Normalization for the simply typed \(\lambda\)-calculus

A trilogy

Thesis MSc Computing Science

Author: Bálint Kocsis

Supervisor: Niels van der Weide

Second reader: Herman Geuvers

March 1, 2024
# Contents

1 Introduction 3  
1.1 Related work 4  
1.2 Overview 4  
1.3 Acknowledgements 5  

2 Syntax and semantics of the simply typed \( \lambda \)-calculus 6  
2.1 Syntax 6  
2.1.1 Types and terms 6  
2.1.2 Substitution 7  
2.1.3 Conversion 8  
2.1.4 Normal forms 9  
2.2 Semantics 10  
2.2.1 Applicative structures, models 10  
2.2.2 Basic model theory 13  
2.2.3 Homomorphisms 16  

3 Normalization for the simply typed \( \lambda \)-calculus 18  
3.1 Weak normalization 19  
3.1.1 “Syntactic” proof 20  
3.1.2 Logical predicates 21  
3.1.3 “Semantic” proof 22  
3.2 Normalization by evaluation 22  
3.2.1 Adding contexts 23  
3.2.2 Kripke models 24  
3.2.3 The algorithm 27  

4 Categorical preliminaries 29  
4.1 Universal properties 29  
4.2 Cartesian closed categories and functors 32  
4.3 Presheaves and representables 35  
4.4 Comma categories 36  

5 Categorical semantics for the simply typed \( \lambda \)-calculus 41  
5.1 Simply typed \( \lambda \)-calculus à la de Bruijn with explicit substitutions 41  
5.2 Semantics 43  
5.2.1 Simply typed categories with families 43  
5.2.2 Abstract syntax and models 49  
5.2.3 Semantics in cartesian closed categories 52  

6 Gluing 54  
6.1 Neutral and normal forms via presheaves 54
Chapter 1

Introduction

There are many different kinds of programming languages. One of the most basic ones is the (pure) simply typed λ-calculus, the basis for all typed functional languages. The simply typed λ-calculus sits in the intersection of logic and programming: via the Curry-Howard isomorphism, its type system corresponds to minimalistic logic, that is, logic with implication as the only connective.

One of the main aspects of programming language theory is the study of the properties of programming languages as formal systems. This branch is also called the metatheory of programming languages. There are several metatheoretic properties one can consider, such as decidable type checking, (strong) normalization, canonicity, and confluence. In some sense, the validity of these properties all contribute to the well-behavedness of a particular programming language.

In this thesis, we are only concerned with the normalization property. Traditionally, normalization has been defined in the context of term rewriting. One defines a collection of rewrite rules (also called reduction rules) on the terms of a particular programming language, expressing how to simplify programs. In this setting, normalization refers to the process of applying the rewrite rules repeatedly until no more reductions are possible. A program in which no reductions are possible is called a normal form.

One proof of normalization for the simply typed λ-calculus is due to Tait [36]. Historically, this proof is important as it crucially relies on the type structure of the simply typed λ-calculus, instead of only considering the shapes of terms and their behavior under reduction. His method uses a so called logical predicate, which is a family of predicates on λ-terms indexed by types and defined by induction on types.

Another important proof of normalization is due to Berger and Schwichtenberg [9]. They construct a normalization function that computes the normal form of a given input term. An interesting aspect of their proof is that it is reduction-free: it does not mention any notion of reduction and does not depend on term rewriting. Instead, it uses denotational semantics, that is, evaluation into a suitable model. This is why their method may be termed normalization by evaluation. Berger and Schwichtenberg’s algorithm, although somewhat technical, is easy to implement and provides an efficient solution to normalization.

There have been multiple accounts of a categorical reconstruction of normalization by evaluation. The first one is due to Altenkirch, Hofmann, and Streicher [2]. They employ a so-called twisted gluing category from which both the normalization function and its correctness can be deduced. They note that normalization by evaluation is related to the categorical Artin-gluing construction [37] without making the relation explicit. The connection to categorical gluing has been made precise by Fiore [17,18].

In this thesis, we present three normalization proofs for the simply typed λ-calculus.
The first one is based on Tait’s idea of a convertible term. The second one is a version of normalization by evaluation, similar to Reynolds’ work [32]. The third one is a categorical proof based on the work of Fiore [17, 18] and Sterling and Spitters [34].

Besides presenting the proofs, we provide a comparison of the structure of the proofs. Comparing the proofs adds to the understanding of the individual proofs themselves on their own. The main message of the thesis is that there are close analogies between the structure of all three proofs.

1.1 Related work

Normalization by evaluation was invented by Berger and Schwichtenberg [9], who called their method ‘inverting the evaluation functional’. However, their coding mechanism for generating fresh variables is rather elaborate. A simpler numbering scheme was used in [7] and [16].

Berger [7] showed that a version of normalization by evaluation can be extracted from a constructive proof of strong normalization, similar to the proof of weak normalization in this thesis. He used Kreisel’s modified realizability interpretation for intuitionistic logic as the program extraction method. Later, he and others [8] formalized these results in various proof assistants.

Dybjer and Filinski [16] used normalization by evaluation for simply typed \( \lambda \)-calculus as the basis for type-directed partial evaluation.

As mentioned already, there have been several attempts at reconstructing Berger and Schwichtenberg’s algorithm using the language of category theory, such as [2] and [14]. Streicher’s short note [35] summarizes the situation.

The method of normalization by evaluation has been extended to more complex systems. Altenkirch, Hofmann, and Streicher presented reduction-free categorical normalization proofs for a combinator version of System F [3] and then System F itself [4]. Altenkirch et al. [1] considered simply typed \( \lambda \)-calculus with strong binary sums (categorical coproducts), while Altenkirch and Kaposi [5] developed normalization by evaluation for dependent type theory.

Kovács [25] formalized normalization by evaluation for an intrinsically scoped and typed version of the simply typed \( \lambda \)-calculus in Agda.

Fiore [17] was the first to relate normalization by evaluation to categorical gluing. Sterling and Spitters [34] prove normalization for free \( \lambda \)-theories generated from many-typed first-order signatures using categorical gluing.

Kaposi et al. [21] transfer the categorical gluing method to type theory and develop gluing for categories with families. Coquand [13] also proves canonicity and normalization for dependent type theory. His construction arises as a special case of the work of Kaposi et al. [21].

1.2 Overview

In Chapter 2, we discuss the syntax and semantics of the simply typed \( \lambda \) calculus using a traditional syntactic presentation and model theory for the simply typed \( \lambda \)-calculus. Then, in Chapter 3, we discuss the first two proofs of normalization for the simply typed \( \lambda \)-calculus. In these two chapters, we try to avoid any reference to category theory to make the methods more accessible.

The goal of the following three chapters is to present the categorical proof of normalization. First, in Chapter 4, we recall some basic definitions, constructions, and theorems from category theory. Then, in Chapter 5 we discuss the syntax and semantics of the simply-typed lambda from a categorical perspective. Finally, in Chapter 6
we present the categorical proof in detail.

1.3 Acknowledgements

I thank Niels van der Weide for his guidance throughout the creation of this work. I also thank Márk Széles for his help in writing.
Chapter 2

Syntax and semantics of the simply typed λ-calculus

In this chapter, we discuss the syntax and semantics of the simply typed λ calculus as can be found in the early literature on the topic (for instance, [20, 31]). The reason for this choice is that this presentation seems to give the simplest proof of normalization.

In Section 2.1, we formally define the syntax of the simply typed λ-calculus. Besides the language of λ-calculus, we discuss important syntactic notions such as substitution and βη-conversion.

In Section 2.2, we present some basic model theory for the simply typed λ-calculus. In particular, we prove a soundness theorem and construct the term model.

2.1 Syntax

2.1.1 Types and terms

Definition 2.1.1 (Types). We fix some nonempty set Σ of base types. The types of the simply typed λ-calculus are generated by the grammar

\[ \sigma, \tau ::= \beta \mid \sigma \rightarrow \tau, \]

where \( \beta \) ranges over \( \Sigma \).

The type former \(-\rightarrow-\) associates to the right. The set of types is denoted by \( Ty \).

Definition 2.1.2 (Terms). We fix a pairwise disjoint collection of countable sets \( V_\sigma \) indexed by types. Elements of \( V_\sigma \) are called variables of type \( \sigma \). We inductively generate a family of sets \( \Lambda_\sigma \) indexed by types \( \sigma \) according to the rules in Figure 2.1; that is, \( (\Lambda_\sigma)_{\sigma \in Ty} \) is the smallest collection of sets closed under the rules. Elements of \( \Lambda_\sigma \) are called terms of type \( \sigma \).

![Figure 2.1: Terms of the simply typed λ-calculus](image-url)
The application operator \( - \cdot - \) associates to the left. We write \( V = \bigcup_{\sigma \in \Ty} V_\sigma \) and \( \Lambda = \bigcup_{\sigma \in \Ty} \Lambda_\sigma \). If \( t \in V_\sigma \) (respectively \( x \in \Lambda_\sigma \)), we also write \( t : \sigma \) (respectively \( x : \sigma \)) and read it as “\( t \) (respectively \( x \)) has type \( \sigma \)”. We also say that \( t \) is an inhabitant of \( \sigma \), or that \( \sigma \) is inhabited by \( t \).

The abstraction operator \( \lambda \) binds the variable \( x \) in the term \( \lambda x^\sigma \cdot t \), similarly how a quantifier binds a variable in a formula. In other words, \( \lambda x^\sigma \cdot t \) introduces the variable \( x \) whose scope extends to \( t \). The occurrences of \( x \) in \( t \) are thus called bound. Variables that are not bound are called free.

**Definition 2.1.3.** (i) The sets of free variables \( \text{FV}(t) \) and bound variables \( \text{BV}(t) \) of a term \( t \) are defined recursively:

\[
\begin{align*}
\text{FV}(-) & : \Lambda \to \mathcal{P}_{\text{fin}}(\mathcal{V}) & \text{BV}(-) & : \Lambda \to \mathcal{P}_{\text{fin}}(\mathcal{V}) \\
\text{FV}(x) & = \{x\} & \text{BV}(x) & = \emptyset \\
\text{FV}(t \cdot u) & = \text{FV}(t) \cup \text{FV}(u) & \text{BV}(t \cdot u) & = \text{BV}(t) \cup \text{BV}(u) \\
\text{FV}(\lambda x^\sigma \cdot t) & = \text{FV}(t) \setminus \{x\} & \text{BV}(\lambda x^\sigma \cdot t) & = \text{BV}(t) \setminus \{x\}
\end{align*}
\]

(ii) \( \text{FV}_\sigma(t) = \text{FV}(t) \cap V_\sigma \) is the set of free variables of \( t \) of type \( \sigma \).

(iii) A term \( t \) is **closed** if \( \text{FV}(t) = \emptyset \).

Generally in mathematics, the names of bound variables do not matter. The formulas \( \forall x.P(x) \) and \( \forall y.P(y) \) both denote the statement “\( P \) holds for all individuals”. Similarly, the terms \( \lambda x^\sigma \cdot x \) and \( \lambda y^\sigma \cdot y \) both denote the identity function that returns its input unchanged.

**Definition 2.1.4** (\( \alpha \)-convertibility). Two terms are **\( \alpha \)-convertible** if they only differ in the names of bound variables. Details can be found in Chapter 2 of [6].

From now on, we identify \( \alpha \)-convertible terms. This means that we do not distinguish between \( \alpha \)-convertible terms and treat them as being equal. Formally, this amounts to taking a quotient of terms by the equivalence relation of \( \alpha \)-convertibility. A consequence of this is that all operations and properties on terms have to be defined on equivalence classes rather than individual terms. In practice, this means that whenever we use a representative of some equivalence class in a definition, we have to prove the invariance of the definition under \( \alpha \)-convertibility. The details of verifying such assertions are tedious and not too interesting, so we leave them out. See also the discussion in Appendix C of [6].

**Remark 2.1.5.** When working with representatives of \( \alpha \)-equivalence classes, we employ Barendregt’s **variable convention** [6]. This means that all bound variables of all terms that occur in a certain mathematical context (e.g. definition, proof) are assumed to be distinct from all free variables of the terms. Given a countable collection of terms, it is always possible choose representatives in such a way that this condition is satisfied. The convention allows for a simpler treatment of substitution, see Definition 2.1.9.

**Remark 2.1.6.** The base types in \( \Sigma \) have no closed inhabitants, so one might wonder why they are necessary. Indeed, the essence of lambda calculus lies in the function types. However, since types are defined inductively, one needs at least one base type as a base case. Otherwise, the set of types would be empty.

### 2.1.2 Substitution

The free variables of a term may be viewed as placeholders in which other terms may be substituted. For instance, if \( t = x \cdot y \) with \( x : \sigma \to \sigma \) and \( u = \lambda z^\sigma \cdot z \), then we can substitute \( u \) for \( x \) in \( t \), denoted by \( t[[x := u]] \), to obtain \((\lambda z^\sigma \cdot z) \cdot y\).
Instead of defining $t[x := u]$, we take a more general approach where multiple terms may be substituted simultaneously for distinct free variables. This operation is also referred to as parallel substitution. We simply call it substitution.

**Definition 2.1.7 (Substitution).** A substitution is a partial function $\gamma : V \rightarrow \Lambda$ with finite domain such that if $x : \sigma$ and $x \in \text{dom}(\gamma)$, then $\gamma(x) : \sigma$.

The set of substitutions is denoted by $\text{Sub}$.

**Notation 2.1.8.**
- The notation $[x_1 := t_1, \ldots, x_n := t_n]$ stands for the substitution $\gamma$ with $\text{dom}(\gamma) = \{x_1, \ldots, x_n\}$ and $\gamma(x_i) = t_i$ $(i = 1, \ldots, n)$.
- For a substitution $\gamma$, variable $x : \sigma$, and term $t : \sigma$, we write $\gamma[x := t]$ for the updated substitution with $\text{dom}(\gamma[x := t]) = \text{dom}(\gamma) \cup \{x\}$ and such that
  
  \[
  \gamma[x := t](y) = \begin{cases} 
  \gamma(y) & \text{if } y \neq x \\
  t & \text{if } y = x
  \end{cases}.
  \]
- For a finite set of variables $\Gamma \subseteq V$, the identity substitution $\text{id}_\Gamma$ on $\Gamma$ is given by
  \[
  \text{id}_\Gamma(x) = x \quad (x \in \Gamma).
  \]
- If $\gamma$ is a substitution, we write
  \[
  \text{FV}(\gamma) = \bigcup_{x \in \text{dom}(\gamma)} \text{FV}(\gamma(x)).
  \]

**Definition 2.1.9 (Substitution in terms).** For $\gamma = [x_1 := t_1, \ldots, x_n := t_n]$, the expression $t\gamma$ denotes the result of simultaneously substituting the terms $t_1, \ldots, t_n$ for the free variables $x_1, \ldots, x_n$ of $t$. Formally, it is defined by recursion:

\[
(-)(-) : \Lambda_\sigma \times \text{Sub} \rightarrow \Lambda_\sigma
\]

\[
x\gamma = \begin{cases} 
\gamma(x) & \text{if } x \notin \text{dom}(\gamma) \\
\gamma & \text{otherwise}
\end{cases}
\]

\[
(t \cdot u)\gamma = t\gamma \cdot u\gamma
\]

\[
(\lambda x^\sigma. t)\gamma = \lambda x^\sigma. t(x := x)
\]

Note that, by definition, substitution preserves types: if $t : \sigma$, then $t\gamma : \sigma$. Furthermore, we have $\text{FV}(t\gamma) \subseteq (\text{FV}(t) \setminus \text{dom}(\gamma)) \cup \text{FV}(\gamma)$.

Note also that we use the term substitution in two senses. It refers both to the data specifying the terms to be substituted (Definition 2.1.7) and to the operation defined in Definition 2.1.9.

**Remark 2.1.10.** The variable convention (see Remark 2.1.5) ensures that we avoid variable capturing, i.e. when a free variable of some $t_i$ becomes bound in the substituted term. Concretely, when writing $t\gamma$, we assume $\text{BV}(t) \cap \text{FV}(\gamma) = \emptyset$.

### 2.1.3 Conversion

A key feature of the $\lambda$-calculus is the so called $\beta$-rule: applying a function $\lambda x^\sigma. t$ to an argument $u$ results in substituting $u$ for $x$ in $t$. This is expressed by an equational theory between $\lambda$-terms.
\[
\begin{array}{c|c|c|c}
\text{REFL} & t : \sigma & \vdash t = \tau & \vdash t = \tau \\
\text{TRANS} & \vdash t = u : \sigma & \vdash u = v : \sigma & \vdash \tau = \tau \\
\text{SYM} & \vdash \tau = \sigma & \vdash u = t : \sigma & \vdash u = t : \sigma \\
\text{CONG-APP} & \vdash t = \tau & \vdash u = \tau & \vdash \tau = \tau \\
& \vdash t \cdot u = \tau & \vdash \tau = \tau & \vdash \tau = \tau \\
\text{CONG-LAM} & \vdash \lambda x : \tau . t = \tau & \vdash \lambda x : \tau . u = \tau & \vdash \tau = \tau \\
\text{BETA} & \vdash (\lambda x : \tau . t) \cdot u = t[x := u] : \tau & \vdash \tau = \tau & \vdash \tau = \tau \\
\text{ETA} & \vdash \lambda x : \tau . t = x : \tau & \vdash \tau = \tau & \vdash \tau = \tau \\
& \vdash \lambda x : \tau . t = x : \tau & \vdash \tau = \tau & \vdash \tau = \tau \\
\end{array}
\]

Figure 2.2: Rules for equations between \(\lambda\)-terms

**Definition 2.1.11** \((\beta\eta\text{-conversion})\). The typed **conversion relation** is a family of relations indexed by types, generated inductively by the rules in Figure 2.2, i.e. it is the least family of relations closed under the rules. We write \(\vdash t = u : \sigma\) to mean \((t, u) \in =_\sigma\).

Note that by definition, only terms of the same type can be related. This is in contrast with the approach taken in operational semantics where a reduction relation is defined on untyped terms.

The following lemma states that substitution respects conversion.

**Lemma 2.1.12.** Suppose \(\vdash t = t' : \sigma\) and \(\gamma, \gamma' \in \text{Sub}\) such that

\[
\text{dom}(\gamma) = \text{dom}(\gamma') \quad \text{and} \quad \vdash \gamma(x) = \gamma'(x) : \tau \quad \text{for all } x : \tau.
\]

Then \(\vdash t \gamma = t' \gamma' : \sigma\).

**Proof.** By induction on the proof of \(\vdash t = t' : \sigma\) using the congruence rules CONG-APP and CONG-LAM. \(\square\)

### 2.1.4 Normal forms

The rule BETA can also be viewed as a reduction rule: \((\lambda x : \tau . t) \cdot u\) reduces to \(t[x := u]\). This rule is then called \(\beta\)-reduction. Introducing a directionality is particularly useful for applications to programming: it allows one to run a program by applying reduction steps in sequence. Evaluation of a program is finished when no more reductions are applicable. In this case, the program is said to be in \(\beta\)-normal form.

In the extensional \(\lambda\)-calculus (\(\lambda\)-calculus with the rule ETA), it is also natural to consider normality with respect to the ETA rule. The natural direction for this rule is reducing \(\lambda x : \tau . t \cdot x\) to \(t\) (with \(x \notin \text{FV}(t)\)). Terms in which no more BETA or ETA-reductions are applicable are said to be in \(\beta\eta\)-normal form.

In a \(\beta\eta\)-normal form, functions can be partially applied. For instance, if \(f : \sigma \rightarrow \sigma \rightarrow \tau\) and \(x : \sigma\) are variables, then \(f\) and \(f \cdot x\) are both in \(\beta\eta\)-normal form. It is also possible to consider another notion of canonical form, where all functions are fully applied. This is achieved by applying the rule ETA in reverse direction, referred to as \(\eta\)-expansion, until all missing arguments are supplied. For instance, \(f \cdot x\) is converted into \(\lambda y : \tau . f \cdot x \cdot y\) (for a fresh variable \(y\)) in this process. These canonical forms are called *long* \(\beta\eta\)-normal forms.

To make the distinction between the two notions of normal form more explicit, the former notion is also referred to as *short* \(\beta\eta\)-normal form. Alternative names for the
two notions include $\eta$-short $\beta$-normal form and $\eta$-long $\beta$-normal form, respectively. The latter terminology emphasizes the fact that $\eta$-long $\beta$-normal forms are normal with respect to $\beta$-reduction but not $\eta$-reduction.

It turns out that long $\beta\eta$-normal forms can be characterized syntactically by a simple set of rules. Since we are only interested in long $\beta\eta$-normal forms, we simply call them normal forms.

**Definition 2.1.13** (Neutral terms, normal forms). We define two families of sets of $\lambda$-terms, the sets of neutral terms $\text{Ne}_\sigma \subseteq \Lambda_\sigma$ and the sets of normal forms $\text{Nf}_\sigma \subseteq \Lambda_\sigma$. The two families are generated mutually inductively by the clauses in Figure 2.3.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR-NE</td>
<td>$x \in V_\sigma$</td>
</tr>
<tr>
<td>APP-NE</td>
<td>$m \in \text{Ne}<em>{\sigma \rightarrow \tau}$, $n \in \text{Nf}</em>\sigma$</td>
</tr>
<tr>
<td>SHIFT</td>
<td>$m \in \text{Ne}_\beta$</td>
</tr>
<tr>
<td>LAM-NF</td>
<td>$\lambda x^\sigma. n \in \text{Nf}_\tau$</td>
</tr>
</tbody>
</table>

Figure 2.3: Normal forms and neutral terms

Neutral terms are an auxiliary class of terms used in the definition of normal forms. They are terms with a variable in head position preventing the application of the rule $\beta$.

**Definition 2.1.14.** Let $t : \sigma$ be a term. We say that $t$ has a normal form if there exists an $n \in \text{Nf}_\sigma$ such that $\vdash t = n : \sigma$. We then also say that $n$ is a normal form of $t$ or $t$ has the normal form $n$.

### 2.2 Semantics

#### 2.2.1 Applicative structures, models

We first define an appropriate notion of model for the simply typed $\lambda$-calculus.

**Definition 2.2.1** (Applicative structure). An applicative structure $A$ consists of

- a family of sets $A_\sigma$ indexed by types $\sigma$, and
- a family of maps $\text{app}^A_{\sigma, \tau} : A_{\sigma \rightarrow \tau} \times A_\sigma \rightarrow A_\tau$ indexed by pairs of types $\sigma, \tau$.

The maps $\text{app}^A_{\sigma, \tau}$ are called application maps.

It is worth noting that even though terms of type $\sigma \rightarrow \tau$ are thought of as functions, $A_{\sigma \rightarrow \tau}$ does not necessarily have to be a set of functions. However, every element $f \in A_{\sigma \rightarrow \tau}$ of an applicative structure $A$ determines a function

$$\text{app}^A_{\sigma, \tau}(f, -) : A_\sigma \rightarrow A_\tau, \ x \mapsto \text{app}^A_{\sigma, \tau}(f, x).$$

which represents the functional behavior of $f$.

**Notation 2.2.2.** We often omit the superscripts and subscripts of the application maps. Furthermore, we often drop the application map completely and write $fx$ for $\text{app}^A_{\sigma, \tau}(f, x)$ when this does not cause confusion.
Definition 2.2.3 (Extensional applicative structure). We say that an applicative structure \( A \) is **extensional** if the following holds for all \( f, g \in A_{\sigma \rightarrow \tau} \):

\[
(\forall x \in A_{\sigma}. fx = gx) \implies f = g.
\]

**Remark 2.2.4.** The assignment \( f \mapsto \text{app}(f, -) \) gives a map \( A_{\sigma \rightarrow \tau} \to A_{\sigma \times \tau} \). The extensionality axiom is equivalent to the statement that this map is injective. Thus, in an extensional applicative structure, we may identify \( A_{\sigma \rightarrow \tau} \) with a subset of \( A_{\sigma \times \tau} \). Under this identification, the application map \( \text{app}_{\sigma, \tau} \) becomes the evaluation map \( (f, x) \mapsto f(x) \).

To be able to interpret \( \lambda \)-terms in an applicative structure, one needs to choose meanings for the free variables of the term. This role is filled by environments.

Definition 2.2.5 (Environment). Let \( A \) be an applicative structure.

(i) An **environment** for \( A \) is a partial function \( \rho : V \rightarrow \bigcup_{\sigma \in \text{Ty} A} A_{\sigma} \) with finite domain such that if \( x : \sigma \) and \( x \in \text{dom}(\rho) \), then \( \rho(x) \in A_{\sigma} \).

(ii) If \( \Gamma \subseteq V \) is a finite set of variables, then a \( \Gamma \)-environment is an environment \( \rho \) such that \( \Gamma \subseteq \text{dom}(\rho) \).

The set of environments for \( A \) is denoted by \( \text{Env}^A \). We write \( \text{Env}^A_{\Gamma} \) for the set of \( \Gamma \)-environments. As usual, the superscript can be omitted. If \( \rho \in \text{Env}_\Gamma \), we also write \( \rho \models \Gamma \). Clearly, if \( \Gamma \subseteq \Gamma' \) and \( \rho \models \Gamma' \), then \( \rho \models \Gamma \).

Note the formal similarity between substitutions and environments. Environments can be seen as the semantic counterpart to substitutions. Whereas a substitution maps free variables to terms, an environment maps them to elements of a structure. The two notions are connected in the term model (Definition 2.2.20), for which environments are essentially the same as substitutions.

**Notation 2.2.6.**

- For an environment \( \rho \), variable \( x : \sigma \), and \( a \in A_{\sigma} \), we write \( \rho[x := a] \) for the **updated environment** with \( \text{dom}(\rho[x := a]) = \text{dom}(\rho) \cup \{x\} \) and such that

\[
\rho[x := a](y) = \begin{cases} 
\rho(y) & \text{if } y \neq x \\
 a & \text{if } y = x.
\end{cases}
\]

- The **empty environment** \( \emptyset \) is the empty function.

The updated environment is the semantic analogue of the updated substitution (Notation 2.1.8).

**Definition 2.2.7 (Environment model).** An **environment model** \( A \) is an extensional applicative structure \( A \) together with an assignment

\[
(t, \rho) \mapsto \llbracket t \rrbracket^A_\rho \in A_{\sigma} \quad \text{for } t : \sigma \text{ and } \rho \models \text{FV}(t)
\]

such that the following equations are satisfied:

\[
\llbracket x \rrbracket^A_\rho = \rho(x) \quad \text{(2.1)}
\]

\[
\llbracket t \cdot u \rrbracket^A_\rho = \llbracket t \rrbracket^A_\rho \llbracket u \rrbracket^A_\rho \quad \text{(2.2)}
\]

\[
\llbracket \lambda x^{\gamma}. t \rrbracket^A_\rho a = \llbracket t \rrbracket^A_{\rho[x := a]} \quad (a \in A_{\gamma}) \quad \text{(2.3)}
\]

The value \( \llbracket t \rrbracket^A_\rho \) is called the **interpretation** of \( t \) at environment \( \rho \) in the model \( A \). As before, we usually omit the superscript \( A \).
Remark 2.2.8. It follows by a simple induction on terms and the extensionality of $A$ that there is at most one possible value for the interpretation $\llbracket t \rrbracket_\rho$ of $t$. Hence, the conditions in Definition 2.2.7 can almost be seen as a recursive definition of the interpretation of $\lambda$-terms. The only problem is that the interpretation of $\lambda x^\tau. t : \sigma \rightarrow \tau$ may not be defined if $A_{\sigma \rightarrow \tau}$ does not contain enough elements.

More specifically, recall from Remark 2.2.4 that if $A$ is an extensional applicative structure, then $A_{\sigma \rightarrow \tau}$ may be identified with a subset of $A_{\sigma \rightarrow \tau}^{A_{\sigma \rightarrow \tau}}$. By (2.3), the interpretation $\llbracket \lambda x^\tau. t \rrbracket_\rho$ has to be equal to the function $A_{\sigma} \rightarrow A_{\tau}$ given by $a \mapsto [\llbracket t \rrbracket_{\rho[x:=a]}]$. However, it is possible that this function is not in $A_{\sigma \rightarrow \tau}$.

An environment model is thus an extensional applicative structure in which it is possible to interpret every $\lambda$-term. Hence, we can say that an extensional applicative structure $A$ is an environment model if the uniquely defined interpretation function is well-defined.

The intended semantics for the simply typed $\lambda$-calculus is given by sets and functions.

Example 2.2.9 (Standard model). Let $X = (X_\beta)_{\beta \in \Sigma}$ be a family of sets indexed by base types. We define the model $S(X)$, called the standard model (over $X$), as follows. The sets $S(X)_\sigma$ are given by recursion on types:

$$S(X)_\sigma = X_\beta \quad (\beta \in \Sigma)$$

$$S(X)_{\sigma \rightarrow \tau} = S(X)^{S(X)_\sigma \times S(X)_\tau}_{S(X)_\sigma}$$

The application map $\text{app}^{S(X)} : S(X)_{\tau}^{S(X)_\sigma \times S(X)_\tau} \rightarrow S(X)_\tau$ is the evaluation map $(f, x) \mapsto f(x)$. It is clear that this applicative structure is extensional. The interpretation of $\lambda$-terms is defined recursively according to the conditions in Definition 2.2.7.

An applicative structure contains an application map to model application of $\lambda$-terms. On the other hand, $\lambda$-abstraction does not have an immediate semantic counterpart, which explains why it is not always possible to interpret a $\lambda$-abstraction. In Definition 2.2.7 this was solved by postulating that the intended interpretation function exists. This might feel strange from a model-theoretic perspective: a semantic structure should be adequate to interpret syntax without us having to verify this explicitly. There is an alternative formulation of when an extensional applicative structure is a model of $\lambda$-calculus which meets this criterion.

Definition 2.2.10 (Combinatory model). A combinatory model $A$ is an extensional applicative structure $A$ with distinguished elements

$$K^A_{\sigma, \tau} \in A_{\sigma \rightarrow \tau} \rightarrow \sigma$$

$$S^A_{\sigma, \tau, \chi} \in A_{(\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \chi) \rightarrow \sigma \rightarrow \chi}$$

for every $\sigma, \tau, \chi \in \text{Ty}$ such that

$$K^A_{\sigma, \tau} x y = x \quad \text{and} \quad S^A_{\sigma, \tau, \chi} f g x = f(x(gx)) \quad (2.4)$$

for all $x, y, f, g$ of the appropriate types.

As for the application maps, we may omit the superscripts and subscripts of $K$ and $S$.

Remark 2.2.11. Combinatory models are so named because they provide semantics for typed combinatory logic, a system equivalent to the simply typed $\lambda$-calculus. Combinatory logic bypasses the use of variables by expressing all functions using certain elementary functions called combinators. Specifically, it has constants

$$K_{\sigma, \tau} : \sigma \rightarrow \tau \rightarrow \sigma$$

$$S_{\sigma, \tau, \chi} : (\sigma \rightarrow \tau \rightarrow \chi) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \chi$$
for every $\sigma, \tau, \chi \in Ty$, satisfying the axioms

$$\vdash K \cdot t \cdot u = t \quad \text{and} \quad \vdash S \cdot f \cdot g \cdot t = f \cdot t \cdot (g \cdot t)$$

for every appropriately typed $t, u, f, g$. It should be clear that the intended semantics of $K_{\sigma,\tau}$ and $S_{\sigma,\tau,\chi}$ in a combinatory model $A$ are $K^A_{\sigma,\tau}$ and $S^A_{\sigma,\tau,\chi}$, respectively.

**Remark 2.2.12.** As in the case for environment models (see Remark 2.2.8), the elements $K$ and $S$ in Definition 2.2.10 are uniquely determined by their defining equations (2.4). Thus, every extensional applicative structure $A$ can be equipped with at most one choice of $K$ and $S$ to make it a combinatory model. In this case, we may say that $A$ is a combinatory model.

**Proposition 2.2.13.** Suppose $A$ is an extensional applicative structure. Then $A$ is an environment model if and only if it is a combinatory model.

**Proof.** The result follows easily from the syntactic equivalence between simply typed $\lambda$-calculus and typed combinatory logic. For example, given an environment model $A$, we can define combinators

$$K_{\sigma,\tau} = \mathcal{K}(\lambda x^{\sigma}. \lambda y^{\tau}. x)$$

$$S_{\sigma,\tau,\chi} = \mathcal{S}(\lambda f^{\sigma \to \tau \to \chi}. \lambda g^{\sigma \to \tau}. \lambda x^{\sigma}. f \cdot x \cdot (g \cdot x))$$

In the other direction, to define $\llbracket t \rrbracket_\rho$ in a combinatory model, we translate the $\lambda$-closure of $t$ to a combinatory term, take its semantics, and apply the resulting function to the values in the environment $\rho$. Details on the translation to combinators for the untyped case can be found in [6, 22].

By Proposition 2.2.13 we may simply speak of a model of $\lambda$-calculus, and use either the environment or the combinatory model formulation to prove that an extensional applicative structure is a model. These kinds of models are also called **Henkin models** in the literature [29].

### 2.2.2 Basic model theory

We now establish some basic results about Henkin models. The most important are the Soundness Theorem 2.2.19 and the existence of a term model (Definition 2.2.20) which satisfies precisely those equations that are provable in the syntax (Proposition 2.2.22).

**Lemma 2.2.14.** Let $t : \sigma$. Then for all environments $\rho, \rho' \models \text{FV}(t)$, we have

$$(\forall x \in \text{FV}(t). \rho(x) = \rho'(x)) \quad \text{implies} \quad \llbracket t \rrbracket_\rho = \llbracket t \rrbracket_{\rho'}.$$  

**Proof.** By induction on $t$:

- **Case** $x$:
  $$\llbracket x \rrbracket_\rho = \rho(x) = \rho'(x) = \llbracket x \rrbracket_{\rho'}$$

- **Case** $t \cdot u$:
  $$\llbracket t \cdot u \rrbracket_\rho = \llbracket t \rrbracket_\rho \llbracket u \rrbracket_\rho = \llbracket t \rrbracket_{\rho'} \llbracket u \rrbracket_{\rho'} = \llbracket t \cdot u \rrbracket_{\rho'}$$

- **Case** $\lambda x^{\sigma}. t$:
  $$\llbracket \lambda x^{\sigma}. t \rrbracket_\rho \rho[a] = \llbracket t \rrbracket_\rho[x := a] = \llbracket t \rrbracket_{\rho'}[x := a] = \llbracket \lambda x^{\sigma}. t \rrbracket_{\rho'} a$$

In order to apply the induction hypothesis, we have used that $\rho[x := a](y) = \rho'[x := a](y)$ for all $y \in \text{FV}(t) \cup \{x\}$. 

13
Lemma 2.2.16 (Substitution lemma). Let $A$ be a model. Then for all $t : \sigma, \gamma \in \text{Sub}$, and $\rho \models (\text{FV}(t) \setminus \text{dom}(\gamma)) \cup \text{FV}(\gamma)$, we have

$$\llbracket t \rrbracket_\rho = \llbracket t \rrbracket_{\gamma \cdot \rho}.$$

Proof. By induction on the term $t$:

- Case $x$: if $x \in \text{dom}(\gamma)$, then we have
  $$\llbracket x \gamma \rrbracket_\rho = \llbracket \gamma(x) \rrbracket_\rho = \llbracket \gamma \rrbracket_\rho(x) = \llbracket x \rrbracket_{\gamma \cdot \rho}.$$

  If $x \notin \text{dom}(\gamma)$, then
  $$\llbracket x \gamma \rrbracket_\rho = \llbracket x \rrbracket_\rho = \rho(x) = \llbracket \gamma \rrbracket_\rho(x) = \llbracket x \rrbracket_{\gamma \cdot \rho}.$$

- Case $t \cdot u$:
  $$\llbracket (t \cdot u) \gamma \rrbracket_\rho = \llbracket t \gamma \cdot u \gamma \rrbracket_\rho = \llbracket t \gamma \rrbracket_\rho \llbracket u \gamma \rrbracket_\rho = \llbracket t \rrbracket_{\gamma \cdot \rho} \llbracket u \rrbracket_{\gamma \cdot \rho} = \llbracket t \cdot u \rrbracket_{\gamma \cdot \rho}.$$

- Case $\lambda x^\sigma. t$: Extensionality implies that it suffices to show
  $$\llbracket (\lambda x^\sigma. t) \gamma \rrbracket_\rho a = \llbracket \lambda x^\sigma. t \rrbracket_{\gamma \cdot \rho} a$$

  for all $a \in A_\sigma$.

$$\llbracket (\lambda x^\sigma. t) \gamma \rrbracket_\rho a = \llbracket \lambda x^\sigma. t(\gamma[x := x]) \rrbracket_\rho a = \llbracket t(\gamma[x := x]) \rrbracket_{\rho[x := a]}$$

$$= \llbracket \gamma \rrbracket_{\rho[x := a]} \llbracket x \rrbracket_{\rho[x := a]} = \llbracket t \rrbracket_{\gamma \cdot \rho} \llbracket x \rrbracket_{\rho[x := a]} = \llbracket \lambda x^\sigma. t \rrbracket_{\gamma \cdot \rho} a.$$

In order to apply the induction hypothesis, we have used that

$$\rho[x := a] \models (\text{FV}(t) \setminus \text{dom}(\gamma[x := x])) \cup \text{FV}(\gamma[x := x]).$$

In the second line, we have used that $\llbracket \gamma[x := x] \rrbracket_{\rho[x := a]} = \llbracket \gamma \rrbracket_\rho[x := a]$. This is true since

$$\llbracket \gamma[x := x] \rrbracket_{\rho[x := a]}(x) = \llbracket \gamma[x := x] \rrbracket_{\rho[x := a]} = \llbracket x \rrbracket_{\rho[x := a]}$$

$$= \rho[x := a](x) = a = \llbracket \gamma \rrbracket_\rho[x := a](x),$$

$$\llbracket \gamma[x := x] \rrbracket_{\rho[x := a]}(y) = \rho(x := a)(y) = \rho(y) = \llbracket \gamma \rrbracket_\rho(y) = \llbracket \gamma \rrbracket_\rho[x := a](y)$$

if $y \notin \text{dom}(\gamma) \cup \{x\} = \text{dom}(\gamma[x := x])$, and

$$\llbracket \gamma[x := x] \rrbracket_{\rho[x := a]}(y) = \llbracket \gamma[x := x] \rrbracket_{\rho[x := a]} = \llbracket \gamma \rrbracket_{\rho[x := a]}$$

$$= \llbracket \gamma \rrbracket_\rho = \llbracket \gamma \rrbracket_\rho(y) = \llbracket \gamma \rrbracket_\rho[x := a](y)$$

if $y \in \text{dom}(\gamma) \setminus \{x\}$. In the last case, the equality $\llbracket \gamma(y) \rrbracket_{\rho[x := a]} = \llbracket \gamma(y) \rrbracket_\rho$ follows from Lemma 2.2.14 because $x \notin \text{FV}(\gamma(y))$. \qed
Corollary 2.2.17. Let \( t : \tau, x : \sigma, \) and \( u : \sigma \). Then for all environments \( \rho \models (\text{FV}(t) \setminus \{x\}) \cup \text{FV}(u) \), we have
\[
\llbracket x := u \rrbracket_{\rho} = \llbracket t \rrbracket_{\rho[x := u]_{\rho}}.
\]

Proof. Follows immediately from Lemma 2.2.16 taking \( \gamma = [x := u] \), using the equation
\[
\llbracket x := u \rrbracket_{\rho} = \rho(x := [u]_{\rho}).
\]

The latter holds since
\[
\llbracket x := u \rrbracket_{\rho}(x) = \llbracket x := u \rrbracket_{\rho}(x) = \rho(x := [u]_{\rho})(x)
\]
and
\[
\llbracket x := u \rrbracket_{\rho}(y) = \rho(y) = \rho(x := [u]_{\rho})(y)
\]
for \( y \in \text{dom}(\rho) \setminus \{x\} \). \qed

Definition 2.2.18 (Satisfaction). Let \( \mathcal{A} \) be a model.

(i) Suppose \( t, u : \sigma \), and \( \rho \models \text{FV}(t) \cup \text{FV}(u) \). We say that the equation \( \vdash t = u : \sigma \) holds at \( \rho \) in \( \mathcal{A} \), written as \( \mathcal{A}, \rho \models t = u : \sigma \), if \( \llbracket t \rrbracket_{\rho}^{\mathcal{A}} = \llbracket u \rrbracket_{\rho}^{\mathcal{A}} \).

(ii) The model \( \mathcal{A} \) satisfies the equation \( \vdash t = u : \sigma \), written as \( \mathcal{A} \models t = u : \sigma \), if \( \mathcal{A}, \rho \models t = u : \sigma \) for all \( \rho \models \text{FV}(t) \cup \text{FV}(u) \).

Theorem 2.2.19 (Soundness). Let \( \mathcal{A} \) be a model. Then
\[
\vdash t = u : \sigma \quad \text{implies} \quad \mathcal{A} \models t = u : \sigma
\]
for all \( t, u : \sigma \).

Proof. By induction on the derivation of \( \vdash t = u : \sigma \). The cases \text{REFL}, \text{TRANS}, and \text{SYM} follow from the fact that \( \mathcal{A} \models \vdash \sigma = \vdash \sigma \) is an equivalence relation. For the cases \text{CONG-APP} and \text{CONG-LAM} we use the compositionality of the interpretation of \( \lambda \)-terms: the meaning of compound terms is defined in terms of the meanings of the subterms. Hence, if the latter are equal at all environments, so are the former.

For \text{BETA}, we use Corollary 2.2.17 Let \( \rho \models (\text{FV}(t) \setminus \{x\}) \cup \text{FV}(u) \). Then
\[
\llbracket (\lambda x^{\tau}. t \cdot u) \rrbracket_{\rho} = \llbracket \lambda x^{\tau}. t \rrbracket_{\rho} \llbracket u \rrbracket_{\rho} = \llbracket t \rrbracket_{\rho [x := u]_{\rho}} = \llbracket t \rrbracket_{\rho[x := u]_{\rho}}.
\]

Finally, \text{ETA} follows from the extensionality of \( \mathcal{A} \) and Lemma 2.2.14
\[
\llbracket \lambda x^{\tau}. t \cdot x \rrbracket_{\rho} = \llbracket t \cdot x \rrbracket_{\rho [x := a]} = \llbracket t \rrbracket_{\rho [x := a]} \llbracket x \rrbracket_{\rho [x := a]} = \llbracket t \rrbracket_{\rho} [a].
\]

Note that to apply Lemma 2.2.14 we use the assumption that \( x \notin \text{FV}(t) \). \qed

Definition 2.2.20 (Term model). The term model \( \mathcal{L} \) is defined as follows. The set \( \mathcal{L}_{\sigma} \) is the set of terms of type \( \sigma \) quotiented by convertibility:
\[
\mathcal{L}_{\sigma} = \Lambda_{\sigma}/\sim_{\sigma}.
\]

That is, \( \mathcal{L}_{\sigma} = \{ [t] \mid t : \sigma \} \), where
\[
[t] = \{ u : \sigma \mid \vdash t = u : \sigma \}
\]
denotes the \( \beta \eta \)-equivalence class of \( t \). The application map is given by
\[
[t][u] = [t \cdot u]
\]
for \( t : \sigma \rightarrow \tau \) and \( u : \sigma \).

By definition, an environment for \( \mathcal{L} \) is a mapping from variables to equivalences of terms. If \( \gamma \) is a substitution, then \([\gamma] \) denotes the environment given by \([\gamma](x) = \gamma(x)\) for all \( x \in \text{dom}(\gamma) \). Every \( \rho \in \text{Env}^\mathcal{L} \) can be written as \([\gamma]\) for some substitution \( \gamma \) by picking representatives in \( \rho(x) \) for each \( x \in \text{dom}(\rho) \). That said, we define the interpretation into \( \mathcal{L} \) as

\[
[t]^{\mathcal{L}}_\gamma = [t\gamma].
\]

**Proposition 2.2.21.** \( \mathcal{L} \) is a well-defined model.

**Proof.** To show well-definedness, we need to check that our definitions are independent of the choices of representatives. For the application map, suppose we pick representatives \( t', u' \) with \( \vdash t = t' : \sigma \rightarrow \tau \) and \( \vdash u = u' : \sigma \). By rule \textsc{cong-app}, we have \( \vdash t \cdot u = t' \cdot u' : \tau \), hence \([t \cdot u] = [t' \cdot u']\).

Now suppose \( \gamma, \gamma' \) are substitutions such that \([\gamma] = [\gamma']\). Then we have \([\gamma(x)] = [\gamma'(x)]\), hence \( \vdash \gamma(x) = \gamma'(x) : \sigma \) for all \( x \in \text{dom}(\gamma) \). By Lemma 2.1.12 this implies \( \vdash t\gamma = t\gamma' : \sigma \), and thus \([t\gamma] = [t\gamma']\). This shows the well-definedness of the interpretation.

To show that the application structure \( \mathcal{L} \) is extensional, let \([t], [t'] \in \mathcal{L}_{\sigma \rightarrow \tau} \) and assume that \([t][u] = [t'][u]\) for all \( u : \sigma \). Let \( x : \sigma \) be a variable such that \( x \notin \text{FV}(t) \cup \text{FV}(u) \). Then we have \([t \cdot x] = [t][x] = [t'][x] = [t' \cdot x] \) by assumption, hence \( \vdash t \cdot x = t' \cdot x : \tau \). By \textsc{cong-lam}, we derive \( \vdash \lambda x. t \cdot x = \lambda x. t' \cdot x : \sigma \rightarrow \tau \). Applying \textsc{eta} twice, we get \( \vdash t = t' : \sigma \rightarrow \tau \). Thus, \([t] = [t']\) as desired.

It remains to check that the interpretation satisfies the equations in Definition 2.2.7:

\[
\begin{align*}
\llbracket t \cdot u \rrbracket_\gamma &= \llbracket (t \cdot u)\gamma \rrbracket = \llbracket t\cdot u\rrbracket_\gamma = \llbracket t\rrbracket_\gamma \llbracket u\rrbracket_\gamma \\
\llbracket \lambda x. t \rrbracket_\gamma[u] &= \llbracket (\lambda x. t)\gamma[u] \rrbracket = \llbracket (\lambda x. t)[x := u] \rrbracket_\gamma = \llbracket t\rrbracket_\gamma[x := u]
\end{align*}
\]

Note that in the last case, \( \llbracket t\rrbracket_\gamma[x := u] = \llbracket t\rrbracket_\gamma[x := u] \) only because \( x \notin \text{FV}(\gamma) \).

**Theorem 2.2.22.** For all \( t \) and \( u \), \( \vdash t = u : \sigma \) if and only if \( \mathcal{L} \vdash t = u : \sigma \).

**Proof.** The only if part is a corollary of Theorem 2.2.19 and Proposition 2.2.21. If \( \mathcal{L} \vdash t = u : \sigma \), then \( \llbracket t \rrbracket^{\mathcal{L}}_{\text{id}} = \llbracket u \rrbracket^{\mathcal{L}}_{\text{id}} \). If \( \mathcal{L} \vdash t = u : \sigma \), then \( \llbracket t \rrbracket^{\mathcal{L}}_{\text{id}} = \llbracket u \rrbracket^{\mathcal{L}}_{\text{id}} = [u(\text{id})] = [u], \)

and hence \( \vdash t = u : \sigma \).

### 2.2.3 Homomorphisms

When studying the properties of a class of mathematical structures, it is worthwhile to consider structure-preserving mappings between the structures. These mappings are usually called **homomorphisms**. For example, functions between groups preserving the group operation are called **group homomorphisms** and they are an essential concept in group theory.

We can apply the same idea to the notions of model defined in Section 2.2.1. Although we do not make use of homomorphisms between Henkin models in this thesis, we believe it is worth mentioning them to make it easier to relate this theory to the categorical semantics to be introduced in Chapter 5.
Definition 2.2.23 (Homomorphism of applicative structures). Let $\mathcal{A}$ and $\mathcal{B}$ be applicative structures. A homomorphism of applicative structures $h : \mathcal{A} \to \mathcal{B}$ is a family of maps $(h_\sigma : A_\sigma \to B_\sigma)_{\sigma \in Ty}$ such that

$$h_\tau(app_{\sigma,\tau}(f, x)) = app_{\sigma',\tau}(h_{\sigma'\tau}(f), h_{\sigma'}(x))$$

for all $\sigma, \tau \in Ty$, $f \in A_{\sigma'\tau}$, and $x \in A_\sigma$.

Notation 2.2.24. Let $h : \mathcal{A} \to \mathcal{B}$ be a homomorphism of applicative structures.

- If $x \in A_\sigma$, then we may write $h(x)$ for $h_\sigma(x)$, omitting the type $\sigma$.
- If $\rho \in \text{Env}^A$ is an environment for $\mathcal{A}$, then $h\rho \in \text{Env}^B$ is the environment defined by $(h\rho)(x) = h_\sigma(\rho(x))$ for all $x \in \text{dom}(\rho) \cap V_\sigma$.

Definition 2.2.25 (Homomorphism of models). Let $\mathcal{A}$ and $\mathcal{B}$ be models. We say that a homomorphism of applicative structures $h : \mathcal{A} \to \mathcal{B}$ is a homomorphism of models if

$$h_\sigma([t]_\rho^A) = [t]_{h\rho}^B$$

for all $t : \sigma$ and $\rho \models \text{FV}(t)$.

Equivalently, using the combinatory model formulation, a homomorphism of models $h : \mathcal{A} \to \mathcal{B}$ can be defined as a homomorphism of applicative structures such that

$$h(K^A_{\sigma,\tau}) = K^B_{\sigma,\tau} \quad \text{and} \quad h(S^A_{\sigma,\tau,\chi}) = S^B_{\sigma,\tau,\chi}.$$

For every model $\mathcal{A}$, there is an identity homomorphism $1_\mathcal{A} : \mathcal{A} \to \mathcal{A}$ which is the identity function at every type. Furthermore, homomorphisms $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{C}$ can be composed pointwise to yield a homomorphism $gf : \mathcal{A} \to \mathcal{C}$. Thus, Henkin models and homomorphisms of models form a category.
Chapter 3

Normalization for the simply typed $\lambda$-calculus

Normalization refers to the process of finding a normal form for a given term. Traditionally, normalization and the notion of normal form have been defined in the context of term rewrite systems. A term rewrite system is given by a set of terms and a set of rewrite rules, also called reduction rules. In this context, a normal form is a term in which no reductions are possible, and normalizing a term consists of applying the rewrite rules repeatedly until a normal form is reached.

There is a computational reading of this process: terms represent programs, and applying rewrite rules corresponds to running a program. The execution is finished when no more reductions are possible, in which case the final term represents the value of the program. Thus, normal forms correspond to values that a program can output, and normalizing corresponds to execution of programs. In particular, in the (simply typed) $\lambda$-calculus, the most important rewrite rule is $\beta$-reduction (the rule $\text{beta}$ applied from left to right).

An interesting question to ask about a term rewrite system is whether every term can be normalized, i.e. whether every term can be reduced to a normal form. A rewrite system in which this is possible is said to satisfy the weak normalization property. There is a stronger property known as strong normalization which states that every possible reduction path is finite, i.e. no infinite reductions are possible.

In a programming language that satisfies the strong normalization property, any sequence of execution steps will eventually lead to a final result, i.e. no program can run forever. In programming language terminology, we say that the programming language is terminating: all programs are guaranteed to terminate and return a result. In contrast, if a language only satisfies weak normalization, then we only know that there is some execution path that leads to a result, but not necessarily that each of them does. However, if evaluation in the language is deterministic (that is, there is at most one possible execution step at each stage), then this property already implies that the language is terminating. Thus, weak normalization can still be a useful property.

In a reduction free setting, we cannot formalize the strong normalization property. However, the weak normalization property still makes sense if reduction is replaced by convertibility. The property then states that every term has a normal form (see Definition 2.1.14). Fiore [17, 18] calls this the extensional normalization problem. Furthermore, it is convenient to characterize normal forms syntactically without reference to any reduction rules. This was done in Definition 2.1.13 for long $\beta\eta$-normal forms.

For practical purposes, the weak normalization property is not sufficient. Instead, we need an effective procedure for determining the normal form of a term, that is, a
function that maps terms to their normal forms. Fiore \[17, 18\] calls this the \textit{intensional normalization problem}. Such a normalization function can then be used for example to decide the convertibility of two terms (Corollary \[3.2.32\]). This is interesting from a logical point of view: the simply typed $\lambda$-calculus can be viewed as an equational theory, and thus the validity of formulas in this theory is decidable.

In Section \[3.1\] we prove weak normalization for the simply typed $\lambda$-calculus using the method of \textit{computability predicates} invented by Tait \[36\] and Gödel \[21\]. We then provide an analysis of this proof in terms of the notion of \textit{logical relation} introduced by Statman \[33\], which is a generalization of Tait’s predicates.

In Section \[3.2\] we consider \textit{normalization by evaluation}, a technique invented by Berger and Schwichtenberg \[9\], providing a solution for the intensional normalization problem. Normalization by evaluation lies on the observation that normalization is essentially evaluation for open terms. However, instead of an operational approach, it employs denotational semantics for evaluation, thereby bypassing reduction. Hence, normalization of a term may be implemented by evaluating it in a context where the free variables are mapped to themselves, and then turning the semantic object back into a normal form.

There is a close analogy between Tait’s proof and normalization by evaluation. In fact, Berger showed \[7, 8\] that a version of normalization by evaluation can be obtained from a Tait-like proof of strong normalization using a mechanized procedure known as \textit{program extraction}. In Section \[3.2.3\] we highlight the correspondences between the two proofs.

3.1 Weak normalization

In this section, we prove the following theorem:

**Theorem 3.1.1** (Weak normalization). \textit{All terms $t : \sigma$ have a normal form.}

A naive attempt of a proof of this theorem would try to prove the statement directly by induction on terms. However, this approach gets stuck at the application case: we cannot prove that $t \cdot u$ has a normal form if $t$ and $u$ have one. A counterexample in the untyped $\lambda$-calculus is the term $(\lambda x. x x) (\lambda x. x x)$.

In Section \[3.1.1\] we present a proof of Theorem 3.1.1 based on the idea of a \textit{convertible term} by Tait \[36\], who used it to prove normalization of terms in Gödel’s system T \[21\]. Tait’s method itself was inspired by Gödel’s notion of a \textit{computable/reckonable functional} (German: \textit{berechenbare Funktion}) \[21\].

The essential idea of the proof is that we define when a term is convertible by induction on the type of the term (Definition \[3.1.3\]). A term of base type is convertible iff it has a normal form; a term of function type is convertible iff it maps convertible inputs to convertible outputs. Now proving that every term is convertible is straightforward using induction on terms (Lemma \[3.1.5\]), solving the issue of the naive attempt. Furthermore, every convertible term has a normal form (Lemma \[3.1.6\]); hence, the theorem follows. On a methodological level, the proof considers a stronger property (convertibility) in order to provide a stronger induction hypothesis, thus making the induction go through. This method is sometimes referred to as \textit{induction loading}.

In Section 3.1.2 we introduce the notion of \textit{logical predicate}, which is a special case of the notion of \textit{logical relation} introduced by Statman \[33\]. Precursors of logical relations and extensions thereof have been widely used in the literature to prove (among other things) definability results about the simply typed $\lambda$-calculus \[31, 33, 23, 19\]. The point of logical predicates is that they generalize the core idea behind Tait’s convertibility predicate, namely, that the property we wish to prove should be defined by induction.
on the type structure. We also prove a generalization of Lemma 3.1.5 (Theorem 3.1.9),
dubbed the Fundamental Theorem of Logical Relations by Statman [33].

Furthermore, we introduce the notion of bilogical predicate (Definition 3.1.10) and
prove a corresponding fundamental theorem (Theorem 3.1.12). The fundamental the-
orem abstracts away the ingredients of the proof in Section 3.1.1. This development
was inspired by Fiore’s Basic Lemma [17,18] and it is essentially an instantiation of his
result phrased in a simpler framework.

Finally, in Section 3.1.3, we give a very short proof of Theorem 3.1.1 employing the
results of Section 3.1.2. This proof is essentially a rephrasing of the proof in Section 3.1.1.

3.1.1 “Syntactic” proof

Definition 3.1.2. Let $T \subseteq \Lambda_\sigma$. We define

$$C(T) = \{t : \sigma \mid \exists u \in T. \vdash t = u : \sigma\}.$$ 

That is, $C(T)$ is the set of terms of type $\sigma$ that are convertible to some term in $T$.
Note that $C(Nf_\sigma)$ is the set of terms of type $\sigma$ that have a normal form.

Definition 3.1.3 (Convertibility predicate). We define a family of predicates $R_\sigma \subseteq \Lambda_\sigma$
on $\lambda$-terms by recursion on types:

$$R_\beta = C(Nf_\beta) \quad (\beta \in \Sigma)$$

$$R_{\sigma \rightarrow \tau} = \{t : \sigma \rightarrow \tau \mid \forall u \in R_\sigma. t \cdot u \in R_\tau\}$$

Lemma 3.1.4. If $\vdash t = t' : \sigma$ and $t' \in R_\sigma$, then $t \in R_\sigma$.

Proof. By induction on $\sigma$:

- Case $\beta \in \Sigma$: Suppose $t' \in R_\beta$, i.e. there is a normal form $n : \beta$ such that
  $\vdash t' = n : \beta$. Since $\vdash t = t' : \beta$, transitivity implies $\vdash t = n : \beta$. Hence,
  $t \in C(Nf_\beta) = R_\beta$.
- Case $\sigma \rightarrow \tau$: Suppose $t' \in R_{\sigma \rightarrow \tau}$, and let $u \in R_\sigma$. Then $t' \cdot u \in R_\tau$. Since
  $\vdash t = t' : \sigma \rightarrow \tau$, we derive $\vdash t \cdot u = t' \cdot u : \tau$. Thus, we can apply the induction
  hypothesis to conclude that $t \cdot u \in R_\tau$. Hence, $t \in R_{\sigma \rightarrow \tau}$.

Lemma 3.1.5 (Fundamental Lemma). Let $t : \sigma$ and $\gamma \in \text{Sub}$ such that $\text{FV}(t) \subseteq \text{dom}(\gamma)$. If $\gamma(x) \in R_\tau$ for all $x \in \text{FV}_\tau(t)$, then $t\gamma \in R_\sigma$.

Proof. By induction on $t$:

- Case $x$: $x\gamma = \gamma(x) \in R_\tau$ by assumption.
- Case $t \cdot u$: We have $t\gamma \in R_{\sigma \rightarrow \tau}$ and $u\gamma \in R_\sigma$ by the induction hypotheses. Thus,
  $(t \cdot u)\gamma = t\gamma \cdot u\gamma \in R_\sigma$ by the definition or $R_{\sigma \rightarrow \tau}$.
- Case $\lambda x^\sigma. t$: We need to show that $(\lambda x^\sigma. t)\gamma = \lambda x^\sigma. t(\gamma[x := x]) \in R_{\sigma \rightarrow \tau}$.
  Let $u \in R_\sigma$. Then $t[x := u]$ satisfies the conditions of the lemma, and hence
  we can apply the induction hypothesis to obtain $t(\gamma[x := u]) \in R_\tau$. Since $\vdash$
  $(\lambda x^\sigma. t(\gamma[x := x])) \cdot u = t(\gamma[x := u]) : \tau$, we infer $(\lambda x^\sigma. t(\gamma[x := x])) \cdot u \in R_\tau$ by
  Lemma 3.1.11. 

Lemma 3.1.6. The following two statements hold for all types $\sigma$:

(i) $R_\sigma \subseteq C(Nf_\sigma)$;
(ii) $C(Nf_\sigma) \subseteq R_\sigma$.

Proof. We prove both statements simultaneously by induction on $\sigma$. 

20
Definition 3.1.10. Suppose \( R \) is a logical predicate over \( A \), then we say that an environment \( \rho \in \text{Env} \) satisfies the predicate \( R \) if \( \rho(x) \in R_{\sigma} \) for all \( x \in \text{dom}(\rho) \cap V_\sigma \).

Theorem 3.1.9 (Fundamental Theorem of Logical Relations). Suppose \( A \) is a model and \( R \) is a logical predicate over \( A \). Then for every term \( t : \sigma \) and environment \( \rho \in \text{Env}^A_{\text{FV}(t)} \) satisfying \( R \), we have \( \llbracket t \rrbracket_\rho \in R_\sigma \).

Proof. By induction on \( t \):

- Case \( x : \tau \): \( \llbracket x \rrbracket_\rho = \rho(x) \in R_\tau \) by assumption.
- Case \( \lambda x. t \): Let \( a \in R_\sigma \). Then \( \rho[x := a] \) satisfies \( R \), and hence we can apply the induction hypothesis to obtain \( \llbracket t \rrbracket_{\rho[x := a]} \in R_\tau \). Thus, \( \llbracket \lambda x. t \rrbracket_\rho a = \llbracket t \rrbracket_{\rho[x := a]} \in R_\tau \) for all \( a \in R_\sigma \).

Definition 3.1.10. Suppose \( A \) is an applicative structure. A bilogical predicate over \( A \) consists of two families of subsets \( Q_\sigma, S_\sigma \subseteq A_\sigma \) indexed by types \( \sigma \), satisfying the following conditions:

(i) \( Q_\beta \subseteq S_\beta \) for all \( \beta \in \Sigma \);
(ii) \( f \in Q_{\sigma \rightarrow \tau} \) and \( x \in S_\tau \) implies \( fx \in Q_\sigma \);
(iii) if \( fx \in S_\tau \) for all \( x \in Q_\sigma \), then \( f \in S_{\sigma \rightarrow \tau} \).

Definition 3.1.11. If \( (Q, S) \) is a bilogical predicate over \( A \), then we say that an environment \( \rho \in \text{Env}^A \) satisfies the predicate \( (Q, S) \) if \( \rho(x) \in Q_\sigma \) for all \( x \in \text{dom}(\rho) \cap V_\sigma \).
Theorem 3.1.12 (Fundamental Theorem of Bilogical Relations). Suppose \( A \) is a model and \( (Q, S) \) is a logical predicate over \( A \). Then for every term \( t : \sigma \) and environment \( \rho \in \text{Env}_{\Lambda}^A \) which satisfies \( (Q, S) \), we have \( \llbracket t \rrbracket^A_\rho \in S_\sigma \).

Proof. Define the logical predicate \( R \) by \( R_\beta = S_\beta \) for all \( \beta \in \Sigma \). We show that

\[
Q_\sigma \subseteq R_\sigma \subseteq S_\sigma \tag{3.2}
\]

for all types \( \sigma \). The proof proceeds by induction on \( \sigma \).

- Case \( \beta \in \Sigma \): We have \( Q_\beta \subseteq R_\beta = S_\beta \) by definition of \( R \) and condition (i) for bilogical predicates.
- Case \( \sigma \rightarrow \tau \): Let \( f \in Q_{\sigma \rightarrow \tau} \) and suppose \( x \in Q_\sigma \). By the induction hypothesis for \( \sigma, x \in S_\sigma \), so condition (ii) for bilogical predicates implies \( f x \in Q_\tau \). Therefore, by the induction hypothesis for \( f, f x \in R_\tau \). Hence, \( f x \in R_\tau \) for all \( x \in R_\sigma \), and thus \( f \in R_{\sigma \rightarrow \tau} \) by definition of \( R_{\sigma \rightarrow \tau} \).

Now let \( f \in R_{\sigma \rightarrow \tau} \) and suppose \( x \in Q_\sigma \). By the induction hypothesis for \( f \), we have \( x \in R_\sigma \), hence \( f x \in R_\tau \). By the induction hypothesis for \( f \), we get \( f x \in S_\tau \). Hence, we have \( f x \in S_\tau \) for all \( x \in Q_\sigma \), so \( f \in S_{\sigma \rightarrow \tau} \) by condition (iii) for bilogical predicates.

This concludes the proof of (3.2). Now let \( t \) and \( \rho \) be as in the statement of the theorem. Since \( Q_\tau \subseteq R_\tau \) for all types \( \tau \), we have that \( \rho \) satisfies \( R \). Hence, by the Fundamental theorem of logical relations (Theorem 3.1.9), we get \( \llbracket t \rrbracket^A_\rho \in R_\sigma \subseteq S_\sigma \).

\[ \square \]

3.1.3 “Semantic” proof

Lemma 3.1.13. Let

\[
M_\sigma = \{ [m] \mid m \in \text{Ne}_\sigma \} \quad \text{and} \quad N_\sigma = \{ [n] \mid n \in \text{Nf}_\sigma \}
\]

for all \( \sigma \). Then \( (M, N) \) is a bilogical predicate over \( L \).

Proof. We verify the conditions of bilogical predicates.

(i) \( M_\beta \subseteq N_\beta \) by rule \textsc{shift}.
(ii) Let \( [m] \in M_{\sigma \rightarrow \tau} \) and \( [n] \in N_\sigma \), where \( m \in \text{Ne}_{\sigma \rightarrow \tau} \) and \( n \in \text{Nf}_\sigma \). Then \( m \cdot n \) by rule \textsc{app-ne}. Hence, \([m][n] = [m \cdot n] \in M_\tau\).
(iii) Let \( [t] \in L_{\sigma \rightarrow \tau} \) such that \([t][m] \in N_\tau \) for all \([m] \in M_\sigma \). Then, in particular, \([t][x] = [t \cdot x] = [n] \) for some \( n \in \text{Nf}_\tau \), where \( x \) is a variable of type \( \sigma \) such that \( x \notin \text{FV}(t) \). This means that \( \vdash t \cdot x = n : \tau \), and by rules \textsc{cong-lam} and \textsc{eta}, we derive \( \vdash t = \lambda x^\sigma . n : \sigma \rightarrow \tau \). Hence, \([t] = [\lambda x^\sigma . n] \in N_{\sigma \rightarrow \tau} \).

\[ \square \]

Alternative proof of Theorem 3.1.12 Let \( (M, N) \) be the bilogical predicate of Lemma 3.1.13 and let \( \text{id} \) be the identity substitution on \( \text{FV}(t) \). Since \( x \in \text{Ne}_\sigma \) for all \( x : \sigma \) by rule \textsc{var-ne}, \([\text{id}] \) satisfies \( (M, N) \). Applying the Fundamental theorem of bilogical relations (Theorem 3.1.12), we get that \( [t] = \llbracket t \rrbracket^A_{\text{id}} \in N_\sigma \). This means that \([t] = [n] \) for some \( n \in \text{Nf}_\sigma \), that is, \( \vdash t = n : \sigma \).

\[ \square \]

3.2 Normalization by evaluation

In this section, we define a normalization function \( \text{nf}_\sigma : \Lambda_\sigma \rightarrow \text{Nf}_\sigma \) that returns the normal form of the input term. It has to satisfy the following requirements:

(i) \( \vdash t = \text{nf}_\sigma(t) : \sigma \) for all \( t : \sigma \), and
(ii) if $t = u : \sigma$, then $\text{nf}_\sigma(t) = \text{nf}_\sigma(u)$.

The normalization function $\text{nf}_\sigma$ will be constructed using the method of normalization by evaluation (NbE) invented by Berger and Schwichtenberg [9]. Informally, NbE works by evaluating the term to be normalized in a suitable model, and then mapping the resulting semantic object back into syntax, producing a normal form evaluating to that object. Since we want the normal form to be convertible to the original term, the mapping provides an inverse to the evaluation function modulo convertibility. Hence, NbE can also be described as inverting the evaluation function ([9]).

To implement this idea, we have to solve some technicalities. By the definition of long $\beta\eta$-normal forms (Definition 2.1.13), a normal form of function type has to be a $\lambda$-abstraction. Hence, when inverting a semantic value of type $\sigma \to \tau$, we need to produce a term of the form $\lambda x^\sigma.t$ where $t \in \text{Nf}_\tau$. To avoid confusion of free and bound variables, the variable $x$ has to be fresh with respect to the whole computation.

We solve the problem by keeping track of free variables in order to generate fresh variables when necessary (see Notation 3.2.25 and Definition 3.2.26). For this reason, in Section 3.2.1 we introduce contexts whose role is to specify the free variables in a term. Thus, terms in this section are indexed by both types and contexts. Accordingly, in Section 3.2.2 we introduce a notion of “indexed” model, called a Kripke model [30], to implement the algorithm. In Section 3.2.3 we construct the algorithm (Definition 3.2.28) and prove its correctness (Theorem 3.2.30).

There are also alternative ways to solve the problem of generating fresh variable names, see for instance [9, 16]. An advantage of our presentation of normalization by evaluation is that it is closer to the more abstract categorical view on normalization to be discussed in Chapter 6. However, it is also possible to implement NbE without referring to contexts and using simple Henkin models, see Dybjer and Filinski [16]. An advantage of Dybjer and Filinski’s approach is that their normalization algorithm is more efficient and easier to describe. We believe, however, that the algorithm presented here is mathematically more elegant. For instance, it gives a stronger specification to the normalization function, guaranteeing that its output is well-typed in the same context as its input.

We remark that NbE can be implemented for all sorts of syntaxes for the simply typed $\lambda$-calculus. For instance, Kovács [25] formalized NbE for a well-scoped version of $\lambda$-calculus with de Bruijn indices in Agda.

### 3.2.1 Adding contexts

**Definition 3.2.1 (Context).** A context $\Gamma$ is a finite set of variables (of any type), that is, an element of $P_{\text{fin}}(V)$.

**Notation 3.2.2.**
- The set of contexts is denoted by $\text{Con}$.
- Let $\Lambda_\sigma(\Gamma) = \{t \in \Lambda_\sigma \mid \text{FV}(t) \subseteq \Gamma\}$. Let $\text{Ne}_\sigma(\Gamma)$ and $\text{Nf}_\sigma(\Gamma)$ denote the subsets of neutral terms and normal forms, respectively, in $\Lambda_\sigma(\Gamma)$.
- If $t \in \Lambda_\sigma(\Gamma)$, then we write $\Gamma \vdash t : \sigma$.
- Similarly, if $t, u \in \Lambda_\sigma(\Gamma)$ and $\Gamma \vdash t = u : \sigma$, then we write $\Gamma \vdash t = u : \sigma$.

We can give “context-aware” derivation rules for terms, displayed in Figure 3.1. The soundness of these rules is easily shown by unfolding the notations in Notation 3.2.2 and using Definition 2.1.2 and Definition 2.1.3. We write $(x : \sigma) \in \Gamma$ to mean $x \in \Gamma \cap \text{V}_\sigma$, and we write $\Gamma, x : \sigma$ to mean $\Gamma \cup \{x\}$ with the assumption that $x : \sigma$.

With contexts, we can also give a more refined specification to the normalization function.
Definition 3.2.3. A normalization function is a family of (computable) functions

$$nf^\sigma_{\Gamma} : \Lambda^\sigma(\Gamma) \rightarrow \text{Nf}^\sigma(\Gamma)$$

indexed by contexts $\Gamma$ and types $\sigma$, satisfying the following requirements:

(i) $\Gamma \vdash t = nf^\sigma_{\Gamma}(t) : \sigma$ for all $t \in \Lambda^\sigma(\Gamma)$, and

(ii) if $\Gamma \vdash t = u : \sigma$, then $nf^\sigma_{\Gamma}(t) = nf^\sigma_{\Gamma}(u)$.

Remark 3.2.4. Let us explain why we put parentheses around the word computable in Definition 3.2.3. Constructing a program that implements a normalization function amounts to showing that it is computable. Hence, for a practical implementation of normalization, we need to make sure that the normalization function is computable.

If we work in a constructive metatheory, then the condition is automatically fulfilled since the definitions directly give rise to a functional program. In a classical metatheory, some additional work could be necessary to implement the mathematical normalization function in a programming language and prove that it corresponds to the mathematical definition.

3.2.2 Kripke models

For a more thorough discussion of Kripke models, see Mitchell and Moggi [30].

Definition 3.2.5 (Kripke applicative structure). Let $(W, \preceq)$ be a poset. A Kripke applicative structure $A$ over $W$ consists of

- a family $A^w_\sigma$ of sets indexed by $\sigma \in \text{Ty}$ and $w \in W$,
- a family of transition maps $i^{w,w'}_\sigma : A^w_\sigma \rightarrow A^{w'}_\sigma$ indexed by $\sigma \in \text{Ty}$ and $w, w' \in W$ such that $w \preceq w'$, and
- a family of application maps $\text{app}^{w}_\sigma : A^{w}_{\sigma \rightarrow \tau} \times A^{w}_{\sigma} \rightarrow A^{w}_{\tau}$

such that

- $i^{w,w'}_{\sigma}$ is the identity function on $A^{w}_{\sigma}$,
- $i^{w,w'}_{\sigma} \circ i^{w,w'}_{\sigma} = i^{w,w'}_{\sigma}$, and
- $\text{app}^{w}_\sigma(i^{w,w'}_{\sigma}(f), i^{w,w'}_{\sigma}(x)) = i^{w,w'}_{\sigma}(\text{app}^{w}_\sigma(f, x))$

for all $\sigma, \tau \in \text{Ty}$ and $w \preceq w' \preceq w''$.

Notation 3.2.6. As before, we may drop the types when they are understood from context, and we may write $fx$ for $\text{app}^{w}_\sigma(f, x)$.

The poset $W$ is viewed as a set of possible worlds partially ordered by accessibility. The transition map $i^{w,w'}_{\sigma}$ allows us to transport an element at world $w$ to any “future” world $w'$.

For the remainder of this section, we fix some poset $(W, \preceq)$. The notation $\uparrow(w)$ stands for the the set $\{w' \in W \mid w' \geq w\}$. 24
Similarly to standard applicative structures, every element \( f \in \mathcal{A}_w^w \) determines a function \( \mathcal{A}_w^w \to \mathcal{A}_w^w \) sending \( x \) to \( fx \). With Kripke applicative structures, more is true: we can use the transition map \( i_{w,w'}^w \) to view \( f \) at world \( w' \). Hence, there is a family of functions \( \mathcal{A}_w^w \to \mathcal{A}_w^w \) given by \( x \mapsto i_{w,w'}^w(f)x \) for every \( w' \geq w \), representing the applicative behaviour of \( f \) in all future worlds.

**Definition 3.2.7** (Extensional Kripke applicative structure). We say that a Kripke applicative structure \( \mathcal{A} \) over \( W \) is **extensional** if the following holds for all \( f, g \in \mathcal{A}_w^w \):

\[
(\forall w' \geq w. \forall x \in \mathcal{A}_w^w. i_{w,w'}^w(f)x = i_{w,w'}^w(g)x) \text{ implies } f = g.
\]

**Remark 3.2.8.** Compare this definition with Definition 2.2.3. The property states that if the applicative behaviour of two elements is the same, then the elements must be identical.

For the definition of a Kripke model, we need the following auxiliary notion.

**Definition 3.2.9** (Global element). Let \( \mathcal{A} \) be a Kripke applicative structure over \( W \). A **global element** \( a \) of \( \mathcal{A} \) of type \( \sigma \) is a family \( a_w \in \mathcal{A}_w^w \) indexed by \( w \in W \) such that \( i_{w,w'}^w(a_w) = a_w' \) for all \( w \leq w' \).

If \( a \) is a global element of \( \mathcal{A} \) of type \( \sigma \), we write \( a \in \mathcal{A}_\sigma \) to express this fact. Furthermore, we write \( a_w \) for the component of \( a \) at \( w \).

**Definition 3.2.10** (Kripke model). A **Kripke model** \( \mathcal{A} \) over \( W \) is an extensional Kripke applicative structure \( \mathcal{A} \) over \( W \) with distinguished global elements

\[
K^{\sigma,\tau} \in \mathcal{A}_{\sigma \to \tau \to \sigma} \quad \text{and} \quad S^{\sigma,\tau,\chi} \in \mathcal{A}_{(\sigma \to \tau \to \chi) \to (\sigma \to \tau \to \chi)}
\]

for every \( \sigma, \tau, \chi \in \mathsf{Ty} \) such that

\[
K_w^{\sigma,\tau}xy = x \quad \text{and} \quad S_w^{\sigma,\tau,\chi}fgx = fx(gx)
\]

for all \( x, y, f, g \) of the appropriate types.

**Remark 3.2.11.** This is an indexed version of Definition 2.2.10.

**Example 3.2.12** (Standard model). Let \( X = (X^w_\beta)_{\beta \in \Sigma, w \in W} \) be a family of base sets and \( j = (j_{w, w'}^\beta : X^w_\beta \to X^{w'}_\beta)_{\beta \in \Sigma, w \leq w'} \) a family of base transition maps such that

\[
- j_{w,w}^\beta \text{ is the identity on } X^w_\beta,
- j_{w',w}^{\beta'} \circ j_{w,w'}^\beta = j_{w',w'}^\beta
\]

for all \( \beta \in \Sigma \) and \( w \leq w' \leq w'' \). We define the model \( S(X,j) \), called the **standard (Kripke) model**, as follows. The sets \( S(X,j)^w_\sigma \) and the transition maps \( i_{w,w'}^w \) are defined simultaneously by recursion on types:

\[
S(X,j)_\beta^w = X^w_\beta \quad (\beta \in \Sigma)
\]

\[
i_{w,w'}^w(x) = j_{w,w'}^w(x) \quad (\beta \in \Sigma)
\]

\[
S(X,j)^{w,\tau}_{w' \to \tau} = \left\{ f \in \prod_{w' \leq w} S(X,j)^w_{w' \to \tau} \mid \forall w'' \geq w. \forall w' \geq w'. \forall x. f_{w''}(i(x)) = i(f_w(x)) \right\}
\]

\[
i_{w,\tau}^w((f)_{w' \to \tau}) = (f_{w'})_{w' \to \tau}
\]
The application map $\text{app}_{\sigma, \tau}^w$ is given by $(f, x) \mapsto f_w(x)$. The global elements $K$ and $S$ are defined as follows:

$$(K_{\sigma, \tau}^w)(x) = \text{app}_{\sigma, \tau}^w(x)$$

$$(S_{\sigma, \tau}^w)(f)(x) = (f_w^w(x)) = (f_w^w(g_w^w(x))).$$

In what follows, $A$ denotes a Kripke model over $W$.

**Definition 3.2.13 (Environment).**

(i) An environment for $A$ is a partial function $\rho : V \times W \to \bigcup_{\sigma \in \mathcal{T}_Y, w \in W} \mathcal{A}_\sigma^w$ with finite domain such that

- if $x : \sigma$ and $(x, w) \in \text{dom}(\rho)$, then $\rho(x, w) \in \mathcal{A}_\sigma^w$, and
- if $(x, w) \in \text{dom}(\rho)$ and $w \leq w'$, then $(x, w') \in \text{dom}(\rho)$ and $\rho(x, w') = \rho(x, w).

(ii) If $\Gamma \in \text{Con}$ and $w \in W$, then an environment $\rho$ is said to satisfy $\Gamma$ at $w$, notation $\rho \vdash \Gamma \mid w$, if $\Gamma \times \{w\} \subseteq \text{dom}(\rho)$.

**Remark 3.2.14.** Compare with Definition 2.2.5.

**Notation 3.2.15.** For an environment $\rho$, variable $x : \sigma$, and $a \in \mathcal{A}_\sigma^w$, we write $\rho[x := a]$ for the updated environment with $\text{dom}(\rho[x := a]) = \text{dom}(\rho) \cup \{x\} \times \uparrow(w)$ and such that $\rho[x := a](y, w') = \begin{cases} \rho(y, w') & \text{if } y \neq x \\ \rho(w', a) & \text{if } y = x \end{cases}$.

**Remark 3.2.16.** Compare with Notation 2.2.6.

**Proposition 3.2.17 (Interpretation of $\lambda$-calculus in a Kripke model).** There is an assignment $(t, \rho) \mapsto \llbracket t \rrbracket_\rho^w \in \mathcal{A}_\sigma^w$ for $\Gamma \vdash t : \sigma$ and $\rho \vdash \Gamma \mid w$ such that the following equations are satisfied:

$$\llbracket x \rrbracket_\rho^w = \rho(x, w)$$

$$\llbracket t \cdot u \rrbracket_\rho^w = \llbracket t \rrbracket_\rho^w \llbracket u \rrbracket_\rho^w$$

$$\llbracket \lambda x. \cdot t \rrbracket_\rho^w \cdot a = \llbracket t \rrbracket_{\rho[x := a]}^w \quad (a \in \mathcal{A}_\sigma^w, w' \geq w)$$

**Remark 3.2.18.** Similarly to Henkin models, the clauses in Proposition 3.2.17 uniquely define the interpretation of a term $t$ in an environment $\rho$. Hence, it can be seen as the definition of interpretation. The well-definedness of the assignment is guaranteed by the existence of the interpretations for the combinators $K$ and $S$. Compare with Remark 2.2.8 Remark 2.2.12 and Proposition 2.2.13.

**Example 3.2.19.** Specializing Proposition 3.2.17 to the standard Kripke model (Example 3.2.12), we get the following standard interpretation function:

$$\llbracket x \rrbracket_\rho^w = \rho(x, w)$$

$$\llbracket t \cdot u \rrbracket_\rho^w = \llbracket t \rrbracket_\rho^w \llbracket u \rrbracket_\rho^w$$

$$\llbracket \lambda x. \cdot t \rrbracket_\rho^w \cdot a = \llbracket t \rrbracket_{\rho[x := a]}^w \quad (a \in \mathcal{A}_\sigma^w, w' \geq w)$$

**Definition 3.2.20 (Satisfaction).**

(i) Suppose $\Gamma \vdash t : \sigma$, $\Gamma \vdash u : \sigma$ and $\rho \vdash \Gamma \mid w$. We say that the equation $\Gamma \vdash t = u : \sigma$ holds at $w$ and $\rho$ in $A$, written as $A, \rho \models \Gamma \vdash t = u : \sigma$, if $\llbracket t \rrbracket_\rho^w = \llbracket u \rrbracket_\rho^w$. 

26
(ii) The model \( \mathcal{A} \) satisfies the equation \( \Gamma \vdash t = u : \sigma \), written as \( \mathcal{A} \models \Gamma \vdash t = u : \sigma \), if \( \mathcal{A}, \rho \models \Gamma \vdash t = u : \sigma \) for all \( w \in W \) and \( \rho \models \Gamma \models w \).

**Remark 3.2.21.** Compare with Definition 2.2.18.

**Theorem 3.2.22 (Soundness).** For all \( \Gamma \vdash t : \sigma \) and \( \Gamma \vdash u : \sigma \), we have

\[
\Gamma \vdash t = u : \sigma \quad \text{implies} \quad \mathcal{A} \models \Gamma \vdash t = u : \sigma.
\]

**Proof.** By induction on the proof of \( \Gamma \vdash t = u : \sigma \). The proof is similar to that of Theorem 2.2.19.

### 3.2.3 The algorithm

We are now ready to define the algorithm. Our algorithm is similar to that of Reynolds [32]. The main steps of the algorithm are as follows:

(i) We construct a suitable model \( \mathcal{S} \) (Definition 3.2.24).

(ii) The model gives rise to an interpretation \( \llbracket - \rrbracket \) of terms in the model.

(iii) We define a family of functions \( q_{\sigma} : S_{\sigma}^{\Gamma} \rightarrow Nf_{\sigma}(\Gamma) \) indexed by types and contexts by recursion on \( \sigma \). These functions provide an inverse to the interpretation/evaluation function \( \llbracket - \rrbracket \). Due to the contravariance of function types in the first argument, we also need a family of functions \( u_{\sigma} : N_{\sigma}(\Gamma) \rightarrow S_{\sigma}^{\Gamma} \) in the other direction, embedding neutral terms (in particular, variables) into the semantics.

(iv) We derive the normalization function \( nf_{\sigma}^{\Gamma} \) from the ingredients above. For this, we need a special environment \( \eta_{\Gamma} \) (Definition 3.2.27) that maps all variables in \( \Gamma \) to their semantic counterpart.

There is a close analogy between these steps and the structure of the proof of Section 3.1.1.

(i) The model \( \mathcal{S} \) corresponds to the convertibility predicate \( R \) (Definition 3.1.3). In particular, both \( S_{\sigma}^{\Gamma} \) and \( R_{\sigma} \) are defined by induction on the type \( \sigma \).

(ii) The fact that there is an interpretation \( \llbracket - \rrbracket \) of terms in the model corresponds the Fundamental Lemma (Lemma 3.1.5).

(iii) The functions \( q_{\sigma} \) and \( u_{\sigma} \) correspond to the two parts of Lemma 3.1.6, respectively.

(iv) The derivation of the normalization function follows the proof of weak normalization in Section 3.1.1. In particular, the environment \( \eta_{\Gamma} \) corresponds to the fact that the identity substitution on \( \Gamma \) satisfies the logical predicate \( R \).

We now present the construction of our normalization function.

**Definition 3.2.23.** We write \( C \) for the poset of contexts ordered by inclusion.

**Definition 3.2.24.** Let \( \mathcal{S} \) be the standard Kripke model over \( C \), where the base families \((X, j)\) are given by \( X^{\Gamma}_{\beta} = Nf_{\beta}(\Gamma) \) and \( j^{\Gamma, \Gamma'}_{\beta}(n) = n \).

**Notation 3.2.25.**

- For a context \( \Gamma \), let \( v_{\Gamma, \sigma} \) denote a fresh variable of type \( \sigma \) for \( \Gamma \), that is, a variable \( v_{\Gamma, \sigma} \in V_{\sigma} \) such that \( v_{\Gamma, \sigma} \notin \Gamma \).
- Let \( \Gamma^{\Gamma'} \) denote the context \( \Gamma \cup \{v_{\Gamma, \sigma}\} \).
Definition 3.2.26. We define two families of functions $q^\Gamma_{\sigma} : S^\Gamma_{\sigma} \to Nf_\sigma(\Gamma)$ and $u^\Gamma_{\sigma} : Ne_\sigma(\Gamma) \to S^\Gamma_{\sigma}$ simultaneously by recursion on $\sigma$:

$$
q^\Gamma_{\sigma, \rightarrow}(f) = \lambda v^\Gamma_{\sigma, \sigma}. (f^\Gamma_{\sigma, \sigma}(q^\Gamma_{\sigma, \sigma}(v)))
$$

$$
u^\Gamma_{\beta}(m) = m
$$

$$
u^\Gamma_{\beta, \rightarrow}(m) = \nu^\Gamma_{\beta}(m \cdot q^\Gamma_{\beta}(a))
$$

Definition 3.2.27. For all contexts $\Gamma$, we define the environment $\eta^\Gamma$ as follows:

$$
\eta^\Gamma(x, \Gamma') = \nu^\Gamma_{\sigma}(x)
$$

for $(x : \sigma) \in \Gamma' \supseteq \Gamma$.

We have that $\eta^\Gamma \models \Gamma \mid \Gamma$.

Definition 3.2.28. We define the function $nf^\Gamma_{\sigma}$ as follows:

$$
nf^\Gamma_{\sigma}(t) = q^\Gamma_{\sigma}(\llbracket t \rrbracket^\Gamma_{\eta^\Gamma})
$$

We now prove the correctness of $nf$ by showing that it is a normalization function in the sense of Definition 3.2.3.

Lemma 3.2.29.

(i) For all $m \in Ne_\sigma(\Gamma)$, we have $\llbracket m \rrbracket^\Gamma_{\eta^\Gamma} = u^\Gamma_{\sigma}(m)$.

(ii) For all $n \in Nf_\sigma(\Gamma)$, we have $nf^\Gamma_{\sigma}(n) = n$.

Proof. By simultaneous induction on neutral terms and normal forms.

Theorem 3.2.30. The function $nf^\Gamma_{\sigma}$ defined in Definition 3.2.28 satisfies the requirements in Definition 3.2.3.

Proof. Condition (ii) follows from Theorem 3.2.22 and the definition of $nf^\Gamma_{\sigma}$. By Theorem 3.1.1 we know that there is $n \in Nf_\sigma$ such that $\vdash t = n : \sigma$. By Condition (ii) and Lemma 3.2.29 we get $nf^\Gamma_{\sigma}(t) = nf^\Gamma_{\sigma}(n) = n$. Hence, $\vdash t = nf^\Gamma_{\sigma}(t) : \sigma$, showing Condition (i). (Note: technically, we need to show $\Gamma \vdash t = \eta^\Gamma_{\sigma}(\llbracket t \rrbracket^\Gamma_{\eta^\Gamma}) : \sigma$. For this, we need that $\Gamma \vdash n : \sigma$, which can be shown by a syntactic argument).

Corollary 3.2.31. For all $\Gamma \vdash t : \sigma$ and $\Gamma \vdash u : \sigma$, we have $\Gamma \vdash t = u : \sigma$ iff $nf^\Gamma_{\sigma}(t) = nf^\Gamma_{\sigma}(u)$.

Proof. The only if direction is Definition 3.2.3 (ii). The if direction follows from Definition 3.2.3 (i), and reflexivity, transitivity, and symmetry of the convertibility relation.

Corollary 3.2.32. Given $\Gamma \vdash t : \sigma$ and $\Gamma \vdash t : \sigma$, it is decidable whether $\Gamma \vdash t = u : \sigma$.

Proof. By Corollary 3.2.31 we can compute $nf^\Gamma_{\sigma}(t)$ and $nf^\Gamma_{\sigma}(u)$ and compare the resulting normal forms for syntactic equality.
Chapter 4

Categorical preliminaries

In this chapter, we recall some important categorical notions, constructions, and theorems that are used throughout the upcoming chapters, and we fix the associated notation.

Categories are denoted by uppercase Latin letters in calligraphic font (e.g., \(A, B, C\)). Specific named categories are written with boldface letters, such as the category \(\text{Set}\) of sets and functions and the category \(\text{Cat}\) of small categories and functors. The collection of objects of a category \(C\) is denoted by \(\text{Ob}(C)\). To indicate that \(A\) is an object of \(C\), we also write \(A \in C\) instead of \(A \in \text{Ob}(C)\). For \(A, B \in C\), the collection of morphisms from \(A\) to \(B\) is denoted by \(C(A, B)\) or \(\text{Hom}(A, B)\) if the category \(C\) is clear from the context. The identity morphism on \(A\) is written as \(1_A\), and the composition of morphisms \(f : B \to C\) and \(g : A \to B\) is written as \(f \circ g\) or \(fg\). The subscript of \(1_A\) is sometimes dropped when understood from context.

The image of an object \(A \in C\) under a functor \(F : C \to D\) is written as \(F(A)\) or \(FA\). Similarly, the image of a morphism \(f\) is written as \(F(f)\) or \(Ff\). The identity functor on \(C\) is denoted by \(\text{Id}_C\) or \(1_C\). Just like for morphisms, the composition of functors \(F : B \to C\) and \(G : A \to B\) is written as \(F \circ G\) or \(FG\). Note that juxtaposition is thus overloaded to mean both composition and application. However, we try to avoid ambiguity by employing appropriate notation.

Given a natural transformation \(\mu : F \to G\) between functors \(F, G : C \to D\), we denote its component at \(A \in C\) by \(\mu_A\). The vertical composition of natural transformations \(\mu : G \to H\) and \(\nu : F \to G\) is again denoted by \(\mu \circ \nu\) or \(\mu \nu\). For a natural transformation \(\mu : F \to G\) between functors \(F, G : C \to D\) and functor \(H : B \to C\), we write \(\mu H : FH \to GH\) for the natural transformation given by \((\mu H)_A = \mu_H A : FHA \to GHA\). The operation sending \(\mu\) to \(\mu H\) is called whiskering (on the right).

4.1 Universal properties

**Definition 4.1.1.** An object \(T\) in a category \(C\) is called terminal if for every object \(X\) of \(C\) there is exactly one morphism \(h : X \to T\). Diagrammatically:

\[
X \xrightarrow{\exists! h} T
\]

Given a terminal object \(T\), we write \(!_X : X \to T\) for the unique morphism into \(T\). If a category \(C\) has a terminal object, it is denoted by \(1_C\) or simply \(1\).

**Example 4.1.2.** In the category \(\text{Set}\), every singleton set is terminal. A canonical choice for the terminal object is the set \(\{\emptyset\}\) containing only the empty set. Since the choice does not matter, we choose some singleton set \(\{\ast\}\) for the terminal object and use the
notation ∗ for its only element. The unique morphism !_X : X → {∗} is the constant map with value ∗.

Example 4.1.3. In the category Cat, any category with exactly one object and one identity morphism on that object is terminal.

The dual notion of a terminal object is an initial object.

Definition 4.1.4. An object I in a category C is called initial if for every object X of C there is exactly one morphism h : I → X. Diagrammatically:

\[ I \rightarrow X \]

Given an initial object I, we write !_X : I → X for the unique morphism out of I. If a category C has an initial object, it is denoted by 0_C or simply 0.

Example 4.1.5. The category of sets has exactly one initial object, namely, the empty set ∅. The unique morphism !_X : ∅ → X is the empty function.

Definition 4.1.6. Let A and B be objects in C. Then a (binary) product of A and B is an object P together with morphisms p₁ : P → A, p₂ : P → B satisfying the following universal property: for every pair of morphisms f : X → A, g : X → B there exists a unique morphism h : X → P such that

\[ p₁ \circ h = f \quad \text{and} \quad p₂ \circ h = g. \]

Diagrammatically:

\[
\begin{array}{c}
X \\
\downarrow f \\
\downarrow g \\
A & \leftarrow & P & \rightarrow & B \\
\downarrow p₁ & \leftarrow & \downarrow !_P & \rightarrow & \downarrow p₂
\end{array}
\]

If (P,p₁,p₂) is a product of A and B, then P is denoted by A × B. The maps p₁ and p₂ are called projections, and they are denoted by fst_{A,B} : A × B → A and snd_{A,B} : A × B → B, respectively. In the literature, the object A × B itself is often referred to as the product of A and B, leaving the projections implicit; we follow this convention. Given maps f : X → A, g : X → B, the unique morphism arising from the universal property is written as ⟨f,g⟩ : X → A × B and is called the pairing of f and g.

Example 4.1.7. The categorical product of sets A and B is the cartesian product A × B together with the projection functions fst_{A,B} : A × B → A and snd_{A,B} : A × B → B sending (a, b) to a and b, respectively. The pairing of functions f : X → A and g : X → B is the function ⟨f,g⟩ : X → A × B given by ⟨f,g⟩(x) = (f(x),g(x)).

Definition 4.1.8. Suppose that for every pair of objects A, B in C we have chosen a product A × B of A and B. Then we can define the product functor − × − : C × C → C that sends a pair of objects ⟨A,B⟩ to A × B and a pair of morphisms ⟨f : A → A',g : B → B'⟩ to

\[ f \times g = ⟨f \circ \text{fst}_{A,B},g \circ \text{snd}_{A,B}⟩ : A × B → A' × B'. \]

Proposition 4.1.9. The operations defined in Definition 4.1.8 constitute a functor.

Definition 4.1.10. Let A and B be objects in C such that for every object X of C we have chosen a product X × A of X and A. Then an exponential of A and B is an
object \( E \) together with a map \( \varepsilon : E \times A \to B \) satisfying the following universal property: for any morphism \( f : X \times A \to B \) there exists a unique morphism \( h : X \to E \) such that

\[
\varepsilon \circ (h \times 1_A) = f.
\]

Diagrammatically:

\[
\begin{array}{c}
E \\
\uparrow \exists h \\
E \times A \xrightarrow{\varepsilon} B \\
\downarrow h \times 1_A \\
X \times A
\end{array}
\]

If \((E, \varepsilon)\) is an exponential of \( A \) and \( B \), then \( E \) is denoted by \( A \Rightarrow B \). The map \( \varepsilon \) is called **evaluation**, and it is denoted by \( \text{ev}_{A,B} : (A \Rightarrow B) \times B \to A \). Similarly to products, the object \( A \Rightarrow B \) itself is often referred to as the exponential of \( A \) and \( B \) in the literature, leaving the evaluation implicit; we follow this convention. Given a map \( f : X \times A \to B \), the unique morphism \( h : X \to A \Rightarrow B \) arising from the universal property is written as \( \lambda(f) \), and the maps \( f \) and \( h \) are called **exponential transposes** of each other. The operation sending \( f \) to \( \lambda(f) \) is also called **currying**.

**Example 4.1.11.** The exponential \( A \Rightarrow B \) in \( \text{Set} \) is given by the set \( B^A \) of functions from \( A \) to \( B \), and the evaluation map \( \text{ev}_{A,B} : B^A \times A \to B \) sends a pair \((f, a)\) to \( f(a) \). The exponential transpose of a function \( f : X \times A \to B \) is the function \( \lambda(f) : X \to B^A \) that sends an element \( x \in X \) to the function \( a \mapsto f(x, a) \).

**Definition 4.1.12.** Suppose that for every pair of objects \( A, B \) in \( C \) we have chosen an exponential \( A \Rightarrow B \) of \( A \) and \( B \) (note that this also requires choices for products). Then we can define the **exponential functor** \(- \Rightarrow - : C^{op} \times C \to C\) that sends a pair of objects \((A, B)\) to \( A \Rightarrow B \) and a pair of morphisms \((f : A \to A', g : B \to B')\) to

\[
f \Rightarrow g = \lambda(g \circ \text{ev}_{A,B} \circ (1 \times f)) : A \Rightarrow B \Rightarrow A' \Rightarrow B'.
\]

**Proposition 4.1.13.** The operations defined in **Definition 4.1.12** constitute a functor.

**Definition 4.1.14.** Let \( f : A \to C \) and \( g : B \to C \) be morphisms in \( C \). Then a **pullback** of \( f \) and \( g \) is an object \( P \) together with morphisms \( p_1 : P \to A \), \( p_2 : P \to B \) satisfying the following universal property: for every pair of morphisms \( a : X \to A \), \( b : X \to B \) such that \( fa = gb \), there exists a unique morphism \( h : X \to P \) such that

\[
p_1 \circ h = a \quad \text{and} \quad p_2 \circ h = b.
\]

Diagrammatically:

\[
\begin{array}{ccc}
X \\
\downarrow \exists h \\
P \\
\downarrow p_1 \quad \downarrow p_2 \\
A \quad B \\
\downarrow f \quad \downarrow g \\
\quad C
\end{array}
\]

The notation and terminology of pullbacks is similar to that of products. If \((P, p_1, p_2)\) is a pullback of \( f : A \to C \) and \( g : B \to C \), then the maps \( p_1 \) and \( p_2 \) are called **projections**. In the literature, the object \( P \) itself is often referred to as the pullback of \( f \) and \( g \), leaving the projections implicit; we follow this convention. Given maps \( a : X \to A \), \( b : X \to B \), the unique morphism arising from the universal property is written as \( \langle a, b \rangle : X \to P \) and is called the **pairing** of \( a \) and \( b \).
Example 4.1.15. In the category Set, the pullback of functions \( f : A \to C \) and \( g : B \to C \) is the set 
\[
P = \{(x, y) \in A \times B \mid f(x) = g(y)\}
\]
together with the projection functions \( \text{fst}_{A,B} \) and \( \text{snd}_{A,B} \) restricted to \( P \). Given a commuting square
\[
\begin{array}{ccc}
X & \xrightarrow{b} & B \\
\downarrow^{a} & & \downarrow^{g} \\
A & \xrightarrow{f} & C
\end{array}
\]
the pairing \( \langle a, b \rangle : X \to A \times B \) factors through \( P \), i.e. we have \( \langle a, b \rangle(x) \in P \) for all \( x \in X \). This is because \( \langle a, b \rangle(x) = (a(x), b(x)) \) and \( f(a(x)) = (fa)(x) = (gb)(x) = g(b(x)) \) for all \( x \in X \) due to the commutativity of the square above.

4.2 Cartesian closed categories and functors

Definition 4.2.1. A cartesian closed category (or CCC for short) is a category equipped with

- a choice of a terminal object \( 1 \),
- an operation which sends a pair of objects \( A \) and \( B \) to a product \((A \times B, \text{fst}_{A,B}, \text{snd}_{A,B})\) of \( A \) and \( B \), and
- an operation which sends a pair of objects \( A \) and \( B \) to an exponential \((A \Rightarrow B, \text{ev}_{A,B})\) of \( A \) and \( B \).

We often omit the objects in the subscripts of the projections and the evaluation when they are understood from context.

Remark 4.2.2. Definition 4.2.1 defines a CCC as a category equipped with the additional data of choices of a terminal object and products and exponentials for every pair of objects. That is, a CCC is a 4-tuple \((\mathcal{C}, 1, - \times -, - \Rightarrow -)\). In the literature, another common definition is to require the mere existence of a terminal object, products, and exponentials, without fixing any particular choices. To elucidate the distinction, we may refer to a CCC in the former sense as a CCC with structure and to a CCC in the latter sense as a CCC with property. The unqualified term CCC refers to a CCC with structure.

Assuming the axiom of choice, the two definitions are equivalent since it is always possible to choose products, exponentials, or other structures defined by universal properties. In constructive mathematics, however, it is often easier to carry around a choice of such structures together with the categories. For instance, to define the product functor \(- \times - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}\) (Definition 4.1.8), it is necessary to have a choice of a product for every pair of objects in \( \mathcal{C} \).

The distinction between structure and property also becomes important (even in the classical setting) once one considers the notion of morphism between structured categories. In particular, one may consider (at least) two kinds of morphisms between CCCs: strict morphisms which preserve the additional structure on the nose, i.e. up to equality, and weak morphisms which preserve it only up to isomorphism. A more detailed explanation can be found in Remark 4.2.6.

Example 4.2.3. The category Set is cartesian closed. This follows from examples 4.1.2, 4.1.7, and 4.1.11.

Definition 4.2.4. Let \( \mathcal{C} \) and \( \mathcal{D} \) be cartesian closed categories and let \( F : \mathcal{C} \to \mathcal{D} \) be a functor. We say that
Definition 4.2.5. A functor $\mathcal{C} \to \mathcal{D}$ up to isomorphism theory, whereby the structure is only preserved with structure since we need to work with strict morphisms in Chapter 6.

Case of products, we could weaken the condition up to equality preserved.

4.2.6 Remark closed if it strictly preserves the terminal object, products, and exponentials.

1 Proposition 4.2.7. Let $\mathcal{C} \to \mathcal{D}$ be a strict cartesian closed functor. Then

(i) $F$ strictly preserves the terminal object if $F(1_{\mathcal{C}}) = 1_{\mathcal{D}}$;
(ii) $F$ strictly preserves products if

$$F(A \times B) = FA \times FB, \quad F(\text{fst}_{A,B}) = \text{fst}_{FA,FB}, \quad \text{and} \quad F(\text{snd}_{A,B}) = \text{snd}_{FA,FB}$$

for all $A, B \in \mathcal{C}$;
(iii) $F$ strictly preserves exponentials if

$$F(A \Rightarrow B) = FA \Rightarrow FB \quad \text{and} \quad F(\text{ev}_{A,B}) = \text{ev}_{FA,FB}$$

for all $A, B \in \mathcal{C}$.

Definition 4.2.5. A functor $F : \mathcal{C} \to \mathcal{D}$ between CCCs is called strict cartesian closed if it strictly preserves the terminal object, products, and exponentials.

Remark 4.2.6. The adjective strict in Definition 4.2.4 indicates that the structure is preserved up to equality: the chosen structure on $\mathcal{C}$ is mapped to the chosen structure on $\mathcal{D}$. Note that this notion of structure preserving morphism does not make sense when $\mathcal{C}$ or $\mathcal{D}$ does not carry a chosen structure. This is our main motivation for adopting CCCs with structure since we need to work with strict morphisms in Chapter 6.

There is another, more widespread, notion of preservation in the literature on category theory, whereby the structure is only preserved up to isomorphism. For instance, in the case of products, we could weaken the condition $F(A \times B) = FA \times FB$ to $F(A \times B) \cong FA \times FB$. We refer to this idea as weak preservation to distinguish it from the strict notion. An advantage of weak preservation is that it is possible to generalize it to the case when $\mathcal{C}$ or $\mathcal{D}$ does not have all (chosen) products; see Definition 4.2.10.

The notions in Definition 4.2.4 also make sense when $\mathcal{C}$ and $\mathcal{D}$ are not necessarily cartesian closed. For instance, if $\mathcal{C}$ and $\mathcal{D}$ only have chosen terminal objects $1_{\mathcal{C}}$ and $1_{\mathcal{D}}$, we still say that $F$ strictly preserves the terminal object whenever $F(1_{\mathcal{C}}) = 1_{\mathcal{D}}$.

Similarly, we only need to assume the existence of chosen products to state that a functor strictly preserves them.

From the universal properties of products and exponentials, it follows that a strict cartesian closed functor strictly preserves pairing and currying too.

Proposition 4.2.7. Let $F : \mathcal{C} \to \mathcal{D}$ be a strict cartesian closed functor. Then

(i) $F(!_X) = !_F X$ for all $X \in \mathcal{C}$;
(ii) $F(\langle f, g \rangle) = \langle Ff, Fg \rangle$ for all $f : X \to A$ and $g : X \to B$;
(iii) $F(f \times g) = Ff \times Fg$ for all $f : A \to A'$ and $g : B \to B'$;
(iv) $F(\lambda(f)) = \lambda(F f)$ for all $f : X \times A \to B$.

Proof. (i) The equality follows since both sides are maps into the terminal object $F(1_{\mathcal{C}}) = 1_{\mathcal{D}}$.
(ii) Note that we have

$$\text{fst}_{FA,FB} \circ F(\langle f, g \rangle) = F(\text{fst}_{A,B}) \circ F(\langle f, g \rangle) = F(\text{fst}_{A,B} \circ (f, g)) = Ff$$

and

$$\text{snd}_{FA,FB} \circ F(\langle f, g \rangle) = F(\text{snd}_{A,B}) \circ F(\langle f, g \rangle) = F(\text{snd}_{A,B} \circ (f, g)) = Fg.$$
We have
\[ \text{ev}_{F_A,F_B} \circ (F(\lambda(f)) \times 1_{F_A}) = F(\text{ev}_{A,B}) \circ (F(\lambda(f)) \times F(1_A)) \]
\[ = F(\text{ev}_{A,B}) \circ F(\lambda(f) \times 1_A) = F(\text{ev}_{A,B} \circ (\lambda(f) \times 1_A)) = F f. \]

Hence, by the universal property of exponentials, \( F(\lambda(f)) = \lambda(F f) \).

The class of strict cartesian closed functors includes the identity functors and is closed under composition. Hence:

**Definition 4.2.8.** Cartesian closed categories and strict cartesian closed functors between them form a category \( \text{CCC} \).

**Definition 4.2.9.** Let \( F : C \to D \) be a functor and suppose that \( T \) is a terminal object in \( C \). We say that \( F \) weakly preserves the terminal object \( T \) if \( FT \) is terminal in \( D \).

**Definition 4.2.10.** Let \( F : C \to D \) be a functor.

(i) Suppose that
\[ A \xleftarrow{p_1} P \xrightarrow{p_2} B \]
is a product of \( A \) and \( B \) in \( C \). We say that \( F \) weakly preserves this product if
\[ FA \xleftarrow{FP_1} FP \xrightarrow{FP_2} FB \]
is a product of \( FA \) and \( FB \) in \( D \).

(ii) We say that \( F \) weakly preserves products if \( F \) weakly preserves all products that exist.

**Notation 4.2.11.** If \( C \) and \( D \) have chosen terminal objects and the functor \( F : C \to D \) weakly preserves the terminal object, then the unique map
\[ !_{F(1_C)} : F(1_C) \to 1_D \]
is an isomorphism. We denote its inverse by \( !_{F(1_C)}^{-1} \).

**Notation 4.2.12.** If \( C \) and \( D \) have chosen products and \( F : C \to D \) is a functor, then there is a canonical map
\[ F(A \times B) \xrightarrow{(F(\text{fst}),F(\text{snd}))} FA \times FB \]
denoted by \( s_{A,B}^F \), which is natural in \( A \) and \( B \). Naturality means that if \( f : A \to A' \) and \( g : B \to B' \), then the square
\[ \begin{array}{ccc}
F(A \times B) & \xrightarrow{s_{A,B}^F} & FA \times FB \\
F(f \times g) & \downarrow & \downarrow F(f \times Fg) \\
F(A' \times B') & \xrightarrow{s_{A',B'}^F} & FA' \times FB'
\end{array} \]
commutes. If \( F \) weakly preserves products, then \( s_{A,B}^F \) is an isomorphism and we denote its inverse by \( (s_{A,B}^F)^{-1} \).
Definition 4.3.5. \( \mu \) exponential transpose \( \lambda \) transformations

Proposition 4.3.6. The category composition functor

Let

Definition 4.3.4. For any small category \( t \) we denote this canonical map by \( t^F_{A,B} \).

As usual, the subscripts and superscripts in \( s^F_{A,B} \) and \( t^F_{A,B} \) may be omitted.

4.3 Presheaves and representables

Given two categories \( C \) and \( D \), the collections of functors \( C \to D \) and natural transformations between them form a category \([C,D]\) called a functor category.

Definition 4.3.1. Let \( C \) be a small category.

(i) A presheaf on \( C \) is a functor \( C^{\text{op}} \to \text{Set} \).
(ii) The category of presheaves on \( C \) is the functor category \([C^{\text{op}}, \text{Set}]\) and is denoted by \( \text{PSh}(C) \).

Definition 4.3.2. For any small category \( C \), there is a functor

\( y : C \to \text{PSh}(C) \),

called the Yoneda-embedding, defined as follows. It sends an object \( C \in C \) to the presheaf \( y_C \) with \( y_C(X) = \text{Hom}(X,C) \) for each object \( X \in C \) and where for each morphism \( f : Y \to X \) the operation \( y_C(f) : y_C(X) \to y_C(Y) \) is given by \( y_C(f)(g) = g \circ f \). Furthermore, it sends a morphism \( g : C \to D \) to the natural transformation \( y_g : y_C \to y_D \) with components \( (y_g)_X : y_C(X) \to y_D(X) \) given by \( (y_g)_X(f) = g \circ f \).

Proposition 4.3.3. The Yoneda-embedding \( y \) is fully faithful.

Proof. For the proof, see [27]. \( \square \)

Definition 4.3.4. Let \( F : C \to D \) be a functor. The reindexing functor or precomposition functor \( F^* : \text{PSh}(D) \to \text{PSh}(C) \) maps a presheaf \( P : D^{\text{op}} \to \text{Set} \) to \( PF^{\text{op}} : C^{\text{op}} \to \text{Set} \) and it maps a natural transformation \( \mu : P \to Q \) to the whiskering \( \mu F^{\text{op}} : PF^{\text{op}} \to QF^{\text{op}} \).

Definition 4.3.5.

- The terminal presheaf \( 1 \) is defined by \( 1(C) = \{ * \} \) and \( 1(f) = 1_\{ * \} \).
- The product \( P \times Q \) of presheaves \( P \) and \( Q \) is pointwise: \( (P \times Q)(C) = P(C) \times Q(C) \) and \( (P \times Q)(f) = P(f) \times Q(f) \). The projections \( \text{fst}_{P,Q} : P \times Q \to P \) and \( \text{snd}_{P,Q} : P \times Q \to Q \) are also pointwise: \( \text{fst}_{P,Q} = \text{fst}_{P,C,Q} \) and \( \text{snd}_{P,Q} = \text{snd}_{P,C,Q} \).
- The exponential \( \Rightarrow Q \) of presheaves \( P \) and \( Q \) is defined as follows: \( (P \Rightarrow Q)(C) = \text{Hom}(y_C \times P, Q) \) and \( (P \Rightarrow Q)(f)(\sigma) = \sigma \circ (y_f \times 1_P) \). The evaluation \( \text{ev}_{P,Q} : (P \Rightarrow Q) \times P \to Q \) is given by \( \text{ev}_{P,Q}(\sigma, x) = \sigma_C(1_C, x) \).

Proposition 4.3.6. The category \( \text{PSh}(C) \) is cartesian closed with terminal object, products, and exponentials as in Definition 4.3.5.

Proof. For a full proof, we refer the reader to standard category theory literature, e.g. [28, 29]. Here, we simply note the following. The pairing \( \langle \mu, \nu \rangle : X \to P \times Q \) of natural transformations \( \mu : X \to P \) and \( \nu : X \to Q \) is pointwise: \( \langle \mu, \nu \rangle_C = \langle \mu_C, \nu_C \rangle \). The exponential transpose \( \lambda(\mu) : X \to P \Rightarrow Q \) of a natural transformation \( \mu : X \times P \to Q \) is given by \( \lambda(\mu)_C(x)_D(h,a) = \mu_D(P(h)(x), a) \). \( \square \)
Definition 4.3.7. Let $\mu : Q \to S$ and $\nu : R \to S$ be morphisms of presheaves. The pullback of $\mu$ and $\nu$ is the presheaf $P$ given by

$$PC = \{(x, y) \in QC \times RC \mid \mu_C(x) = \nu_C(y)\}$$

$$\quad (Pf)(x, y) = ((Qf)(x), (Rf)(y))$$

together with projections $p_1 : P \to Q$ and $p_2 : P \to R$ given by $(p_1)_C(x, y) = x$ and $(p_2)_C(x, y) = y$.

Proposition 4.3.8. The category $\text{PSh}(\mathcal{C})$ has pullbacks as given in Definition 4.3.7.

Proof. We again refer the reader to basic category theory literature, e.g. [26].

Note that the construction of pullbacks in $\text{PSh}(\mathcal{C})$ is also pointwise. That is, the pullback $P$ of $\mu : Q \to S$ and $\nu : R \to S$ in $\text{PSh}(\mathcal{C})$ evaluated at $C$ is the pullback of $\mu_C : QC \to SC$ and $\nu_C : RC \to SC$ in $\text{Set}$.

Proposition 4.3.9. The Yoneda-embedding weakly preserves the terminal object and products.

Proof. The proof follows from reformulating the universal property of terminal objects and products by stating that we have a natural bijection on the hom-sets.

Proposition 4.3.10. Let $F : \mathcal{C} \to \mathcal{D}$ be a functor. The reindexing functor $F^* : \text{PSh}(\mathcal{D}) \to \text{PSh}(\mathcal{C})$ weakly preserves the terminal object and products.

Proof. The reindexing functor $F^* : \text{PSh}(\mathcal{D}) \to \text{PSh}(\mathcal{C})$ has a left adjoint given by Kan-extension ([27], Chapter X, Section 4, Theorem 1). Hence, it weakly preserves limits, in particular, the terminal object and products.

Definition 4.3.11. Let $P : \mathcal{C}^{\text{op}} \to \text{Set}$ be a presheaf.

(i) We say that $P$ is representable if there exists an object $C \in \mathcal{C}$ such that $y_C \cong P$.

(ii) If $C \in \mathcal{C}$ and $\rho : y_C \to P$ is an isomorphism, then we say that the pair $(C, \rho)$ is a representation of $P$.

Note the difference between the two parts of Definition 4.3.11. A representation of a presheaf $P$ is extra structure, namely the pair $(C, \rho)$, attached to $P$. In contrast, representability is a property of a presheaf $P$, stating that there merely exists a representation of $P$. The distinction is analogous to the discussion regarding CCCs with structure versus CCCs with property (Remark 4.2.2).

Remark 4.3.12. There may be multiple representations of the same presheaf. However, it follows from the properties of the Yoneda-embedding (Proposition 4.3.3) that any two representations are isomorphic in an appropriate sense. Hence, if a presheaf is representable, then its representation is essentially unique, meaning unique up to isomorphism.

4.4 Comma categories

Definition 4.4.1. Let

$$\begin{array}{ccc}
A & \xrightarrow{F} & C \\
\downarrow & & \\
\mathcal{B} & \xrightarrow{\mathcal{G}} & \mathcal{C}
\end{array}$$

be a diagram of categories and functors. The comma category $F \downarrow G$ is defined as follows:
- objects are triples \((A, B, p)\) where \(A \in A\), \(B \in B\), and \(p : FA \to GB\);
- morphisms \((A, B, p) \to (A', B', p')\) are pairs \((f, g)\) with \(f : A \to A'\) and \(g : B \to B'\) such that the square

\[
\begin{array}{ccc}
FA & \xrightarrow{Ff} & FA' \\
\downarrow p & & \downarrow p' \\
GB & \xrightarrow{Gg} & GB'
\end{array}
\]

commutes;
- identities and composition are componentwise.

There is a projection functor \(P_{F,G} : F \downarrow G \to A\) sending \((A, B, p)\) to \(A\) and \((f, g)\) to \(f\). Similarly, \(Q_{F,G} : F \downarrow G \to B\) projects the components in \(B\). Furthermore, there is a natural transformation \(\theta_{F,G} : FP_{F,G} \to GQ_{F,G}\) given by \((\theta_{F,G})_{(A,B,p)} = p\). These objects can be organized into a diagram

\[
\begin{array}{ccc}
F \downarrow G & \xrightarrow{P_{F,G}} & A \\
\downarrow Q_{F,G} & & \downarrow \theta_{F,G} \\
B & \xrightarrow{G} & C
\end{array}
\]

There are important special cases of the comma category construction for which we introduce notation.

**Notation 4.4.2.**
- If \(F\) or \(G\) is the identity functor \(\text{Id}_C\), then we replace the name of the functor by the category \(C\). For instance, \(C \downarrow G\) stands for \(\text{Id}_C \downarrow G\).
- If the domain of \(F\) or \(G\) is the terminal category \(1\), then the functor can be identified with an object \(X\) in \(C\), and we use the object in place of the functor. For instance, \(F \downarrow X\) stands for \(F \downarrow G\) where \(G : 1 \to C\) sends the unique object of \(1\) to \(X\).

**Lemma 4.4.3.** Suppose \(C\) and \(D\) are categories with a terminal object and \(F : C \to D\) weakly preserves the terminal object. Then \((1_D, 1_C, 1^1_F)\) is a terminal object in \(D \downarrow F\).

**Proof.** Given an object \((X, A, p) \in D \downarrow F\), we have a morphism

\[
\begin{array}{ccc}
X & \xrightarrow{1_X} & 1 \\
\downarrow p & & \downarrow 1^1_F \\
FA & \xrightarrow{F(1_A)} & F1
\end{array}
\]

in \(D \downarrow F\). The square commutes since \(F1\) is terminal. The uniqueness of this morphism follows from the fact that its components are maps to terminal objects. \(\square\)

**Lemma 4.4.4.** Suppose \(C\) and \(D\) are categories with products and \(F : C \to D\) weakly preserves products. Then the product of \((X, A, p)\) and \((Y, B, q)\) in \(D \downarrow F\) is

\[
(X, A, p) \xleftarrow{(\text{fst}_X, Y, \text{fst}_A, h)} (X \times Y, A \times B, r) \xrightarrow{(\text{snd}_X, Y, \text{snd}_A, n)} (Y, B, q)
\]

where \(r\) is the composite

\[
X \times Y \xrightarrow{p \times q} FA \times FB \xrightarrow{s^3_{A,B}} F(A \times B)
\]
Proof. The first projection \((\text{fst}_{X,Y}, \text{fst}_{A,B})\) is a morphism in \(D \downarrow F\) since

\[
\begin{array}{ccc}
X \times Y & \xrightarrow{\text{fst}_{X,Y}} & X \\
p \times q & \downarrow & \\
FA \times FB & \xrightarrow{p} & FA \\
\end{array}
\]

commutes. The proof for the second projection \((\text{snd}_{X,Y}, \text{snd}_{A,B})\) is similar.

Now suppose we have two morphisms \((f, a) : (Z, C, u) \to (X, A, p)\) and \((g, b) : (Z, C, u) \to (Y, B, q)\) in \(D \downarrow F\), i.e. commuting squares

\[
\begin{array}{ccc}
Z & \xrightarrow{\ f\ } & X \\
\uparrow & & \downarrow p \\
FC & \xrightarrow{\Fa\ } & FA \\
\end{array} \quad \begin{array}{ccc}
Z & \xrightarrow{\ g\ } & Y \\
\uparrow & & \downarrow q \\
FC & \xrightarrow{\Fc\ } & FB \\
\end{array}
\]

The pairing \(\langle (f, a), (g, b) \rangle : (Z, C, u) \to (X \times Y, A \times B, r)\) is defined as \(\langle (f, g), \langle a, b \rangle \rangle\). This is a morphism since

\[
\begin{array}{ccc}
Z & \xrightarrow{\ (f,g)\ } & X \times Y \\
\uparrow & & \downarrow p \times q \\
FC & \xrightarrow{\Fa, \Fc\ } & FA \times FB \\
\end{array}
\]

commutes. Using the universal properties of \(X \times Y\) and \(A \times B\), it can be shown that it is the unique morphism satisfying the universal property of the product. \(\square\)

**Lemma 4.4.5.** Suppose \(C\) and \(D\) are categories with exponentials, \(D\) has chosen pullbacks, and \(F : C \to D\) weakly preserves products. Then the exponential of \((X, A, p)\) and \((Y, B, q)\) in \(D \downarrow F\) is

\[
(R, A \Rightarrow B, r) \times (X, A, p) \xrightarrow{(ev_{X,Y} (k \times 1), ev_{A,B})} (Y, B, q)
\]

where \(R, r,\) and \(k\) are given by the pullback diagram

\[
\begin{array}{ccc}
R & \xrightarrow{k} & X \Rightarrow Y \\
\downarrow r & & \downarrow 1 \Rightarrow q \\
F(A \Rightarrow B) & \xrightarrow{\La \cdot r} & FA \Rightarrow FB \\
\end{array} \quad \begin{array}{ccc}
& & \\
& & \downarrow p \Rightarrow q \\
& & FB \\
\end{array}
\]
Proof. First, we check that the evaluation is a morphism in $\mathcal{D} \downarrow F$, i.e. that the diagram

$$
\begin{array}{ccc}
R \times X & \xrightarrow{k \times 1} & (X \Rightarrow Y) \times X \\
\downarrow r \times p & & \downarrow \text{ev}_{X,Y} \\
F(A \Rightarrow B) \times FA & \xrightarrow{q} & F((A \Rightarrow B) \times A) \\
\downarrow s_{A,B,A}^{-1} & & \downarrow F(\text{ev}_{A,B}) \\
F((A \Rightarrow B) \times A) & \xrightarrow{F(\text{ev}_{A,B})} & FB
\end{array}
$$

commutes. This follows from the pullback diagram (4.1) by taking the exponential transposes of the two composites.

Next, suppose we have a morphism $(f, g) : (Z, C, u) \times (X, A, p) \to (Y, B, q)$ in $\mathcal{D} \downarrow F$, i.e. a commuting diagram

$$
\begin{array}{ccc}
Z \times X & \xrightarrow{f} & Y \\
\downarrow u \times p & & \downarrow q \\
FC \times FA & \xrightarrow{s_{C,A}^{-1}} & F(C \times A) \\
\downarrow F(\lambda g) & & \downarrow Fg \\
F((A \Rightarrow B) \times A) & \xrightarrow{F(\text{ev}_{A,B})} & FB
\end{array}
$$

Taking the exponential transposes of the two composites, using the naturality of $s^{-1}$, we get that the diagram

$$
\begin{array}{ccc}
Z & \xrightarrow{\lambda(f)} & X \Rightarrow Y \\
\downarrow u & & \downarrow 1 \Rightarrow q \\
FC & \xrightarrow{F(\lambda(g))} & F((A \Rightarrow B) \times A) \\
\downarrow F(\lambda(g)) & & \downarrow Fg \\
F((A \Rightarrow B) \times A) & \xrightarrow{F(\text{ev}_{A,B})} & FB
\end{array}
$$

commutes. Hence, applying the pullback property of (4.1), we obtain a morphism $\bar{f} : Z \to R$ such that $k \bar{f} = \lambda(f)$ and such that the diagram

$$
\begin{array}{ccc}
Z & \xrightarrow{f} & R \\
\downarrow u & & \downarrow r \\
FC & \xrightarrow{F(\lambda(g))} & F((A \Rightarrow B) \times A)
\end{array}
$$

commutes. This shows that $(\bar{f}, \lambda(g)) : (Z, C, u) \to (R, A \Rightarrow B, r)$ is a morphism in $\mathcal{D} \downarrow F$. Using the universal properties of $R, X \Rightarrow Y$, and $A \Rightarrow B$, it can be shown that this is the unique morphism satisfying the universal property of the exponential. 

**Proposition 4.4.6.** Let $C$ and $\mathcal{D}$ be cartesian closed categories and $F : C \to \mathcal{D}$ be a functor. Suppose $F$ weakly preserves the terminal object and products, and suppose $\mathcal{D}$ has chosen pullbacks. Then
(i) the comma category $\mathcal{D} \downarrow F$ is cartesian closed, and
(ii) the projection functor $Q_{\mathcal{D}, F} : \mathcal{D} \downarrow F \to \mathcal{C}$ is strict cartesian closed.

Proof. The first claim is a corollary of the previous three lemmas. The second claim follows immediately from the definitions of the terminal object, products, and exponentials in $\mathcal{D} \downarrow F$. \qed
Chapter 5

Categorical semantics for the simply typed \( \lambda \)-calculus

In Chapter 2 we discussed the syntax and semantics of simply typed \( \lambda \)-calculus in a traditional way. In this chapter, we will discuss the categorical semantics of simply typed \( \lambda \)-calculus.

To motivate the categorical semantics, we first look at a different version of the syntax. This is done in Section 5.1. There are two main differences between the new and the old syntax.

- In the new syntax, we use de Bruijn indices instead of variable names. The goal of de Bruijn indices is to have a variable numbering scheme which eliminates the need for \( \alpha \)-conversion. With this scheme, each occurrence of a variable is replaced by the number of \( \lambda \)-binders between the occurrence of the variable and the binder the variable refers to. For example, the term
  \[
  \lambda x^{\sigma \rightarrow \tau} \cdot \lambda y^{\sigma} \cdot x \cdot y
  \]

  is expressed as
  \[
  \lambda^{\sigma \rightarrow \tau} \cdot \lambda^{\sigma} \cdot 1 \cdot 0
  \]
  with de Bruijn indices.

- Substitutions are made explicit in the new syntax. That is, while in the old syntax, substitution was an operation defined on terms, in the new syntax, substitution in terms is a primitive term former. Accordingly, instead of defining a substitution as a mapping from variables to terms, there are rules for the formation of substitutions in the syntax. Moreover, there are two convertibility relations: one for terms, one for substitutions.

After presenting the new syntax, we discuss the categorical semantics of the simply typed \( \lambda \)-calculus, including notions of models and morphisms, in Section 5.2. We also give a general method to construct models from cartesian closed categories.

5.1 Simply typed \( \lambda \)-calculus à la de Bruijn with explicit substitutions

Types are as before (Definition 2.1.1):

\[
\sigma, \tau ::= \beta \mid \sigma \rightarrow \tau
\]
where $\beta$ ranges over $\Sigma$.

Contexts are defined as follows.

**Definition 5.1.1 (Context).** A **context** is a list of types.

**Notation 5.1.2.**
(i) We denote the set of contexts by $\text{Con}$ and range over its elements by $\Gamma, \Delta, \Theta, \Xi$.
(ii) The **empty context** is denoted by $[]$.
(iii) If $\Gamma \in \text{Con}$ and $\sigma \in \text{Ty}$, then we write $\Gamma, \sigma$ for the **extended context** obtained by appending $\sigma$ at the end of the list $\Gamma$.

Now we define terms and substitutions.

**Definition 5.1.3.** We generate two families $Tm(\Gamma; \sigma)$ and $\text{Sub}(\Delta; \Gamma)$ of sets, called **terms** and **substitutions**, respectively, by mutual induction. Terms are indexed in contexts and types, substitutions are indexed in pairs of contexts. The rules are displayed in Figure 5.1. We write $\Gamma \vdash t : \sigma$ for $t \in Tm(\Gamma; \sigma)$ and $\Delta \vdash \gamma : \Gamma$ for $\gamma \in \text{Sub}(\Delta; \Gamma)$.

**Figure 5.1: Terms and substitutions**

As mentioned before, the main difference between this syntax and the previous one is that the new syntax uses de Bruijn indices instead of variable names, and that substitutions are part of the rules for generating the syntax. Note however, that only the zero de Bruijn index $v^0$ is part of the generators for the syntax. The other de Bruijn indices are expressed in terms of this and the weakening substitution $p^\Gamma$ as follows.

**Definition 5.1.4 (De Bruijn indices).** We define a subset $\text{Var}(\Gamma; \sigma) \subseteq Tm(\Gamma; \sigma)$ of terms, called **de Bruijn indices**, by induction. The rules are displayed in Figure 5.2. We write $\Gamma \vdash v x : \sigma$ for $x \in \text{Var}(\Gamma; \sigma)$.

**Figure 5.2: Rules for de Bruijn indices**

Thus, a de Bruijn index is of the form $vp^n$ for some natural number $n$, where $p^n$ denotes an $n$-fold composition of $p$ of the appropriate types. Such a de Bruijn index points to the $n$-th free variable in the context, counting from zero.
Notation 5.1.5.  
(i) For $\Gamma \vdash t : \sigma$, let $t^* \in \text{Sub}(\Gamma; [\sigma])$ be the substitution $((\iota)_\Gamma, t)$.
(ii) For $\Gamma \vdash t : \sigma$, let $\langle t \rangle \in \text{Sub}(\Gamma; \Gamma, \sigma)$ denote the singleton substitution $(\text{id}_\Gamma, t)$.
(iii) For $\Delta \vdash \gamma : \Gamma$ and $\sigma \in \text{Ty}$, let $\gamma^\sigma \in \text{Sub}(\Delta, \sigma; \Gamma, \sigma)$ denote the lifted substitution $(\gamma \circ p^\sigma_\Delta, v^\sigma_\Delta)$.

The convertibility relation is defined similarly to the traditional presentation. However, we also need to define convertibility for substitutions.

Definition 5.1.6 (Convertibility). We generate two families of equivalence relations

$$\equiv_{\Gamma, \sigma} \subseteq \text{Tm}(\Gamma; \sigma) \times \text{Tm}(\Gamma; \sigma) \quad \text{and} \quad \equiv_{\Delta, \Gamma} \subseteq \text{Sub}(\Delta; \Gamma) \times \text{Sub}(\Delta; \Gamma)$$

by mutual induction. The rules are displayed in Figure 5.3. We write $\Gamma \vdash t = t' : \sigma$ for $(t, t') \in \equiv_{\Gamma, \sigma}$ and $\Delta \vdash \gamma = \gamma' : \Gamma$ for $(\gamma, \gamma') \in \equiv_{\Delta, \Gamma}$.

Finally, we define normal forms and neutral terms. These are defined similarly to the traditional presentation.

Definition 5.1.7. We define two families $\text{Ne}(\Gamma; \sigma)$ and $\text{Nf}(\Gamma; \sigma)$ of subsets of $\text{Tm}(\Gamma; \sigma)$, called neutral terms and normal forms, respectively, by mutual induction. The rules are displayed in Figure 5.4. We write $\Gamma \vdash \text{ne} m : \sigma$ for $m \in \text{Ne}(\Gamma; \sigma)$ and $\Gamma \vdash \text{nf} n : \sigma$ for $n \in \text{Nf}(\Gamma; \sigma)$.

For the gluing proof, we also need neutral substitutions, which are essentially lists of neutral terms. Note that a substitution is equivalently given by a list of terms.

Definition 5.1.8. A substitution is neutral if every term in it is neutral.

The identity substitution is convertible to a neutral by definition, and thus it is neutral as well.

Proposition 5.1.9. The identity substitution is neutral.

Proof. The identity substitution can be expressed as a list of variables, all of which are neutral terms.

5.2 Semantics

In this section, we introduce a notion of model for the simply typed $\lambda$-calculus which we call $\lambda$-domain, based on simply typed categories with families. We also define morphisms of such models, leading to the category of $\lambda$-domains. Next, we construct an initial model from the syntax of $\lambda$-calculus. Finally, we show how every cartesian closed category gives rise to a $\lambda$-domain.

5.2.1 Simply typed categories with families

Categories with families provide a nice categorical setting for dealing with type systems with variable binding, especially those with dependent types. They were originally invented by Dybjer [15] to model a basic framework for dependent types (essentially, Martin-Löf type theory without any type formers). The axioms of categories with families thus serve to formalize the notions of context, substitution, and variables corresponding to the judgmental framework of dependent type theories.

Categories with families can be seen as models of the structural components of dependent type theory. However, they lie rather close to the syntax of dependent type theory – specifically, Martin-Löf’s substitution calculus. Thus, they can act as an intermediary.
between syntax (formal systems) and semantics (categorical/algebraic notions). This
double role enables one to prove equivalences between certain type theories and some
classes of models for those type theories. For instance, see \[12\] for the case of Martin-Löf
type theory, and \[11\] for results about various simpler type systems.

In this section, we introduce a kind of semantic domain (Definition 5.2.9) in which to
interpret the simply typed \(\lambda\)-type theory, and \[11\] for results about various simpler type systems.

| REFL | \(\Gamma \vdash t : \sigma\) |
| TRANS | \(\Gamma \vdash t = u : \sigma\) |
| SYM | \(\Gamma \vdash t = u : \sigma\) |
| CONG-APP | \(\Gamma \vdash t = t' : \sigma \rightarrow \tau\) |
| CONG-LAM | \(\Gamma, \sigma \vdash t = t' : \tau\) |
| CONG-SUBST | \(\Gamma \vdash u = u' : \sigma\) |
| CONG-EXT | \(\Gamma \vdash t \cdot u = t' \cdot u' : \tau\) |
| CONG-COMP | \(\Gamma \vdash t \gamma = \gamma' : \sigma\) |
| CONG-FUN | \(\Gamma \vdash \lambda \gamma. t = \lambda \gamma'. t' : \sigma \rightarrow \tau\) |
| ETA | \(\Gamma \vdash t : \sigma \rightarrow \tau\) |
| EMP-ETA | \(\Gamma \vdash \gamma : []\) |
| ID-L | \(\Gamma \vdash \gamma : \Gamma\) |
| ID-R | \(\Gamma \vdash \lambda \delta. \text{tp} : \tau \vdash \delta : \tau\) |
| COMPO-ASSOC | \(\Gamma \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |
| PROJ | \(\Gamma \vdash \gamma : \Gamma\) |
| APP-SUBST | \(\Gamma \vdash t : \sigma \rightarrow \tau\) |
| LAM-SUBST | \(\Gamma, \sigma \vdash \gamma : \Gamma\) |
| SUBS-ID | \(\Gamma \vdash \gamma : \Gamma\) |
| LAM-SUBST | \(\Gamma, \sigma \vdash t : \tau\) |
| SUBS-COMP | \(\Gamma \vdash t : \sigma\) |
| EXT-ETA | \(\Gamma \vdash \gamma : \Gamma, \sigma\) |
| \(\Delta \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |
| \(\Theta \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |
| \(\Theta \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |
| \(\Theta \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |
| \(\Theta \vdash \gamma \sigma \text{ id} \Gamma = \gamma : \Gamma\) |

Figure 5.3: Equations between terms and substitutions
VAR
\[ \Gamma \vdash^\gamma x : \sigma \]
\[ \Gamma \vdash^\text{ne} x : \sigma \]

APP
\[ \frac{\Gamma \vdash^\text{ne} m : \sigma \rightarrow \tau \quad \Gamma \vdash^\text{nf} n : \sigma}{\Gamma \vdash^\text{nf} m \cdot n : \tau} \]

SHIFT
\[ \frac{\Gamma \vdash^\text{ne} m : \beta}{\Gamma \vdash^\text{nf} m : \beta} \quad (\beta \in \Sigma) \]

LAM
\[ \frac{\Gamma, \sigma \vdash^\text{nf} n : \tau}{\Gamma \vdash^\text{nf} \lambda^\sigma. n : \sigma \rightarrow \tau} \]

Figure 5.4: Normal forms and neutral terms

for a bare-bones simple type system with no type formers and only variables as terms.

As such, simply typed categories with families are a suitable basis for modelling more complex type systems. For instance, we shall see (Definition 5.2.5) how to incorporate function types, which are necessary to model \(\lambda\)-calculus.

For a more thorough introduction to categories with families in general, we refer the reader to [15] and [11].

Definition 5.2.1 (Scwf). A simply typed category with families, or scwf, consists of:

(i) a category \(\mathcal{C}\) with a terminal object \(1\),
(ii) a set \(\text{Ty}\),
(iii) for every \(A \in \text{Ty}\), a presheaf \(\text{Tm}_A : \mathcal{C}^{\text{op}} \rightarrow \text{Set}\), and
(iv) for every \(\Gamma \in \text{Con}\) and \(A \in \text{Ty}\), a representation of the presheaf \(y_\Gamma \times \text{Tm}_A : \mathcal{C}^{\text{op}} \rightarrow \text{Set}\).

Definition 5.2.1 is rather concise, but may be hard to understand for a reader not trained in category theory, also known as general abstract nonsense. It turns out that the data packed in the definition matches the syntax of simply typed \(\lambda\)-calculus presented in Section 5.1, excluding function types, application, and abstraction. Let us unpack the definition to illustrate this point. The notation and terminology regarding scwfs also help make the connection clearer.

(i) The category \(\mathcal{C}\) is called the base category of the scwf. Its objects are referred to as contexts, and its morphisms as substitutions. For this reason, we also write \(\text{Con}\) for \(\text{Ob}(\mathcal{C})\) and \(\text{Sub}(\Delta; \Gamma)\) for \(\mathcal{C}(\Delta, \Gamma)\). Since \(\mathcal{C}\) is a category, we have the usual composition and identity operations on substitutions.

The terminal object \(1\) is called the empty context and written as \(\bullet\). The unique map \(\Gamma \rightarrow \bullet\) is called the empty substitution and written as \(\eta_\Gamma\).

We use capital Greek letters (\(\Gamma, \Delta, \Theta, \ldots\)) to range over contexts, and lowercase Greek letters (\(\gamma, \delta, \theta, \ldots\)) to range over substitutions.

(ii) The elements of the set \(\text{Ty}\) are referred to as types. We use uppercase Latin letters (\(A, B, C, \ldots\)) to range over types.

(iii) For \(\Gamma \in \text{Con}\) and \(A \in \text{Ty}\), the elements of \(\text{Tm}_A(\Gamma)\) are called terms of type \(A\) in context \(\Gamma\). The set \(\text{Tm}_A(\Gamma)\) is also written as \(\text{Tm}(\Gamma; A)\).

For a substitution \(\gamma : \Delta \rightarrow \Gamma\), the functorial action of \(\text{Tm}_A\) gives a function

\[ \text{Tm}_A(\gamma) : \text{Tm}(\Gamma; A) \rightarrow \text{Tm}(\Delta; A) \]

referred to as substitution in terms. If \(t \in \text{Tm}(\Gamma; A)\), then we write \(t[\gamma]\) for \(\text{Tm}_A(\gamma)(t)\).

We use lowercase Latin letters (\(t, u, v, \ldots\)) to range over terms.
of the structure base category for the other components. For instance, Con and other parts of the structure implicit. If necessary for disambiguation, we indicate the

Thus, a representation of this presheaf amounts to a context $\Gamma \cdot A$ together with a substitution $p^\Gamma_A : \Gamma \cdot A \to \Gamma$ and a term $q^\Gamma_A \in \operatorname{Tm}(\Gamma \cdot A; A)$ satisfying the following universal property: for every context $\Delta$, substitution $\gamma : \Delta \to \Gamma$, and term $t \in \operatorname{Tm}(\Delta; A)$, there exists a unique substitution $\gamma.t : \Delta \to \Gamma \cdot A$ such that

$$p^\Gamma_A \circ (\gamma.t) = \gamma \quad \text{and} \quad q^\Gamma_A[\gamma.t] = t.$$  

We say that the triple $(\Gamma \cdot A, p^\Gamma_A, q^\Gamma_A)$ is a context comprehension of $\Gamma$ and $A$. We also say that $\Gamma \cdot A$ is obtained by extending the context $\Gamma$ with the type $A$.

We emphasize that the constants and operations $\bullet$, $\operatorname{Ty}$, $\operatorname{Tm}$, $\cdot$, $p$, $q$, and $\cdot$ are part of the structure of an scwf. Hence, an scwf is formally a tuple $(C, \bullet, \operatorname{Ty}, \operatorname{Tm}, \cdot, p, q, \cdot)$. In this thesis, however, we refer to scwfs only by their base category, and leave the other parts of the structure implicit. If necessary for disambiguation, we indicate the base category for the other components. For instance, $\operatorname{Con}^\Gamma$, $\operatorname{Sub}^C(\Delta; \Gamma)$, $\operatorname{Ty}^C$, and $\operatorname{Tm}^C(\Gamma; A)$ denote, respectively, the collections of contexts, substitutions, types, and terms of the scwf $C$. As before, we often omit superscripts and subscripts (e.g. in $p^\Gamma_A$) to simplify notation.

Example 5.2.2. We define the scwf $\operatorname{Set}$ of sets as follows.

- Its base category is the category $\operatorname{Set}$ of sets.
- Types are sets.
- Terms $\operatorname{Tm}_A$ are functions $\Gamma \to A$. Substitution in terms is given by precomposition of functions.
- Context comprehension is given by cartesian products of sets and the projection functions.

Remark 5.2.3. The notion of scwf can also be presented as a generalized algebraic theory [10]. The collections of contexts, substitutions, types, and terms become the sorts, and the empty context, empty substitution, substitution in terms, and context comprehension become the operations of the theory, subject to the equations arising from the functoriality of $\operatorname{Tm}_A$ and the universal properties of the terminal object and representability.

A full presentation of (dependently typed) cwfs as a generalized algebraic theory can be found in [13]. A similar presentation for scwfs can be obtained by removing the dependency of types on terms. The resulting formal theory is essentially the same as the syntax of simply typed $\lambda$-calculus presented in Section 5.1.4 (again, excluding functions).

The following notations are the semantic counterpart of Notation 5.1.5.

Notation 5.2.4. (i) For $t \in \operatorname{Tm}(\Gamma; A)$, let $t^\Gamma : \Gamma \to \bullet \cdot A$ be the substitution $\eta_T.t$.
(ii) For $t \in \operatorname{Tm}(\Gamma; A)$, let $\langle t \rangle : \Gamma \to \Gamma \cdot A$ denote the singleton substitution $\operatorname{id}_{\Gamma.t}$.
(iii) For $\gamma : \Delta \to \Gamma$ and $A \in \operatorname{Ty}$, let $\gamma^A : \Delta \cdot A \to \Gamma \cdot A$ denote the lifted substitution $(\gamma \circ p).q$.

To model type formers, we need to require additional structure on scwfs. This structure essentially consists of operations on types and terms, corresponding to the type
formation, introduction, and elimination rules of type systems, satisfying certain equations corresponding to $\beta$ and optionally $\eta$-rules for the type former. For our minimalistic simply typed $\lambda$-calculus presented in Section 5.1, it suffices to define what it means for an scwf to support function types.

**Definition 5.2.5** (Function-structure). A function-structure on an scwf $C$ consists of a type former $\rightarrow : Ty \times Ty \rightarrow Ty$ and for each $\Gamma \in C$ and $A, B \in Ty$, term formers $\Gamma, A, B \rightarrow : Tm(\Gamma; A \Rightarrow B) \times Tm(\Gamma; A) \rightarrow Tm(\Gamma; B)$ $\lambda A, B \Gamma (\rightarrow) : Tm(\Gamma \cdot A; B) \rightarrow Tm(\Gamma; A \Rightarrow B)$ satisfying the equations

$$\lambda A, B \Gamma (s) \kappa_{\Gamma, A, B} u = s[\langle u \rangle]$$

$$\lambda A, B \Gamma (t[\gamma]) \kappa_{\Gamma, A, B} u[\gamma] = t$$

$$\lambda A, B \Gamma (t \kappa_{\Gamma, A, B} u)[\gamma] = t \kappa_{\Delta, A, B} u[\gamma]$$

for all $s \in Tm(\Gamma \cdot A; B), t \in Tm(\Gamma; A \Rightarrow B), u \in Tm(\Gamma; A)$, and $\gamma : \Delta \rightarrow \Gamma$.

Often, we drop the subscripts of $\lambda A, B \Gamma (\rightarrow)$ and $\kappa_{\Gamma, A, B}$ to improve readability.

In Definition 5.2.5, the operation $\kappa_{\Gamma, A, B}$ corresponds to function application in $\lambda$-calculus, and $\lambda A, B \Gamma (\rightarrow)$ corresponds to $\lambda$-abstraction. The first and second equations correspond to the $\beta$ and $\eta$-rules, respectively. The third equation describes how to perform a substitution in an application.

There is also a substitution law for abstraction: for each $t \in Tm(\Gamma \cdot A; B)$ and $\gamma : \Delta \rightarrow \Gamma$, we have

$$\lambda A, B \Gamma (t[\gamma]) = \lambda A, B \Gamma (t[\gamma A]).$$

This law can be derived from the axioms of scwfs and function structures. Compare these equations with the corresponding rules in Figure 5.3 (beta, eta, app-subst, and lam-subst).

A function-structure on an scwf $C$ is a tuple $(\rightarrow, \kappa, \lambda(\rightarrow))$. To simplify notation, we usually omit the explicit reference to the function structure.

**Example 5.2.6.** We define the function-structure on the scwf of sets as follows. The function type $A \Rightarrow B$ is given by the set of functions $B^A$. The operations $\lambda A, B \Gamma (\rightarrow)$ and $\kappa_{\Gamma, A, B}$ are given by currying and function application, respectively.

We now introduce a notion of morphism between scwfs.

**Definition 5.2.7** (Strict scwf-morphism). Let $C$ and $D$ be scwfs. A strict scwf-morphism from $C$ to $D$ consists of

(i) a functor $F : C \rightarrow D$ between the base categories,

(ii) a function $T : Ty^C \rightarrow Ty^D$, and

(iii) for each $A \in Ty^C$, a natural transformation $\tau_A : Tm^C_A \rightarrow Tm^D_{TA} \circ F$ such that $F$ strictly preserves the empty context and $\tau$ strictly preserves context comprehension.

Again, the compactness of the categorical definition above might obscure the intuitive idea behind strict scwf-morphisms. Thus, we spell out Definition 5.2.7 in detail while also introducing notation for the components of a strict scwf-morphism.
(i) We have a mapping $F^\text{Con} : \text{Con}^C \to \text{Con}^D$ of contexts, corresponding to the object part of the functor $F : C \to D$. The morphism part of $F$ is given by mappings $F^\text{Sub}_{\Delta; \Gamma} : \text{Sub}^C(\Delta; \Gamma) \to \text{Sub}^D(F^\text{Con}_{\Delta}; F^\text{Con}_{\Gamma})$ of substitutions for each $\Delta, \Gamma \in \text{Con}^C$.

The functoriality of $F$ means that composition and identities are preserved: we have

$$F^\text{Sub}_{\Theta, \Gamma}(\gamma \circ \delta) = F^\text{Sub}_{\Delta, \Gamma}(\gamma) \circ F^\text{Sub}_{\Theta, \Delta}(\delta)$$

for all $\gamma \in \text{Sub}^C(\Delta; \Gamma), \delta \in \text{Sub}^C(\Theta; \Delta)$, and

$$F^\text{Sub}_{\Gamma, \Gamma}(\text{id}_\Gamma) = \text{id}_{F^\text{Con}_{\Gamma}}$$

for all $\Gamma \in \text{Con}^C$.

(ii) The function $T : \text{Ty}^C \to \text{Ty}^D$ provides a mapping from the types of $C$ to the types of $D$ and is written as $F^\text{Ty}$.

(iii) For each $A \in \text{Ty}^C$, the natural transformation $\tau_A : \text{Ty}^C_A \to \text{Ty}^D_{F^A}$ is denoted by $F^\text{Tm}_A$. Its components $(\tau_A)_\Gamma : \text{Ty}^C(\Gamma; A) \to \text{Ty}(F^\text{Con}_{\Gamma}; F^\text{Ty} A)$ for $\Gamma \in \text{Con}^C$ are mappings of terms and are denoted by $F^\text{Tm}_{\Gamma, A}$. Naturality of $F^\text{Tm}_A$ amounts to preservation of substitution in terms. That is, for all $t \in \text{Ty}^C(\Gamma; A)$ and $\gamma \in \text{Sub}^C(\Delta; \Gamma)$, we have

$$F^\text{Tm}_{\Delta, A}(t[\gamma]) = F^\text{Tm}_{\Gamma, A}(t)[F^\text{Sub}_{\Delta, \Gamma}(\gamma)].$$

(iv) Strict preservation of the empty context means that $F^\text{Con}(\bullet^C) = \bullet^D$.

(v) Finally, strict preservation of context comprehension means that

$$F^\text{Con}(\Gamma \cdot A) = F^\text{Con}_{\Gamma} \cdot F^\text{Ty} A$$

$$F^\text{Sub}_{\Gamma, \Gamma A}(p_A) = p_{F^\text{Con} \Gamma}$$

$$F^\text{Tm}_{\Gamma, A}(q_A) = q_{F^\text{Con} \Gamma}.$$

We can also rephrase this condition by saying that, for all $\Gamma \in \text{Con}^C$ and $A \in \text{Ty}^C$, the triple

$$(F^\text{Con}_{\Gamma} \cdot A, F^\text{Sub}_{\Gamma, \Gamma A}(p_A), F^\text{Tm}_{\Gamma, A}(q_A))$$

is a context comprehension of $F^\text{Con}_{\Gamma} \in \text{Con}^D$ and $F^\text{Ty} A \in \text{Ty}^D$.

Similarly to scwfs, scwf-morphisms have several components. Formally, an scwf-morphism from $C$ to $D$ is a tuple $(F, T, \tau)$. By convention, we use the symbol for the base functor to denote all three components of an scwf-morphism. However, as demonstrated above, the notation can get quite verbose. To simplify notation, we drop the superscripts and subscripts most of the time, since they can usually be inferred from the argument of $F$.

Note that our notion of scwf-morphism is exactly the same as one would expect when viewing scwfs as generalized algebraic structures: they are structure preserving mappings between the sorts of the structures. If one thinks about the sorts (contexts, substitutions, types, terms) syntactically, then a good intuition for scwf-morphisms is that they are syntactic translations from one theory to another one.

**Definition 5.2.8.** Suppose $C$ and $D$ are scwfs with function-structures. An scwf-morphism $F : C \to D$ is said to **strictly preserve the function-structure** if

$$F(A \Rightarrow B) = FA \Rightarrow FB$$

for all $\Gamma \in \text{Con}^C, A, B \in \text{Ty}^C, t \in \text{Tm}(\Gamma; A \Rightarrow B)$, and $u \in \text{Tm}(\Gamma; A)$.
In Definition 5.2.8 we only require $F$ to preserve the application operation. By the axioms of scwfs and function-structures, this already implies that it also preserves abstraction. That is, we have

$$F(\lambda^A_{\Gamma,B}(t)) = \lambda^{FA}_{\Gamma,F\cdot B}(Ft)$$

for all $t \in \text{Term}((\cdot; A; B))$.

Scwfs equipped with a function structure provide the main notion of semantic domain for the simply typed $\lambda$-calculus in this and the next chapter. Hence, we introduce the following terminology.

**Definition 5.2.9.** (i) A $\lambda$-domain is an scwf equipped with a function-structure.

(ii) A morphism of $\lambda$-domains is an scwf-morphism preserving the function-structure.

It is easy to see that scwf-morphisms are componentwise composable, and that the identity morphism at each component gives an identity scwf-morphism on each scwf. Moreover, the property of preservation of function-structure is stable under composition, and the identity scwf-morphisms trivially preserve the function-structure. Hence:

**Definition 5.2.10.** $\lambda$-domains and their morphisms form a category $\lambda$-Dom.

### 5.2.2 Abstract syntax and models

The goal of this section is to show that there exists a syntactic $\lambda$-domain (Definition 5.2.15) which satisfies the universal property of being a free $\lambda$-domain over the set of base types $\Sigma$ (Theorem 5.2.16). We then define a notion of model for the simply typed $\lambda$-calculus (Definition 5.2.17) and show that there is a syntactic model (Definition 5.2.20) which is initial in the category of models (Theorem 5.2.21).

As a first step, we discuss how to interpret the syntax of simply typed $\lambda$-calculus presented in Section 5.1 in a $\lambda$-domain $C$. To do this, we need to choose an interpretation $J : \Sigma \to \text{Ty}^C$ for the base types. Then, we can extend the interpretation $J$ to an interpretation $[\cdot]_J$ of the syntax of $\lambda$-calculus by structural recursion on the syntax. The definitions are entirely straightforward: we simply replace each syntactic operation by the corresponding semantic operation. We spell out the interpretation in steps for the different kinds of syntactic entities.

**Definition 5.2.11** (Interpretation of $\lambda$-calculus in a $\lambda$-domain). Let $C$ be a $\lambda$-domain and let $J : \Sigma \to \text{Ty}^C$ be an interpretation for the base types. The interpretation of $\lambda$-calculus in $C$ with respect to $J$ is defined as follows.

(i) The interpretation of types is defined by structural recursion on types:

$$[\cdot]_{\text{Ty}}^J : \text{Ty} \to \text{Ty}^C$$

$$[\beta]_{\text{Ty}}^J = J(\beta) \quad (\beta \in \Sigma)$$

$$[\sigma \to \tau]_{\text{Ty}}^J = [\sigma]_{\text{Ty}}^J \Rightarrow [\tau]_{\text{Ty}}^J$$

(ii) The interpretation of contexts is defined by structural recursion on lists:

$$[\cdot]_{\text{Con}}^J : \text{Con} \to \text{Con}^C$$

$$[[\cdot]]_{\text{Con}}^J = \bullet^C$$

$$[\Gamma, \sigma]_{\text{Con}}^J = [\Gamma]_{\text{Con}}^J \cdot [\sigma]_{\text{Ty}}^J$$

49
(iii) The interpretation of terms and substitutions is defined by mutual recursion on the generating rules for terms and substitutions (see Figure 5.1).

\[ \llbracket \cdot \rrbracket^\text{Tm}_J : \text{Tm}(\Gamma; \sigma) \to \text{Tm}^\mathcal{C}(\llbracket \Gamma \rrbracket^\text{Con}_J; \llbracket \sigma \rrbracket^\text{Ty}_J) \]

\[ \llbracket \cdot \rrbracket^\text{Sub}_J : \text{Sub}(\Delta; \Gamma) \to \text{Sub}^\mathcal{C}(\llbracket \Delta \rrbracket^\text{Con}_J; \llbracket \Gamma \rrbracket^\text{Con}_J) \]

\[ \llbracket v \rrbracket^\text{Tm}_J = \text{q}_{\text{Tm}}^\mathcal{C}(\llbracket v \rrbracket^\text{Ty}_J) \]

\[ \llbracket t \cdot u \rrbracket^\text{Tm}_J = \text{Product}^\mathcal{C}(\llbracket t \rrbracket^\text{Tm}_J; \llbracket u \rrbracket^\text{Tm}_J) \]

\[ \llbracket \lambda^A. t \rrbracket^\text{Tm}_J = \lambda^A(\llbracket t \rrbracket^\text{Tm}_J) \]

\[ \llbracket (\gamma, t) \rrbracket^\text{Sub}_J = \text{Product}^\mathcal{C}(\llbracket \gamma \rrbracket^\text{Sub}_J; \llbracket t \rrbracket^\text{Tm}_J) \]

\[ \llbracket \text{id}_1 \rrbracket^\text{Sub}_J = \text{id}_{\text{Tm}}^\mathcal{C}(\llbracket \cdot \rrbracket^\text{Ty}_J) \]

\[ \llbracket \gamma \circ \delta \rrbracket^\text{Sub}_J = \text{Product}^\mathcal{C}(\llbracket \gamma \rrbracket^\text{Sub}_J; \llbracket \delta \rrbracket^\text{Sub}_J) \]

\[ \llbracket p_\alpha \rrbracket^\text{Sub}_J = \text{Product}^\mathcal{C}(\llbracket \Gamma \rrbracket^\mathfrak{G}_\alpha^\text{Con}) \]

To improve readability, we often drop the superscripts from \( \llbracket \cdot \rrbracket^\text{Tm}_J \), \( \llbracket \cdot \rrbracket^\text{Con}_J \), etc. and write \( \llbracket \cdot \rrbracket_J \) uniformly for all the interpretation functions.

An important property of the interpretation is that convertible terms and substitutions are mapped to the same semantic object. This property is referred to as soundness.

**Theorem 5.2.12 (Soundness of the interpretation).** Let \( \mathcal{C} \) be a \( \lambda \)-domain and \( J : \Sigma \to \text{Ty}^\mathcal{C} \) an interpretation for the base types.

(i) For all \( t \) and \( t' \), if \( \Gamma \vdash t = t' : \sigma \), then \( \llbracket t \rrbracket_J = \llbracket t' \rrbracket_J \in \text{Tm}^\mathcal{C}(\llbracket \Gamma \rrbracket^\text{Con}_J; \llbracket \sigma \rrbracket^\text{Ty}_J) \).

(ii) For all \( \gamma \) and \( \gamma' \), if \( \Delta \vdash \gamma = \gamma' : \Gamma \), then \( \llbracket \gamma \rrbracket_J = \llbracket \gamma' \rrbracket_J \in \text{Sub}^\mathcal{C}(\llbracket \Delta \rrbracket^\text{Con}_J; \llbracket \Gamma \rrbracket^\text{Con}_J) \).

**Proof.** Both statements are proved simultaneously by mutual induction on the derivation of \( \Gamma \vdash t = t' : \sigma \), respectively \( \Delta \vdash \gamma = \gamma' : \Gamma \). We omit the unsurprising details. \( \square \)

The next step is to define the *syntactic \( \lambda \)-domain*. We do this in stages corresponding to the structural complexity of \( \lambda \)-domains. First, we define the so called *syntactic category* (Definition 5.2.13), which serves as the base category for the syntactic \( \lambda \)-domain. Next, we endow the syntactic category with the structure of an scwf to obtain the *syntactic scwf* (Definition 5.2.14). Finally, we equip the syntactic scwf with the *syntactic function-structure* resulting in the syntactic \( \lambda \)-domain (Definition 5.2.15).

**Definition 5.2.13 (Syntactic category).** The *syntactic category* \( \mathcal{L} \) of \( \lambda \)-calculus is defined as follows.

(i) Its objects are syntactic contexts, i.e. lists of types.

(ii) For \( \Gamma, \Delta \in \mathcal{L} \), \( \mathcal{L}(\Delta, \Gamma) \) is the set of syntactic substitutions \( \text{Sub}(\Delta; \Gamma) \) quotiented by the convertibility relation \( \equiv_{\Delta; \Gamma} \). That is,

\[ \mathcal{L}(\Delta, \Gamma) = \{ [\gamma] \mid \Delta \vdash \gamma : \Gamma \} \]

where

\[ [\gamma] = \{ \Delta \vdash \gamma' : \Gamma \mid \Delta \vdash \gamma = \gamma' : \Gamma \} \]

denotes the equivalence class of \( \Delta \vdash \gamma : \Gamma \).

50
(iii) Composition of substitutions is done on representatives of equivalences classes:

\[ [\gamma] \circ^L [\delta] = [\gamma \circ \delta] \]

for \( \Delta \vdash \gamma : \Gamma \) and \( \Theta \vdash \delta : \Delta \). This operation is well-defined since, by the congruence rule cong-comp (Figure 5.3), if \( \gamma' \) and \( \delta' \) are also representatives of the equivalences classes \([\gamma]\) and \([\delta]\), respectively, then \([\gamma \circ \delta] = [\gamma' \circ \delta']\).

(iv) The identity substitution of \( L \) on the context \( \Gamma \) is the equivalence class of the syntactic identity substitution on \( \Gamma \):

\[ \text{id}_L^\Gamma = [\text{id}_\Gamma] \]

The category axioms are satisfied due to the associativity (comp-assoc) and identity (id-l, id-r) laws of the syntax (Figure 5.3).

To summarize the construction of Definition 5.2.13, the syntactic category \( L \) is obtained by using corresponding syntactic objects and operations for each component of the category. That is, contexts are syntactic contexts, substitutions are syntactic substitutions, etc. However, there is a subtlety involved for morphisms: instead of using the substitutions themselves, we quotient them by convertibility, thereby identifying convertible substitutions. Taking the quotient is necessary to satisfy the category axioms: without quotienting, they would only hold up to convertibility instead of equality.

The downside of quotienting is that all operations defined on equivalence classes must be proved to be well-defined, that is, independent of the chosen representatives. In the syntactic category, this was ensured by the congruence law cong-comp. In the rest of this section, we omit such verifications, as they all follow immediately from one of the congruence rules (cong-app, cong-lam, cong-subst, cong-ext, cong-comp in Figure 5.3) for the syntax.

Definition 5.2.14 (Syntactic scwf). The syntactic scwf has as base category the syntactic category \( L \). The remaining structure is defined as follows.

(i) The terminal object \( \bullet^L \) is the empty list \([\,]\) and the unique map \( \eta^L : \Gamma \to [\,] \) is the equivalence class of the empty tuple \((\,\rangle_\Gamma\). (Note that this equivalence class is a singleton set since \((\,\rangle_\Gamma\) is the unique element of \(\text{Sub}(\Gamma|[\,])\).)

(ii) The set \( \text{Ty}^L \) is the set \( \text{Ty} \) of types of the simply typed \( \lambda \)-calculus.

(iii) Given \( \sigma \in \text{Ty} \), the presheaf \( \text{Tm}^L_\sigma \) is defined as follows. It sends a context \( \Gamma \) to the set of syntactic terms \( \text{Tm}_\Gamma(\Gamma;\sigma) \) quotiented by the convertibility relation \( \equiv_{\Gamma,\sigma}^\text{Tm} \), and it sends an equivalence class \([\gamma]:\Delta \to \Gamma\) of substitutions to the function:

\[ [t] \mapsto [t\gamma], \text{ where } \Delta \vdash \gamma : \Gamma \text{ and } \Gamma \vdash t : \sigma. \]

The presheaf axioms are satisfied due to the rules subst-id and subst-comp (Figure 5.3).

(iv) Context comprehension is given by context extension, and the operations on substitution are defined by the analogous operations on the syntax.

Definition 5.2.15 (Syntactic \( \lambda \)-domain). The syntactic function-structure on the syntactic scwf \( L \) is defined as follows.

(i) We define \( \sigma \Rightarrow^L \tau \) to be \( \sigma \to \tau \).

(ii) We define \([t]\downarrow^L_{\Gamma,\sigma,\tau} [u]\) to be \([t \cdot u]\).

(iii) Finally, we define \( \lambda^L_{\Gamma,\tau}([t]) \) to be \([\lambda^\tau. t]\).

The syntactic \( \lambda \)-domain is the syntactic scwf \( L \) equipped with the syntactic function-structure.
We are now ready to state the universal property of the syntactic $\lambda$-domain.

**Theorem 5.2.16.** For every $\lambda$-domain $C$ and function $J : \Sigma \rightarrow Ty^C$, there is a unique morphism of $\lambda$-domains $\llbracket - \rrbracket_J : L \rightarrow C$ such that $\llbracket \beta \rrbracket_J = J(\beta)$ for all $\beta \in \Sigma$.

**Proof.** The components of the desired morphism of $\lambda$-domains $\llbracket - \rrbracket_J$ are given by the interpretation functions of Definition 5.2.11. These definitions on the equivalence classes of terms and substitutions are well-defined due to the soundness of the interpretation (Theorem 5.2.12). The structure of $\lambda$-domains is preserved by the definition of $\llbracket - \rrbracket_J$. Uniqueness of the morphism follows by induction on the syntax of the $\lambda$-calculus given in Section 5.1. □

In categorical terms, the above theorem states that $L$ is a free $\lambda$-domain over the set $\Sigma$ of base types. We can rephrase the result in terms of initiality. For this, we introduce a category of models (Definition 5.2.19) and show that there is a syntactic model which is an initial object in this category.

**Definition 5.2.17** (Model). A model of simply typed $\lambda$-calculus is a $\lambda$-domain $C$ together with an interpretation function $J : \Sigma \rightarrow Ty^C$ for the base types.

**Definition 5.2.18** (Model morphism). Let $(C, J)$ and $(D, K)$ be models. A model morphism from $(C, J)$ to $(D, K)$ is a morphism of $\lambda$-domains $F : C \rightarrow D$ such that $F(J(\beta)) = K(\beta)$ for all $\beta \in \Sigma$.

Obviously, the composite of model morphisms is a model morphism, and the identity morphism on each $\lambda$-domain is a model morphism. Hence:

**Definition 5.2.19.** Models and model morphisms form a category $\text{Mod}$.

**Definition 5.2.20.** (i) The canonical interpretation $I : \Sigma \rightarrow Ty^L$ of base types is the inclusion of $\Sigma$ into $Ty$.

(ii) The syntactic model is the syntactic $\lambda$-domain $L$ together with the canonical interpretation $I$ of base types.

**Theorem 5.2.21.** The syntactic model $(L, I)$ is an initial object in $\text{Mod}$.

**Proof.** This theorem is a rephrasing of Theorem 5.2.16. □

### 5.2.3 Semantics in cartesian closed categories

In this section, we show that cartesian closed categories provide a wide range of models for the simply typed $\lambda$-calculus. Instead of defining the interpretation directly, we use the framework of the previous section.

**Proposition 5.2.22.** We have a functor $S : \text{CCC} \rightarrow \lambda\text{-Dom}$.

**Proof.** We only discuss the object part of the functor. Given a CCC $C$, we let $S(C)$ be the $\lambda$-domain defined as follows:

(i) Its base category is $C$.

(ii) Types are objects of $C$.

(iii) Terms $\text{Tm}(\Gamma; A)$ are morphisms $\Gamma \rightarrow A$ in $C$. Substitution in terms is given by precomposition of morphisms.

(iv) Context comprehension is given by the categorical product in $C$.

(v) The function type $A \Rightarrow B$ is the exponential $A \Rightarrow B$ in $C$. The operation $\lambda^A_{\Gamma,B}(-)$ sends a term $\Gamma \times A \rightarrow B$ to its exponential transpose. Finally, the operation $\$^A_{\Gamma,A,B}$ maps the pair $(t : \Gamma \rightarrow (A \Rightarrow B), u : \Gamma \rightarrow A)$ to $ev \circ (t, u) : \Gamma \rightarrow B$. □
Corollary 5.2.23 (Interpretation of λ-calculus in a CCC). Let $\mathcal{C}$ be a CCC and let $J : \Sigma \to \text{Ob}(\mathcal{C})$ be an interpretation for the base types. The interpretation of λ-calculus in $\mathcal{C}$ with respect to $J$ is given by the following clauses.

(i) Types:

$\left\llbracket - \right\rrbracket_{J}^{Ty} : \text{Ty} \to \text{Ob}(\mathcal{C})$

$\left\llbracket \beta \right\rrbracket_{J} = J(\beta) \quad (\beta \in \Sigma)$

$\left\llbracket \sigma \to \tau \right\rrbracket_{J} = \left\llbracket \sigma \right\rrbracket_{J} \Rightarrow \left\llbracket \tau \right\rrbracket_{J}$

(ii) Contexts:

$\left\llbracket - \right\rrbracket_{J}^{Con} : \text{Con} \to \text{Ob}(\mathcal{C})$

$\left\llbracket [] \right\rrbracket_{J} = 1_{\mathcal{C}}$

$\left\llbracket \Gamma, \sigma \right\rrbracket_{J} = \left\llbracket \Gamma \right\rrbracket_{J} \times \left\llbracket \sigma \right\rrbracket_{J}$

(iii) Terms and substitutions:

$\left\llbracket - \right\rrbracket_{J}^{Tm} : \text{Tm}(\Gamma; \sigma) \to \mathcal{C}(\left\llbracket \Gamma \right\rrbracket_{J}, \left\llbracket \sigma \right\rrbracket_{J})$

$\left\llbracket - \right\rrbracket_{J}^{Sub} : \text{Sub}(\Delta; \Gamma) \to \mathcal{C}(\left\llbracket \Delta \right\rrbracket_{J}, \left\llbracket \Gamma \right\rrbracket_{J})$

$\left\llbracket v_{\Gamma}^{\sigma} \right\rrbracket_{J} = \text{snd} \left\llbracket \Gamma \right\rrbracket_{J}, \left\llbracket \sigma \right\rrbracket_{J}$

$\left\llbracket t \cdot u \right\rrbracket_{J} = \text{ev} \circ \left\llbracket t \right\rrbracket_{J}, \left\llbracket u \right\rrbracket_{J}$

$\left\llbracket \lambda A. t \right\rrbracket_{J} = \lambda \left\llbracket t \right\rrbracket_{J}$

$\left\llbracket t_{\gamma} \right\rrbracket_{J} = \left\llbracket t \right\rrbracket_{J} \circ \left\llbracket \gamma \right\rrbracket_{J}$

$\left\llbracket ()_{\Gamma} \right\rrbracket_{J} = 1_{\left\llbracket \Gamma \right\rrbracket_{J}}$

$\left\llbracket (\gamma, t) \right\rrbracket_{J} = \left\llbracket \gamma \right\rrbracket_{J}, \left\llbracket t \right\rrbracket_{J}$

$\left\llbracket \text{id}_{\Gamma} \right\rrbracket_{J} = 1_{\left\llbracket \Gamma \right\rrbracket_{J}}$

$\left\llbracket \gamma \circ \delta \right\rrbracket_{J} = \left\llbracket \gamma \right\rrbracket_{J} \circ \left\llbracket \delta \right\rrbracket_{J}$

$\left\llbracket p_{\sigma}^{\Gamma} \right\rrbracket_{J} = \text{fst} \left\llbracket \Gamma \right\rrbracket_{J}, \left\llbracket \sigma \right\rrbracket_{J}$

Proof. These clauses arise from the interpretation morphism $\left\llbracket - \right\rrbracket_{J}^{S(\mathcal{C})} : \mathcal{L} \to S(\mathcal{C})$. ∎
Chapter 6

Gluing

In this chapter, we look at a categorical reconstruction of normalization by evaluation using categorical gluing. The presentation is similar to and is inspired by the work of Fiore [17, 18] and Sterling and Spitters [34].

The structure of the proof follows closely the components of normalization by evaluation (Section 3.2.3).

1. First, we define a suitable model, namely the gluing category \( \mathcal{G} \) (Definition 6.2.4) together with the interpretation \( \rho \) of base types (Definition 6.2.7). The gluing category \( \mathcal{G} \) can be seen as the category of logical predicates.

2. The model \( (\mathcal{G}, \rho) \) gives rise to an interpretation functor \( \lceil - \rceil^\mathcal{G}_\rho: \mathcal{L} \rightarrow \mathcal{G} \) (Definition 6.2.7).

3. Next, we define families of natural transformations \( q^\sigma \) and \( u^\sigma \) (Definition 6.3.1). These natural transformations correspond to the ‘quote’ and ‘unquote’ functions of normalization by evaluation.

4. Finally, we define the normalization function as interpretation followed by quote (Definition 6.4.1). For this, we need to extend the unquote map to contexts (Definition 6.3.2), which is used to embed substitutions into the semantics. The environment \( \eta^\Gamma \) corresponds to unquoting the identity substitution.

6.1 Neutral and normal forms via presheaves

In this section, we define presheaves \( \text{Ne}_\sigma \) and \( \text{Nf}_\sigma \) of neutral and normal terms, respectively, for all types \( \sigma \in \text{Ty} \). Intuitively, \( \text{Ne}_\sigma \) and \( \text{Nf}_\sigma \) are variable sets indexed by contexts, i.e. \( \text{Ne}_\sigma(\Gamma) = \text{Ne}(\Gamma; \sigma) \) and \( \text{Nf}_\sigma(\Gamma) = \text{Nf}(\Gamma; \sigma) \). The presheaf action on morphisms corresponds to substitution. Since neutral and normal terms are not closed under arbitrary substitutions, we need to restrict the base category of the presheaves to an appropriate subcategory of \( \mathcal{L} \).

In this thesis, we choose the category of weakenings for the restricted base category, which is isomorphic to the dual of the poset of contexts under the prefix ordering. Alternative base categories are the category of order-preserving embeddings or the category of renamings. The former is used in [2] (in which it is confusingly called the category of weakenings) and [25], and the latter in [17, 34, 18].

Definition 6.1.1 (Poset of contexts). We order the set \( \text{Con} \) of contexts by the prefix ordering: \( \Gamma \leq \Delta \) iff \( \Gamma \) is a prefix of \( \Delta \). We write \( \geq \) for the dual order, which we call the extension order: \( \Delta \geq \Gamma \) iff \( \Gamma \leq \Delta \). If \( \Delta \geq \Gamma \), we also say that \( \Delta \) is an extension of \( \Gamma \).

From now on, the set \( \text{Con} \) is assumed to carry the ordering \( \geq \) of Definition 6.1.1 unless
otherwise stated. We view the poset Con as a category whose objects are contexts and such that Con(Δ, Γ) is a singleton \{∗\} if Δ ≥ Γ and Con(Δ, Γ) is empty otherwise.

Lemma 6.1.2. For every Δ, Γ ∈ Con, we have

\[ \Delta \geq \Gamma \iff \Delta = \Gamma \vee \exists \Delta' \in \text{Con}, \sigma \in \text{Ty}. \Delta = (\Delta', \sigma) \wedge \Delta' \geq \Gamma \]

Proof. Follows by induction on the difference of the lengths of Δ and Γ.

In what follows, we often employ Lemma 6.1.2 to define mathematical objects depending on a pair of contexts Δ and Γ such that Δ ≥ Γ by case distinction (for instance, Definition 6.1.3). In the second case, the definition may refer recursively to the object being defined but depending on the “smaller” pair Δ' ≥ Γ. This type of definition can be seen as a sort of recursion principle for the partial order ≥. Alternatively, one may think of this definition scheme as recursion on lists but where the base case is Γ instead of the empty list.

Definition 6.1.3. We define substitutions Δ ⊢ p_Γ : Γ for any pair of contexts Δ ≥ Γ:

\[ p_\Delta = \begin{cases} \text{id}_\Gamma & \text{if } \Delta = \Gamma \\ p_{\Delta'} \circ p_\sigma & \text{if } \Delta = (\Delta', \sigma) \wedge \Delta' \geq \Gamma \end{cases} \]

The substitution \( p_\Delta \) is a sort of generalized weakening. The primitive substitution \( p_\sigma \) only adds the single type \( \sigma \) to a context \( \Gamma \). In contrast, \( p_\Delta \) extends \( \Gamma \) with any number of types.

Lemma 6.1.4. \( \Theta \vdash p_\Delta \circ p_\Delta = p_\Theta : \Gamma \)

Proof. Using Lemma 6.1.2 on the proof that \( \Theta \geq \Delta \).

Proposition 6.1.5. There is an inclusion functor \( i : \text{Con} \to \mathcal{L} \).

Proof. The functor \( i : \text{Con} \to \mathcal{L} \) is the identity on contexts, and it sends the unique morphism \( \Delta \geq \Gamma \) in \( \text{Con} \) to the equivalence class of \( p_\Delta \in \text{Sub}(\Delta; \Gamma) \). Composition is preserved by Lemma 6.1.3 and identities are preserved by the definition of \( p_\Delta \).

The functor \( i \) is bijective on objects and faithful. Hence, it allows us to view Con as a wide subcategory of \( \mathcal{L} \), containing only morphisms of the form \( p_\Delta \) for \( \Delta \geq \Gamma \). This subcategory is also called the category of weakenings.

Recall from Definition 4.3.4 that the inclusion \( i : \text{Con} \to \mathcal{L} \) gives rise to a reindexing functor \( i^* : \text{PSh}(\mathcal{L}) \to \text{PSh}(\text{Con}) \).

Definition 6.1.6. We define the functor \( \text{Sub} : \mathcal{L} \to \text{PSh}(\text{Con}) \) as the composite

\[ \mathcal{L} \xrightarrow{i^*} \text{PSh}(\mathcal{L}) \xrightarrow{i_*} \text{PSh}(\text{Con}). \]

We write \( \text{Sub}_\Gamma \) for \( \text{Sub}(\Gamma) \). Intuitively, \( \text{Sub}_\Gamma \) is the presheaf of substitutions with codomain \( \Gamma \). The presheaf action weakens the domain of a substitution, allowing extra free variables in the context.

We are now ready to define the presheaves of neutral and normal terms [2, 34].

Proposition 6.1.7. We have presheaves \( \text{Ne}_\sigma \in \text{PSh}(\text{Con}) \) and \( \text{Nf}_\sigma \in \text{PSh}(\text{Con}) \) whose object parts are given by

\[ \text{Ne}_\sigma(\Gamma) = \text{Ne}(\Gamma; \sigma) \quad \text{and} \quad \text{Nf}_\sigma(\Gamma) = \text{Nf}(\Gamma; \sigma). \]
Proof. The morphism parts of the presheaves $N_{e_{\sigma}}$ and $N_{f_{\sigma}}$ are essentially given by
\[ N_{e_{\sigma}}(\Delta \geq \Gamma)(\Gamma \vdash_{ne} m : \sigma) = mp_{\Delta}^{\Gamma} \]
and
\[ N_{f_{\sigma}}(\Delta \geq \Gamma)(\Gamma \vdash_{nf} n : \sigma) = np_{\Delta}^{\Gamma}, \]
where, abusing notation slightly, we write $\Delta \geq \Gamma$ for the unique element of $\text{Con}(\Delta, \Gamma)$. However, we cannot use the substitution term former (\textsc{subst} in Figure 5.1) since terms of the form $t \gamma$ for a term $t$ and substitution $\gamma$ are neither neutral nor normal. Instead, we need to define these actions using mutual recursion on the syntax of neutral terms and normal forms. We omit the details. \hfill \Box

Proposition 6.1.8. We have the following morphisms:
\[ \begin{align*}
- N_{e_{\beta}} & \xrightarrow{\cong} N_{f_{\beta}} \\
- \text{var}_{\sigma} : \text{Sub}_{[\sigma]} & \to N_{e_{\sigma}} \\
- \text{app}_{\sigma, \tau} : N_{e_{\sigma}} \times N_{f_{\sigma}} & \to N_{e_{\tau}} \\
- \text{lam}_{\sigma, \tau} : (\text{Sub}_{[\sigma]} \Rightarrow N_{f_{\tau}}) & \xrightarrow{\cong} N_{f_{\sigma}} \Rightarrow \tau
\end{align*} \]

Proof. The first morphism coincides with the morphism in [18, Figure 13]. The final three morphisms coincide with those in [18, Figure 14]. \hfill \Box

Similarly to the presheaves of neutral terms, we have presheaves $N_{e_{\Gamma}} \in \text{PSh}(\text{Con})$ of neutral substitutions. These are necessary for Definition 6.3.2.

6.2 Gluing category

In this section, we construct the gluing category (Definition 6.2.4) and prove that it is cartesian closed and hence a model of the simply typed $\lambda$-calculus. We also define the gluing interpretation (Definition 6.2.7) and discuss how to interpret the syntax of $\lambda$-calculus in the gluing category (Corollary 6.2.10).

As a first step, we need to show that the syntactic category $\mathcal{L}$ is cartesian closed.

Notation 6.2.1. Let $\Gamma = [A_1, \ldots, A_n] \in \text{Con}$ and $A \in \text{Ty}$. Then $\Gamma \to A \in \text{Ty}$ is shorthand for $A_1 \to \cdots \to A_n \to A$.

Definition 6.2.2. (i) The terminal object in $\mathcal{L}$ is the empty context $[]$.
(ii) The product $\Gamma \times \Delta$ of contexts $\Gamma$ and $\Delta$ in $\mathcal{L}$ is their concatenation $\Gamma, \Delta$.
(iii) If $\Delta = [B_1, \ldots, B_n]$, then the exponential $\Gamma \Rightarrow \Delta$ is the context $[\Gamma \to B_1, \ldots, \Gamma \to B_n]$.

Lemma 6.2.3. The syntactic category $\mathcal{L}$ is cartesian closed with structure given in Definition 6.2.2.

Proof. The lemma can be proved using constructions similar to Čubrić et al. [14, Proposition 3.6]. There are two differences: they use variable names in the context of $\text{P}$-category theory, whereas we use de Bruijn indices and explicit substitutions in the context of ordinary category theory. \hfill \Box

We can now define the gluing category $\mathcal{G}$.
**Definition 6.2.4** (Gluing category). We define the **gluing category** $\mathcal{G}$ to be the comma category $\text{PSh}(\text{Con}) \downarrow \text{Sub}$. The functors $\pi_1$ and $\pi_2$ are given by the following diagram:

$$
\begin{array}{ccc}
\mathcal{G} & \xrightarrow{\pi_1} & \text{PSh}(\text{Con}) \\
\downarrow{\pi_2} & & \downarrow{\text{Id}} \\
\text{Sub} & \xrightarrow{} & \text{PSh}(\text{Con})
\end{array}
$$

Recall that objects of $\mathcal{G}$ are triples $(P, \Gamma, p)$ where $P \in \text{PSh}(\text{Con})$ is a presheaf over $\text{Con}$, $\Gamma \in \mathcal{L}$ is a syntactic context, and $p : P \to \text{Sub}_\Gamma$ is a natural transformation. The morphism $p$ can be seen as an indexed predicate over substitutions (where the index is the domain of the substitutions). Naturality of $p$ means that the predicate is stable under weakening.

Now we show that $\mathcal{G}$ is cartesian closed and that $\pi_2$ is strict cartesian closed. For this, we need the following lemma.

**Lemma 6.2.5.** The functor $\text{Sub} : \mathcal{L} \to \text{PSh}(\text{Con})$ weakly preserves the terminal object and products.

**Proof.** This is a consequence of Proposition 4.3.9 and Proposition 4.3.10. □

**Proposition 6.2.6.** (i) The category $\mathcal{G}$ is cartesian closed.

(ii) The functor $\pi_2 : \mathcal{G} \to \mathcal{L}$ is strict cartesian closed.

**Proof.** Follows immediately from Proposition 4.4.6, Lemma 6.2.3, Proposition 4.3.6, Lemma 6.2.5, and Proposition 4.3.8. □

We now define the gluing interpretation. Recall that for every term $\Gamma \vdash t : \sigma$ we have the substitution $t^* \in \text{Sub}(\Gamma; [\sigma])$ (Notation 5.1.5).

**Definition 6.2.7** (Gluing interpretation). We define the **gluing interpretation** $\rho : \Sigma \to \mathcal{G}$ as follows:

$$
\rho(\beta) = (\text{Nf}_\beta, [\beta], n_\beta),
$$

where $n_\beta : \text{Nf}_\beta \to \text{Sub}_\beta$ sends a normal form $s \in \text{Nf}_\beta(\Gamma)$ to $[n^*] \in \mathcal{L}(\Gamma, [\beta])$.

By Theorem 5.2.16, we get an interpretation functor $\llbracket - \rrbracket^\mathcal{G} : \mathcal{L} \to \mathcal{G}$ which is a morphism of $\lambda$-domains. This satisfies the following proposition.

**Proposition 6.2.8.** The following diagram of categories and functors commutes:

$$
\begin{array}{ccc}
\mathcal{L} & \xrightarrow{\llbracket - \rrbracket^\mathcal{G}} & \mathcal{G} \\
\downarrow{\text{Id}_\mathcal{L}} & & \downarrow{\pi_2} \\
\mathcal{L} & \xrightarrow{} & \mathcal{G}
\end{array}
$$

**Proof.** By definition, the interpretation functor $\llbracket - \rrbracket^\mathcal{G}$ is the interpretation morphism $\llbracket - \rrbracket^\mathcal{G}_S$, where $S$ is the functor from Proposition 5.2.22. Applying the functor $S$ to $\pi_2 : \mathcal{G} \to \mathcal{L}$, we get the following commutative diagram of $\lambda$-domains due to the initiality of $\mathcal{L}$:

$$
\begin{array}{ccc}
\mathcal{L} & \xrightarrow{\llbracket - \rrbracket^\mathcal{G}} & S(\mathcal{G}) \\
\downarrow{S(\text{Id}_\mathcal{L})} & & \downarrow{S(\pi_2)} \\
\mathcal{L} & \xrightarrow{} & S(\mathcal{L})
\end{array}
$$
We now obtain the desired diagram by forgetting the extra structure of \( \lambda \)-domains and their morphisms.

Remark 6.2.9. The above proof crucially depends on the strictness of the functor \( \pi_2 : G \to L \). This is why we decided to adopt CCCs with structure and strict cartesian closed functors in Chapter 4. See also Remark 4.2.6.

We now characterize how the syntax of \( \lambda \)-calculus is interpreted in the gluing category.

**Corollary 6.2.10.** The following statements hold for the interpretation functor \( \llbracket - \rrbracket^G_\rho \).

(i) For every \( \sigma \in Ty \), we have

\[
\llbracket \sigma \rrbracket^{Ty}_\rho = (R_\sigma, [\sigma], r_\sigma)
\]

for some \( R_\sigma \in \text{PSh}(\text{Con}) \) and \( r_\sigma : R_\sigma \to \text{Sub}_{[\sigma]} \).

(ii) For every \( \Gamma \in \text{Con} \), we have

\[
\llbracket \Gamma \rrbracket^{\text{Con}}_\rho = (R_\Gamma, \Gamma, r_\Gamma)
\]

for some \( R_\Gamma \in \text{PSh}(\text{Con}) \) and \( r_\Gamma : R_\Gamma \to \text{Sub}_\Gamma \).

(iii) For every \( \Gamma \vdash t : \sigma \), we have

\[
\llbracket t \rrbracket^{\text{Tm}}_\rho = (\tilde{t}, [t^*]) : (R_\Gamma, \Gamma, r_\Gamma) \to (R_\sigma, [\sigma], r_\sigma)
\]

for some \( \tilde{t} : R_\Gamma \to R_\sigma \). That is, the diagram

\[
\begin{array}{ccc}
R_\Gamma & \xrightarrow{\tilde{t}} & R_\sigma \\
r_\Gamma \downarrow & & \downarrow r_\sigma \\
\text{Sub}_\Gamma & \xrightarrow{[t^*]} & \text{Sub}_{[\sigma]}
\end{array}
\]

commutes.

(iv) For every \( \Delta \vdash \gamma : \Gamma \), we have

\[
\llbracket \gamma \rrbracket^{\text{Sub}}_\rho = (\tilde{\gamma}, [\gamma^*]) : (R_\Delta, \Delta, r_\Delta) \to (R_\Gamma, \Gamma, r_\Gamma)
\]

for some \( \tilde{\gamma} : R_\Delta \to R_\Gamma \). That is, the diagram

\[
\begin{array}{ccc}
R_\Delta & \xrightarrow{\tilde{\gamma}} & R_\Gamma \\
r_\Delta \downarrow & & \downarrow r_\Gamma \\
\text{Sub}_\Delta & \xrightarrow{[\gamma^*]} & \text{Sub}_\Gamma
\end{array}
\]

commutes.

**Proof.** This follows from Proposition 6.2.8 because \( \pi_2 \) picks out the second coordinate from the objects of the gluing category.

\[\square\]

### 6.3 Quote and unquote

The goal of this section is to define the quote and unquote functions of normalization by evaluation.

58
Proposition 6.3.1. We have two families of natural transformations

\[ u^\sigma : \mathrm{Ne}_\sigma \to R_\sigma \quad \text{and} \quad q^\sigma : R_\sigma \to \mathrm{Nf}_\sigma. \]

Proof. We define \( u^\sigma \) and \( q^\sigma \) by mutual recursion on the type \( \sigma \). For the definitions, we use the morphisms in Proposition 6.1.8.

(i) For base types \( \beta \in \Sigma \), we have \( R_\beta = \mathrm{Nf}_\beta \). Thus, we put \( u^\beta = (\mathrm{Ne}_\beta \cong - \to \mathrm{Nf}_\beta) \) and \( q^\beta = 1_{\mathrm{Nf}_\beta} \).

(ii) For function types \( \sigma \to \tau \), by Corollary 6.2.10, we have that \( R_\sigma \to \tau \) is given by the pullback diagram

\[
\begin{array}{ccc}
R_\sigma & \to & R_\tau \\
\downarrow r_\sigma & & \downarrow r_\tau \\
\mathrm{Sub}_{[\sigma \to \tau]} & \cong & \mathrm{Sub}_{[\tau]} \\
\end{array}
\]

We define \( u^{\sigma \to \tau} : \mathrm{Ne}_{\sigma \to \tau} \to R_{\sigma \to \tau} \) using the universal property of the pullback square (6.1). For that, we need natural transformations \( f : \mathrm{Ne}_{\sigma \to \tau} \to \mathrm{Sub}_{[\sigma \to \tau]} \) and \( g : \mathrm{Ne}_{\sigma \to \tau} \to R_\sigma \Rightarrow \mathrm{Sub}_{[\tau]} \) such that the square

\[
\begin{array}{ccc}
\mathrm{Ne}_{\sigma \to \tau} & \to & R_\sigma \\
\downarrow f & & \downarrow u_\sigma \\
\mathrm{Sub}_{[\sigma \to \tau]} & \cong & \mathrm{Sub}_{[\tau]} \\
\end{array}
\]

commutes. For \( f \), we choose the morphism \( m_{\sigma \to \tau} \) that sends a neutral term \( m \in \mathrm{Ne}_{\sigma \to \tau} (\Gamma) \) to \( [m^*] \in \mathcal{L}(\Gamma, [\sigma \to \tau]) \). For \( g \), we take the exponential transpose of the composite

\[
\begin{array}{ccc}
\mathrm{Ne}_{\sigma \to \tau} \times R_\sigma & \cong & \mathrm{Ne}_{\sigma \to \tau} \times \mathrm{Nf}_\sigma \\
\downarrow 1 \times q^\sigma & & \downarrow \text{app} \times \gamma \\
\mathrm{Ne}_{\sigma \to \tau} & \Rightarrow & \mathrm{Nf}_\tau \\
\end{array}
\]

We now extend the unquote function to contexts. This is necessary to define the normalization function. Note that \( R_{[]} = 1 \) and \( R_{\Gamma, \sigma} = R_\Gamma \times R_\sigma \) by Corollary 6.2.10.

Definition 6.3.2. We define \( u^\Gamma : \mathrm{Ne}_\Gamma \to R_\Gamma \) by recursion on the context \( \Gamma \).

\[
\begin{array}{l}
u^\Gamma_{[]} = * \\
u^\Gamma_{\gamma, t} = (u^\Gamma_{\gamma}, u^\Gamma_{\Delta}(t))
\end{array}
\]

6.4 Normalization function

In this section, we construct the normalization function and prove its correctness.

Recall from Corollary 6.2.10 that the interpretation \( \llbracket t \rrbracket_\rho^\Gamma \) of a term \( \Gamma \vdash t : \sigma \) in the category \( \mathcal{G} \) takes the form \( (t, \llbracket t \rrbracket_\rho^\sigma) \) for some \( t : R_\Gamma \to R_\sigma \).
**Definition 6.4.1.** We define the normalization function $\text{nf}_\sigma^\Gamma : \text{Tm}(\Gamma; \sigma) \to \text{Nf}(\Gamma; \sigma)$ as

$$\text{nf}_\sigma^\Gamma(t) = q_{\Gamma}^\sigma(\tilde{t}_\Gamma(\mu_{\Gamma}(\text{id}_{\Gamma}))).$$

In essence, the definition of the normalization function has the same form as the one for normalization by evaluation in Section 3.2.3.

Diagrammatically, the normalization function is the composite

$$\text{Tm}(\Gamma; \sigma) \xrightarrow{1-\text{Tm}} G(\llbracket \Gamma \rrbracket_\rho, \llbracket \sigma \rrbracket_\rho) \xrightarrow{\sigma_1} \text{PSh}(\text{Con})(R_{\Gamma}, R_{\sigma}) \xrightarrow{\varphi} R_{\sigma}(\Gamma) \xrightarrow{q_{\Gamma}^\sigma} \text{Nf}(\Gamma; \sigma)$$

where $\varphi$ is given by

$$\varphi(\tilde{t}) = \tilde{t}_\Gamma(\mu_{\Gamma}(\text{id}_{\Gamma})).$$

The correctness of the normalization function can be expressed via the following two propositions.

**Proposition 6.4.2.** If $\Gamma \vdash t = u : \sigma$, then $\text{nf}_\sigma^\Gamma(t) = \text{nf}_\sigma^\Gamma(u)$.

**Proof.** Note that $\text{nf}_\sigma^\Gamma(t)$ is defined in terms of the interpretation $\llbracket t \rrbracket_\rho^\sigma$. Hence, by the soundness of the interpretation (Theorem 5.2.12), convertible terms have equal normal forms. \qed

**Proposition 6.4.3.** $\Gamma \vdash \text{nf}_\sigma^\Gamma(t) = t : \sigma$.

**Proof.** This is Theorem 3.6 in [34]. \qed
Chapter 7

Conclusion

We presented and compared three reduction-free normalization proofs for the simply typed λ-calculus, all of which share a similar structure. The proofs have advantages and disadvantages.

(i) First, we proved weak normalization using Tait’s idea of a convertibility predicate (Section 3.1.1). An advantage of this proof is that it is rather short and easy to understand. A disadvantage is that the weak normalization theorem does not tell us how to compute the normal form of a term.

(ii) The second proof constructed a normalization function which computes the normal form of a given input term (Section 3.2.3). This proof used the technique of normalization by evaluation, whereby normalization is implemented by interpreting the term in a suitable model and then turning the semantic object back into a normal form. An advantage of this proof is that it provides an efficient normalization algorithm which does not rely on concepts from term rewriting. A disadvantage is that the proof itself is more involved.

(iii) The third proof was a categorical reconstruction of normalization by evaluation using the categorical gluing construction (Chapter 6). The main advantage of this proof is that it is independent of the syntactic presentation of λ-calculus: it employs categorical semantics where the syntax can be characterized as an initial model. A disadvantage is that the constructions in this proof are complicated and abstract, and thus this proof is much harder to understand.

On the way to presenting the proofs, we also provided introductions to various flavors of syntax and semantics for the simply typed λ-calculus. In particular, we discussed Henkin models, Kripke models, simply typed categories with families, and cartesian closed categories.

As an attempt to generalize the structure of the Tait-like normalization proof, we introduced the notion of bilogical predicates. Using this notion, we were able to give a very short proof of weak normalization for the simply typed λ-calculus. As future work, one could consider a similar generalization for normalization by evaluation.

Another, orthogonal direction of research is to consider more complex type theories, such as dependent type theory or type theory with simple inductive types. A comparison between the different proof methods for such systems would be interesting.
Bibliography


