :: l)$. To do so, notice that $(w, \mathbf{inl}^{\chi\vee\chi'}\ \_ :: l) \rightsquigarrow (\mathbf{inl}^{\chi\vee\chi'}\ w, l)$. Now, by definition of R, we see $R_{\chi\vee\chi'}(\mathbf{inl}^{\chi\vee\chi'}\ w)$. From $Q_{\chi\vee\chi'}(l)$ it now follows that $\mathcal{N}_\rho(\mathbf{inl}^{\chi\vee\chi'}\ w, l)$.

Case $(\vee E)$. Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{vcase}(L; x.M; y.N) : \sigma$$

We derive

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} L : \chi \vee \chi'$$
$$x_1 : \varphi_1, ..., x_n : \varphi_n, x : \chi; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \sigma$$
$$x_1 : \varphi_1, ..., x_n : \varphi_n, y : \chi'; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} N : \sigma$$

We will prove that for all continuations $l : \sigma$ with $Q_\sigma(l)$, we have

$$\mathcal{N}_\rho((\mathbf{vcase}(L; x.M; y.N))[...], l)$$

We see $(\mathbf{vcase}(L; x.M; y.N))[...] = \mathbf{vcase}(L[...]; x.M[...]; y.N[...])$ and $(\mathbf{vcase}(L[...]; x.M[...]; y.N[...]), l) \rightsquigarrow (L[...], \mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l)$. We

will show that $\mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l$ is well-typed and satisfies $Q_{\chi\vee\chi'}(\mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l)$. From the induction hypothesis it then follows that $\mathcal{N}_\rho(L[...], \mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l)$. Again, the well-typing is clear. To prove $Q_{\chi\vee\chi'}(\mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l)$, assume some $w : \chi \vee \chi'$ with $R_{\chi\vee\chi'}(w)$. We will have to show $\mathcal{N}_\rho(w, \mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l)$. We know that $(w, \mathbf{vcase}(\_; x.M[...]; y.N[...]) :: l) \rightsquigarrow (\mathbf{vcase}(w; x.M[...]; y.N[...]), l)$. From $R_{\chi\vee\chi'}(w)$ we derive two cases. If w is $\mathbf{inl}^{\chi\vee\chi'}\ u$ with $R_\chi(u)$, then $(\mathbf{vcase}(w; x.M[...]; y.N[...]), l) \rightsquigarrow (M[...][u/x], l)$. From the induction hypothesis and the fact that $Q_\sigma(l)$, it then follows that $\mathcal{N}_\rho(M[...][u/x], l)$. If, on the other hand, w is $\mathbf{inr}^{\chi\vee\chi'}\ u$ with $R_{\chi'}(u)$, then $(\mathbf{vcase}(w; x.M[...]; y.N[...]), l) \rightsquigarrow (N[...][u/y], l)$. Again from the induction hypothesis and $Q_\sigma(l)$, it follows that $\mathcal{N}_\rho(N[...][u/y], l)$.

Case (**throw**). Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{throw}\ k\ M : \bot$$

We derive for some type χ

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi$$
$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{false}} k : \chi$$

We want to prove that for all continuations $l : \bot$ with $Q_\bot(l)$, we have

$$\mathcal{N}_\rho((\mathbf{throw}\ k\ M)[...], l)$$

We can write $(\mathbf{throw}\ k\ M)[...]$ as $\mathbf{throw}\ k[...]\ M[...]$. We know that k has to be a continuation variable, so $k[...] = k_i$ with $\chi = \psi_i$ and $Q_{\psi_i}(k_i)$. Moreover we know $(\mathbf{throw}\ k_i\ M[...], l) \rightsquigarrow (M[...], k_i)$. Now, by the induction hypothesis we have $\mathcal{N}_\rho(M[...], l')$ for all $l' : \psi_i$ with $Q_{\psi_i}(l')$. Hence, $\mathcal{N}_\rho(M[...], k[...])$.

Case (**letcc**). Suppose

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \mathbf{halt} : \rho \vdash^{\text{true}} \mathbf{letcc}\ \beta^\chi\ .\ M : \chi$$

From this, it follows that

$$x_1 : \varphi_1, ..., x_n : \varphi_n; \alpha_1 : \psi_1, ..., \alpha_m : \psi_m, \beta : \chi, \mathbf{halt} : \rho \vdash^{\text{true}} M : \chi$$

We prove that for all continuations $l : \chi$ with $Q_\chi(l)$, we get

$$\mathcal{N}_\rho((\mathbf{letcc}\ \beta^\chi\ .\ M)[...], l)$$

From the induction hypothesis it follows that for all $l' : \chi$ with $Q_\chi(l')$ and $l'' : \chi$ with $Q_\chi(l'')$, we obtain $\mathcal{N}_\rho(M[...][l'/\beta], l'')$. So in particular, we get $\mathcal{N}_\rho(M[...][l'/\beta], l')$. Also, for our configuration, we have $(\mathbf{letcc}\ \beta^\chi\ .\ M)[...] = \mathbf{letcc}\ \beta^\chi\ .\ M[...]$ and $(\mathbf{letcc}\ \beta^\chi\ .\ M[...], l) \rightsquigarrow (M[...][l/\beta], l)$. Now, notice that we can let l equal l', which gives us the desired result.

□

Corollary 4.8. *For every well-typed initial configuration $(M, \boldsymbol{halt})$ of type φ the statement $\mathcal{N}_\varphi(M, \boldsymbol{halt})$ holds. Hence, every well-typed configuration terminates.*

Chapter 5

Discussion

In this thesis we have studied the Curry-Howard isomorphism with respect to both intuitionistic and classical logic. We have presented a typed language and provided an operational semantics with the proven properties of determinism, progress, preservation and termination.

The main observations of the Curry-Howard correspondence are presented in Figure 5.1. We will review the most interesting conclusions here.

First of all, we have seen that the correspondence gives us a dynamic interpretation for logical proofs. The 'running' of a program —or more formally, the process of reducing a term— has been shown to correspond to normalizing a proof. This has substantiated a way of thinking about proofs as dynamic entities, as opposed to merely static objects.

Secondly, the expansion of the isomorphism into classical logic has resulted in a correspondence between classical axioms and control operators. We have seen that each axiom results in a different control operator. We have opted to define classical logic by adding Peirce's law to intuitionistic logic. This axiom was selected because of its clear computational interpretation, which the law of excluded middle (yet) lacks. Furthermore, Peirce's law does not imply *ex falso quodlibet*. Choosing double negation elimination, which does imply EFQ, as our classical axiom and obtaining the corresponding control operator, would have made the **abort**-expression superfluous. With Peirce's law this is not the case.

Classical proofs have the computational effect of changing the evaluation of a program; it adds non-local control flow. While continuations should be handled with caution, they can be used for constructing often wanted features such as exception handling.

The observations discussed in this thesis have all been made before. However, by presenting a simple language and consistently using that language to discuss various principles, we hope to contribute to the understanding of the Curry-Howard isomorphism with respect to intuitionistic and classical logic. We believe the insights the correspondence provides, are interesting and worth studying, both from a logical and from a computational point of view.

Logic	**Computation**
Formulae	Types
Proofs	Programs
Implication	Function type
Conjunction	Product type
Disjunction	Sum type
Classical axiom	Control operator
Proof checking	Type checking
Formula inference	Type inference
Provability	Type inhabitation
Normalization	Reduction
Normal proof	Value
Double negation translation	CPS transformation

Figure 5.1: Summary Curry-Howard isomorphism

Further research

While we have attempted to create an overview that is as comprehensive as possible, many more interesting aspects remain to be discussed. We mention some topics here which would be interesting to include in a more elaborate examination.

Firstly, there exist other classical axioms that might have an interesting computational interpretation. We have examined the three most commonly used axioms, but there are many others, such as the stronger law of excluded middle —$\varphi \vee \varphi \to \psi$— and the classical deMorgan law —$\neg(\varphi \wedge \psi) \to (\neg\varphi \vee \neg\psi)$.

Secondly, we have briefly mentioned other research that has introduced a notion of judgements and a corresponding typing to better deal with classical logic [2, 20]. To keep our presentation as uniform as possible, we have chosen not to adjust our language and typing accordingly. However, changing the language as proposed might provide a more intuitive reading of the computational content of classical logic.

The discussed $\lambda_{\mathcal{C}}$-calculus was the first model of computation which was shown to correspond to classical logic. However, clearer calculi have been introduced since, such as the $\lambda\mu$-calculus [24] and variations of the $\lambda_{\mathcal{C}}$- and $\lambda\mu$-calculus in [2]. An examination of those calculi and a correspondence with our constructed language would be interesting to include.

The presented language can be interpreted as a very simple programming language. It does not, however, contain more elaborate data-types or primitive recursion. Such constructs could be incorporated, following [18].

Finally, this thesis only deals with propositional logic. However, the isomorphism can be extended into higher-order logic and provide interesting computational interpretations [27].

Bibliography

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

[2] Z. M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *Higher-order and symbolic computation*, 20(4): 403–429, 2007.

[3] M. Biernacka and D. Biernacki. A context-based approach to proving termination of evaluation. *Electronic Notes in Theoretical Computer Science*, 249:169–192, 2009.

[4] K. Bimbó. Combinatory logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition, 2018.

[5] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

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

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

[8] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical structures in computer science*, 2(4):361–391, 1992.

[9] B. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1991.

[10] M. Felleisen and D. P. Friedman. *Control Operators, the SECD-machine, and the lambda-calculus*. Indiana University, Computer Science Department, 1986.

[11] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical computer science*, 52(3):205–237, 1987.

[12] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1): 3–25, 2009.

[13] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

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

[15] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

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

[17] L. Keiff. Dialogical logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2011 edition, 2011.

[18] R. Krebbers. *Classical logic, control calculi and data types*. PhD thesis, Master's thesis, Radboud Universiteit Nijmegen, 2010.

[19] P. B. Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer Science & Business Media, 2012.

[20] D. Licata and F. Pfenning. Computational meaning of classical logic [lecture notes], September 2008.

[21] L. Moreau. *Sound evaluation of parallel functional programs with first-class continuations*. PhD thesis, Université de Liège, Faculté des sciences appliqúées, 1995.

[22] J. Moschovakis. Intuitionistic logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2018 edition, 2018.

[23] C. R. Murthy. An evaluation semantics for classical proofs. Technical report, Cornell University, 1991.

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

[25] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[26] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. 2002.

[27] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[28] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP): 1–31, 2019.