Saarland University

Faculty of Mathematics and Computer Science

Bachelor's Thesis

Formalising the Undecidability of Higher-Order Unification

Author
Simon Spies

Supervisor
Prof. Gert Smolka

Advisor
Yannick Forster

Reviewers
Prof. Gert Smolka
Prof. Bernd Finkbeiner

Submitted: 29th March 2019

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 29th March, 2019

Abstract

In this thesis we formally verify the undecidability of higher-order unification in the proof assistant Coq. Higher-order unification procedures underlie many modern day proof assistants including Coq. Higher-order unification is the process of finding an instantiation of the free variables in two typed terms such that after substitution the resulting terms are convertible.

While it is well-known that unification of first-order terms is decidable, for terms of higher order the problem has been shown to be undecidable. In 1973 Gerard Huet proved that third-order unification is undecidable with a reduction from the Post correspondence problem. Warren D. Goldfarb improved on this result in 1981 by proving that even unification in second-order languages is undecidable. The proof is obtained by a reduction from Hilbert's tenth problem.

In this work we formalise both reductions in Coq. We simplify Huet's proof by reducing from the modified Post correspondence problem and give an intuitive explanation of Goldfarb's construction. Furthermore, we show that the undecidability of second and third-order unification is sufficient to conclude the undecidability of higher-order unification in general and how Huet's result can be obtained as a corollary of Goldfarb's result.

Acknowledgements

First and foremost, I want to thank my advisor Yannick Forster. Yannick convinced me to explore the area of computational logic early on in my Bachelor's studies and helped foster my interest in an academic future. His advice and support were far beyond what can be expected for which I am truly grateful. This thesis would not exist without him.

Furthermore, I want to express my gratitude to my supervisor Professor Smolka for offering me this thesis. He introduced me to the fields of programming language semantics and formal verification. I am thankful for his support and his mentorship throughout my Bachelor's studies. In addition, I want to thank Professor Finkbeiner for reviewing this thesis.

Lastly, I want to thank my friends and family for their unwavering support during these last months. In particular, I want to thank Marie and Dominik for proofreading this thesis.

Contents

Chapter 1

Introduction

When working in a proof assistant such as Coq, the problem of higher-order unification naturally arises. For example, higher-order unification is needed whenever functional arguments of a universally quantified proposition are to be inferred by the proof assistant. Consider the proposition $\forall n.\ n+0=n$. If we prove this claim by induction on n, logically an application of the induction principle for natural numbers $\forall P.\ P(0) \to (\forall n.\ P(n) \to P(n+1)) \to \forall n.\ P(n)$ is required. For the proof assistant this means that $\forall n.\ P(n)$ has to be specialised to the proposition $\forall n.\ n+0=n$. In particular, an instantiation of the predicate variable P has to be chosen. In a type theory such as the one underlying Coq a predicate on natural numbers is a function transforming a natural number n into a proposition over n. Here the predicate $\lambda n.n+0=n$ can be chosen as an instantiation of P. If said predicate is chosen, the resulting proposition $\forall n.\ (\lambda n.n+0=n)\ n$ is convertible to $\forall n.\ n+0=n$. While using a proof assistant, it can become tedious to give such simple instantiations manually. For this reason one would expect that a predicate for P can be inferred automatically from the context. The process involved in finding such an instantiation is called higher-order unification.

In general, higher-order unification is the process of finding a substitution for the free variables in two typed terms such that under the substitution both terms are convertible. When analysing the problem of higher-order unification in the simply typed λ-calculus, we can distinguish different classes of sub-problems based on the types of variables and constants that may appear in the terms. Types are distinguished based on their order. First-order types are base types such as `int` or `bool`. Second-order types are the types of functions with base type arguments. Third-order types are the types of functions with second-order arguments. In general, nth-order types for $n > 1$ are the types of functions with arguments of order $n-1$. We speak of nth-order unification if the types of all variables are of at most order n and the types of all constants are at most of order $n+1$. It is well known that unification of first-order languages is decidable [45]. In 1972 Lucchesi [34] and in 1973 Huet [28] independently

discovered that for terms of third order the problem of unification is undecidable. In 1981 Goldfarb [25] improved on this result by showing that unification is already undecidable in second-order languages provided they contain a 2-ary function constant. Unification in second-order languages with only unary function constants, also referred to as monadic second-order unification, is still decidable as proved by Farmer [16]. Despite higher-order unification being undecidable, both higher-order unification and nth-order unification are enumerable using the unification algorithm by Huet [29].

In this thesis we formalise the reductions given by Huet and Goldfarb in the constructive type theory of Coq, establishing that second-order and third-order unification are undecidable. Huet [28] reduces the Post correspondence problem [42] to third-order unification by encoding cards over a binary alphabet as λ-terms. We simplify Huet's proof by reducing from the modified Post correspondence problem [26] instead. Goldfarb on the other hand gives a reduction from Hilbert's tenth problem to second-order unification. He encodes diophantine equations, polynomial equations over natural numbers, as unification equations. While Huet conducts his analysis in a Church-typed version of the λ-calculus where every term is well-typed, Goldfarb employs a second-order language without abstractions using hereditary substitutions. In this thesis, we adapt both results to a Curry-style simply-typed λ-calculus and prove them to be sufficient to conclude the undecidability of higher-order unification in general. In addition, we investigate the role of constants in unification. Moreover, we verify a unification algorithm for the first-order fragment of the simply-typed λ-calculus thus formally proving first-order unification to be decidable.

Synthetic Undecidability For formalising reductions we utilise the approach of synthetic computability theory. Synthetic computability theory differs from traditional computability theory in the notion of reduction and the results obtained by reduction. In traditional computability theory one fixes a model of computation — say Turing machines — and then proves results about the expressiveness of Turing machines as a computational model. In particular, one establishes that there does not exist a Turing machine deciding the halting problem on Turing machines. From there on, one can show a number of problems undecidable by means of reduction, for example the Post correspondence problem [42]. A problem $\mathbf{P}$ is undecidable in this setting if there does not exist a Turing machine yielding a decision whether a given instance of $\mathbf{P}$ has a solution or not. A reduction from a problem $\mathbf{P}$ to a problem $\mathbf{Q}$, written $\mathbf{P} \preceq \mathbf{Q}$, is a function f, computable by a Turing machine, which transforms instances of the problem $\mathbf{P}$ into instances of the problem $\mathbf{Q}$ such that an instance x of $\mathbf{P}$ has a solution if and only if the transformation $f(x)$ has a solution. The result obtained through reduction is that there is a Turing machine deciding $\mathbf{P}$ whenever there is a Turing machine deciding $\mathbf{Q}$. In particular, if $\mathbf{P}$ is undecidable $\mathbf{Q}$ is also undecidable.

However, in practice it is tedious to construct Turing machines computing the reduction function f and formally verify their correctness. Moreover, the undecidability of the problem of interest is usually not dependent on the particular model of computation that was chosen. For this reason, we work in the setting of synthetic computability theory. Synthetic in this context means that reduction functions are defined and the verification thereof is conducted in the constructive type theory of Coq [58]. In contrast to classical computability theory, we do not fix a model of computation but rely on the fact that all functions constructible in Coq are by definition computable. In practice, this alleviates the need for proving the correctness of a program computing the reduction function leaving just the verification of the function. This style corresponds well to the style of reductions found in the literature where reduction functions are constructed without reference to a fixed model of computation [28, 25].

A problem in the synthetic setting is a predicate $\mathbf{P}$ on some type X. An instance of $\mathbf{P}$ is then a value x of type X. Given two problems $\mathbf{P}$ and $\mathbf{Q}$ where $\mathbf{P}$ is on type X and $\mathbf{Q}$ is on type Y, a reduction $\mathbf{P} \preceq \mathbf{Q}$ consists of a reduction function $f : X \to Y$ and a proof that $\mathbf{P}(x)$ iff $\mathbf{Q}(f(x))$. A problem $\mathbf{P}$ is said to be decidable if there exists a decision function d deciding whether $\mathbf{P}(x)$ holds or not. However, the notion of undecidability differs in the synthetic setting. In contrast to the traditional approach, undecidability in the synthetic sense is not simply the negation of decidability. While every function constructible in Coq is computable in a model of computation, the type theory underlying the proof assistant is consistent with the assumption of noncomputable functions. In particular, one may consistently assume a decision procedure for arbitrary problems. Thus, synthetic undecidability of a problem $\mathbf{P}$ is not characterised as the nonexistence of a decision procedure for $\mathbf{P}$ but as the existence of a chain of reductions from a problem which is widely accepted as undecidable, for example the halting problem on Turing machines.

The present work is part of an ongoing project to formalise computability theory as a library of problems in the synthetic setting. In [21] the theory of synthetic undecidability is developed formally in Coq and in [21, 20, 19, 18, 32] several problems are proven to be synthetically undecidable, including the Post correspondence problem [20] and Hilbert's tenth problem [32].

Formalisation The contents of this thesis are formalised in the proof assistant Coq. The formalisation is self contained and we remark in Chapter 10 on details of the formalisation. The only axiom we assume is functional extensionality. As a consequence of the accompanying formalisation, we do not prove every claim on paper. Instead, the theorems, facts and lemmas of this thesis are hyperlinked with their formal counterparts. In addition, the entire formalisation is available at:

`http://www.ps.uni-saarland.de/~spies/bachelor.php`

In particular, we do not give all lemmas present in the Coq formalisation in this thesis. Instead we present a selected subset which we believe to be helpful in understanding the subject.

Overview In Chapter 2, we give an informal overview of the results presented in this thesis and explain the main ideas behind their proofs. It is to be understood as a gentle introduction into the contents of this thesis. In Chapter 3, we briefly introduce the common operations used in subsequent chapters. In Chapter 4, we formally introduce the Curry-style simply-typed λ-calculus with unrestricted β-reduction which will be used throughout this thesis. We establish an equational theory for the calculus and formally introduce the notion of order. In Chapter 5, we precisely define the problem of higher-order unification $\mathbf{U}$ and unification in the nth-order fragment $\mathbf{U}_n$. For the remainder of this thesis we mean $n > 0$ when we write $\mathbf{U}_n$ since no type can ever have order zero and thus terms in $\mathbf{U}_0$ cannot contain variables. In Chapter 6, we formalise and simplify the proof of the undecidability of third-order unification by Huet [28]. In Chapter 7, we formalise the reduction presented by Goldfarb [25] and therefore formally prove second-order unification to be undecidable. For the Goldfarb reduction, we follow the explanation given by Dowek [13] using Church numerals to encode Diophantine equations. This method yields the undecidability of higher-order unification in general, but not of the second-order fragment. The undecidability of second-order unification is achieved by moving to Goldfarb numerals, a different encoding of natural numbers. Apart from the equations generated for multiplication, this encoding is very close to the approach using Church numerals. In Chapter 8, we state and verify a unification algorithm for the first-order fragment of the Curry-style simply-typed λ-calculus. In Chapter 9, we prove the conservativity result $\mathbf{U}_n \preceq \mathbf{U}_m \preceq \mathbf{U}$ for $n \leq m$, justifying the popular reasoning that the undecidability of second or third-order unification implies the undecidability of higher-order unification in general. The reduction functions used in the reductions can be viewed as identity functions and therefore one can say we show $\mathbf{U}$ subsumes $\mathbf{U}_n$. Furthermore, one can say we prove $\mathbf{U} = \bigcup_{n\in\mathbb{N}} \mathbf{U}_n$ because every well-typed term can be assigned some order n. In addition, we investigate the role of constants with respect to unification. We show how constants can be added without affecting unifiability in the nth-order fragment and we show how certain constants may be removed without affecting unifiability. In Chapter 10, we remark on the formalisation. In particular, we comment on the overhead generated by formalising the work in a proof assistant and give an overview of the development. In Chapter 11, we touch upon prospective future work and present related work in the fields of synthetic computability theory, formalisations of unification, and the undecidability of higher-order unification.

Contribution We present the problem of higher-order unification in a unified setting and investigate the role of constants with respect to unification. We contribute a formalisation of the undecidability of higher-order unification using reductions presented by Huet and Goldfarb. We give a formalisation of a first-order unification algorithm for the simply-typed λ-calculus. We showcase an application of the strong normalisation proof technique presented in [22] adjusted to yield a weak normalisation proof for the simply-typed λ-calculus with constants.

Chapter 2

Informal Overview

In this chapter we give an overview of the results presented in this thesis and sketch the ideas behind them. We refrain from digressing into formal details and focus on the intuitions underlying our constructions — a formal discussion of all results presented here is conducted in subsequent chapters.

λ-calculus As the underlying language for unification we employ a Curry-style simply-typed λ-calculus with unrestricted β-reduction and constants. We use the letters s, t for terms of the calculus and write $\Gamma \vdash s : A$ if the term s can be assigned the type A in the context Γ. Types consist of base types $\alpha, \beta, \ldots$ and function types $A \to B$. For the remainder of this chapter we focus only on well-typed terms and leave the typing information implicit whenever possible.

The letters σ, τ denote substitutions — infinite maps from variables to terms. The operation $s[\sigma]$ expresses the capture-avoiding replacement of all free variables in s according to the substitution σ. As a convention, when defining substitutions in examples, we only give the terms that are assigned to the free variables. All other variables are left unchanged. For example, consider the term $\lambda y.x\ y$. Here the variable y occurs bound and the variable x occurs free. A substitution may replace only the variable x and if the term inserted for x contains the variable y as a free variable, then the bound variable y is renamed to a fresh variable. Thus, for the substitution $\sigma x = z$ we have $(\lambda y.x\ y)[\sigma] = \lambda y.z\ y$ and for $\tau x = y$ we have $(\lambda y.x\ y)[\tau] = \lambda z.y\ z$. Constants, denoted by the letter c, cannot be substituted but may occur in the terms inserted by substitutions. We refer to both constants and variables as *atoms*.

We write $s \succ t$ if a term s reduces in a single step to the term t and say s is β-equivalent ("equivalent" for short) to t, written $s \equiv t$, if s and t are in the equivalence closure of the reduction relation $\succ$. For example, the following two terms are equivalent:

$$\lambda xy.(\lambda\_.z)\ x \equiv \lambda xy.z \equiv \lambda xy.(\lambda\_.z)\ y \tag{2.1}$$

Every well-typed term is weakly normalising, meaning for every term $\Gamma \vdash s : A$ there exists a term t such that s reduces to t and t is a normal form. Due to the nature of $\succ$ this term t is unique.

2.1 Higher-Order Unification

Higher-order unification is the process of finding a well-typed substitution for the free variables in two typed terms s and t such that under the substitution both terms are equivalent. We write $\Gamma \vdash s \stackrel{?}{=} t : A$ for the unification equation between s and t provided both terms are of type A under the typing context Γ, meaning $\Gamma \vdash s : A$ and $\Gamma \vdash t : A$. A substitution σ is said to unify an equation $\Gamma \vdash s \stackrel{?}{=} t : A$, if s and t are equivalent under the substitution, meaning $s[\sigma] \equiv t[\sigma]$. The substitution σ is said to agree with the type constraints of Γ under the typing context Δ, written $\Delta \vdash \sigma : \Gamma$, if for every binding $(x : A) \in \Gamma$ the term inserted for x, i.e. σx, is of type A under context Δ. For a given equation $\Gamma \vdash s \stackrel{?}{=} t : A$, the problem of higher-order unification asks whether there exists a context Δ and a substitution σ such that σ agrees with the type constraints of Γ under Δ and σ unifies the equation. Explicitly,

$$\mathbf{U}(\Gamma \vdash s \stackrel{?}{=} t : A) \quad \text{iff} \quad \Delta \vdash \sigma : \Gamma \text{ and } s[\sigma] \equiv t[\sigma] \text{ for some } \Delta, \sigma$$

There is a distinction to be made whether σ may insert terms containing free variables or not. For instance, consider the unification equation $\Gamma \vdash \lambda xy.f\ x \stackrel{?}{=} \lambda xy.f\ y : \alpha \to \alpha \to \alpha$ where $\Gamma = (f : \alpha \to \alpha)$. As we allow open terms, the example Eq. (2.1) shows that the substitution $\sigma f = \lambda\_.z$ in the context $\Delta = (z : \alpha)$ unifies both terms. If we would only allow closed terms, then there is no substitution which both respects the type of f and unifies s and t in a language where the type α is empty. All closed terms of type $\alpha \to \alpha$ are equivalent to identity functions and thus cannot unify the equation.

When speaking of higher-order unification we may freely assume that s and t are normal and that σ only inserts normal terms, since unifiability is defined up to equivalence. To simplify matters, in this informal overview we do not explicitly annotate each equation with its type and a typing context. Instead, we remark on the types of the free variables contained in Γ such that the reader can infer a type of the entire equation from context. For example, the above equation $\lambda xy.f\ x \stackrel{?}{=} \lambda xy.f\ y$ is fully specified by the knowledge that f is of type $\alpha \to \alpha$.

Systems of Equations Unification, especially first-order unification, is frequently formulated as a problem over multiple equations. We refer to multiple equations $s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$ as a *system of equations* [52] ("system" for short) and denote them by the letter E. In a system all equations have to be typed in the same typing context. We show in the following that both formulations are interreducible.

We say σ unifies E if σ unifies all equations in E. The problem of unifying a system of equations, *system unification* **SU**, is described by $\mathbf{SU}(E)$ iff $\forall s \stackrel{?}{=} t \in E.\ s[\sigma] \equiv t[\sigma]$ for some context Δ and substitution $\Delta \vdash \sigma : \Gamma$ where Γ is the typing context used to type the equations of E. The reduction $\mathbf{U} \preceq \mathbf{SU}$ is easy since every equation $s \stackrel{?}{=} t$ can be interpreted as a singleton system. For the converse direction, $\mathbf{SU} \preceq \mathbf{U}$, we combine a system of equations $s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$ into a single equation:

$$\lambda h.h\ s_1 \cdots s_n \stackrel{?}{=} \lambda h.h\ t_1 \cdots t_n$$

The key insight behind the transformation is that whenever an atom is applied to two or more arguments, then those arguments are independent from each other. By independent we mean that from $h\ s_1\ s_2 \equiv h\ t_1\ t_2$ we can conclude the equivalences $s_1 \equiv t_1$ and $s_2 \equiv t_2$ and vice versa. To ensure that h cannot be substituted, we capture the variable with an abstraction. The argument generalises to the case of n equations, as witnesses by:

$$\forall i.\ s_i \equiv t_i \quad \text{iff} \quad \lambda h.h\ s_1 \cdots s_n \equiv \lambda h.h\ t_1 \cdots t_n$$

Enumerability Using standard techniques, we show that higher-order unification is enumerable. We enumerate all equations $\Gamma \vdash s \stackrel{?}{=} t : A$ and well-typed instantiations of the free variables of s and t. From this we select the instantiations under which s and t are equivalent. Since all terms involved are well-typed, we can decide whether two terms are equivalent by deciding whether their normal forms are equal or not.

2.2 Undecidability of Higher-Order Unification

We proceed by proving that higher-order unification is undecidable by reduction from Hilbert's tenth problem. The proof follows an explanation by Dowek [13]. Hilbert's tenth problem, also referred to as **H10**, in its original form asks whether a given Diophantine equation is satisfiable. Diophantine equations are equations over natural numbers (or equivalently integers) involving addition, multiplication and constants. In general, it is undecidable whether a Diophantine equation is satisfiable as proven by the combined work of Davis [7], Putnam [8], Robinson [46, 9], and Matijasevivc [35]. Nowadays, the result is known as the DPRM-theorem and it was only proved a few years before Huet proved third-order unification to be undecidable [28].

To motivate why the question of satisfiability is not easy to answer, consider the equation:

$$(x+1)^3 + (y+1)^3 \doteq (z+1)^3 \quad \text{where } x, y, z : \mathbb{N}$$

This equation is a special case of Fermat's last theorem and thus there is no solution to it, meaning it is not satisfiable. An algorithm deciding the non-existence of a solution for this equation would implicitly prove a non-trivial special case of Fermat's last theorem.

Since its original formulation there have been found several equivalent ways to express this problem. Following Goldfarb [25] we use a formulation based on simple Diophantine equations over the natural numbers. A simple Diophantine equation is an equation over the natural numbers involving a single addition, multiplication or constant. The corresponding formulation of Hilbert's tenth problem then asks whether a system of such simple Diophantine equations has a solution. For example, the following equations form a system:

$$x \doteq 42 \qquad y \doteq x \cdot y \qquad z \doteq z + z$$

A solution for such a system is an assignment θ mapping variables to natural numbers such that under this assignment the resulting equations hold. This system has exactly one solution, namely $\theta x = 42$ and $\theta y = \theta z = 0$.

The proof that Hilbert's tenth problem reduces to higher-order unification is based on two core ideas. The first idea is to express natural numbers in the λ-calculus as Church numerals and to encode simple Diophantine equations as unification equations. The second idea is to ensure that the "domain" of the equations only allows Church encoded natural numbers. The word "domain" in this context is to be understood as the collection of all the terms which may be inserted for the variables occurring in the encoding of the simple Diophantine equations. The second idea is realised by adding a characteristic equation for every variable which precisely characterises what it means to be a Church numeral.

Church numerals are a common way to express natural numbers in the λ-calculus. Every natural number n is represented by the abstraction $[\![n]\!] := \lambda af.f^n\ a$ of type $\alpha \to (\alpha \to \alpha) \to \alpha$. f^n expresses n-fold iteration of f on an initial value a. Since $f^{n+m}\ a = f^n\ (f^m\ a)$ and $f^{n \cdot m}\ a = (f^m)^n\ a$, addition and multiplication are characterised by

$$\mathsf{add}\ s\ t := \lambda af.s\ (t\ a\ f)\ f \qquad \mathsf{mul}\ s\ t := \lambda af.s\ a\ (\lambda b.t\ b\ f)$$

satisfying $\mathsf{add}\ [\![n]\!]\ [\![m]\!] \equiv [\![n+m]\!]$ and $\mathsf{mul}\ [\![n]\!]\ [\![m]\!] \equiv [\![n \cdot m]\!]$. What remains is the restriction of the domain with the help of a characteristic equation which is only satisfied by valid Church encodings. As it turns out, the distribution property of iteration suffices for this equation, i.e. $f\ (f^n\ a) = f^n\ (f\ a)$. For normal forms s we show:

$$\lambda af.s\ (f\ a)\ f \equiv \lambda af.f\ (s\ a\ f) \quad \text{iff} \quad s = [\![n]\!] \text{ for some } n : \mathbb{N}$$

and thus define $\mathsf{CN}\ x := \lambda af.x\ (f\ a)\ f \stackrel{?}{=} \lambda af.f\ (x\ a\ f)$. In the reduction we add one characteristic equation for every variable and one corresponding equation for every simple Diophantine equation. Recall our example from the beginning: $x \doteq 42$, $y \doteq x \cdot y$ and $z \doteq z + z$. For this system of Diophantine equations we produce the system of unification equations:

$$x \stackrel{?}{=} [\![42]\!] \qquad y \stackrel{?}{=} \mathsf{mul}\ x\ y \qquad z \stackrel{?}{=} \mathsf{add}\ z\ z \qquad \mathsf{CN}\ x \qquad \mathsf{CN}\ y \qquad \mathsf{CN}\ z$$

2.3 Nth-Order Unification

When analysing the problem of higher-order unification, we distinguish different fragments of the problem based on the types of variables and constants that may occur in s and t. We differentiate types based on their *order*, where the order of a type A is given by $\mathsf{ord}\ \alpha = 1$ and $\mathsf{ord}\ (A \to B) = \max\{\mathsf{ord}\ A + 1, \mathsf{ord}\ B\}$. We say a term s is of order n, if the order of the types of its variables is at most n and the order of the types of its constants is at most $n + 1$. Formally, we introduce the predicate $\Gamma \vdash_n s : A$ characterising the nth-order fragment of the calculus.

The fragment of higher-order unification where all terms are at most of order n is called nth-order unification and denoted by $\mathbf{U}_n$. An instance of this problem is an equation $\Gamma \vdash_n s \stackrel{?}{=} t : A$ where both s and t are of type A under the context Γ in the nth-order fragment of the calculus. Analogously to $\mathbf{U}$, $\mathbf{U}_n$ is characterised by

$$\mathbf{U}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A) \quad \text{iff} \quad \Delta \vdash_n \sigma : \Gamma \text{ and } s[\sigma] \equiv t[\sigma] \text{ for some } \Delta, \sigma$$

where $\Delta \vdash_n \sigma : \Gamma$ expresses that σ agrees with the type constraints of Γ under Δ and σ inserts terms of at most order n.

Analogously to the higher-order case, we introduce systems of nth-order equations and the corresponding unification problem $\mathbf{SU}_n$. Moving from higher-order unification to nth-order unification, the reduction $\mathbf{U} \preceq \mathbf{SU}$ carries over unchanged, meaning $\mathbf{U}_n \preceq \mathbf{SU}_n$. However, the proof of $\mathbf{SU} \preceq \mathbf{U}$ is not preserved. By transforming $s_1 \stackrel{?}{=} t_1, \ldots, s_k \stackrel{?}{=} t_k$ into $\lambda h.h\ s_1 \cdots s_k \stackrel{?}{=} \lambda h.h\ t_1 \cdots t_k$ the order of the terms is affected due to the variable h. If $A_1, \ldots, A_k$ are the types of the equations of the system, then the variable h is of type $A_1 \to \cdots \to A_k \to \alpha$ which has order $1 + \max\{\mathsf{ord}\ A_1, \ldots, \mathsf{ord}\ A_k\}$.

2.4 Third-Order Unification

Huet [28] and Lucchesi [34] were the first to prove that third-order unification and higher-order unification in general are undecidable. They both give proofs by reduction from the Post correspondence problem. In this thesis we simplify Huet's

construction to yield an undecidability proof of third-order unification by reduction from the modified Post correspondence problem [26].

We call a pair of two strings l and r over the binary alphabet $\{0, 1\}$ a *card*, written l/r, and a collection of cards $l_1/r_1, \ldots, l_n/r_n$ a *stack* of cards. Intuitively, the modified Post correspondence problem **MPCP** can be described as follows: Given an *initial card* and a stack of cards, such as

$$\boxed{\frac{11}{1}} \quad \text{and} \quad \boxed{\frac{0}{110}} \quad \boxed{\frac{101}{000}}$$

does there exist a sequence of the cards, starting with the initial card, possibly containing cards multiple times or not at all, in which the cards produce the same string in the top and bottom row. In the case of our example we start the sequence by using the initial card twice and follow up with the second card.

$$\boxed{\begin{array}{c|c|c} \frac{11}{1} & \frac{11}{1} & \frac{0}{110} \end{array}}$$

The top and bottom rows then both read 11110.

When reducing **MPCP** to third-order unification, we have to transform an initial card l_0/r_0 and a stack of cards $l_1/r_1, \ldots, l_n/r_n$ into a unification equation $s \stackrel{?}{=} t$. This equation has to be chosen such that any solution to the **MPCP** instance yields a substitution unifying s and t. Furthermore, any substitution unifying s and t has to implicitly pick a sequence of the cards such that the top and bottom rows agree. In the reduction, we transform $l_0/r_0, \ldots, l_n/r_n$ into the unification equation:

$$\lambda u_1 u_0.\overline{l_0}\ (x_f\ \overline{l_0} \cdots \overline{l_n}) \stackrel{?}{=} \lambda u_1 u_0.\overline{r_0}\ (x_f\ \overline{r_0} \cdots \overline{r_n})$$

where $x_f : (\alpha \to \alpha)^{n+1} \to \alpha$ and $(\alpha \to \alpha)^{n+1}$ means $n + 1$ arguments of type $\alpha \to \alpha$. We encode strings over the binary alphabet $\{0, 1\}$ into λ-terms using a modified Church encoding of binary strings. We fix the variables u_1, u_0 and map the string 110 to the abstraction $\overline{110} := \lambda x.u_1\ (u_1\ (u_0\ x))$. Note that the proper Church encoding of binary strings would take u_1, u_0 as arguments of the abstraction as well.

To motivate the transformation, we consider closed terms $f\colon (\alpha \to \alpha)^{n+1} \to \alpha \to \alpha$. and to simplify matters, we assume that α is an empty type allowing us for now to ignore constants. The notion of parametricity [43] suggests

$$f\ g_0 \cdots g_n\ s \equiv g_{i_1}\ (\cdots (g_{i_k}\ s)) \text{ for some order } i_1, \ldots, i_k.$$

meaning f applies its arguments $g_0, \ldots, g_n$ in some order to s. Moreover, the notion of parametricity suggests that the order $i_1, \ldots, i_k$ in which the arguments are applied is independent of the choice of $g_0, \ldots, g_n$ and s. In particular, if we have $f\ \overline{l_0} \cdots \overline{l_n}\ s \equiv f\ \overline{r_0} \cdots \overline{r_n}\ s$, then we can conclude $\overline{l_{i_1}}\ (\cdots (\overline{l_{i_k}}\ s)) \equiv \overline{r_{i_1}}\ (\cdots (\overline{r_{i_k}}\ s))$.

Due to the definition of the encoding of strings, the successive application of encoded strings is equivalent to applying the concatenated string. Thus, the above equivalence holds if and only if $\overline{l_{i_1} \cdots l_{i_k}}\ s \equiv \overline{r_{i_1} \cdots r_{i_k}}\ s$. If we pick for s a variable, such as ϵ, this equivalence allows us to conclude the equality of $l_{i_1} \cdots l_{i_k}$ and $r_{i_1} \cdots r_{i_k}$.

Hence, one might falsely think that the equation $x_f\ \overline{l_0} \cdots \overline{l_n}\ \epsilon \stackrel{?}{=} x_f\ \overline{r_0} \cdots \overline{r_n}\ \epsilon$ is sufficient for the reduction. Note that in this equation x_f takes an additional argument and thus is of type $(\alpha \to \alpha)^{n+1} \to \alpha \to \alpha$. However, since the variables u_1, u_0 occur free in this equation, there is no restriction preventing a substitution from inserting $\lambda x.x$ for both u_1 and u_0. The solution for this problem is to transform their free occurrences into bound occurrences: $\lambda u_1 u_0.x_f\ \overline{l_0} \cdots \overline{l_n}\ \epsilon \stackrel{?}{=} \lambda u_1 u_0.x_f\ \overline{r_0} \cdots \overline{r_n}\ \epsilon$. Nevertheless, we are not guaranteed that the first card of a solving sequence is always the initial card. We can address this problem by applying the initial card to the result of x_f, resulting in $\lambda u_1 u_0.\overline{l_0}\ (x_f\ \overline{l_0} \cdots \overline{l_n}\ \epsilon) \stackrel{?}{=} \lambda u_1 u_0.\overline{r_0}\ (x_f\ \overline{r_0} \cdots \overline{r_n}\ \epsilon)$.

As a last step in the transformation, we remove the variable ϵ. In a language with constants there might be terms of type α. Even if there are none, the definition of higher-order unification allows substitutions to insert free variables as long as they are assigned a type in some context Δ. Thus, ϵ has no influence on the existence of solutions. We change the type of x_f to $(\alpha \to \alpha)^{n+1} \to \alpha$ and obtain: $\lambda u_1 u_0.\overline{l_0}\ (x_f\ \overline{l_0} \cdots \overline{l_n}) \stackrel{?}{=} \lambda u_1 u_0.\overline{r_0}\ (x_f\ \overline{r_0} \cdots \overline{r_n})$.

In the verification of the reduction, we have to show that a **MPCP** instance has a solution if and only if the above equation has a solution. To motivate why this is true we recall the example from the beginning. The cards $11/1$, $0/110$, $101/000$ are transformed into the equation:

$$\lambda u_1 u_0.\overline{11}\ (x_f\ \overline{11}\ \overline{0}\ \overline{101}) \stackrel{?}{=} \lambda u_1 u_0.\overline{1}\ (x_f\ \overline{1}\ \overline{110}\ \overline{000})$$

For the "if" direction, we can use the solving sequence to construct a unifying substitution. We pick $\sigma x_f = \lambda x_0 x_1 x_2.x_0\ (x_1\ \epsilon)$ in the context $\Delta = (\epsilon : \alpha)$. Under this substitution we have:

$$\overline{11}\ (\sigma x_f\ \overline{11}\ \overline{0}\ \overline{101}) \equiv \overline{11}\ (\overline{11}\ (\overline{0}\ \epsilon)) \equiv \overline{11110}\ \epsilon \equiv \overline{1}\ (\overline{1}\ (\overline{110}\ \epsilon)) \equiv \overline{1}\ (\sigma x_f\ \overline{1}\ \overline{110}\ \overline{000})$$

For the "only if" direction, a normal form analysis on σx_f reveals a term structure which implicitly contains the sequence of the cards. For example, σx_f might be of the shape $\lambda x_0 x_1 x_2.x_0\ (x_1\ t)$ where t does not start with any x_i. The equivalence $\overline{11}\ (\sigma x_f\ \overline{11}\ \overline{0}\ \overline{101}) \equiv \overline{1}\ (\sigma x_f\ \overline{1}\ \overline{110}\ \overline{000})$ then allows us to conclude that the sequence "initial card, initial card, second card" is a valid solution for the **MPCP** instance.

2.5 Conservativity

In [28] Huet proves the undecidability of higher-order unification and remarks that since the proof only requires terms of order three, he actually gives a proof of the undecidability of third-order unification. When formalising proofs such a claim cannot be made as easily. The original proof would have to be duplicated and adapted to yield a proof of the undecidability of third-order unification.

An alternative technique to conclude both the undecidability of third-order unification and higher-order unification is to prove $\mathbf{MPCP} \preceq \mathbf{U}_3 \preceq \mathbf{U}$. For the latter reduction we establish the *conservativity* of unification. By conservativity we mean that any nth-order equation $\Gamma \vdash_n s \stackrel{?}{=} t : A$ is unifiable iff it is unifiable in the nth-order fragment of the calculus. One can say that higher-order unification subsumes nth-order unification and the same can be said for the system variants of the problem. Thus, for $n \leq m$ we show:

$$\begin{array}{ccccc} \mathbf{U}_n & \subseteq & \mathbf{U}_m & \subseteq & \mathbf{U} \\ \downarrow & & \downarrow & & \updownarrow \\ \mathbf{SU}_n & \subseteq & \mathbf{SU}_m & \subseteq & \mathbf{SU} \end{array}$$

where $\mathbf{P} \to \mathbf{Q}$ means $\mathbf{P}$ reduces to $\mathbf{Q}$ and $\mathbf{P} \subseteq \mathbf{Q}$ means $\mathbf{Q}$ subsumes $\mathbf{P}$.

Every higher-order substitution $\Delta \vdash \sigma : \Gamma$ unifying $\Gamma \vdash_n s \stackrel{?}{=} t : A$ is implicitly also a mth-order substitution where m is the maximum order of the terms that are inserted for free variables of s and t. Thus, in the following we focus on proving $\mathbf{U}_n \subseteq \mathbf{U}_m$ for $n \leq m$. Proving $\mathbf{U}_n \subseteq \mathbf{U}_m$ essentially boils down to proving the following claim:

$s[\sigma] \equiv t[\sigma]$ for some Δ and $\Delta \vdash_n \sigma : \Gamma \quad$ iff $\quad s[\tau] \equiv t[\tau]$ for some Σ and $\Sigma \vdash_m \sigma : \Gamma$

The "if" direction being trivial, we focus on the "only if" direction. The key insight for the "only if" direction is that variables and constants whose order is too high can be replaced. As an example, consider the equation

$$\lambda x.\mathsf{g}\ \mathsf{a}\ y \stackrel{?}{=} \lambda x.f\ x\ x$$

where $\mathsf{g} : \alpha \to \alpha \to \alpha$ and $\mathsf{a} : \alpha$ are constants and the free variables $f : \alpha \to \alpha \to \alpha$ and $y : \alpha$ have to be replaced. A simple second-order substitution unifying both terms would be $\sigma y = z$, $\sigma f = \lambda x_1 x_2.\mathsf{g}\ \mathsf{a}\ z$ in the context $\Delta = (z : \alpha)$. However, there is also the third-order substitution $\tau y = h\ (\mathsf{g}\ \mathsf{a})$ and $\tau f = \lambda x_1 x_2.\mathsf{g}\ \mathsf{a}\ (h\ (\mathsf{g}\ \mathsf{a}))$ in the context $\Delta = (h : (\alpha \to \alpha) \to \alpha)$ unifying both terms. In the following, we show how the former can be recovered from the latter.

We observe that the following three criteria can artificially increase the order of a unifying substitution σ up to m when the original terms are of order n.

1. σ can insert free variables of order m.
2. σ can insert constants of order $m+1$ which do not occur in s or t.
3. σ can insert non-normal terms containing bound variables of order m. For example, the variable y could have been instantiated with the third-order term $(\lambda g.g\ z)\ (\lambda x.x)$ in the previous equation.

We show that whenever none of the above criteria apply, a normal term can be shown to be of order n. In the following, we address each criteria individually and explain how we transform τ stepwise to obtain a substitution σ of order n.

1. Whenever we encounter a free variable $x : A_1 \to \cdots \to A_k \to \alpha$ in τ of order m, we replace it with the first-order term $\lambda x_1 \cdots x_k.z$ where $z : \alpha$. The justification for this is given by the compatibility of $\equiv$ with substitution:
$$s \equiv t \quad \text{implies} \quad s[\sigma] \equiv t[\sigma] \text{ for all } \sigma, s, t$$
2. Whenever we encounter a constant $c : A_1 \to \cdots \to A_k \to \alpha$ which does not appear in s or t, we replace it with the first-order term $\lambda x_1 \cdots x_k.z$ where $z : \alpha$ analogously to the previous technique. The key observation here is that replacing only the constants which do not occur in s or t does not affect s and t and thus does not affect unifiability.
3. We normalise all terms after applying transformations 1 and 2.

In the example $\tau y = h\ (\mathsf{g}\ \mathsf{a})$ and $\tau f = \lambda x_1 x_2.\mathsf{g}\ \mathsf{a}\ (h\ (\mathsf{g}\ \mathsf{a}))$, after transformation 1 we obtain the substitution $\tau_1 y = (\lambda g.z)\ (\mathsf{g}\ \mathsf{a})$ and $\tau_1 f = \lambda x_1 x_2.\mathsf{g}\ \mathsf{a}\ ((\lambda g.z)\ (\mathsf{g}\ \mathsf{a}))$. After the second and third transformation this substitution is simplified to $\tau_2 y = z$ and $\tau_2 f = \lambda x_1 x_2.\mathsf{g}\ \mathsf{a}\ z$.

2.6 Second-Order Unification

Goldfarb [25] improved on the result of Huet by showing that unification is already undecidable in second-order languages provided they contain at least a single 2-ary function constant $\mathsf{g} : \alpha \to \alpha \to \alpha$. He establishes the undecidability of second-order unification by a reduction from Hilbert's tenth problem. In its essence, the structure of Goldfarb's proof follows the structure of the undecidability proof we gave in Section 2.1. The difference between both proofs lies in the encoding of natural numbers.

While we use Church numerals in Section 2.1, Goldfarb gives an encoding based on a 2-ary function constant $\mathsf{g} : \alpha \to \alpha \to \alpha$. We call natural numbers encoded using this constant *Goldfarb numerals*. A Goldfarb numeral $[\![n]\!]$ is obtained from the Church numeral by fixing the function variable f to the term $\mathsf{g}\ \mathsf{a}$ where $\mathsf{a} : \alpha$ is a constant. Explicitly, the Goldfarb numeral for n is the term $[\![n]\!] = \lambda a.(\mathsf{g}\ \mathsf{a})^n\ a$. In contrast to

Church numerals, which are of type $\alpha \to (\alpha \to \alpha) \to \alpha$, Goldfarb numerals are of type $\alpha \to \alpha$. As a consequence, the undecidability of second-order unification can be proved. However, the price of changing the encoding is that multiplication can no longer be encoded in the style of Church numerals. Consequently, Goldfarb gives a different encoding of multiplication which we motivate in the following.

Consider the iterative algorithm mult for computing the product $m \cdot n = \mathsf{mult}(0,0)$:

$$\begin{aligned} \mathsf{mult}(a,i) &= a && \text{if } i = m \\ \mathsf{mult}(a,i) &= \mathsf{mult}(a+n, i+1) && \text{if } i \neq m \end{aligned}$$

We focus on the transformation $\mathsf{succ}(a,i) := (a+n, i+1)$ which is applied to a and i in each step of the computation. If we start in the pair $(0,0)$ and repeatedly iterate the successor function succ, we produce the sequence: $(0,0);(n,1);\ldots;(m\cdot n, m)$. The correctness of mult then justifies the idea that a finite sequence of the form

$$(0,0);(n,1);\cdots;(p,m)$$

generated by iteratively applying succ on $(0,0)$ serves as a proof of $m \cdot n = p$. As a consequence, to express the equation $m \cdot n = p$ it suffices to give an equation which can only be satisfied if there exists a finite sequence obeying the same iterative structure and terminating in the pair (p,m). We encode this concept in the equation:

$$X;(p,m) = (0,0);\mathsf{succ}(X)$$

where succ is lifted to finite sequences by pointwise application. This equation is satisfiable by a finite sequence X if and only if $m \cdot n = p$.

To understand why this equation expresses an iterative structure, we consider the infinite sequence $(0,0);(n,1);(2n,2);\ldots$ This sequence is characterised by the recursive equation $X = (0,0);\mathsf{succ}(X)$ since for any sequence X satisfying this equation, we can show

$$X = (0,0);\mathsf{succ}(X) = (0,0);(n,1);\mathsf{succ}(\mathsf{succ}(X)) = \ldots$$

In the finite case the pair (p,m) ensures that the iteration has to end after m steps. For $t_i := (i \cdot n, i)$ we show that $X = t_0;\cdots;t_{m-1}$ is the solution for the equation $(0,0);(n,1);\cdots;(p,m)$ if and only if $m \cdot n = p$.

All that remains is to express the above equation as a unification equation. This is accomplished by finding a suitable second-order encoding of finite sequences and encoding the succ function by means of unification.

2.7 Constants

Goldfarb remarks in his work [25] that the proof remains valid even in languages without the constants $\mathsf{a}, \mathsf{b} : \alpha$. The straightforward way to verify this claim formally is to duplicate the original proof and to adapt it to a language without a and b. However, there is a more elegant way to solve this problem. In the following we motivate techniques useful for introducing and removing constants from a language without affecting unifiability. We write $\mathbf{U}_n^{\mathcal{C}}$ for the unification problem where all constants are drawn from $\mathcal{C}$. While formally we represent $\mathcal{C}$ as a Coq type, to simplify matters in this informal overview we pretend $\mathcal{C}$ is a set of constants and use set notation.

The first technique allows for the introduction of arbitrary constants into a language. Explicitly, we prove $\mathbf{U}_n^{\mathcal{C}} \preceq \mathbf{U}_n^{\mathcal{D}}$ whenever $\mathcal{C} \subseteq \mathcal{D}$. The key insight in this reduction is that whenever s and t draw their constants exclusively from $\mathcal{C}$, then they are unifiable if and only if they are unifiable with constants from $\mathcal{C}$. Thus, we may remove all constants which do not occur in s or t from a unifying substitution τ to obtain a unifying substitution σ which draws its constants from $\mathcal{C}$. We cannot simply replace constants by variables as they may have a type of order $n+1$ whereas the order of the types of variables is bounded by n. Similarly to the proof of conservativity, if a constant $c : A_1 \to \cdots \to A_k \to \alpha$ is to be removed, we replace it with the term $\lambda x_1 \cdots x_k.z$ in a context Δ where $z : \alpha$.

The second technique allows for the elimination of constants of an order strictly smaller than n. Explicitly, we prove $\mathbf{U}_n^{\mathcal{D}} \preceq \mathbf{U}_n^{\mathcal{C}}$ whenever $\mathcal{C} \subseteq \mathcal{D}$ and all $d \in \mathcal{D} - \mathcal{C}$ are of an order strictly smaller than n. The main idea is to replace constants by bound variables. For example, consider the terms $\mathsf{g}\ x \stackrel{?}{=} \mathsf{g}\ \mathsf{a}$ where $\mathsf{g} : \alpha \to \alpha \to \alpha$ and $x, \mathsf{a} : \alpha$. Any substitution σ with $\sigma x \equiv \mathsf{a}$ unifies both terms. When eliminating the constant a from the language, a first attempt could be to introduce a new variable x_{a} and to replace every occurrence of a with x_{a}. We ensure that x_{a} is not affected by substitutions by capturing x_{a}. Applying both transformations to $\mathsf{g}\ x \stackrel{?}{=} \mathsf{g}\ \mathsf{a}$ results in the equation $\lambda x_{\mathsf{a}}.\mathsf{g}\ x \stackrel{?}{=} \lambda x_{\mathsf{a}}.\mathsf{g}\ x_{\mathsf{a}}$. However, under these transformations the terms are no longer unifiable since inserting the variable x_{a} for x would cause a renaming of the bound variable x_{a}. We fix this problem by ensuring that the new variables introduced by this transformation are made available to all free variables. To this end, we transform the type of x from α to $\alpha \to \alpha$ and replace every occurrence of x with $x\ x_{\mathsf{a}}$. Hence, we obtain $\lambda x_{\mathsf{a}}.\mathsf{g}\ (x\ x_{\mathsf{a}}) \stackrel{?}{=} \lambda x_{\mathsf{a}}.\mathsf{g}\ x_{\mathsf{a}}$. In the case of our example, it remains to prove that

$$(\mathsf{g}\ x)[\sigma] \equiv (\mathsf{g}\ \mathsf{a})[\sigma] \text{ for some } \sigma \quad \text{iff} \quad (\lambda x_{\mathsf{a}}.\mathsf{g}\ (x\ x_{\mathsf{a}}))[\tau] \equiv (\lambda x_{\mathsf{a}}.\mathsf{g}\ x_{\mathsf{a}})[\tau] \text{ for some } \tau$$

For the sake of simplicity we only show how specific substitutions σ and τ are transformed. For the "if" direction let $\sigma x = \mathsf{a}$. Then we pick the substitution $\tau x = \lambda x_{\mathsf{a}}.x_{\mathsf{a}}$

which can be obtained from σ by taking x_{a} as an argument and replacing the constant a with x_{a}. For the "only if" direction let $\tau x = \lambda x_{\mathsf{a}}.x_{\mathsf{a}}$. Then we pick $\sigma x = (\lambda x_{\mathsf{a}}.x_{\mathsf{a}})\ \mathsf{a}$ which can be obtained from τ by applying τx to the argument a.

Combining both techniques, we can conclude $\mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}} \preceq \mathbf{U}_2^{\{\mathsf{g}\}} \preceq \mathbf{U}_2^{\mathcal{C}}$ where $\mathsf{g} \in \mathcal{C}$. Furthermore, we can obtain Huet's result from Goldfarb's result

$$\mathbf{U}_2^{\{\mathsf{g}\}} \preceq \mathbf{U}_3^{\{\mathsf{g}\}} \preceq \mathbf{U}_3^{\emptyset} \preceq \mathbf{U}_3$$

Farmer [16] shows that monadic second-order unification, i.e. unification where all function constants are at most unary, is decidable. We remark that eliminating constants of an order strictly smaller than n is the best we can hope for in general. If we could eliminate constants of order n, we would obtain the reduction $\mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}} \preceq \mathbf{U}_2^{\emptyset}$. However, $\mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}}$ being undecidable and $\mathbf{U}_2^{\emptyset}$ being decidable, the assumption that constants of order n can be eliminated as well is contradictory.

2.8 First-Order Unification

We give a decision procedure for first-order unification in the λ-calculus. In the context of first-order unification one usually considers a language similar to $s, t ::= x \mid c \mid s\ t$. However, when unifying first-order terms in the λ-calculus subtle differences arise. For example, consider the first-order terms $\mathsf{g}\ x\ \mathsf{a}$ and $\mathsf{g}\ \mathsf{b}\ \mathsf{a}$ where $\mathsf{g} : \alpha \to \alpha \to \alpha$ and $\mathsf{a}, \mathsf{b}, x : \alpha$. Using the substitution $\sigma x = \mathsf{b}$ they can be unified. If we turn the free occurrence of x into a bound occurrence using a λ-abstraction, then the resulting terms $\lambda x.\mathsf{g}\ x\ \mathsf{a}$ and $\lambda x.\mathsf{g}\ \mathsf{b}\ \mathsf{a}$ can no longer be unified.

The above example shows that when it comes to first-order unification in the λ-calculus, then we have to distinguish between bound and free variables. In essence, unification in the first-order fragment of the λ-calculus may be understood as unification in the language $s, t ::= x \mid c \mid s\ t$ where only a subset of the variables can be instantiated. More explicitly, to answer the question of unifiability for two first-order terms s and t it suffices to normalise s and t to normal forms $\lambda x_1 \cdots x_k.s'$ and $\lambda x_1 \cdots x_l.t'$ where s' and t' are free of abstractions, decide whether $k = l$ and if so decide whether the terms s' and t' are unifiable when the variables $x_1, \ldots, x_k$ are considered bound.

Chapter 3

Formal Preliminaries

In this chapter we introduce common operations which are used throughout the thesis.

We write $\mathcal{O}X$ for the option type over X, $\emptyset$ for the empty option. The option containing the value x is simply denoted by x as well.

We write $\mathcal{L}X$ for the type of lists over X, nil for the empty list and $x :: A$ for the list A which is extended by the element x. For the concatenation of A and B we write $A +\!\!+ B$ and $\overleftarrow{A}$ for the list obtained by reversing A. $|A|$ denotes the length of A and $A[i]$ the element at position i of A. The first position of A is 0 and if i not strictly smaller than the length of A, then $A[i]$ is $\emptyset$. $x \in A$ means the element x is contained in the list A and $A \subseteq B$ means every element of A is contained in B. We write $[fx \mid x \in A]$ for the list which results from applying the function f to every element in A and we write a^n for the list obtained by repeating the term a exactly n times.

If the pair (ι, ϱ) forms a retraction from X to Y, we write $X \hookrightarrow Y$. The proposition $x \in \mathsf{im}\ \iota$ means $\iota y = x$ for some y. We assume retractions are always tight, meaning $x \notin \mathsf{im}\ \iota$ iff $\varrho x = \emptyset$.

We write $X + Y$ for the sum type of X and Y with the injections $\mathsf{L}_+ : X \to X + Y$ and $\mathsf{R}_+ : Y \to X + Y$.

Chapter 4

λ-calculus

In this thesis, we consider a Curry-style simply-typed λ-calculus [6, 40, 2] with unrestricted β-reduction, meaning β-reductions are allowed in all subterms. For a discrete type of constants $\mathcal{C}$ we define *terms*, *types* and *typing contexts* by:

$$\begin{aligned} s, t, u, v &::= x \mid c \mid \lambda x.s \mid s\ t && (x : \mathbb{N}, c : \mathcal{C}) \\ A, B &::= \alpha \mid A \to B && (\alpha : \mathbb{N}) \\ \Gamma, \Delta, \Sigma &::= x_1 : A_1, \ldots, x_n : A_n \end{aligned}$$

Variables and constants will sometimes be referred to as *atoms*, denoted by the letter a. On paper we use named syntax whereas in Coq we use De Bruijn indices [10]. As a consequence, abstractions are of the form λs instead of $\lambda x.s$ in the formal development. The variable n in De Bruijn notation indicates that n binders have to be skipped until the binder of the variable is reached. For example, the term $(\lambda xyz.z)\ (\lambda y.y)$ can be expressed as $(\lambda\lambda\lambda 2)\ (\lambda 0)$. In the formalisation, we represent type contexts Γ by lists of types and the proposition $(x : A) \in \Gamma$ is to be interpreted as $\Gamma[x] = A$. We write $\mathsf{dom}\ \Gamma$ for the list of variables contained in the context Γ and $\mathsf{vars}\ s$ for the free variables of the term s.

$\mathsf{dom}\ \Gamma$

$\mathsf{vars}\ s$

Substitutions In this work we use parallel substitutions [10], i.e. maps from variables to terms, denoted by σ, τ. We write $s[\sigma]$ for the result of applying the substitution σ to all free variables of s. Following Barendregt's variable convention [2], we assume that the free variables occurring in our terms are always distinct from bound variables. As a consequence, on paper we assume that substitution $s[\sigma]$ is always capture-avoiding. If σ is a substitution, we write $\sigma[x := s]$ for the substitution obtained by extending σ with a binding of x to s. We write $\sigma[\tau]$ for the composition of σ and τ and s/x for the single-point substitution replacing the variable x with the term s. Analogously, we denote parallel renamings by ρ, δ. Parallel renamings are parallel substitutions which insert only variables. We do not state every lemma regarding substitution on paper but remark on important ones such as:

Fact 4.1 *If $\sigma x = \tau x$ for all $x \in \mathsf{vars}\ s$, then $s[\sigma] = s[\tau]$.*

4.1 Simply-Typed λ-calculus

In accordance to Huet [28], we employ unrestricted β-reduction as a reduction strategy. We write $s \succ t$ if the term s reduces in a single step to the term t.

$s \succ t$
$s \succ^* t$

Definition 4.2

$$\frac{}{(\lambda x.s)\ t \succ s[x/t]} \qquad \frac{s \succ s'}{\lambda x.s \succ \lambda x.s'} \qquad \frac{s \succ s'}{s\ t \succ s'\ t} \qquad \frac{t \succ t'}{s\ t \succ s\ t'}$$

The reflexive, transitive closure of $\succ$ is denoted by $\succ^*$.

Fact 4.3

1. *$\succ$ is compatible with renaming and substitution.*
2. *$\succ^*$ is a compatible with the term structure, substitutions, and renamings.*

$\mathsf{normal}\ s$
$s \triangleright t$

We say a term s is normal, written $\mathsf{normal}\ s$, if it does not reduce any further and write $s \triangleright t$ if s reduces to t and t is normal. We establish that normality is preserved under renaming and in some cases even under substitution. In addition, we give an alternative characterisation of normality which follows the term structure.

Fact 4.4

1. *Normality is decidable.*
2. *If s is normal, then $s[\rho]$ is normal for every renaming ρ.*
3. *If s is normal and σx is normal and not an abstraction for all x, then $s[\sigma]$ is normal.*

Fact 4.5 *The following rules and their inversions hold for normality:*

$$\frac{}{\mathsf{normal}\ x} \qquad \frac{}{\mathsf{normal}\ c} \qquad \frac{\mathsf{normal}\ s}{\mathsf{normal}\ (\lambda x.s)} \qquad \frac{\mathsf{normal}\ s \quad \mathsf{normal}\ t \quad \neg\mathsf{isLam}\ s}{\mathsf{normal}\ (s\ t)}$$

$\mathsf{head}\ s$

In the context of reduction, we will frequently be concerned with the applicative head of a term. The *applicative head* of a term s, written $\mathsf{head}\ s$, is the left-most subterm with respect to application. For example, we have $\mathsf{head}\ (a\ y\ (\lambda z.z)) = a$ and $\mathsf{head}\ ((\lambda xy.x\ y)\ \mathsf{a}\ z) = \lambda xy.x\ y$. Notably, if the applicative head of a term s is an abstraction and the term s is an application, then there must exist a β-redex which can be reduced. In this case s is of the shape $s = (\lambda x.s')\ t_1 \cdots t_n$. To invert reduction sequences, we use the following fact:

Fact 4.6 *Let ρ be a renaming.*

1. *If $\lambda x.s \succ^* t$ then $s \succ^* s'$ and $t = \lambda x.s'$ for some s'.*

2. *If* $s\ t \succ^* u$ *then either there exist* s', t' *with* $s \succ^* s'$, $t \succ^* t'$ *and* $u = s'\ t'$ *or there exists* s' *with* $s \succ^* \lambda x.s'$ *and the applicative head of* s *is an abstraction.*
3. *If* $s[\rho] \succ^* t$ *then there exists* s' *with* $s \succ^* s'$ *and* $t = s'[\rho]$.

4.1.1 Equational Theory

When reasoning about the equivalence of two terms up to reduction, we shall use the equivalence closure of $\succ$, denoted by $\equiv$. Confluence of $\succ$ entails that $\equiv$ satisfies the Church-Rosser property which allows us to lift compatibility properties from $\succ^*$ to $\equiv$. We defer a confluence proof of $\succ$ to Section 4.4.

$\boxed{s \equiv t}$

Fact 4.7

1. *If* $s \equiv s'$, *then there is a term* t *such that* $s \succ^* t$ *and* $s' \succ^* t$.
2. $\equiv$ *is compatible with the term structure, substitutions, and renamings.*
3. *If* $\sigma x \equiv \tau x$ *for all* $x \in \mathsf{vars}\ s$, *then* $s[\sigma] \equiv s[\tau]$.

Similar to syntactic equality, we can recover the injectivity and disjointness of term constructors. Clearly, we cannot expect application to be injective without further side conditions, since the applicative head might be an abstraction which could be reduced.

Fact 4.8

1. *If* $x \equiv y$, *then* $x = y$.
2. *If* $c \equiv c'$, *then* $c = c'$.
3. *If* $\lambda x.s \equiv \lambda x.t$, *then* $s \equiv t$.
4. *If the applicative heads of* s, s' *are atoms and* $s\ t \equiv s'\ t'$, *then* $s \equiv s'$ *and* $t \equiv t'$.

Fact 4.9 *Let the applicative head of* s_1 *be an atom.*

$$x \not\equiv c \qquad x \not\equiv \lambda y.s \qquad c \not\equiv \lambda x.s \qquad x \not\equiv s_1\ s_2 \qquad c \not\equiv s_1\ s_2 \qquad \lambda x.s \not\equiv s_1\ s_2$$

4.1.2 Simple Typing

Huet [28] and Snyder and Gallier [52] present their versions of the simply-typed λ-calculus in a Church-typed fashion, meaning only well-typed terms exist. As a consequence, functions operating on terms are formally always defined on typing derivations. This complicates the formalisation in Coq significantly. We remark on this aspect in Chapter 10. Furthermore, even if one does not require every term to be

well-typed, the inclusion of types into the syntax introduces a certain overhead with respect to formalisation. The Type system cannot be exchanged without reproving lemmas about for example reduction, equivalence, and substitution.

For these reasons, we present unification in a Curry-style type system. Let Ω be a signature assigning types to constants. We write $\Gamma \vdash s : A$, if s can be assigned the type A under context Γ and $\Delta \vdash \sigma : \Gamma$ if σ agrees with the constraints of Γ under Δ. Analogously, we introduce $\Delta \vdash \rho : \Gamma$ for renamings.

$\boxed{\Gamma \vdash s : A}$
$\boxed{\Delta \vdash \rho : \Gamma}$
$\boxed{\Delta \vdash \sigma : \Gamma}$

Definition 4.10

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash c : \Omega c} \qquad \frac{\Gamma \vdash s : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B}$$

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \to B}$$

$$\frac{\forall (x : A) \in \Gamma.\ (\rho x : A) \in \Delta}{\Delta \vdash \rho : \Gamma} \qquad \frac{\forall (x : A) \in \Gamma.\ \Delta \vdash \sigma x : A}{\Delta \vdash \sigma : \Gamma}$$

As a consequence of choosing this type system, terms do not have unique types. For example, the term $\lambda x.x$ can be typed as $\alpha \to \alpha$ and as $(\alpha \to \alpha) \to (\alpha \to \alpha)$ in every context Γ. With respect to unification, it is still possible to choose the types of the terms inserted by substitutions since the types of the free variables are always determined by the context Γ. We prove that the type of a term is preserved under renaming, substitution, and reduction.

Fact 4.11 (Preservation) *Let* $\Gamma \vdash s : A$.

1. *If* $\Delta \vdash \rho : \Gamma$, *then* $\Delta \vdash s[\rho] : A$.
2. *If* $\Delta \vdash \sigma : \Gamma$, *then* $\Delta \vdash s[\sigma] : A$.
3. *If* $s \succ s'$, *then* $\Gamma \vdash s' : A$.
4. *If* $s \succ^* s'$, *then* $\Gamma \vdash s' : A$.

We remark that substitutions $\Delta \vdash \sigma : \Gamma$ are only required to insert terms which are well-typed for the variables of Γ. For all other variables a substitution may return untyped terms. In the context of unification all free variables in s and t will always be contained in Γ and therefore only well-typed terms will be inserted into s and t. To this end, we establish:

Lemma 4.12 *If* $\Gamma \vdash s : A$ *and* $x \in \mathsf{vars}\ s$, *then* $x \in \mathsf{dom}\ \Gamma$.

4.2 Order

In this section we formally introduce the notion of *order*. As a consequence of using a calculus without explicit type annotations we cannot simply compute "the" order of a term. We compensate this problem by introducing an *order type system* $\Gamma \vdash_n s : A$

which captures the nth-order fragment of the calculus. The type system $\Gamma \vdash_n s : A$ is monotone with respect to order and may be understood as a more fine-grained version of $\Gamma \vdash s : A$.

We define the order of a type in accordance to [52] and extend the terminology to typing contexts.

Definition 4.13

$\boxed{\mathsf{ord}\ A}$ $\boxed{\mathsf{ord}\ \Gamma}$

$$\mathsf{ord}\ \alpha = 1 \qquad \mathsf{ord}\ (A \to B) = \max(1 + \mathsf{ord}\ A, \mathsf{ord}\ B) \qquad \mathsf{ord}\ \Gamma = \max_{(x:A)\in\Gamma} \mathsf{ord}\ A$$

The fragment of the language containing only variables with a type of at most order n and constants of at most order $n + 1$ is characterised by the order type system $\Gamma \vdash_n s : A$.

Definition 4.14

$\boxed{\Gamma \vdash_n s : A}$ $\boxed{\Delta \vdash_n \rho : \Gamma}$ $\boxed{\Delta \vdash_n \sigma : \Gamma}$

$$\frac{(x : A) \in \Gamma \quad \mathsf{ord}\ A \leq n}{\Gamma \vdash_n x : A} \qquad \frac{\mathsf{ord}\ (\Omega c) \leq n + 1}{\Gamma \vdash_n c : \Omega c}$$

$$\frac{\Gamma \vdash_n s : A \to B \quad \Gamma \vdash_n t : A}{\Gamma \vdash_n s\ t : B} \qquad \frac{\Gamma, x : A \vdash_n s : B}{\Gamma \vdash_n \lambda x.s : A \to B}$$

$$\frac{\forall (x : A) \in \Gamma.\ (\rho x : A) \in \Delta}{\Delta \vdash_n \rho : \Gamma} \qquad \frac{\forall (x : A) \in \Gamma.\ \Delta \vdash_n \sigma x : A}{\Delta \vdash_n \sigma : \Gamma}$$

For renamings the definition of $\Delta \vdash_n \rho : \Gamma$ coincides with $\Delta \vdash \rho : \Gamma$. Note that under this definition the order of a term is neither always unique nor can it always be computed by inspecting only the term itself. For example, the term $\lambda x.x$ can be typed as $\Gamma \vdash_1 \lambda x.x : \alpha \to \alpha$ and as $\Gamma \vdash_2 \lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha)$. When it comes to unification, we are mostly interested in the order of the free variables which is always determined by Γ.

Every well-typed term can be typed at some order and every order-typed term is well-typed. Furthermore, the judgement is monotone with respect to the order.

Fact 4.15

1. *If* $\Gamma \vdash_n s : A$, *then* $\Gamma \vdash s : A$.
2. *If* $\Gamma \vdash s : A$, *then* $\Gamma \vdash_n s : A$ *for some* n.
3. *If* $\Gamma \vdash_n s : A$ *and* $n \leq m$, *then* $\Gamma \vdash_m s : A$.

All properties also hold for substitutions.

We can generalise Lemma 4.12. All properties stated in Fact 4.11 also hold for the order type system $\Gamma \vdash_n s : A$.

Lemma 4.16 *If* $\Gamma \vdash_n s : A$ *and* $x \in \mathsf{vars}\ s$*, then* $(x : B) \in \Gamma$ *for some* B *with* $\mathsf{ord}\ B \leq n$*.*

Fact 4.17 (Order-Preservation) *Let* $\Gamma \vdash_n s : A$*.*

1. *If* $\Delta \vdash_n \rho : \Gamma$*, then* $\Delta \vdash_n s[\rho] : A$*.*
2. *If* $\Delta \vdash_n \sigma : \Gamma$*, then* $\Delta \vdash_n s[\sigma] : A$*.*
3. *If* $s \succ s'$*, then* $\Gamma \vdash_n s' : A$*.*
4. *If* $s \succ^* s'$*, then* $\Gamma \vdash_n s' : A$*.*

The statement for preservation under substitution and renaming can be strengthened such that σ and ρ only have to be typed properly for the free variables of s.

Fact 4.18 *Let* $\Gamma \vdash_n s : A$*.*

1. *If for all* $(x : B) \in \Gamma$ $x \in \mathsf{vars}\ s$ *implies* $(\rho x, B) \in \Delta$*, then* $\Delta \vdash_n s[\rho] : A$*.*
2. *If for all* $(x : B) \in \Gamma$ $x \in \mathsf{vars}\ s$ *implies* $\Delta \vdash_n \sigma x : B$*, then* $\Delta \vdash_n s[\sigma] : A$*.*

4.3 Lists of Terms

On paper, terms like $s_1(\cdots(s_n\ t)\cdots)$ are easy to read and understand. Working in a proof assistant on the other hand, writing "$\cdots$" is not an option. However, the previous term can be seen as the application of a list of terms $[s_1, \ldots, s_n]$ from the left to the term t. Therefore we generalise several notions of the previous sections to lists of terms S, T, lists of variables X, Y, and lists of types L, K. Substitutions and renamings are extended to lists by applying them successively to each element of the list, explicitly $S[\rho] := [s[\rho] \mid s \in S]$ and $S[\sigma] := [s[\sigma] \mid s \in S]$. By $\sigma[X := S]$ we denote the substitution obtained through extending σ by pointwise binding the terms S to the variables X.

We lift both reduction and typing to lists yielding a type system $\Gamma \vdash S : L$, an order type system $\Gamma \vdash_n S : L$, and a small-step semantics $S \succ S'$.

$\boxed{S \succ S'}$
$\boxed{\Gamma \vdash S : L}$
$\boxed{\Gamma \vdash_n S : L}$

Definition 4.19

$$\frac{s \succ s'}{s :: S \succ s' :: S} \qquad \frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{nil}} \qquad \frac{}{\Gamma \vdash_n \mathsf{nil} : \mathsf{nil}}$$

$$\frac{S \succ S'}{s :: S \succ s :: S'} \qquad \frac{\Gamma \vdash s : A \quad \Gamma \vdash S : L}{\Gamma \vdash (s :: S) : (A :: L)} \qquad \frac{\Gamma \vdash_n s : A \quad \Gamma \vdash_n S : L}{\Gamma \vdash_n (s :: S) : (A :: L)}$$

$\boxed{\mathsf{ord}\ L}$

The order of a list of types is defined by $\mathsf{ord}\ L := \max_{A \in L} \mathsf{ord}\ A$. Confluence of the semantics is inherited from confluence of the single term reduction, which entails that the equivalence closure of the list reduction $\succ$, also denoted by $\equiv$, satisfies the Church-Rosser property.

Fact 4.20 *The reduction strategy $\succ$ is confluent on lists.*

Instead of extending the syntax of the calculus itself, we introduce metalevel types and operations on lists. The type constructor $\to$ is lifted to lists by:

Definition 4.21 $\boxed{L \to A}$

$$\mathsf{nil} \to A = A \qquad (B :: L) \to A = B \to (L \to A)$$

The notions of list application to a term from the left, written $S\ t$, or from the right, written $s\ T$, as well as a list indexed abstraction, written $\Lambda X.s$, are defined as:

Definition 4.22

$\boxed{S\ t}$ $\boxed{s\ T}$ $\boxed{\Lambda X.s}$

$$\mathsf{nil}\ t = t \qquad s\ \mathsf{nil} = s \qquad \Lambda\mathsf{nil}.s = s$$

$$(s :: S)\ t = s\ (S\ t) \qquad s\ (t :: T) = (s\ T)\ t \qquad \Lambda x :: X.s = \lambda x.\Lambda X.s$$

In the Coq formalisation, because of the De Bruijn representation, we index the list abstraction Λ by a natural number n instead of a list of variables X.

By nature of the extension, these new operators inherit many properties of their single term counterparts. We do not explicitly spell out the details of how substitution and renaming behave in the context of the list operators but summarize results regarding the operational semantics in the following lemma:

Lemma 4.23

1. *$\succ$, $\succ^*$, and $\equiv$ are compatible with list application from the left, list application from the right, and list abstractions.*
2. *In the presence of list application from the left, list application from the right, and list abstractions normality is preserved through the term structure.*

Using the list operators, we can generalise the β-rule to:

Lemma 4.24

$$\frac{|X| = |T|}{(\Lambda X.s)\ T \succ^* s[X := T]}$$

Proof By generalising the claim to $(\Lambda X +\!\!+ Y.s)[\sigma]\ T \succ^* \Lambda Y.s[\sigma[X := T]]$ and induction on T. □

We derive typing rules of the list operations for both $\vdash$ and $\vdash_n$. Note that the typing rule for application from the left is only a special case of a more general rule. However, for our purposes this one will suffice.

Lemma 4.25

$$\frac{|L| = |X| \quad \Gamma, X : \overleftarrow{L} \vdash s : A}{\Gamma \vdash \Lambda X.s : L \to A} \qquad \frac{\Gamma \vdash S : (A \to A)^n \quad \Gamma \vdash t : A}{\Gamma \vdash S\ t : A}$$

$$\frac{\Gamma \vdash s : \overleftarrow{L} \to A \quad \Gamma \vdash T : L}{\Gamma \vdash s\ T : A}$$

All rules also hold for order typing.

In accordance with Chapter 3 $(A \to A)^n$ here denotes the list containing $A \to A$ exactly n times. The effect of the above rules can be inverted using the following inversion principles. For list application from the left we cannot recover the premise of the above rule as it is only a special case of a more general rule. Thus, we settle on S and t being well-typed in the context Γ.

Lemma 4.26

1. *If* $\Gamma \vdash \Lambda X.s : B$, *then* $\Gamma, X : \overleftarrow{L} \vdash s : A$ *and* $B = L \to A$ *and* $|L| = |X|$ *for some* L, A.
2. *If* $\Gamma \vdash S\ t : B$, *then* $\Gamma \vdash S : L$ *and* $\Gamma \vdash t : A$ *for some* L, A.
3. *If* $\Gamma \vdash s\ T : B$, *then* $\Gamma \vdash T : L$ *and* $\Gamma \vdash s : \overleftarrow{L} \to A$ *for some* L.

All properties also hold for order typing.

As the applicative head is defined by recursion only the left side of applications, we can decompose every term s into its applicative head and a list of arguments being applied from the right.

Lemma 4.27 $s = (\mathsf{head}\ s)\ T$ *for some list* T.

Whenever we encounter a normal term s, s can be written as a sequence of abstractions followed by some atom applied to a list of normal arguments. We capture this form with an inductive predicate $\mathsf{nf}\ s$.

$\boxed{\mathsf{nf}\ s}$

Lemma 4.28 *Every normal term* s *satisfies the inductive predicate*

$$\frac{s = \Lambda X.a\ T \quad \forall t \in T.\ \mathsf{nf}\ t}{\mathsf{nf}\ s}$$

This predicate gives rise to a useful induction principle on normal forms. Instead of proving claims by induction on the structure of the normal form s, we can do an induction on $\mathsf{nf}\ s$.

4.4 Confluence, Normalisation & Evaluation

In this section we establish standard results about our version of the simply-typed λ-calculus. We prove the reduction strategy $\succ$ is confluent. Furthermore, we prove that every term $\Gamma \vdash s : A$ is weakly normalising. Weak normalisation will allow us to reason about normal forms of typed terms $\Gamma \vdash s : A$ in subsequent chapters. In particular, we use weak normalisation to obtain an evaluator ξ for typed terms which is used to normalise typed substitutions in subsequent chapters.

4.4.1 Confluence

Following the standard technique introduced by Tait and Martin-Löf and refined by Takahashi [57], we prove confluence of $\succ$. As the proof is well known, we only give the definition of the parallel reduction relation $\succcurlyeq$ and the maximal reduction function ς and state the necessary lemma.

Definition 4.29

$$\begin{aligned} \varsigma x &= x \\ \varsigma c &= c \\ \varsigma(\lambda x.s) &= \lambda x.\varsigma s \\ \varsigma((\lambda x.s)\ t) &= (\varsigma s)[\varsigma t/x] \\ \varsigma(s\ t) &= (\varsigma s)\ (\varsigma t) \quad \textit{othw.} \end{aligned}$$

$$\frac{}{x \succcurlyeq x} \qquad \frac{}{c \succcurlyeq c}$$

$$\frac{s \succcurlyeq s'}{\lambda x.s \succcurlyeq \lambda x.s'} \qquad \frac{s \succcurlyeq s' \quad t \succcurlyeq t'}{(\lambda x.s)\ t \succcurlyeq s'[t'/x]}$$

$$\frac{s \succcurlyeq s' \quad t \succcurlyeq t'}{s\ t \succcurlyeq s'\ t'}$$

$\boxed{\varsigma s}$

$\boxed{s \succcurlyeq t}$

The function ς can be interpreted as a reduction strategy reducing all β-redexes that occur inside of its argument. The relation $\succcurlyeq$ allows for the reduction of multiple β-redexes in the left term. The following lemma ensures that ς is an upper bound as to how far $\succcurlyeq$ can reduce in one step.

Lemma 4.30

1. *$\succcurlyeq$ is reflexive and compatible with both renamings and substitutions.*
2. *$\succcurlyeq^*$ is compatible with abstractions and applications.*
3. $\succ \subseteq \succcurlyeq \subseteq \succ^*$
4. *If $s \succcurlyeq t$, then $t \succcurlyeq \varsigma s$.*

These properties are sufficient to conclude the confluence of $\succ$.

Fact 4.31 (Confluence) *If $s \succ^* t_1$ and $s \succ^* t_2$, then there is a term t such that $t_1 \succ^* t$ and $t_2 \succ^* t$.*

4.4.2 Normalisation

Girard [24] was the first to give a strong normalisation proof of the simply-typed λ-calculus with unrestricted reduction. In contrast to normalisation proofs for restricted reduction strategies [15], unrestricted reduction entails that abstractions are no longer normal forms. We compensate this difference by adapting a technique presented in [22]. The authors prove strong normalisation for the call-by-push-value λ-calculus with unrestricted reduction. We modify their technique to yield a weak normalisation proof for this version of the λ-calculus.

The main idea of the proof is to establish that well-typed terms $\Gamma \vdash s : A$ are semantically well-typed, written $\Gamma \vDash s : A$, a property we will refer to as semantic soundness. $\Gamma \vDash s : A$ in turn will entail that s is weakly normalising. Similar to [22] the judgement $\vDash$ relies on the definition of logical relations, in our case $\mathcal{V}[A]$, $\mathcal{E}[A]$ and $\mathcal{G}[\Gamma]$. In order to define these relations, we adapt the notion of an *active* term to the simply-typed λ-calculus. A term s is active if and only if s is an abstraction. We group all normal, active terms that behave similar to expressions of type A in the value relation $\mathcal{V}[A]$. The expression relation $\mathcal{E}[A]$ is the extension of $\mathcal{V}[A]$ by all normalising terms, whose normal form is in the relation $\mathcal{V}[A]$ if it is active.

Definition 4.32

$$\mathcal{V}[\alpha] \coloneqq \emptyset \qquad \mathcal{V}[A \to B] \coloneqq \{\lambda x.s \mid \mathsf{normal}\ s \textit{ and } \forall t \in \mathcal{E}[A].\forall \rho.(\lambda x.s)[\rho]\ t \in \mathcal{E}[B]\}$$

$$\mathcal{E}[A] \coloneqq \{s \mid \exists t.\ s \triangleright t \textit{ and if } t \textit{ is active, then } t \in \mathcal{V}[A]\}$$

$$\mathcal{G}[\Gamma] \coloneqq \{\sigma \mid \forall (x : A) \in \Gamma.\ \sigma x \in \mathcal{E}[A]\}$$

$$\Gamma \vDash s : A \coloneqq \forall \sigma \in \mathcal{G}[\Gamma].\ s[\sigma] \in \mathcal{E}[A]$$

The relation $\mathcal{E}[A]$ only requires *active* normal forms to be in $\mathcal{V}[A]$. As a consequence, the normalisation of open terms is possible. Variables are contained in the $\mathcal{E}[A]$ relation and therefore the identity substitution is a possible choice for σ in the definition of $\vDash$. To ease the proof of semantic soundness, we fix some properties about our logical relations. By construction, the relations $\mathcal{V}[A], \mathcal{E}[A]$ and $\mathcal{G}[\Gamma]$ are Kripke logical relations. The relation $\mathcal{E}[A]$ is closed under both expansion and reduction.

Lemma 4.33

1. *$x \in \mathcal{E}[A]$, $id \in \mathcal{G}[\Gamma]$.*
2. *If $s \in \mathcal{E}[A]$ and $\sigma \in \mathcal{G}[\Gamma]$, then $\sigma[x := s] \in \mathcal{G}[\Gamma, x : A]$.*
3. *Let $s \succ^* t$. Then $s \in \mathcal{E}[A]$ iff $t \in \mathcal{E}[A]$.*
4. *The relations $\mathcal{V}[A], \mathcal{E}[A]$ and $\mathcal{G}[\Gamma]$ are closed under renaming. Explicitly, if $R \in \{\mathcal{V}[A], \mathcal{E}[A], \mathcal{G}[\Gamma]\}$ and $x \in R$, then $x[\rho] \in R$.*

The proof of semantic soundness is by induction on the typing judgement using the following compatibility lemmas.

Lemma 4.34

1. *If* $(x : A) \in \Gamma$ *and* $\sigma \in \mathcal{G}[\Gamma]$, *then* $\sigma x \in \mathcal{E}[A]$.
2. $c \in \mathcal{E}[\Omega c]$
3. *If* $s \in \mathcal{E}[B]$ *and* $\forall t \rho.\ t \in \mathcal{E}[A] \to (\lambda x.s)[\rho]\ t \in \mathcal{E}[B]$, *then* $\lambda x.s \in \mathcal{E}[A \to B]$.
4. *If* $s \in \mathcal{E}[A \to B]$ *and* $t \in \mathcal{E}[A]$, *then* $s\ t \in \mathcal{E}[B]$.

Proof The third and fourth claim follow using Lemma 4.33 as well as the definition of $\mathcal{V}[A \to B]$. The others are trivial. □

Lemma 4.35 *If* $\Gamma \vdash s : A$, *then* $\Gamma \vDash s : A$.

Proof By induction on $\Gamma \vdash s : A$ using Lemma 4.34. All cases are trivial except for the case of abstractions. Let $\Gamma, x : A \vDash s : B$ and $\sigma \in \mathcal{G}[\Gamma]$. We establish $s \in \mathcal{E}[B]$ by using the inductive hypothesis and Lemma 4.33. Using Lemma 4.34 it remains to prove that under the assumption $t \in \mathcal{E}[A]$ we can show $(\lambda x.s)[\rho]\ t \in \mathcal{E}[B]$. Using Lemma 4.33 it suffices to prove $s[\sigma[\rho][x := t]] \in \mathcal{E}[B]$ which follows with the inductive hypothesis and Lemma 4.33. □

Corollary 4.36

1. *If* $\Gamma \vdash s : A$, *then* $s \triangleright t$ *for some* t.
2. *If* $\Gamma \vdash_n s : A$, *then* $s \triangleright t$ *for some* t.

Proof Follows from Lemma 4.35 with $\sigma := id$. □

4.4.3 Evaluation

As a by-product of the above proofs we can utilise the reduction strategy ς to obtain a step-indexed interpreter ξ_n for $\succ^*$ and an evaluator ξ of typed terms.

Definition 4.37 $\boxed{\xi_n s}$

$$\xi_0 s = \emptyset \qquad \xi_{n+1} s = s \quad \textit{if } \delta s \qquad \xi_{n+1} s = \xi_n(\varsigma s) \quad \textit{othw.}$$

where δ *is a decider for the normality of terms.*

For the correctness of the step indexed interpreter we establish:

Lemma 4.38

1. $s \triangleright t$ *iff* $\xi_n s = t$ *for some* n.
2. *If* $n \leq m$ *and* $\xi_n s = t$, *then* $\xi_m s = t$.
3. *If* $s \triangleright t$ *for some* t, *then we can compute the normal form of* s.

Weak normalisation of typed terms allows us to lift the step indexed interpreter ξ_n to a full interpreter ξ on typed terms.

$\boxed{\xi s}$

Definition 4.39 *By Lemma 4.38 and Corollary 4.36 there exists a function mapping well-typed terms* s *to their normal form. We denote this function with* ξs *and leave the typing information implicit on paper.*

Lemma 4.40

$$s \triangleright \xi s \qquad \mathsf{normal}\ (\xi s) \qquad \frac{\Gamma \vdash s : A}{\Gamma \vdash \xi s : A} \qquad \frac{\Gamma \vdash_n s : A}{\Gamma \vdash_n \xi s : A}$$

Chapter 5

Unification

Higher-order unification is the process of finding a well-typed substitution for the free variables in two typed terms such that under the substitution both terms are equivalent. As an example, we consider $\lambda xy.fx$ and $\lambda xy.fy$ of type $\alpha \to \alpha \to \alpha$ in the context $\Gamma = (f : \alpha \to \alpha)$. Recall from Section 2.1 that if we replace the free variable f with the term $\Delta \vdash \lambda\_.z : \alpha \to \alpha$ in the context $\Delta = (z : \alpha)$, then the resulting terms are equivalent, i.e. $\lambda xy.(\lambda\_.z)x \equiv \lambda xy.z \equiv \lambda xy.(\lambda\_.z)y$.

Note that unifiability in the above example crucially depends on the fact that the substitution may introduce a new variable of type α. If α is an empty type, then there is no closed, well-typed substitution unifying $\lambda xy.fx$ and $\lambda xy.fy$ because the variable f has to be replaced by some term of type $\alpha \to \alpha$. All closed terms of type $\alpha \to \alpha$ are equivalent to the identity function $\lambda x.x$ and inserting the identity function for f does not unify both terms.

5.1 Higher-Order Unification

In general, a substitution σ unifies two terms s and t if $s[\sigma] \equiv t[\sigma]$ and σ agrees with the types of the variables of s and t. Explicitly, if s and t have the same type in context Γ, then σ must satisfy $\Delta \vdash \sigma : \Gamma$ for some typing context Δ. While Δ may be chosen by the substitution, the requirement $\Delta \vdash \sigma : \Gamma$ enforces that the same context Δ is used to type all the terms inserted by the substitution. We speak of higher-order unification **U**, if no restriction is applied to the order of terms.

Definition 5.1
A higher-order unification instance is a dependent tuple $(\Gamma, s, t, A, H_1, H_2)$ *consisting of a typing context* Γ*, two terms* s *and* t*, a type* A*, a proof* H_1 *of* $\Gamma \vdash s : A$*, and a proof* H_2 *of* $\Gamma \vdash t : A$*. We write* $\Gamma \vdash s \stackrel{?}{=} t : A$ *if we want to refer to* Γ, s, t, A *explicitly and leave the typing information implicit.*

$\boxed{\Gamma \vdash s \stackrel{?}{=} t : A}$

$$\mathbf{U}(\Gamma \vdash s \stackrel{?}{=} t : A) := \exists \Delta \sigma.\ \Delta \vdash \sigma : \Gamma \text{ and } s[\sigma] \equiv t[\sigma] \qquad \boxed{\mathbf{U}}$$

In the previous example, we establish $\mathbf{U}(\Gamma \vdash \lambda xy.fx \stackrel{?}{=} \lambda xy.fy : \alpha \to \alpha)$ for $\Gamma = (f : \alpha \to \alpha)$ with the substitution $\sigma f = \lambda\_.z$ in context $\Delta = (z : \alpha)$. The terms $\Delta \vdash \lambda x.x \stackrel{?}{=} \lambda x.z : \alpha \to \alpha$ are an example of two terms which are not unifiable. There does not exist a term which can be substituted for z such that the resulting terms would be equivalent since we utilise capture-avoiding substitution.

Normalisation In this paragraph we show that reduction has no effect on unifiability since unifiability is defined up to equivalence. In particular, we show that arbitrary parts of the definition may be assumed to be normal. The higher-order unification problem **U** postulates the existence of a well-typed substitution σ without reference to normality. While it will sometimes be convenient to give a substitution which is not normal, we prove in the following that there always exists a corresponding substitution inserting normal forms only. The key to obtaining this substitution is the evaluator ξ introduced in Section 4.4.

NU

Definition 5.2

$$\mathbf{NU}(\Gamma \vdash s \stackrel{?}{=} t : A) := \exists \Delta \sigma.\ \Delta \vdash \sigma : \Gamma \textit{ and } s[\sigma] \equiv t[\sigma] \textit{ and } \forall x.\ \mathsf{normal}\ (\sigma x)$$

Fact 5.3

1. *If $\Delta \vdash \sigma : \Gamma$, then we can compute a substitution τ such that $\Delta \vdash \tau : \Gamma$, $\forall x \in \mathsf{dom}\ \Gamma.\ \mathsf{normal}\ (\tau x)$, and $\sigma x \succ^* \tau x$ for all x.*
2. *$\mathbf{U}(\Gamma \vdash s \stackrel{?}{=} t : A)$ iff $\mathbf{NU}(\Gamma \vdash s \stackrel{?}{=} t : A)$.*

Proof

1. Pick the substitution τ such that $\tau x = \sigma x$ if $x \notin \mathsf{dom}\ \Gamma$ and $\tau x = \xi(\sigma x)$ if $x \in \mathsf{dom}\ \Gamma$.
2. The "only if" direction is trivial. For the "if" direction assume $\Delta \vdash \tau : \Gamma$ and $s[\tau] \equiv t[\tau]$ for some substitution τ and context Δ. By Item 1 we obtain a substitution τ' with $\forall x \in \mathsf{dom}\ \Gamma.\ \mathsf{normal}\ (\tau' x)$ and $\forall x.\ \sigma x \succ^* \tau' x$. Pick $\sigma x := \tau' x$ if $x \in \mathsf{dom}\ \Gamma$ and $\sigma x := x$ otherwise. Since σ and τ are extensionally equivalent on $\mathsf{dom}\ \Gamma \supseteq \mathsf{vars}\ [s, t]$, meaning $\sigma x \equiv \tau x$ for all $x \in \mathsf{dom}\ \Gamma$, we obtain $s[\sigma] \equiv t[\sigma]$ with Fact 4.7. □

We prove that unification is invariant under reduction in the sense that the terms s, t may be replaced with equivalent terms s', t' as long as s' and t' have the same type. As a consequence, unifiability of two typed terms s and t is equivalent to the unifiability of their normal forms. This justifies extending the evaluator ξ to unification instances by $\xi(\Gamma \vdash s \stackrel{?}{=} t : A) = \Gamma \vdash \xi s \stackrel{?}{=} \xi t : A$.

Fact 5.4 *Let $\Gamma \vdash s \stackrel{?}{=} t : A$ and $\Gamma \vdash s' \stackrel{?}{=} t' : A$ such that $s \equiv s'$ and $t \equiv t'$.*

1. *$\mathbf{U}(\Gamma \vdash s \stackrel{?}{=} t : A)$ iff $\mathbf{U}(\Gamma \vdash s' \stackrel{?}{=} t' : A)$.*
2. *$\mathbf{U}(\Gamma \vdash s \stackrel{?}{=} t : A)$ iff $\mathbf{U}(\xi(\Gamma \vdash s \stackrel{?}{=} t : A))$.*

5.2 Systems of Equations

While Huet [28], Goldfarb [25] and Snyder and Gallier [52] define higher-order unification as the problem of unifying a single equation, Dowek [13] considers unification the problem of unifying multiple equations. Following Snyder and Gallier [52], we refer to multiple equations as a *system of equations*. In the following, we define the problem of system unification **SU** and relate it to higher-order unification **U**.

We write $s \stackrel{?}{=} t$ for a single equation, i.e. the pair (s, t) and denote systems of equations, i.e. lists of equations by the letter E. We extend typing to systems of equations with the type system $\Gamma \vdash E : L$ and say a system of equations $\Gamma \vdash E : L$ is unifiable, if there exists a substitution $\Delta \vdash \sigma : \Gamma$ unifying all equations in E.

Definition 5.5 $\boxed{\Gamma \vdash E : L}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{nil}} \qquad \frac{\Gamma \vdash s : A \quad \Gamma \vdash t : A \quad \Gamma \vdash E : L}{\Gamma \vdash (s \stackrel{?}{=} t :: E) : (A :: L)}$$

For the sake of readability we also write $\Gamma \vdash E : L$ *for instances of system unification. Instances of system unification are dependent tuples of the form* (Γ, E, L, H) *where* H *is a proof of* $\Gamma \vdash E : L$.

$$\mathbf{SU}(\Gamma \vdash E : L) := \exists \Delta \sigma.\ \Delta \vdash \sigma : \Gamma \text{ and } E_{\mathsf{L}}[\sigma] \equiv E_{\mathsf{R}}[\sigma] \qquad \boxed{\mathbf{SU}}$$

where E_{L} *refers to the terms on the left of the equations in* E *and* E_{R} *refers to those on the right.*

We show that higher-order unification and system unification are interreducible. To prove $\mathbf{SU} \preceq \mathbf{U}$ we exploit that bound variables can neither be replaced nor reduced. For example, given the equations $s_1 \stackrel{?}{=} t_1$ and $s_2 \stackrel{?}{=} t_2$, we produce the terms $\lambda h.h\ s_1 s_2 \stackrel{?}{=} \lambda h.h\ t_1 t_2$, which are unifiable iff the original terms are unifiable by the same substitution.

Fact 5.6 $\mathbf{U} \preceq \mathbf{SU}$ *and* $\mathbf{SU} \preceq \mathbf{U}$.

Proof The first claim follows by lifting terms and types to singleton lists, explicitly $f(\Gamma \vdash s \stackrel{?}{=} t : A) := \Gamma \vdash [s \stackrel{?}{=} t] : [A]$. For the second claim pick the reduction function $f(\Gamma \vdash E : L) := \Gamma \vdash \lambda h.h\ E_{\mathsf{L}} \stackrel{?}{=} \lambda h.h\ E_{\mathsf{R}} : (\overleftarrow{L} \to \alpha) \to \alpha$. The result follows with Fact 4.8 and Lemma 4.23. □

Analogously to single equations, whenever a system of equations is unifiable, then there exists a normal substitution unifying the system.

Fact 5.7 $\mathbf{SU}(\Gamma \vdash E : L)$ *iff* $\mathbf{NSU}(\Gamma \vdash E : L)$ *where* $\boxed{\mathbf{NSU}}$

$$\mathbf{NSU}(\Gamma \vdash E : L) := \exists \Delta \sigma.\ \Delta \vdash \sigma : \Gamma \text{ and } E_{\mathsf{L}}[\sigma] \equiv E_{\mathsf{R}}[\sigma] \text{ and } \forall x.\ \mathsf{normal}\ (\sigma x)$$

5.3 Nth-Order Unification

When analysing the problem of higher-order unification **U**, we can distinguish different fragments. We speak of nth-order unification if the terms are of the nth-order fragment of the calculus. In accordance with Snyder and Gallier [52] we define the problem of nth-order unification $\mathbf{U}_n$.

$\boxed{\Gamma \vdash_n s \stackrel{?}{=} t : A}$

Definition 5.8
An instance of nth-order unification is a dependent tuple $(\Gamma, s, t, A, H_1, H_2)$ consisting of a typing context Γ, two terms s and t, a type A, a proof H_1 of $\Gamma \vdash_n s : A$, and a proof H_2 of $\Gamma \vdash_n t : A$. We write $\Gamma \vdash_n s \stackrel{?}{=} t : A$ if we want to refer to Γ, s, t, A explicitly and leave the typing information.

$\boxed{\mathbf{U}_n}$

$$\mathbf{U}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A) := \exists \Delta \sigma.\ \Delta \vdash_n \sigma : \Gamma \text{ and } s[\sigma] \equiv t[\sigma]$$

For the remainder of this thesis, we will always assume $n > 0$ when speaking of $\mathbf{U}_n$ or any related problems involving order. While $\mathbf{U}_0$ is formally defined, its instances cannot contain free variables. Furthermore, the problem is not well behaved in the sense that it is not conservative. Consider $\Gamma \vdash_0 \mathsf{g} \stackrel{?}{=} \mathsf{g} : \alpha \to \alpha \to \alpha$ in the context $\Gamma = (x : \alpha)$ for a language with only the single constant $\mathsf{g} : \alpha \to \alpha \to \alpha$. We cannot prove $\mathbf{U}_0$ of this instance because there is no substitution $\Delta \vdash_0 \sigma : \Gamma$ as there is no term $\Delta \vdash_0 s : \alpha$. However, there is the substitution $\sigma x = z$ with $z : \alpha \vdash_1 \sigma : \Gamma$ unifying both terms at order one.

System of Equations Analogously to the higher-order case, we consider systems of equations in the nth-order fragment of the calculus. We characterise nth-order systems of equations with the order type system $\Gamma \vdash_n E : L$ and the problem of nth-order system unification $\mathbf{SU}_n$ as the counterpart to **SU**.

$\boxed{\Gamma \vdash_n E : L}$

Definition 5.9

$$\frac{}{\Gamma \vdash_n \mathsf{nil} : \mathsf{nil}} \qquad \frac{\Gamma \vdash_n s : A \quad \Gamma \vdash_n t : A \quad \Gamma \vdash_n E : L}{\Gamma \vdash_n (s \stackrel{?}{=} t :: E) : (A :: L)}$$

For the sake of readability we also write $\Gamma \vdash_n E : L$ for instances of nth-order system unification. Instances of nth-order system unification are dependent tuples of the form (Γ, E, L, H) where H is a proof of $\Gamma \vdash_n E : L$.

$\boxed{\mathbf{SU}_n}$

$$\mathbf{SU}_n(\Gamma \vdash_n E : L) := \exists \Delta \sigma.\ \Delta \vdash_n \sigma : \Gamma \text{ and } E_{\mathsf{L}}[\sigma] \equiv E_{\mathsf{R}}[\sigma]$$

In contrast to **SU**, the situation for $\mathbf{SU}_n$ is somewhat different. While we can establish $\mathbf{U}_n \preceq \mathbf{SU}_n$, the proof of $\mathbf{SU} \preceq \mathbf{U}$ is not preserved. By transforming $\Gamma \vdash E : L$ into $\Gamma \vdash \lambda h.h\ E_{\mathsf{L}} \stackrel{?}{=} \lambda h.h\ E_{\mathsf{R}} : (\overleftarrow{L} \to \alpha) \to \alpha$, the order of the types involved is affected as well. The variable h has type $\overleftarrow{L} \to \alpha$ thus the resulting terms are at least of order $\mathsf{ord}\ \overleftarrow{L} + 1$. As L depends on the system unification instance, we cannot give a general reduction. However, we can establish the following result which is still useful for subsequent reductions:

Lemma 5.10
For $\mathsf{ord}\ L < n$ *define* $f(\Gamma \vdash_n E : L) := \Gamma \vdash_n \lambda h.h\ E_{\mathsf{L}} \stackrel{?}{=} \lambda h.h\ E_{\mathsf{R}} : (\overleftarrow{L} \to \alpha) \to \alpha$.
$\mathbf{SU}_n(\Gamma \vdash_n E : L)$ *iff* $\mathbf{U}_n(f(\Gamma \vdash_n E : L))$.

We conjecture that in a calculus with η-reduction the above technique can be adapted to yield a proof of $\mathbf{U}_n \preceq \mathbf{SU}_n$. The results on normalisation for **U** and **SU** can be lifted to the nth-order fragment.

Fact 5.11 *Define*

$\boxed{\mathbf{NU}_n}$ $\boxed{\mathbf{NSU}_n}$

$$\mathbf{NU}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A) := \exists \Delta\sigma.\ \Delta \vdash_n \sigma : \Gamma \textit{ and } s[\sigma] \equiv t[\sigma] \textit{ and } \forall x.\ \mathsf{normal}\ (\sigma x)$$

$$\mathbf{NSU}_n(\Gamma \vdash_n E : L) := \exists \Delta\sigma.\ \Delta \vdash_n \sigma : \Gamma \textit{ and } E_{\mathsf{L}}[\sigma] \equiv E_{\mathsf{R}}[\sigma] \textit{ and } \forall x.\ \mathsf{normal}\ (\sigma x)$$

1. $\mathbf{U}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A)$ *iff* $\mathbf{NU}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A)$
2. $\mathbf{SU}_n(\Gamma \vdash_n E : L)$ *iff* $\mathbf{NSU}_n(\Gamma \vdash_n E : L)$

Retyping Unused arguments inside a term $\Gamma \vdash_n s : A$ may increase the order of its type A beyond $n+1$. As an example consider $\lambda xy.x$. The term can be typed both as $\vdash_1 \lambda xy.x : \alpha \to \alpha \to \alpha$ and as $\vdash_3 \lambda xy.x : \alpha \to ((\alpha \to \beta) \to \beta) \to \alpha$. In the following we show that such terms with types of artificially inflated order can be retyped to types of order $n+1$ in a context of order n.

Definition 5.12 *Let* $\lceil A \rceil_n := A$ *if* $\mathsf{ord}\ A \leq n$ *and* $\lceil A \rceil_n := \alpha$ *otherwise.*

$\boxed{\mathsf{retype}_n\ A}$ $\boxed{\mathsf{retype}_n\ \Gamma}$

$$\mathsf{retype}_n\ \alpha = \alpha \qquad \mathsf{retype}_n\ (A \to B) = \lceil A \rceil_n \to \mathsf{retype}_n\ B$$

$$\mathsf{retype}_n\ \Gamma = [(x : \lceil A \rceil_n) \mid (x : A) \in \Gamma]$$

Clearly $\mathsf{ord}\ (\mathsf{retype}_n\ A) \leq n+1$ and $\mathsf{ord}\ (\mathsf{retype}_n\ \Gamma) \leq n$. We show that every normal term $\Gamma \vdash_n s : A$ can be retyped.

Lemma 5.13 $\dfrac{\Gamma \vdash_n s : A \quad \mathsf{normal}\ s}{\mathsf{retype}_n\ \Gamma \vdash_n s : \mathsf{retype}_n\ A}$

Proof By induction on the structure of $\mathsf{nf}\ s$. □

When establishing the decidability of first-order unification, it will be convenient to work with normal terms in a first-order context with a second-order type. Thus, we define $\mathsf{retype}_n\ (\Gamma \vdash_n s \stackrel{?}{=} t : A) := (\mathsf{retype}_n\ \Gamma) \vdash_n \xi s \stackrel{?}{=} \xi t : (\mathsf{retype}_n\ A)$ and establish:

Fact 5.14 $\mathbf{U}_n(\Gamma \vdash_n s \stackrel{?}{=} t : A)$ *iff* $\mathbf{U}_n(\mathsf{retype}_n\ (\Gamma \vdash_n s \stackrel{?}{=} t : A))$

The proof of the above fact requires techniques introduced in Chapter 9. Hence, in the formalisation this proof occurs alongside the results of Chapter 9.

5.4 Enumerability

Despite unification being undecidable, we can still prove that the problem is recursively enumerable. In our setting, recursive enumerability of a problem $\mathbf{P}$ is established by giving an enumeration of all instances of $\mathbf{P}$ that have a solution. Explicitly, if $\mathbf{P}$ is a problem on type X, then an enumeration of $\mathbf{P}$ is a function $e : \mathbb{N} \to \mathcal{O}X$, such that $\mathbf{P}(x)$ iff $\exists n.\ e(n) = x$ for all x. We say a type X is enumerable if we can find an enumeration of all values of the type.

Following Forster et al. [21], there is a particularly convenient style for establishing enumerability — list enumerators. List enumerators are similar to enumerations in the sense that they enumerate all values of type X which satisfy $\mathbf{P}$. However, while enumerations may return at most one value for every natural number n, a list enumerator may return finitely many elements for every natural number n in form of a list. Furthermore, it will be convenient to only work with cumulative list enumerators. A list enumerator L is cumulative, if $L(n)$ is always a prefix of $L(n+1)$. For every enumeration there exists a cumulative list enumerator and vice versa [21].

In the following we give a proof that higher-order unification is enumerable. Explicitly, this means we enumerate all equations $\Gamma \vdash s \stackrel{?}{=} t : A$ where s and t are unifiable. Using list enumerations, it is straightforward to enumerate simple types such as terms. We do nothing more than to successively apply term constructors to produce every possible term eventually.

Lemma 5.15 *Terms, types, and typing contexts are enumerable.*

Proof In order to give a list enumeration L_{tm} of terms, we assume list enumerations of constants $\mathsf{L}_{\mathcal{C}}$ and define

$$\begin{aligned} \mathsf{L}_{\mathsf{tm}} 0 &:= \mathsf{nil} \\ \mathsf{L}_{\mathsf{tm}}(n+1) &:= \mathsf{L}_{\mathsf{tm}} n +\!\!+ [x \mid x \in \mathsf{L}_{\mathbb{N}} n] \\ &\quad +\!\!+ [c \mid c \in \mathsf{L}_{\mathcal{C}} n] \\ &\quad +\!\!+ [\lambda x.s \mid (x,s) \in \mathsf{L}_{\mathbb{N}} n \times \mathsf{L}_{\mathsf{tm}} n] \\ &\quad +\!\!+ [s\ t \mid (s,t) \in \mathsf{L}_{\mathsf{tm}} n \times \mathsf{L}_{\mathsf{tm}} n] \end{aligned}$$

Types and typing contexts are enumerated analogously. □

Enumerating all unifiable equations $\Gamma \vdash s \stackrel{?}{=} t : A$ is more complicated. Besides enumerating s and t, we enumerate proofs of the well-typedness of s and t. Recall that $\Gamma \vdash s \stackrel{?}{=} t : A$ is just notation for a dependent tuple of the form $(\Gamma, s, t, A, H_1, H_2)$ where H_1 is a proof of $\Gamma \vdash s : A$ and H_2 is a proof of $\Gamma \vdash t : A$. Thus, we need an enumeration of all proofs of the proposition $\Gamma \vdash s : A$. For this purpose we define a recursive function $\mathsf{L}_{\Gamma \vdash s:A} n$ parametric over the context Γ, term s and type A yielding a list of proofs of $\Gamma \vdash s : A$ for every n. While the function is recursive on n, the recursive structure of the typing judgement on the term s is reflected by a case distinction on s.

Lemma 5.16 *The proofs of* $\Gamma \vdash s : A$ *are enumerable.*

Proof Pick

$$\mathsf{L}_{\Gamma\vdash s:A}0 := \mathsf{nil} \qquad \mathsf{L}_{\Gamma\vdash x:A}(n+1) := \mathsf{L}_{\Gamma\vdash x:A}n +\!\!+ \left[\frac{H}{\Gamma \vdash x : A} \text{ where } H : (x, A) \in \Gamma\right]$$

$$\mathsf{L}_{\Gamma\vdash c:A}(n+1) := \mathsf{L}_{\Gamma\vdash c:A}n +\!\!+ \left[\overline{\Gamma \vdash c : \Omega c} \text{ where } A = \Omega c\right]$$

$$\mathsf{L}_{\Gamma\vdash \lambda x.s:A\to B}(n+1) := \mathsf{L}_{\Gamma\vdash \lambda x.s:A\to B}n +\!\!+ \left[\frac{H}{\Gamma \vdash \lambda x.s : A \to B} \;\middle|\; H \in \mathsf{L}_{\Gamma,x:A\vdash s:B}n\right]$$

$$\mathsf{L}_{\Gamma\vdash s\ t:B}(n+1) :=$$

$$\mathsf{L}_{\Gamma\vdash s\ t:B}n +\!\!+ \left[\frac{H_1 \quad H_2}{\Gamma \vdash s\ t : B} \;\middle|\; (H_1, H_2) \in \mathsf{L}_{\Gamma\vdash s:A\to B}n \times \mathsf{L}_{\Gamma\vdash t:A}n, A \in \mathsf{L}_{\mathsf{ty}}n\right]$$

$$\mathsf{L}_{\Gamma\vdash s:A}(n+1) := \mathsf{L}_{\Gamma\vdash s:A}n \quad \text{othw.}$$

where $[f\ H \text{ where } H : P]$ is a shorthand for $[f\ H]$ if P is provable by some proof H and nil otherwise. We omit H if the P does not occur as a premise of the rule. The claim follows with a routine induction on $\Gamma \vdash s : A$. □

Corollary 5.17

1. *The dependent tuples* $\Gamma \vdash s \stackrel{?}{=} t : A$ *are enumerable by a list enumerator* L_{eq}.
2. *The predicate* $\vdash$ *is enumerable by a list enumerator* $\mathsf{L}_{\vdash}$.

Every equation $\Gamma \vdash s \stackrel{?}{=} t : A$ contains only finitely many variables. Since only the occurring variables are of interest for substitutions, enumerating all relevant substitutions is accomplished by enumerating all instantiations of those variables. Note that we cannot hope for a technique enumerating well typed substitutions in general as they are uncountable. However, for a fixed substitution τ we can enumerate all contexts Δ, Γ and substitutions $\Delta \vdash \sigma : \Gamma$ such that $\sigma x = \tau x$ for all $x \notin \mathsf{dom}\ \Gamma$. Since unifiability is not affected by variables not occurring in Γ, we may pick $\tau = id$ later on.

Lemma 5.18
We can enumerate all (Δ, σ, Γ) *such that* $\Delta \vdash \sigma : \Gamma$ *and* $\sigma x = \tau x$ *for all* $x \notin \mathsf{dom}\ \Gamma$.

Proof Extend $\mathsf{L}_{\vdash}$ to substitutions by

$$\begin{aligned}
\mathsf{L}_{\vdash}0 &:= \mathsf{nil}\\
\mathsf{L}_{\vdash}(n+1) &:= \mathsf{L}_{\vdash}n\\
&\quad +\!\!+ [(\Delta, \tau, \mathsf{nil}) \mid \Delta \in \mathsf{L}_{\mathsf{ctx}}n]\\
&\quad +\!\!+ [(\Delta, \sigma[x := s], (\Gamma, x : A)) \mid ((\Delta, \sigma, \Gamma), (\Delta, s, A), x) \in \mathsf{L}_{\vdash}n \times \mathsf{L}_{\vdash}n \times \mathsf{L}_{\mathbb{N}}n]
\end{aligned}$$

□

To conclude the enumerability of $\mathbf{U}$, we single out all the unifiable equations from the enumeration L_{eq}.

Theorem 5.19 $\mathbf{U}$ *is enumerable.*

Proof Instead of enumerating $\mathbf{U}$ directly, we enumerate the related problem $\mathbf{U}'(\Gamma \vdash s \stackrel{?}{=} t : A, \Delta, \sigma) := \Delta \vdash \sigma : \Gamma$ and $s[\sigma] \equiv t[\sigma]$ and $\forall x \notin \mathsf{dom}\ \Gamma.\ \sigma x = x$.

Pick

$$\begin{aligned} \mathsf{L}0 &:= \mathsf{nil} \\ \mathsf{L}(n+1) &:= \mathsf{L}n \\ &\quad +\!\!+ [(I, \Delta, \sigma) \mid (I, (\Delta, \sigma, \Gamma)) \in \mathsf{L}_{\mathsf{eq}} n \times \mathsf{L}_{\vdash} n \text{ where } \xi_n(s[\sigma]) = \xi_n(t[\sigma])] \end{aligned}$$

where $I = (\Gamma, s, t, A, H_1, H_2)$ and $\xi_n(s[\sigma]) = \xi_n(t[\sigma])$ is to be read as $\xi_n(s[\sigma]) = u = \xi_n(t[\sigma])$ for some term u. Using Lemma 4.38, the proof is routine. The enumerability of $\mathbf{U}$ follows from a projection of $\mathbf{U}'$ on Δ, σ, Fact 4.1 and Lemma 4.12. □

Chapter 6

Third-Order Unification

In this chapter, we formally prove the undecidability of third-order unification following a proof by Huet [28]. Huet proves that third-order unification in a language of arbitrary constants is undecidable by reduction from the Post correspondence problem **PCP** [42]. We simplify his undecidability proof by reducing from the modified Post correspondence problem **MPCP** [26] instead.

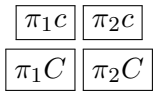
Recall from Section 2.4 that an **MPCP** instance consists of an initial card c_0 and a stack of cards C such as $11/1$ and $0/110$, $101/000$. Formally, *symbols*, *words*, *cards*, and *stacks of cards* are generated by the following abstract syntax:

$$b ::= 1 \mid 0 \qquad w, l, r ::= \epsilon \mid bw \qquad c ::= l/r \qquad C ::= \mathsf{nil} \mid c, C$$

and we write W for a list of words. In the formalisation we realise symbols b with booleans, words w with lists of symbols, cards c with pairs of words and stacks C with lists of cards. We call the upper half of a card c the *top* of c and the lower half the *bottom* of c. We introduce the operations $\pi_1 c$, $\pi_2 c$ which can be used to obtain the top and bottom of a card and lift them to stacks by $\pi_1 C$ and $\pi_2 C$.

Definition 6.1

$$\pi_1(l/r) = l \qquad \pi_2(l/r) = r \qquad \pi_1 C = [\pi_1 c \mid c \in C] \qquad \pi_2 C = [\pi_2 c \mid c \in C]$$



The Post correspondence problem **PCP** asks for a sequence of cards drawn from a stack C, possibly containing cards multiple times or not at all, in which the cards produce the same string in the top and bottom row. The modified Post correspondence problem **MPCP** is a special case of the Post correspondence problem where the first card of the sequence is fixed to some initial card c_0. Intuitively, given the cards $l_0/r_0, \ldots, l_n/r_n$ a solution of **PCP** is a non-empty sequence of indices $i_1, \ldots, i_k$ such that $l_{i_1} \cdots l_{i_k} = r_{i_1} \cdots r_{i_k}$. For a solution of **MPCP**, we require $i_1 = 0$. We call such a sequence of indices an *ordering* and represent orderings with lists of indices $I = [i_1, \ldots, i_k]$.

Forster and Larchey-Wendling [18] propose a formal characterisation of **PCP** over a binary alphabet which we adapt in this thesis. Forster et al. [20] propose a formal characterisation of **MPCP** which is equivalent to the one we provide in the following. In the context of **PCP** and **MPCP**, we will sometimes require the elements of a list A in some ordering I. For this purpose we introduce the operation $A[I]$. Furthermore, since we are interested in the concatenation of the top and bottom row of a sequence of cards, we introduce an operation for the concatenation of a list of words $\sum W$.

$\boxed{A[I]}$ $\boxed{\sum W}$

Definition 6.2

$$\begin{aligned} A[\mathsf{nil}] &= \mathsf{nil} & & & \textstyle\sum \mathsf{nil} &= \mathsf{nil} \\ A[i :: I] &= A[i] :: A[I] & i < |A| & & \textstyle\sum (w :: W) &= w \textstyle\sum W \\ A[i :: I] &= A[I] & i \geq |A| & \end{aligned}$$

The Post correspondence problem **PCP** can then formally be defined as follows: Given a stack of cards $C = l_1/r_1, \ldots, l_n/r_n$, is there an ordering $I \subseteq [1, \ldots, n]$ such that $I = [i_1, \ldots, i_k]$ is not empty and the concatenation of the upper and lower halves results in the same string, i.e. $l_{i_1} \cdots l_{i_k} = r_{i_1} \cdots r_{i_k}$. For the modified Post correspondence problem **MPCP** we are given an initial card l_0/r_0 and a stack of cards $C = l_1/r_1, \ldots, l_n/r_n$ and we ask for an ordering $I \subseteq [0, \ldots, n]$ such that $l_0 l_{i_1} \cdots l_{i_k} = r_0 r_{i_1} \cdots r_{i_k}$.

$\boxed{\textbf{PCP}}$ $\boxed{\textbf{MPCP}}$

Definition 6.3

$$\textbf{PCP}(C) := \exists I \subseteq [0, \ldots, |C| - 1].\ \textstyle\sum\big((\pi_1 C)[I]\big) = \sum\big((\pi_2 C)[I]\big) \text{ and } I \neq \mathsf{nil}$$

$$\textbf{MPCP}(c_0, C) := \exists I \subseteq [0, \ldots, |C|].\ \pi_1 c_0 \textstyle\sum\big((\pi_1 C_0)[I]\big) = \pi_2 c_0 \sum\big((\pi_2 C_0)[I]\big)$$

where $C_0 := c_0, C$

6.1 Encoding

Recall that when reducing the modified Post correspondence problem to third-order unification, we have to transform a pair $C_0 = c_0, C$ of initial card c_0 and stack of cards C into a unification equation $s \stackrel{?}{=} t$. In particular, we have to transform strings w over the alphabet $\{0, 1\}$ into λ-terms.

$\boxed{\overline{b}}$ $\boxed{\overline{w}}$ $\boxed{\overline{W}}$

Definition 6.4 *Fix the variables* u_1, u_0.

$$\overline{1} = u_1 \qquad \overline{0} = u_0 \qquad \overline{w} = \lambda x.\big[\overline{b} \;\big|\; b \in w\big]\ x \qquad \overline{W} = \big[\overline{w} \;\big|\; w \in W\big]$$

The encoding is a modified Church encoding of boolean strings. It is modified in the sense that u_1 and u_0 are not taken as arguments but instead are fixed. In the case of the cards 0/110, 101/000, the string 101 is encoded as $\lambda x.u_1\ (u_0\ (u_1\ x))$. Provided u_0, u_1 have type $\alpha \to \alpha$, our encoding returns terms of type $\alpha \to \alpha$. The typing can then be lifted to lists of encoded words.

Lemma 6.5 *Let* $\Gamma \vdash u_0 : \alpha \to \alpha$ *and* $\Gamma \vdash u_1 : \alpha \to \alpha$.

$$\frac{}{\Gamma \vdash \overline{b} : \alpha \to \alpha} \qquad \frac{}{\Gamma \vdash \overline{w} : \alpha \to \alpha} \qquad \frac{}{\Gamma \vdash \overline{W} : (\alpha \to \alpha)^{|W|}}$$

We observe that applying an encoded word $\overline{w} = \overline{b_1 \cdots b_n}$ corresponds to applying all symbols $b_1, \ldots, b_n$ from the left. For this reason concatenation of words is successive application and applying lists of encoded strings $\overline{W}$ corresponds to applying all encoded strings from the left. Intuitively, $\overline{w_1 \cdots w_n}\ s \equiv \overline{w_1}\ (\cdots (\overline{w_n}\ s) \cdots)$.

Lemma 6.6

$$\overline{\epsilon}\ s \equiv s \qquad \overline{bw}\ s \equiv \overline{b}\ (\overline{w}s) \qquad \overline{ww'}\ s \equiv \overline{w}\ (\overline{w'}s) \qquad \overline{\textstyle\sum W}\ s \equiv \overline{W}\ s$$

We establish that substitutions σ, not affecting u_1 and u_0, do not affect encoded terms.

Lemma 6.7 *If* $\sigma u_1 = u_1$ *and* $\sigma u_0 = u_0$, *then* $\overline{b}[\sigma] = \overline{b}$, $\overline{w}[\sigma] = \overline{w}$, *and* $\overline{W}[\sigma] = \overline{W}$

The above lemma is heavily used in the formal development. For the sake of simplicity we do not use it on paper explicitly. Instead, we assume that substitutions solving the $\mathbf{U}_3$ instance of the reduction do not contain u_0 or u_1 as free variables.

In proof of the reduction we will also need to conclude the equality of two encoded words from their equivalence when applied to arguments. In general, this is clearly not the case. Take $\overline{11}\ (u_0\ z)$ and $\overline{110}\ z$ for instance. While both terms are equivalent, the encoded strings are not the same. However, considering equations of the form $\overline{w}\ s \equiv \overline{w'}\ t$, we can conclude the equality of w and w' if we can ensure that s and t never produce u_1 or u_0 on the left side of an application.

Lemma 6.8 *Let* $s \not\equiv u_0\ s'$, $s \not\equiv u_1\ s'$, $t \not\equiv u_0 s'$, $t \not\equiv u_1\ s'$ *for all* s'.

1. *If* $\overline{b_1} \equiv \overline{b_2}$, *then* $b_1 = b_2$.
2. *If* $\overline{w}\ s \equiv \overline{w'}\ t$, *then* $w' = w'$.

Proof

1. Case analysis on b.
2. By induction on w with w' generalised using Lemma 6.6 and Item 1. □

6.2 MPCP Reduction

Given a pair $C_0 = c_0, C$ consisting of a stack of cards $C = c_1, \ldots, c_n$ and an initial card c_0, we construct an equation $\hat{\Gamma} \vdash_3 \hat{s} \stackrel{?}{=} \hat{t} : \hat{A}$ such that $\hat{s}$ and $\hat{t}$ are unifiable if and only if the cards can form a solving sequence. Pick

$$\begin{aligned}
\hat{\Gamma} &:= x_f : (\alpha \to \alpha)^{|C_0|} \to \alpha \\
\hat{s} &:= \lambda u_0 u_1.\overline{\pi_1 c_0}\ (x_f\ \overline{\pi_1 C_0}) \\
\hat{t} &:= \lambda u_0 u_1.\overline{\pi_2 c_0}\ (x_f\ \overline{\pi_2 C_0}) \\
\hat{A} &:= (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha
\end{aligned}$$

The typing of both terms follows with Lemmas 4.25 and 6.5. A substitution σ cannot instantiate u_1, u_0 in the above equations because they are bound.

As a consequence of the definition of applying a list of terms from the right, the order in which the encoded card halves are applied to x_f is not the same as the order in which they appear in the stack and in Chapter 2. For instance, the term $\hat{s}$, when unfolding the definition of list application, reads $\lambda u_0 u_1.\overline{l_0}\ (x_f\ \overline{l_n} \cdots \overline{l_1}\ \overline{l_0})$ for $l_i := \pi_1 c_i$. While this order may be counter-intuitive, it simplifies the formalisation. With this order of arguments the De Bruijn indices of the variables coincide with the indices in a corresponding ordering. Recall the solution of the above example. While on paper we write $\lambda x_2 x_1 x_0.x_0\ (x_1\ z)$, in the formalisation this term is represented by $\lambda\lambda\lambda 0\ (1\ 3)$. In the solution of this particular choice of cards, we use 0 to refer to the initial card and 1 to refer to the second card. Hence we do not have to apply any transformation from the variables to the indices of an ordering. If the arguments were given in ascending order, each De Bruijn index i would have to be inverted, i.e. transformed into $n - i$ to obtain I.

The direction from **MPCP** to $\mathbf{U}_3$ is straightforward. Given a solution $i_1, \ldots, i_k$ for cards $l_0/r_0, \ldots, l_n/r_n$, we substitute the variable x_f with the term

$$\lambda x_n \cdots x_1 x_0.x_{i_1}\ (\cdots (x_{i_k}\ z) \cdots)$$

which unifies both sides of the equation since $l_0 l_{i_1} \cdots l_{i_k} = r_0 r_{i_1} \cdots r_{i_k}$.

Lemma 6.9 *If* $\mathbf{MPCP}(C_0)$*, then* $\mathbf{U}_3(\hat{\Gamma} \vdash_3 \hat{s} \stackrel{?}{=} \hat{t} : \hat{A})$.

Proof Let $I \subseteq [0, \ldots, n]$ be an ordering solving the **MPCP** instance. Fix variables $X = [x_n, \ldots, x_1, x_0]$ and pick

$$\Delta := (z : \alpha) \qquad\qquad \sigma x_f := \lambda X.X[I]\ z$$

With Lemma 4.25 typing $\Delta \vdash \sigma : \hat{\Gamma}$ follows and it remains to show that $s[\sigma] \equiv t[\sigma]$ which under the use of Fact 4.7 amounts to $\overline{\pi_1 c_0}\ (\sigma x_f\ \overline{\pi_1 C_0}) \equiv \overline{\pi_2 c_0}\ (\sigma x_f\ \overline{\pi_2 C_0})$:

$$\overline{\pi_1 c_0}\ ((\lambda X.X[I]\ z)\ \overline{\pi_1 C_0}) \equiv \overline{\pi_2 c_0}\ ((\lambda X.X[I]\ z)\ \overline{\pi_2 C_0}) \quad (\sigma x_f = \lambda X.X[I]\ z)$$

$$\text{iff} \quad \overline{\pi_1 c_0}\ (\overline{\pi_1 C_0}[I]\ z) \equiv \overline{\pi_2 c_0}\ (\overline{\pi_2 C_0}[I]\ z) \quad \text{(Lemma 4.24)}$$

$$\text{iff} \quad \overline{\pi_1 c_0}\ (\overline{(\pi_1 C_0)[I]}\ z) \equiv \overline{\pi_2 c_0}\ (\overline{(\pi_2 C_0)[I]}\ z) \quad (\overline{W}[I] = \overline{W[I]})$$

$$\text{iff} \quad \overline{\pi_1 c_0 \textstyle\sum\big((\pi_1 C_0)[I]\big)}\ z \equiv \overline{\pi_2 c_0 \textstyle\sum\big((\pi_2 C_0)[I]\big)}\ z \quad \text{(Lemma 6.6)}$$

which follows since by assumption I is a solution to the **MPCP** instance, meaning $\pi_1 c_0 \sum((\pi_1 C_0)[I]) = \pi_2 c_0 \sum((\pi_2 C_0)[I])$. □

For the converse direction we start from $\mathbf{NU}_3$ instead of $\mathbf{U}_3$. This step is justified by Fact 5.11. The idea of the proof is that the term inserted for x_f must be equivalent to some term of the shape $\lambda x_n x_{n-1} \ldots x_m . x_{i_1}\ (\cdots (x_{i_k}\ t) \cdots)$ for some t where t is not an application of any of the x_i's from the left. To obtain such decompositions, we show:

Lemma 6.10 *For every decidable predicate on terms P and term s, we can compute the longest list S satisfying P such that $s = S\ t$ for some t.*

As in the introductory example the ordering $[i_1, \ldots, i_k]$ in which the arguments are applied yields a solution to the unification instance. From the unifiability of both terms we can derive the equation $\overline{l_0 l_{i_1} \cdots l_{i_k}}\ s' \equiv \overline{r_0 r_{i_1} \cdots r_{i_k}}\ t'$ for some s', t' which do not produce any more u_0, u_1's. This allows us to conclude the equality of both encoded strings.

Lemma 6.11 *If* $\mathbf{NU}_3(\hat{\Gamma} \vdash_3 \hat{s} \stackrel{?}{=} \hat{t} : \hat{A})$, *then* $\mathbf{MPCP}(C_0)$.

Proof Assume $\Delta \vdash_3 \sigma : \hat{\Gamma}$, such that $\forall x.\ \mathsf{normal}\ (\sigma x)$ and $\hat{s}[\sigma] \equiv \hat{t}[\sigma]$. With Lemma 4.28 we know that $\sigma x_f = \Lambda X.e$ with $e = a\ T$ for some atom a, some list of variables X, and some list of terms T. Using Fact 4.8 we obtain the equivalence

$$\overline{\pi_1 c_0}\ ((\Lambda X.e)\ \overline{\pi_1 C_0}) \equiv \overline{\pi_2 c_0}\ ((\Lambda X.e)\ \overline{\pi_2 C_0})$$

From the judgement $\Delta \vdash_3 \sigma : \hat{\Gamma}$ we know $\Delta \vdash \Lambda X.e : (\alpha \to \alpha)^{|C_0|} \to \alpha$. Thus with Lemma 4.26 we deduce

$$\Delta, X : (\alpha \to \alpha)^{|X|} \vdash e : (\alpha \to \alpha)^{|C_0| - |X|} \to \alpha$$

We decompose C_0 into two lists C_1, C_2 such that $C_0 = C_1 +\!\!+ C_2$ and $|X| = |C_2|$ and proceed by case analysis on C_1.

1. Let $C_1 = \mathsf{nil}$. This means $|X| = |C_0|$ and $\Delta, X : (\alpha \to \alpha)^{|C_0|} \vdash e : \alpha$. Using Lemma 6.10 we decompose e into $S\ t$ for the longest list S such that for all $s \in S$ the predicate $P(s) := s \in X$ holds. "Longest" in this context means that if $t = s_1\ s_2$ for some s_1, s_2, then $P(s_1)$ does not hold. Clearly all elements of S are variables hence we have some ordering $I \subseteq [0, \ldots, n]$ such that $S = X[I]$.

   Define $t_1 := t[X := \overline{\pi_1 C_0}]$ and $t_2 := t[X := \overline{\pi_2 C_0}]$. Then

   $$\overline{\pi_1 c_0}\ ((\Lambda X.X[I]\ t)\ \overline{\pi_1 C_0}) \equiv \overline{\pi_2 c_0}\ ((\Lambda X.X[I]\ t)\ \overline{\pi_2 C_0}) \quad (e = X[I]\ t)$$
   $$\text{iff} \quad \overline{\pi_1 c_0}\ (\overline{\pi_1 C_0}[I]\ t_1) \equiv \overline{\pi_2 c_0}\ (\overline{\pi_2 C_0}[I]\ t_2) \quad \text{(Lemma 4.24)}$$
   $$\text{iff} \quad \overline{\pi_1 c_0}\ (\overline{(\pi_1 C_0)[I]}\ t_1) \equiv \overline{\pi_2 c_0}\ (\overline{(\pi_2 C_0)[I]}\ t_2)$$
   $$\text{iff} \quad \overline{\pi_1 c_0 \textstyle\sum\big((\pi_1 C_0)[I]\big)}\ t_1 \equiv \overline{\pi_2 c_0 \textstyle\sum\big((\pi_2 C_0)[I]\big)}\ t_2 \quad \text{(Lemma 6.6)}$$

   Using Lemma 6.8 we obtain $\pi_1 c_0 \sum\big((\pi_1 C_0)[I]\big) = \pi_2 c_0 \sum\big((\pi_2 C_0)[I]\big)$ provided $t_1 \not\equiv u_0\ s'$, $t_1 \not\equiv u_1\ s'$, $t_2 \not\equiv u_0\ s'$, $t_2 \not\equiv u_1\ s'$ for all s'. All those cases are analogous. We consider $t_1 \not\equiv u_0\ s'$. Assume $t_1 \equiv u_0\ s'$. Since t is normal and u_0 may not occur in t, it is easy to see that the only way this is possible is if t is of the form $t = x\ T'$ for $x \in X$. If $T' = \mathsf{nil}$, then $t_1 = \overline{w}$ for some word w. Since this is an abstraction, we arrive at a contradiction to $t_1 \equiv u_0\ s'$. If $T' = [t']$ for some t', we have a contradiction, since in this case $t = x\ t'$ and $P(x)$ holds. Recall $x \in X$ and thus $x : \alpha \to \alpha$. If T' has at least two elements, we arrive at a contradiction since the type of x is $\alpha \to \alpha$ and therefore x may not have more than one argument.

2. Let $C_1 = c' :: C_1'$. Recall that $\sigma x_f = \Lambda X.e$ with $e = a\ T$. Then

   $$\overline{\pi_1 c_0}\ \Big((\Lambda X.a\ T)\ \overline{\pi_1 C_2}\ \overline{\pi_1 C_1'}\ \overline{\pi_1 c'}\Big) \equiv \overline{\pi_2 c_0}\ \Big((\Lambda X.a\ T)\ \overline{\pi_2 C_2}\ \overline{\pi_2 C_1'}\ \overline{\pi_2 c'}\Big)$$

   We proceed by case analysis on a.

   (a) Case $a = x$ for some variable x. Due to the types of its arguments a cannot have the type $\alpha \to \alpha$ and therefore $x \notin X$. This entails that x is the new applicative head of the list applications after reduction. Using Lemma 6.8 we can conclude that $\pi_1 c_0 = \pi_2 c_0$, meaning $I = \mathsf{nil}$ solves the **MPCP** instance.

   (b) Case $a = c$ for some constant c. Then c is the new applicative head of the list applications after reduction. Using Lemma 6.8 we can conclude that $\pi_1 c_0 = \pi_2 c_0$, meaning $I = \mathsf{nil}$ solves the **MPCP** instance. □

Theorem 6.12 $\mathsf{MPCP} \preceq \mathsf{U}_3$

Proof Follows with Lemmas 6.9 and 6.11. □

6.3 Remarks

We reduce from the modified Post correspondence problem whereas Huet reduces from the Post correspondence problem. As a consequence, Huet has to ensure that the ordering I that is implicitly picked by the substitution is never empty. In essence, he accomplishes this by adding the constraint $\lambda u_0.x_f\ u_0 \cdots u_0 \stackrel{?}{=} \lambda u_0.u_0\ (x_g\ u_0)$ which forces the term inserted for x_f to use at least one of its arguments. More precisely, he picks the equation $e_1 \stackrel{?}{=} e_2$ where

$$e_1 := \lambda u_0 u_1 x_h.x_h\ (x_f\ \overline{l_1} \cdots \overline{l_n})\ (x_f\ u_0 \cdots u_0)$$
$$e_2 := \lambda u_0 u_1 x_h.x_h\ (x_f\ \overline{r_1} \cdots \overline{r_n})\ (u_0\ (x_g\ u_0))$$

We can avoid a second equation by reducing from **MPCP** which makes the empty ordering $I = \mathsf{nil}$ an acceptable solution. In our reduction, even if $I = \mathsf{nil}$ meaning the term inserted for x_f just returns some term t of type α without using its arguments, we can still construct a solution for the **MPCP** instance. In this case the top and the bottom of the initial card c_0 have to be the same.

The original proof by Huet uses Church-typing meaning every term s also has a type A_s. In this work we used a Curry-style type system on top of untyped syntax. We do not see any differences arising from this decision in the above proof. In the Coq development, we have also formalised the original proof in our Curry-typed calculus.

Huet conducts his reduction in a calculus with tuples and remarks that the version in a calculus without tuples, as we present it here, also allows for an undecidability proof. We remark that our encoding differs marginally from the encoding of Huet. Huet encodes words as $\overline{\epsilon} = \lambda x.x$, $\overline{\mathsf{1}w} = \lambda x.u_1\ (\overline{w}\ x)$, and $\overline{\mathsf{0}w} = \lambda x.u_0\ (\overline{w}\ x)$ whereas our encoding is the normal form of Huet's encoding.

Since Huet only considers well-typed terms, all terms are normalising. Thus, he defines the equivalence of two terms as syntactic equality of the normal forms. The following lemma shows that this definition coincides with ours for normalising terms:

Fact 6.13 *If $s \triangleright v_1$ and $t \triangleright v_2$, then $s \equiv t$ iff $v_1 = v_2$.*

Chapter 7

Second-Order Unification

In this chapter we formalise the undecidability of second order unification as first presented by Goldfarb [25]. Recall from Chapter 2 that Goldfarb improved on the result of Huet by showing that unification is already undecidable in second-order languages provided they contain at least a single 2-ary function constant $\mathsf{g} : \alpha \to \alpha \to \alpha$. He establishes the undecidability of second-order unification by a reduction from Hilbert's tenth problem **H10**. In general, Hilbert's tenth problem asks whether a Diophantine equation has a solution. In this thesis, we use the formulation of **H10** over systems of simple Diophantine equations. Simple Diophantine equations ("Diophantine equations" in the following) are given by the abstract syntax:

$$d ::= x \doteq c \mid x + y \doteq z \mid x \cdot y \doteq z \quad (xyz : \mathbb{N}, c : \mathbb{N})$$

We denote systems of Diophantine equations by the letter D and represent them as lists. Solutions, i.e. variable assignments for Diophantine equations, are denoted by the letter θ. We write $\theta \vDash d$ if the variable assignment θ satisfies the equation d and lift the notation to systems of equations. Hilbert's tenth problem can then be defined as the *satisfiability* of a system of equations, i.e. the existence of a satisfying variable assignment.

Definition 7.1

$\boxed{\theta \vDash d}$ $\boxed{\theta \vDash D}$ $\boxed{\textbf{H10}}$

$$\begin{aligned} \theta \vDash x \doteq c &\quad \textit{iff} \quad \theta x = c \\ \theta \vDash x + y \doteq z &\quad \textit{iff} \quad \theta y + \theta y = \theta z \\ \theta \vDash x \cdot y \doteq z &\quad \textit{iff} \quad \theta y \cdot \theta y = \theta z \\ \theta \vDash D &\quad \textit{iff} \quad \theta \vDash d \ \textit{ for all } d \in D \end{aligned}$$

$$\textbf{H10}(D) := \exists \theta.\, \theta \vDash D$$

In the remainder of this chapter, we formalise Goldfarb's reduction from Hilbert's tenth problem to second-order unification. Following the explanation by Dowek [13],

in Section 7.1 we motivate Goldfarb's construction by reducing Hilbert's tenth problem to higher-order unification in general. Recall from Section 2.2 that we use Church numerals to encode natural numbers in the λ-calculus and encode Diophantine equations as the unification equations resulting from translating addition, multiplication, and constants into the setting of Church numerals. In this translation it is crucial that the "domain" of the variables occurring in encoded equations consists of Church numerals. We achieve this by adding a characteristic equation for every occurring variable x which asserts that the only valid instantiation of x is a proper Church encoding. This construction only yields a proof for the undecidability of higher-order unification. A proof of the undecidability of second-order unification can be obtained by Goldfarb's original construction based on Goldfarb numerals which we explain and formalise in Section 7.2.

7.1 Higher-Order Motivation

In his construction, Dowek encodes natural numbers as Church numerals. Every natural number n corresponds to the Church numeral $[\![n]\!]$ expressing n-fold iteration. We express addition and multiplication with the corresponding operations on Church numerals.

$[\![n]\!]$
$\mathsf{add}\ s\ t$
$\mathsf{mul}\ s\ t$

Definition 7.2

$$[\![n]\!] := \lambda af.f^n\ a \qquad \mathsf{add}\ s\ t := \lambda af.s\ (t\ a\ f)\ f \qquad \mathsf{mul}\ s\ t := \lambda af.s\ a\ (\lambda b.t\ b\ f)$$

In the formalisation we represent $f^n\ a$ as the list containing f exactly n times applied from the left to a.

Lemma 7.3

1. $[\![n]\!]\ s\ f \equiv f^n\ s$
2. $[\![m+n]\!] \equiv \mathsf{add}\ [\![m]\!]\ [\![m]\!]$
3. $[\![m \cdot n]\!] \equiv \mathsf{mul}\ [\![m]\!]\ [\![m]\!]$
4. $\mathsf{normal}\ [\![n]\!]$
5. *If* $[\![n]\!] \equiv [\![m]\!]$, *then* $n = m$.
6. $\dfrac{}{\Gamma \vdash_3 [\![n]\!] : \alpha \to (\alpha \to \alpha) \to \alpha}$

In the reduction we need to construct variable assignments for systems of Diophantine equations from substitutions. In particular, we have to decide whether some term represents an encoded natural number n and if so compute n. This is achieved by means of the following lemma:

Lemma 7.4 *For every term s we can either compute a natural number n such that $[\![n]\!] = s$, or prove that no such n exists.*

In the reduction we ensure that all terms inserted for variables behave like encodings of some natural number. This is realised by a characteristic equation describing what it means to be a Church encoding.

Lemma 7.5 *Let s be a normal term.*

$$\lambda af.f\ (s\ a\ f) \equiv \lambda af.s\ (f\ a)\ f \quad \textit{iff} \quad s = [\![n]\!] \ \textit{for some}\ n : \mathbb{N}$$

Reduction The characteristic equation is given by

$$\mathsf{CN}\ x := \lambda af.x\ (f\ a)\ f \stackrel{?}{=} \lambda af.f\ (x\ a\ f)$$

We encode a system of Diophantine equations D into a system of unification equations $\overline{D}$. In particular, we transform every equation $d \in D$ into a unification equation $\overline{d}$.

$$\overline{x \doteq c} := x \stackrel{?}{=} [\![c]\!] \qquad \overline{x + y \doteq z} := \mathsf{add}\ x\ y \stackrel{?}{=} z \qquad \overline{x \cdot y \doteq z} := \mathsf{mul}\ x\ y \stackrel{?}{=} z$$

To ensure that variables are only replaced by valid Church encodings, we add the characteristic equation $\mathsf{CN}\ x$ for every variable. Furthermore, we prove that the equations inherit their types from the type of Church numerals.

Lemma 7.6 *Let $[\![\mathbb{N}]\!] := \alpha \to (\alpha \to \alpha) \to \alpha$ and $\Gamma_D := [(x : [\![\mathbb{N}]\!]) \mid x \in \mathsf{vars}\ D]$.*

$$\frac{x \in \mathsf{vars}\ D}{\Gamma_D \vdash_3 \mathsf{CN}\ x : [\![\mathbb{N}]\!]} \qquad \frac{x \doteq c \in D}{\Gamma_D \vdash_3 \overline{x \doteq c} : [\![\mathbb{N}]\!]} \qquad \frac{x + y \doteq z \in D}{\Gamma_D \vdash_3 \overline{x + y \doteq z} : [\![\mathbb{N}]\!]}$$

$$\frac{x \cdot y \doteq z \in D}{\Gamma_D \vdash_3 \overline{x \cdot y \doteq z} : [\![\mathbb{N}]\!]}$$

In analogy to the notation $\theta \vDash d$ we define $\sigma \vDash s \stackrel{?}{=} t$ which we use if σ unifies s and t. In particular, we mean σ unifies the encoding of d when we write $\sigma \vDash \overline{d}$.

Definition 7.7

$$\sigma \vDash s \stackrel{?}{=} t \quad \textit{iff} \quad s[\sigma] \equiv t[\sigma]$$

$\boxed{\sigma \vDash s \stackrel{?}{=} t}$

For the correctness of the translation we establish that the characteristic equation is only satisfied by valid Church encodings and that the operations on Church encoded terms mirror the operations on natural numbers.

Lemma 7.8 *Let σx be normal for all x.*

1. $\sigma \vDash \mathsf{CN}\ x$ *iff* $\sigma x = [\![n]\!]$ *for some* $n : \mathbb{N}$.

2. *Let* $\sigma x = [\![m]\!]$, $\sigma y = [\![n]\!]$ *and* $\sigma z = [\![p]\!]$.

$$\sigma \vDash \overline{x \doteq c} \text{ iff } m = c \qquad \sigma \vDash \overline{x + y \doteq z} \text{ iff } m + n = p$$

$$\sigma \vDash \overline{x \cdot y \doteq z} \text{ iff } m \cdot n = p$$

Theorem 7.9 $\mathbf{H10} \preceq \mathbf{SU}_3$

Proof For a system of Diophantine equations D we produce the system of equations:

$$f(D) = \Gamma_D \vdash_3 E_D : (\alpha \to (\alpha \to \alpha) \to \alpha)^{|E_D|}$$
$$\text{where } E_D = [\mathsf{CN}\ x \mid x \in \mathsf{vars}\ D] +\!\!+ \left[\overline{d} \;\middle|\; d \in D\right]$$

It remains to prove that $\mathbf{H10}(D)$ iff $\mathbf{SU}_3(f(D))$.

1. Let θ be a solution for D. Pick $\Delta := \emptyset$ and $\sigma x := [\![\theta x]\!]$. Typing follows with Lemma 7.3 and the equivalences with Lemma 7.8.
2. By Fact 5.11 let $\Delta \vdash_3 \sigma : \Gamma_D$ be a normal substitution unifying the equations E_D. Using Lemma 7.4 we define $\theta x := n$ if $\sigma x = [\![n]\!]$ and $\theta x := 0$ otherwise. We establish the equations using Lemma 7.8. □

Remarks Note that in the formalisation it is more convenient to add the equation $\mathsf{CN}\ x$ once for every equation containing x.

We remark that our explanation differs from the one presented by Dowek [13] in two key aspects. The first aspect is that we encode systems of simple Diophantine equations instead of encoding entire polynomials. The second aspect concerns the characteristic equation for variables. Dowek gives the equation $\lambda a.x\ a\ (\lambda y.y) \stackrel{?}{=} \lambda y.y$. In this work, we choose a different equation because we believe our equation is better fit to motivate Goldfarb's characteristic equation. In addition, its type is $[\![\mathbb{N}]\!]$ which is also the type of all other equations hence simplifying the formalisation.

7.2 Second-Order Realisation

Recall from Section 2.6 that in its essence, the structure of Goldfarb's proof is the same as the structure of the above proof. The difference between both proofs lies in the encoding of natural numbers. The Church encoding of a natural number n, $[\![n]\!] = \lambda a f.f^n\ a$, takes as one of its arguments a function $f : \alpha \to \alpha$. Working in a language with constants $\mathsf{a}, \mathsf{b} : \alpha$ and $\mathsf{g} : \alpha \to \alpha \to \alpha$, Goldfarb fixes for f the term $\mathsf{g}\ \mathsf{a} : \alpha \to \alpha$. We write $[\![n]\!]$ for the Goldfarb numeral corresponding to the number n and $\overline{n}t$ for the result of applying a Goldfarb numeral to t. The mathematical function $\overline{n}\cdot$ may be understood as the metalevel equivalent to $[\![n]\!]$. Furthermore, we

introduce a relation $s \approx [\![n]\!]$ which expresses the notion that s behaves like the Goldfarb numeral $[\![n]\!]$.

Definition 7.10

$$[\![n]\!] := \lambda a.(\mathsf{g}\ \mathsf{a})^n\ a \qquad \overline{n}t := (\mathsf{g}\ \mathsf{a})^n\ t \qquad s \approx [\![n]\!] := \forall t.\ s\ t \equiv \overline{n}t$$

$[\![n]\!]$ $\overline{n}t$

$s \approx [\![n]\!]$

The reason for introducing $s \approx [\![n]\!]$ is that the characteristic equation for Goldfarb numerals can only ensure that any s which satisfies the equation behaves like a Goldfarb numeral. Frequently, reasoning about $\overline{n}t$ instead of $[\![n]\!]$ will allow us to use syntactic equality whereas proofs using $[\![n]\!]$ are based on equivalence. Thus, we focus primarily on $\overline{n}t$ in the following and establish:

Lemma 7.11

1. $\overline{0}t = t$ *and* $\overline{n+1}t = \mathsf{g}\ \mathsf{a}\ (\overline{n}t)$ *and* $\overline{n+m}t = \overline{n}(\overline{m}t)$.
2. *If* $\mathsf{normal}\ t$, *then* $\mathsf{normal}\ (\overline{n}t)$.
3. *If* t_1, t_2 *are atoms and* $\overline{n}t_1 = \overline{m}t_2$, *then* $n = m$ *and* $t_1 = t_2$.
4. $\dfrac{\Gamma \vdash_2 t : \alpha}{\Gamma \vdash_2 \overline{n}t : \alpha}$

Similar to Section 7.1, in the reduction we need to construct variable assignments for systems of Diophantine equations from substitutions. In particular, we have to decide whether some term represents an encoded natural number and if so compute which natural number is represented by the term. This is achieved by means of the following lemma:

Lemma 7.12

1. *For every term* s *we can either compute a natural number* n *such that* $s = \overline{n}\mathsf{a}$, *or we can prove that no such* n *exists.*
2. *For every normal term* s *we can either compute a natural number* n *such that* $s\ \mathsf{a} \equiv \overline{n}\mathsf{a}$, *or we can prove that no such* n *exists.*

The characteristic equation we gave in the previous section can be reused, if f is fixed to $\mathsf{g}\ \mathsf{a}$ and a is instantiated with a, meaning $\mathsf{GN}\ x := x\ (\mathsf{g}\ \mathsf{a}\ \mathsf{a}) \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ (x\ \mathsf{a})$. With this equation we do not axiomatise being a Goldfarb numeral directly. Instead, we axiomatise behaving like a Goldfarb numeral. Explicitly:

Lemma 7.13 *Let* s *be some normal term.*

$$s(\mathsf{g}\ \mathsf{a}\ \mathsf{a}) \equiv \mathsf{g}\ \mathsf{a}\ (s\ \mathsf{a}) \quad \textit{iff} \quad s \approx [\![n]\!]\ \textit{for some}\ n : \mathbb{N}$$

As a consequence of this encoding, the undecidability of second-order unification can be proven. However, the price of changing the encoding is that multiplication can no longer be encoded in the style of Church numerals. For Church numerals we encoded the multiplication $m \cdot n$ as an application of m to an abstraction dependent on n. Since for Goldfarb numerals f is fixed to $\mathsf{g}\ \mathsf{a}$, this is not possible.

Motivation Multiplication Goldfarb solves this problem by encoding the multiplication computation into unification itself in a certain sense. For every equation $x \cdot y = z$, Goldfarb constructs two unification equations which are satisfiable by a substitution σ with $\sigma x \approx [\![m]\!]$, $\sigma x \approx [\![n]\!]$ and $\sigma x \approx [\![p]\!]$ iff $m \cdot n = p$. Note that $\sigma x \approx [\![m]\!]$, $\sigma x \approx [\![n]\!]$ and $\sigma x \approx [\![p]\!]$ for some m, n, p is not guaranteed by the multiplication equations themselves but can be guaranteed by adding characteristic equations.

Recall from Section 2.6 the characterisation of multiplication using finite sequences. We represent finite sequences as lists in the following and define the successor function $\mathsf{succ}(a, i) := (a + n, b + i)$ and and $\mathsf{succ}(X) := [\mathsf{succ}\ (a, i) \mid (a, i) \in X]$. Abstracting over the start values, we can generalise the characterisation to:

$$X +\!\!+ [(a + p, b + m)] = (a, b) :: \mathsf{succ}(X)$$

Lemma 7.14 *$m \cdot n = p$ iff $X +\!\!+ [(a + p, b + m)] = (a, b) :: \mathsf{succ}(X)$ for some X.*

Proof Let $t_i := (a + i \cdot n, b + i)$.

1. "If": The claim follows for $X := [t_i \mid i = 0, \ldots, m - 1]$.
2. "Only if": It suffices to prove that $X +\!\!+ [x] = (a, b) :: \mathsf{succ}(X)$ implies $x = t_{|X|}$ for all x. The claim then follows with $x = (a + p, b + m)$.

   We proceed by induction on X with a, b generalised. For the empty list the claim is trivial. For $X = (k, l) :: X'$ we have $k = a$ and $l = b$. Thus $X' +\!\!+ [(a + p, b + m)] = (a + n, b + 1) :: \mathsf{succ}(X')$. The claim follows with the inductive hypothesis. □

To conclude our motivation of the encoding of multiplication, we dispense with the list concatenation in our equation. Under the encoding which we will use for lists and pairs, we cannot express list concatenation as a λ-term. Therefore, we remove list concatenation by considering finite, unfinished sequences of pairs. By unfinished we mean that they contain a hole $\bullet$ at the end which can be used to concatenate finite sequences. For example, an unfinished sequence may look like this:

$$(a, b); (a + n, b + 1); (a + 2n, b + 2); \bullet$$

We introduce an operation $X[s_1, \ldots, s_n]$ which fills the hole of X with the finite sequence $s_1; \cdots; s_n$. Using this hole filling operation, we characterise multiplication by

$$X[(a+p, b+m)] = (a, b) :: \mathsf{succ}(X[])$$

The only valid solution to this equation is the finite, unfinished sequence

$$(a, b); \ldots; (a + (m-1) \cdot n, b + (m-1)); \bullet$$

In the remainder of this paragraph, we motivate how this technique of axiomatising multiplication can be transformed into the setting of unification. In the setting of unification, the "hole filling" operation can be understood as a substitution of the variable $\bullet$ and every unfinished finite sequence X can be interpreted as a function which cannot analyse its arguments. For the encoding of lists and pairs, we fix constants $\mathsf{a}, \mathsf{b} : \alpha$ and $\mathsf{g} : \alpha \to \alpha \to \alpha$ and define:

$$(s, t) := \mathsf{g}\ s\ t \qquad \mathsf{nil} := \mathsf{a} \qquad s :: t := \mathsf{g}\ s\ t$$

We write $[s_1, \ldots, s_n]$ for the list $s_1 :: \cdots :: s_n :: \mathsf{nil}$. Note that the encoding for :: and pairs is the same. We add both encodings to ease readability. Using pairs, lists, and a fresh variable M_{xyz} of type $\alpha \to \alpha \to \alpha$ we encode $x \cdot y \doteq z$ with the equations:

$$M_{xyz}\ \mathsf{a}\ \mathsf{b}\ [(z\ \mathsf{a}, x\ \mathsf{b})] \stackrel{?}{=} (\mathsf{a}, \mathsf{b}) :: M_{xyz}\ (y\ \mathsf{a})\ (\overline{1}\mathsf{b})\ \mathsf{nil}$$

$$M_{xyz}\ \mathsf{b}\ \mathsf{a}\ [(z\ \mathsf{b}, x\ \mathsf{a})] \stackrel{?}{=} (\mathsf{b}, \mathsf{a}) :: M_{xyz}\ (y\ \mathsf{b})\ (\overline{1}\mathsf{a})\ \mathsf{nil}$$

To understand the connection between these two equations and the equation between finite, unfinished sequences, we consider what happens in the presence of a substitution σ. Due to the characteristic equations for Goldfarb numerals, we may assume that $\sigma x \approx [\![m]\!]$, $\sigma y \approx [\![n]\!]$, $\sigma z \approx [\![p]\!]$. We abbreviate σM_{xyz} with X. In the presence of this substitution, we obtain:

$$X\ \mathsf{a}\ \mathsf{b}\ [(\overline{p}\mathsf{a}, \overline{m}\mathsf{b})] \equiv (\mathsf{a}, \mathsf{b}) :: X\ (\overline{n}\mathsf{a})\ (\overline{1}\mathsf{b})\ \mathsf{nil}$$

$$X\ \mathsf{b}\ \mathsf{a}\ [(\overline{p}\mathsf{b}, \overline{m}\mathsf{a})] \equiv (\mathsf{b}, \mathsf{a}) :: X\ (\overline{n}\mathsf{b})\ (\overline{1}\mathsf{a})\ \mathsf{nil}$$

For now, focus on the first equation. The term $X\ \mathsf{a}\ \mathsf{b}$ can be thought of as the finite, unfinished sequence which starts in (a, b). The application to $[(\overline{p}\mathsf{a}, \overline{m}\mathsf{b})]$ corresponds to filling the hole $\bullet$ with the finite sequence $[(p+a, m+b)]$. The term $X\ \overline{n}\mathsf{a}\ (\overline{1}\mathsf{b})$ can be thought of as the finite, unfinished sequence which is started in $(n+a, 1+b)$. As a consequence, the term $X\ \mathsf{a}\ \mathsf{b}\ [(\overline{p}\mathsf{a}, \overline{m}\mathsf{b})]$ can be thought of as $X\ [(a+p, b+m)]$ and $(\mathsf{a}, \mathsf{b}) :: X\ (\overline{n}\mathsf{a})\ (\overline{1}\mathsf{b})\ \mathsf{nil}$ can be thought of as the finite sequence $(a, b) :: \mathsf{succ}(X\ [])$.

The dual use of a and b enforces that X is parametric over a and b. For the sake of simplicity assume $X = \lambda w_1 w_2 w_3.u$ for some term u. Using the duality of a and b in the above equations, we can show that the terms corresponding to t_i in the setting of unification are $(\overline{i \cdot n} w_1, \overline{i} w_2)$. Thus, we define $t_i := (\overline{i \cdot n} w_1, \overline{i} w_2)$. The reader may think of the application to w_1 and w_2 as the addition of a and b. Furthermore, the variable w_3 represents the hole of the unfinished finite sequence.

Reduction In essence, operations of Diophantine equations are mirrored by their counterparts on Goldfarb numerals in the reduction. We encode a system of Diophantine equations D into a system of unification equations $\overline{D}$. In particular, we transform every equation $d \in D$ into unification equations $\overline{d}$.

$$\overline{x \doteq c} := x\ \mathsf{a} \stackrel{?}{=} \overline{c}\mathsf{a} \qquad\qquad \overline{x + y \doteq z} := x\ (y\ \mathsf{a}) \stackrel{?}{=} z\ \mathsf{a}$$

For multiplication $\overline{x \cdot y \doteq z}$, we give two equations

$$M_{xyz}\ \mathsf{a}\ \mathsf{b}\ [(z\ \mathsf{a}, x\ \mathsf{b})] \stackrel{?}{=} (\mathsf{a}, \mathsf{b}) :: M_{xyz}\ (y\ \mathsf{a})\ (\overline{1}\mathsf{b})\ \mathsf{nil}$$

$$M_{xyz}\ \mathsf{b}\ \mathsf{a}\ [(z\ \mathsf{b}, x\ \mathsf{a})] \stackrel{?}{=} (\mathsf{b}, \mathsf{a}) :: M_{xyz}\ (y\ \mathsf{b})\ (\overline{1}\mathsf{a})\ \mathsf{nil}$$

where lists and tuples are to be understood as introduced in the motivation. We index the variable M by xyz to ensure that we have a fresh variable for every choice of x, y and z. In the formalisation, we realise this by drawing variables from two different copies of the natural numbers and injecting them into the natural numbers. Explicitly, we use a bijection $\langle \cdot, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and a bijection $\mathsf{I}_+ : \mathbb{N} + \mathbb{N} \to \mathbb{N}$ to represent x as $\mathsf{I}_+(\mathsf{L}_+\ x)$ and M_{xyz} as $\mathsf{I}_+(\mathsf{R}_+\ \langle x, \langle y, z \rangle\rangle)$.

We prove that all equations are of type α.

Lemma 7.15 *Let*

$$\Gamma_D := [(x : \alpha \to \alpha) \mid x \in \mathsf{vars}\ D] \cup \left[(M_{xyz} : \alpha^3 \to \alpha) \;\middle|\; x \cdot y \doteq z \in D\right]$$

$$\frac{x \in \mathsf{vars}\ D}{\Gamma_D \vdash_2 \mathsf{GN}\ x : \alpha} \qquad \frac{x \doteq c \in D}{\Gamma_D \vdash_2 \overline{x \doteq c} : \alpha} \qquad \frac{x + y \doteq z \in D}{\Gamma_D \vdash_2 \overline{x + y \doteq z} : \alpha}$$

$$\frac{x \cdot y \doteq z \in D}{\Gamma_D \vdash_2 \overline{x \cdot y \doteq z} : \alpha^2}$$

For the correctness of the translation we establish that the characteristic equation is only satisfied by terms behaving like Goldfarb numerals and that the operations on Goldfarb encoded terms mirror the operations on natural numbers.

Lemma 7.16 *Let σx be normal for all x.*

1. $\sigma \vDash \mathsf{GN}\ x$ *iff* $\sigma x \approx [\![n]\!]$ *for some* $n : \mathbb{N}$.
2. *Let* $\sigma x \approx [\![m]\!]$, $\sigma y \approx [\![n]\!]$ *and* $\sigma z \approx [\![p]\!]$.

$$\sigma \vDash \overline{x = c} \text{ iff } m = c \qquad \sigma \vDash \overline{x + y \doteq z} \text{ iff } m + n = p$$

Since this is similar to the reduction in Section 7.1, we primarily focus on the encoding of multiplication. Recall the unification equations generated for $x \cdot y = z$.

$$M_{xyz}\ \mathsf{a}\ \mathsf{b}\ [(z\ \mathsf{a}, x\ \mathsf{b})] \stackrel{?}{=} (\mathsf{a}, \mathsf{b}) :: M_{xyz}\ (y\ \mathsf{a})\ (\overline{1}\mathsf{b})\ \mathsf{nil}$$

$$M_{xyz}\ \mathsf{b}\ \mathsf{a}\ [(z\ \mathsf{b}, x\ \mathsf{a})] \stackrel{?}{=} (\mathsf{b}, \mathsf{a}) :: M_{xyz}\ (y\ \mathsf{b})\ (\overline{1}\mathsf{a})\ \mathsf{nil}$$

In Lemma 7.8 we prove $m \cdot n = p$ if and only if $\sigma \vDash \overline{x \cdot y = z}$. For Goldfarb numerals we are forced to adapt this claim to take the variable M_{xyz} into account. To this end, assume a substitution σ which only inserts normal forms such that $\sigma x \approx [\![m]\!]$, $\sigma y \approx [\![n]\!]$ and $\sigma z \approx [\![p]\!]$. For the "if" direction, we adapt the claim to

Lemma 7.17 *If* $m \cdot n = p$ *and* $\sigma M_{xyz} = \lambda w_1 w_2 w_3.t_0 :: \ldots :: t_{m-1} :: w_3$, *then* $\sigma \vDash \overline{x \cdot y = z}$.

Proof The first equation simplifies to $(\mathsf{a}, \mathsf{b}) :: \ldots :: (\overline{m \cdot n}\mathsf{a}, \overline{m}\mathsf{b}) :: \mathsf{nil}$ on both sides and the second equation to $(\mathsf{b}, \mathsf{a}) :: \ldots :: (\overline{m \cdot n}\mathsf{b}, \overline{m}\mathsf{a}) :: \mathsf{nil}$. □

The term $t_0 :: \ldots :: t_{m-1} :: w_3$ represents a finite, unfinished sequence with the hole w_3. We proceed by proving the "only if" direction which is left unchanged, meaning we prove:

$$\textit{If } \sigma \vDash \overline{x \cdot y \doteq z}, \quad \textit{then} \quad m \cdot n = p.$$

For the sake of simplicity, we assume $\sigma M_{xyz} = \lambda w_1 w_2 w_3.u$ for some u in the following and address this assumption in Lemma 7.22. Given $\sigma M_{xyz} = \lambda w_1 w_2 w_3.u$, we obtain by β-reduction

$$\sigma \vDash \overline{x \cdot y \doteq z} \quad \text{iff} \quad u[\sigma_k] \equiv t_0[\sigma_k] :: u[\tau_k] \text{ for } k = 1, 2$$

where the substitutions $\sigma_1, \sigma_2, \tau_1, \tau_2$ are given by:

	w_1	w_2	w_3
σ_1	a	b	$[(\overline{p}\mathsf{a}, \overline{m}\mathsf{b})]$
σ_2	b	a	$[(\overline{p}\mathsf{b}, \overline{m}\mathsf{a})]$

	w_1	w_2	w_3
τ_1	$\overline{n}\mathsf{a}$	$\overline{1}\mathsf{b}$	nil
τ_2	$\overline{n}\mathsf{b}$	$\overline{1}\mathsf{a}$	nil

In this context, the use of τ_1, τ_2 corresponds to succ and w_3 represents the hole $\bullet$. Here, $\overline{1}\cdot$ represents the successor function and $\overline{n}\cdot$ the addition of n. On the left this

hole may be filled with $[(\overline{p}\mathsf{a}, \overline{m}\mathsf{b})]$ and $[(\overline{p}\mathsf{b}, \overline{m}\mathsf{a})]$ respectively, whereas on the right side this hole may be filled with the empty sequence nil.

Since u is normal and the substitutions insert normal terms which are free of abstractions, we conclude that all terms occurring in the above equations are normal. As a consequence, we deduce that the terms on the left and right of the above equalities are syntactically equal, meaning

$$u[\sigma_k] = t_0[\sigma_k] :: u[\tau_k] \text{ for } k = 1, 2$$

We analyse how the dual definition of σ_1 and σ_2 can be used to enforce that u actually inserts tuples of the shape of t_i.

Lemma 7.18

1. *If* $s[\sigma_1] = \mathsf{a}$ *and* $s[\sigma_2] = \mathsf{b}$*, then* $s = w_1$.
2. *If* $s[\sigma_1] = \mathsf{b}$ *and* $s[\sigma_2] = \mathsf{a}$*, then* $s = w_2$.
3. *If* $s[\sigma_k] = t_i[\sigma_k]$ *for* $k = 1, 2$*, then* $s = t_i$.

The following lemma shows how we can interpret τ_1, τ_2 as the succ function.

Lemma 7.19 $t_{i+1}[\sigma_k] = t_i[\tau_k]$ *for* $k = 1, 2$.

Using this lemma, we establish the iterative character of these equations with the following lemma:

Lemma 7.20 *Let* $s[\sigma_k] = t_i[\sigma_k] :: s[\tau_k]$ *for* $k = 1, 2$.

1. $s = w_3$ *or* $s = t_i :: s'$ *for some* s'.
2. $s = t_i :: t_{i+1} :: \ldots :: t_{i+(l-1)} :: w_3$ *for some* l.

Proof

1. By analysis of the structure of u using Lemma 7.18.
2. By size induction on s using Item 1. □

Corollary 7.21 *If* $u[\sigma_k] = t_0[\sigma_k] :: u[\tau_k]$ *for* $k = 1, 2$*, then* $m \cdot n = p$.

Proof By Lemma 7.20, we have $u = t_0 :: \ldots :: t_{l-1} :: w_3$ for some l. Case analysis on l. If $l = 0$, then $(\overline{p}\mathsf{a}, \overline{m}\mathsf{b}) = (\mathsf{a}, \mathsf{b})$ which means $m = p = 0$. If $l = l' + 1$, then by definition of σ_1 and τ_1, we have

$$(\mathsf{a}, \mathsf{b}) :: \ldots :: (\overline{n \cdot l'}\mathsf{a}, \overline{l'}\mathsf{b}) :: (\overline{p}\mathsf{a}, \overline{m}\mathsf{b}) :: \mathsf{nil} = (\mathsf{a}, \mathsf{b}) :: \ldots :: (\overline{n \cdot l}\mathsf{a}, \overline{l}\mathsf{b}) :: \mathsf{nil}$$

In particular, $(\overline{p}\mathsf{a}, \overline{m}\mathsf{b}) = (\overline{n \cdot l}\mathsf{a}, \overline{l}\mathsf{b})$ which means $p = l \cdot n$ and $m = l$, thus $m \cdot n = p$. □

We now address the assumption $\sigma M_{xyz} = \lambda w_1 w_2 w_3.u$. As it turns out, there is one more possible instantiation of M_{xyz}. For the special case where $m = 1$ and $m \cdot n = p$, both $\lambda w_1 w_2 w_3.\mathsf{g}\ (\mathsf{g}\ w_1\ w_2)\ w_3$ and $\lambda w_1 w_2.\mathsf{g}\ (\mathsf{g}\ w_1\ w_2)$ are valid instantiations of M_{xyz}. The latter is constructed from the former by one η-reduction.

Lemma 7.22 *If $\sigma \vDash \overline{x \cdot y \doteq z}$, then $\sigma M_{xyz} = \lambda w_1 w_2.\mathsf{g}\ (\mathsf{g}\ w_1\ w_2)$ or σM_{xyz} is of the shape $\lambda w_1 w_2 w_3.u$ for some u.*

We combine this lemma with the previous results to obtain:

Lemma 7.23 *If $\sigma \vDash \overline{x \cdot y \doteq z}$, then $m \cdot n = p$.*

Proof Case analysis on the shape of σM_{xyz} using Lemma 7.22.

1. Let $\sigma M_{xyz} = \lambda w_1 w_2.\mathsf{g}\ (\mathsf{g}\ w_1\ w_2)$. Then $(\mathsf{a},\mathsf{b}) :: (\overline{p}\mathsf{a}, \overline{m}\mathsf{b}) :: \mathsf{nil} = (\mathsf{a},\mathsf{b}) :: (\overline{n}\mathsf{a}, \overline{1}\mathsf{b}) :: \mathsf{nil}$. Thus, we have $n = p$ and $m = 1$.
2. Let $\sigma M_{xyz} = \lambda w_1 w_2 w_3.u$ for some u. Then $u[\sigma_k] \equiv t_0[\sigma_k] :: u[\tau_k]$ for $k = 1, 2$. Since all terms involved are normal and no abstractions are inserted by the substitutions, we may conclude $u[\sigma_k] = t_0[\sigma_k] :: u[\tau_k]$ for $k = 1, 2$. The claim follows with Corollary 7.21. □

We write $\mathbf{SU}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}}$ for the second-order system unification problem with constants $\mathsf{a}, \mathsf{b}, \mathsf{g}$ and analogously $\mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}}$ for the second-order unification problem with constants $\mathsf{a}, \mathsf{b}, \mathsf{g}$.

Theorem 7.24 $\mathbf{H10} \preceq \mathbf{SU}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}}$

Proof For a system of Diophantine equations D we produce the system of equations:

$$f(D) = \Gamma_D \vdash_3 E_D : \alpha^{|E_D|}$$
$$\text{where } E_D = \left[\overline{x} \;\middle|\; x \in \mathsf{vars}\ D\right] +\!\!+ \left[\overline{d} \;\middle|\; d \in D\right]$$

where the notation $\left[\overline{d} \;\middle|\; d \in D\right]$ is here to be understood as adding two equations in the case of multiplication and one otherwise. We show $\mathbf{H10}(D)$ iff $\mathbf{SU}_2(f(D))$:

1. Let θ be a solution for D. Pick $\Delta := \emptyset$ and σ such that $\sigma x = [\![\theta x]\!]$ and $\sigma M_{xyz} = \lambda w_1 w_2 w_3.t_0 :: \ldots :: t_{\theta x - 1} :: w_3$ where $t_i := (\overline{\theta y \cdot i} w_1, \overline{i} w_2)$. Typing follows with Lemma 7.11 and the equivalences with Lemmas 7.16 and 7.17.
2. By Fact 5.11 let $\Delta \vdash_2 \sigma : \Gamma_D$ be a normal substitution unifying the equations E_D. Using Lemma 7.12 we define $\theta x := n$ if $\sigma x\ \mathsf{a} \equiv \overline{n}\mathsf{a}$ and $\theta x := 0$ otherwise. We establish the equations using Lemmas 7.16 and 7.23. □

Corollary 7.25 (Goldfarb) $\mathbf{H10} \preceq \mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}}$

Proof Analogous to Theorem 7.24 using Lemma 5.10. □

Chapter 8

First-Order Unification

In this chapter we give a decision procedure for first-order unification in the λ-calculus. Our work is based on the formalisation by Smolka and Husson [51].

When speaking of first-order unification, one usually considers languages consisting of variables, constants, and application [39, 36] such as $s, t ::= x \mid c \mid s\ t$ or equivalently languages consisting of variables and $n$ary function constants applied to all their arguments [1, 30]. Recall from Section 2.8 that subtle differences arise when we consider unification of first-order terms in the λ-calculus. For example, consider the first-order terms $\mathsf{g}\ x\ \mathsf{a}$ and $\mathsf{g}\ \mathsf{b}\ \mathsf{a}$ where $\mathsf{g} : \alpha \to \alpha \to \alpha$ and $\mathsf{a}, \mathsf{b}, x : \alpha$. They can be unified using the substitution $\sigma x := \mathsf{b}$. However, if we turn the free occurrence of x into a bound occurrence using a λ-abstraction, then the resulting terms $\lambda x.\mathsf{g}\ x\ \mathsf{a}$ and $\lambda x.\mathsf{g}\ \mathsf{b}\ \mathsf{a}$ can no longer be unified. Consequently, when constructing a unification algorithm we have to take free and bound variables into account.

8.1 Simplified First-Order Unification

In general, we mean $\mathbf{U}_1$ when we speak of first-order unification. To simplify matters, we first prove the decidability of first-order unification in the λ-free fragment of the calculus and proceed by proving the decidability of first-order unification in the full calculus. A term is λ-free, if it does not contain any abstractions. For example, $\mathsf{g}\ \mathsf{b}\ \mathsf{a}$ is λ-free whereas $f\ (\lambda x.x)$ is not. In the following we assume that all equations $s \stackrel{?}{=} t$ and systems of equations E only contain λ-free terms. This assumption is addressed in Section 8.2.

For first-order unification in the λ-free fragment we assume a decidable predicate free which distinguishes free variables from bound variables. We say a substitution σ *respects bound variables* if $\sigma x = x$ for all bound variables x and no bound variable occurs free in the terms inserted by σ.

free x

Note that λ-free terms are always normal since they cannot contain any β-redexes. As a consequence, when it comes to unifiability we reason about syntactic equality instead of equivalence. We use analogous notation to Chapter 7, with syntactic equal-

ity instead of equivalence. Explicitly, we write $\sigma \vDash s \stackrel{?}{=} t$ for $s[\sigma] = t[\sigma]$ and $\sigma \vDash E$ if $\sigma \vDash s \stackrel{?}{=} t$ for all $s \stackrel{?}{=} t \in E$.

To decide whether λ-free first-order terms containing bound and free variables are unifiable, we give a computable relation $E \mapsto \sigma$ operating on systems of equations. The operational reading of $E \mapsto \sigma$ yields the implementation rules for a unification procedure which we implement using the Equations tool [54]. To prove the correctness of the procedure, we verify three properties.

1. **Computability.** For every list of equations E we can compute a substitution σ with $E \mapsto \sigma$ or we can prove that no such σ exists.
2. **Soundness.** Whenever we can derive a substitution $E \mapsto \sigma$, then σ solves the equations in E, meaning $\sigma \vDash E$. Furthermore, σ is λ-free, respects bound variables, and if $\Gamma \vdash_1 E : L$, then $\Delta \vdash_1 \sigma : \Gamma$ for some context Δ.
3. **Completeness.** Whenever the equations in E are solved by a substitution σ respecting bound variables, then there exists a substitution τ such that $E \mapsto \tau$.

8.1.1 Term Decomposition

To define $E \mapsto \sigma$ we first define an auxiliary procedure decomp which decomposes systems of equations into simpler systems where the left side is always a variable. For instance, take the system of equations $\mathsf{g}\ \mathsf{a}\ \mathsf{b} \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ \mathsf{b}, \mathsf{g}\ x\ y \stackrel{?}{=} \mathsf{g}\ (\mathsf{g}\ y\ \mathsf{a})\ (\mathsf{g}\ \mathsf{a}\ \mathsf{a})$. We can decompose these equations into the simpler system $x \stackrel{?}{=} \mathsf{g}\ y\ \mathsf{a}$ and $y \stackrel{?}{=} \mathsf{g}\ \mathsf{a}\ \mathsf{a}$ which is unifiable if and only if the original system is unifiable. We remove equations where the left and the right side are the same and we decompose equations with applications on both sides such as $s_1\ s_2 \stackrel{?}{=} t_1\ t_2$ into the equations resulting from simplifying $s_1 \stackrel{?}{=} t_1$ and $s_2 \stackrel{?}{=} t_2$.

$\boxed{\mathsf{decomp}\ (s \stackrel{?}{=} t)}$
$\boxed{\mathsf{decomp}\ E}$

Definition 8.1

$$\mathsf{decomp}\ (s \stackrel{?}{=} s) = \mathsf{nil} \qquad \mathsf{decomp}\ (x \stackrel{?}{=} s) = [x \stackrel{?}{=} s] \qquad \mathsf{decomp}\ (s \stackrel{?}{=} x) = [x \stackrel{?}{=} s]$$

$$\mathsf{decomp}\ (s_1\ s_2 \stackrel{?}{=} t_1\ t_2) = \mathsf{decomp}\ (s_1 \stackrel{?}{=} t_1) +\!\!+ \mathsf{decomp}\ (s_2 \stackrel{?}{=} t_2)$$

$$\mathsf{decomp}\ (\_ \stackrel{?}{=} \_) = \emptyset \quad \textit{othw.}$$

$$\mathsf{decomp}\ \mathsf{nil} = \mathsf{nil} \qquad \mathsf{decomp}\ (s \stackrel{?}{=} t :: E) = \mathsf{decomp}\ (s \stackrel{?}{=} t) +\!\!+ \mathsf{decomp}\ E$$

The concatenation $A +\!\!+ B$ is to be interpreted such that $A +\!\!+ B = \emptyset$ whenever $A = \emptyset$ or $B = \emptyset$.

We show that decomposition of systems does not affect unifiability. To this end, we introduce the equivalence relation $E_1 \approx E_2$ which expresses that E_1 and E_2 are unifiable by the same substitutions. Explicitly, $E_1 \approx E_2 := \forall\sigma.\ \sigma \vDash E_1$ iff $\sigma \vDash E_2$.

Lemma 8.2

1. $\approx$ *is compatible with* $+\!\!+$
2. $[s \stackrel{?}{=} s] \approx \mathsf{nil}$
3. $[s \stackrel{?}{=} t] \approx [t \stackrel{?}{=} s]$
4. $[s_1\ s_2 \stackrel{?}{=} t_1\ t_2] \approx [s_1 \stackrel{?}{=} t_1, s_2 \stackrel{?}{=} t_2]$
5. $[s \stackrel{?}{=} t] \approx \mathsf{decomp}\ (s \stackrel{?}{=} t)$.
6. $E \approx \mathsf{decomp}\ E$

where decomp *is assumed to return a nonemty option in Items 5 and 6.*

Unification equations with different constants on both sides, such as $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or equations where constants are equated with applications such as $\mathsf{g\ a\ a} \stackrel{?}{=} \mathsf{a}$ are not unifiable. In these cases the decomposition returns $\emptyset$ which we justify with the following lemma:

Lemma 8.3

1. *If* $\mathsf{decomp}\ (s \stackrel{?}{=} t) = \emptyset$*, then there is no* σ *such that* $\sigma \vDash s \stackrel{?}{=} t$.
2. *If* $\mathsf{decomp}\ E = \emptyset$*, then there is no* σ *such that* $\sigma \vDash E$.

Formally, decomp could also encounter an abstraction and return $\emptyset$ which we ignore since we only consider λ-free terms.

Furthermore, we prove that decomposition retains well-typedness in the sense that well-typed systems of equations are decomposed into systems of equations which are also well-typed. In addition, we establish that decomp does not introduce new variables.

Lemma 8.4

1. *If* $\Gamma \vdash_1 s \stackrel{?}{=} t : A$ *and* $\mathsf{decomp}\ (s \stackrel{?}{=} t) = E$*, then* $\Gamma \vdash_1 E : L$ *for some* L.
2. *If* $\Gamma \vdash_1 E : L$ *and* $\mathsf{decomp}\ E = E'$*, then* $\Gamma \vdash_1 E' : L'$ *for some* L'.
3. *If* $\mathsf{decomp}\ (s \stackrel{?}{=} t) = E$*, then* $\mathsf{vars}\ E \subseteq \mathsf{vars}\ [s, t]$.
4. *If* $\mathsf{decomp}\ E = E'$*, then* $\mathsf{vars}\ E' \subseteq \mathsf{vars}\ E$.

8.1.2 Unification Relation

Using decomp we introduce the relation $E \mapsto \sigma$ whose operational reading yields a unification procedure. Before we formally introduce the relation, we use the example from above to motivate the relation. Given the system $\mathsf{g\ a\ b} \stackrel{?}{=} \mathsf{g\ a\ b}, \mathsf{g}\ x\ y \stackrel{?}{=} \mathsf{g}\ (\mathsf{g}\ y\ \mathsf{a})\ (\mathsf{g\ a\ a})$ a decomposition yields the system $x \stackrel{?}{=} \mathsf{g}\ y\ \mathsf{a}$ and $y \stackrel{?}{=} \mathsf{g\ a\ a}$. To unify both equations we can choose $\sigma y = \mathsf{g\ a\ a}$ and $\sigma x = (\mathsf{g}\ y\ \mathsf{a})[\mathsf{g\ a\ a}/y] = \mathsf{g}\ (\mathsf{g\ a\ a})\ \mathsf{a}$.

$\boxed{E \mapsto \sigma}$

Definition 8.5

$$\frac{\mathsf{decomp}\ E = \mathsf{nil}}{E \mapsto id}$$

$$\frac{\mathsf{decomp}\ E = x \stackrel{?}{=} s :: E' \quad \mathsf{free}\ x \quad x \notin \mathsf{vars}\ s \quad \forall y \in \mathsf{vars}\ s.\ \mathsf{free}\ y \quad E'[s/x] \mapsto \sigma}{E \mapsto \sigma[x := s[\sigma]]}$$

In general, whenever the decomposition of E returns an empty system, all equations of E are trivially unifiable. If the decomposition yields a nonempty system $x \stackrel{?}{=} s :: E'$, then we can analyse whether x is a bound variable or s contains a bound variable. If this is the case, then $x \stackrel{?}{=} s$ cannot be solved with a substitution that respects bound variables. Note that in the case where s is the variable x, the decomposition would have eliminated the equation. If x occurs as a proper subterm of s, then there cannot exist a substitution unifying both sides. Otherwise, we can substitute s for x in the remaining system E and recursively obtain a substitution σ unifying this modified system. If such a substitution σ is found, then we extend it by a binding for x. As the above example shows, s can contain free variables thus we have to ensure that those variables are replaced by their instantiations from σ.

Computability We observe that the number of variables decreases with every recursive step in the $E \mapsto \sigma$ relation. This motivates the following induction principle for systems of equations.

Lemma 8.6

$$\frac{\forall E.\ (\forall E'.\ \mathsf{vars}\ E' \subsetneq \mathsf{vars}\ E \to P\ E') \to P\ E}{\forall E.\ P\ E}$$

Proof Follows from the well-foundedness of $\subsetneq$. □

Fact 8.7 (Computability) *For every list of equations E we can compute a substitution σ with $E \mapsto \sigma$, or we can prove that no such σ exists.*

Proof By induction on E using the induction principle from Lemma 8.6. Case analysis on $\mathsf{decomp}\ E$:

1. Case $\mathsf{decomp}\ E = \mathsf{nil}$. Then $E \mapsto id$.
2. Case $\mathsf{decomp}\ E = x \stackrel{?}{=} s :: E'$. We decide $\mathsf{free}\ x, x \notin \mathsf{vars}\ s$ and $\forall x \in \mathsf{vars}\ s.\ \mathsf{free}\ x$. If one of the above does not hold, then there exists no such σ. By the inductive hypothesis we decide whether there exists σ with $E'[s/x] \mapsto \sigma$, since $x \notin \mathsf{vars}\ (E'[s/x])$. If this is the case, then $E \mapsto \sigma[x := s[\sigma]]$. Otherwise, there cannot be a σ which could be derived from the rules of $E \mapsto \sigma$.
3. Otherwise, no such σ can be derived. □

Soundness We show that whenever we can derive a substitution $E \mapsto \sigma$, then σ is sufficient to unify the equations in E. The replacement of x with s in the relation $E \mapsto \sigma$ is justified by the following lemma:

Lemma 8.8 $x \stackrel{?}{=} s :: E \approx x \stackrel{?}{=} s :: E[s/x]$

To conclude soundness, we show that σ is λ-free, respects bound variables, and that σ is well-typed if E was well-typed.

Fact 8.9 (Soundness) *Let $E \mapsto \sigma$.*

1. *If $\Gamma \vdash_1 E : L$ and $\mathsf{ord}\ \Gamma \leq 1$, then $\Gamma \vdash_1 \sigma : \Gamma$.*
2. *$\sigma \vDash E$.*
3. *σx is λ-free for all x.*
4. *σ respects bound variables.*

Proof By induction on $E \mapsto \sigma$. The first claim follows with Lemma 8.4. The second claim follows using Lemmas 8.2 and 8.8. The others are trivial. □

Completeness We show that whenever the a system of equations E is unifiable by a substitution τ respecting the bound variables, then we can derive a substitution σ unifying E using the rules of $E \mapsto \sigma$. The following lemma ensures that all premises of the second rule are fulfilled, whenever there is a unifying substitution τ.

Lemma 8.10 *Let $\tau \vDash x \stackrel{?}{=} s$ such that τ respects the bound variables.*

1. *If $x \neq s$, then $x \notin \mathsf{vars}\ s$.*
2. *If $x \notin \mathsf{vars}\ s$, then $\mathsf{free}\ x$.*
3. *If $x \notin \mathsf{vars}\ s$ and $\mathsf{free}\ x$, then $\mathsf{free}\ y$ for all $y \in \mathsf{vars}\ s$.*

Using the above lemma we establish completeness.

Fact 8.11 (Completeness) *Let τ be a substitution that respects bound variables. If $\tau \vDash E$, then there exists σ with $E \mapsto \sigma$.*

Proof By induction on E using the induction principle of Lemma 8.6, and Lemmas 8.2, 8.3, 8.8 and 8.10. □

8.2 Full First-Order Unification

In general, we consider the problem $\mathbf{U}_1$ to be the problem of first-order unification. As a consequence, there are first-order unifiable terms which are not λ-free. Take $\Gamma \vdash_1 \lambda y.x \stackrel{?}{=} \lambda y.\mathsf{a} : \alpha \to \alpha$ in the context $\Gamma = (x : \alpha)$ for example. Both terms are unifiable with the substitution $\sigma x = \mathsf{a}$. We observe that in a first-order equation

$\Gamma \vdash_1 s \stackrel{?}{=} t : A$ all occurring variables are at most of order one. Thus any normal form that is inserted for a variable in the process of unification cannot be an abstraction. The following lemma ensures that the terms inserted for variables are λ-free.

Lemma 8.12 *If* $\Gamma \vdash_1 s : A$, $\mathsf{normal}\ s$, *and the applicative head of* s *is an atom, then* s *is* λ*-free.*

Proof The claim follows by induction on $\mathsf{nf}\ s$. □

We justify reasoning about equality instead of equivalence even in the first-order fragment containing abstractions with the following lemma:

Lemma 8.13 *Let* $\Gamma \vdash_1 s \stackrel{?}{=} t : A$ *for normal terms* s *and* t. *If* $\Delta \vdash_1 \sigma : \Gamma$ *and* $\mathsf{normal}\ (\sigma x)$ *for all* x, *then* $s[\sigma] \equiv t[\sigma]$ *implies* $s[\sigma] = t[\sigma]$.

Theorem 8.14 $\mathbf{U}_1$ *is decidable.*

Proof By Fact 5.11 it suffices to decide whether there exists a unifying substitution inserting normal forms only. Let $\Gamma \vdash_1 s \stackrel{?}{=} t : A$. Due to the retyping presented in Fact 5.14, we can assume wlog. that s and t are normal, $\mathsf{ord}\ \Gamma \leq 1$, and $\mathsf{ord}\ A \leq 2$. Lemma 8.13 justifies deciding whether there exists a substitution σ and a context Δ such that:

$$\Delta \vdash_1 \sigma : \Gamma,\ s[\sigma] = t[\sigma],\ \text{and}\ \mathsf{normal}\ \sigma x\ \text{for all}\ x.$$

Since s and t are normal, we can do a normal form analysis on $\mathsf{nf}\ s$ and $\mathsf{nf}\ t$. Let $s = \lambda x_1 \cdots x_k.a_1\ T_1$ and $t = \lambda x_1 \cdots x_l.a_2\ T_2$.

1. Case $k \neq l$. Wlog. let $k < l$. We show both terms cannot be unifiable. By way of contradiction assume both terms are unifiable. If a_1 is a constant, then substitution does not affect a_1 and thus we can obtain an equation between an abstraction and a constant. This is clearly a contradiction. If a_1 is a variable, then the type of this variable must be a function type. A contradiction, since first-order terms cannot contain function variables.

2. Case $k = l$. Using Lemma 8.12 we establish that $a_1\ T_1$ and $a_2\ T_2$ are λ-free. Thus, we can use the decision procedure from Fact 8.7 to decide whether $[a_1\ T_1 \stackrel{?}{=} a_2\ T_2] \mapsto \sigma$ for some σ where there variables $x_1, \ldots, x_k$ are considered bound and all others are free. If this is the case, then soundness (Fact 8.9) guarantees that σ unifies s and t. If this is not the case, then completeness (Fact 8.11) guarantees that no σ exists unifying s and t. □

8.3 Remarks

1. Note that in the formalisation the above theorem has to be adapted to the De Bruijn setting. In particular, the substitution σ has to be adapted when moving from the λ-free fragment to the full first-order fragment.

2. Using the Equations tool [54] we define a Coq function $\mathsf{unif}\ E$ which computes a substitution σ if E is unifiable and returns $\emptyset$ otherwise. The function can be understood as the unification procedure computing the relation $E \mapsto \sigma$. In particular, we show:

   Lemma 8.15 $\mathsf{unif}\ E = \sigma$ *iff* $E \mapsto \sigma$.

   Using the well-foundedness of $\subsetneq$ and Lemma 8.4 it is straightforward to argue termination using Equations.

Chapter 9

Conservativity & Constants

Huet [28] and Goldfarb [25] give proofs of the undecidability of third and second-order unification which we formalise in Chapters 6 and 7. In this chapter, we show how these results can be extended to yield the undecidability of higher-order unification in general. Explicitly, we establish the conservativity of unification meaning a nth-order equation $\Gamma \vdash_n s \stackrel{?}{=} t : A$ is unifiable if and only if it is unifiable in the nth-order fragment of the calculus. From this we conclude $\mathsf{U}_n \preceq \mathsf{U}_m \preceq \mathsf{U}$ for $n \leq m$. In addition, we present techniques for the introduction and elimination of constants without affecting unifiability. Using these techniques, we can obtain Goldfarb's strengthened result that unification is already undecidable in languages with only a single binary constant g. The combined results of this chapter yield:

$$\mathsf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}} \quad \preceq \quad \mathsf{U}_2^{\{\mathsf{g}\}} \quad \preceq \quad \mathsf{U}_3^{\{\mathsf{g}\}} \quad \preceq \quad \mathsf{U}_3^{\emptyset} \quad \preceq \quad \mathsf{U}_3 \quad \preceq \quad \mathsf{U}$$

where $\mathsf{U}_n^{\{\mathsf{g}\}}$ is nth-order unification in the fragment where g is the only constant and $\mathsf{U}_3^{\emptyset}$ is third-order unification in the fragment without constants.

9.1 Conservativity

In this section, we establish the conservativity of unification and prove for $n+1 \leq m$:

$$\mathsf{U}_n \quad \preceq \quad \mathsf{U}_{n+1} \quad \preceq \quad \cdots \quad \preceq \quad \mathsf{U}_m \quad \preceq \quad \cdots \quad \preceq \quad \mathsf{U}$$

As the reduction functions used in these reductions are identity functions up to the typing judgements, one can say that mth-order unification subsumes nth-order unification and higher-order unification subsumes mth-order unification. Similarly, we establish that a system $\Gamma \vdash_n E : A$ is unifiable, if it is already unifiable by a substitution of order n, yielding a proof of

$$\mathsf{SU}_n \quad \preceq \quad \mathsf{SU}_{n+1} \quad \preceq \quad \cdots \quad \preceq \quad \mathsf{SU}_m \quad \preceq \quad \cdots \quad \preceq \quad \mathsf{SU}$$

Operations on Constants In the proof of $\mathbf{U}_n \preceq \mathbf{U}_{n+1}$ we need to transform a substitution $\Delta \vdash_{n+1} \sigma : \Gamma$ into a substitution $\Sigma \vdash_n \tau : \Gamma$ which still unifies the terms. In particular, we have to transform terms with constants of order $n+2$ into terms with constants of at most order $n+1$. We write $\mathsf{consts}\ s$ for the list containing all constants of s and $\mathsf{consts}\ S$ for the list containing all the constants of terms from S.

$\boxed{\mathsf{consts}\ s}$
$\boxed{\mathsf{consts}\ S}$

Lemma 9.1

1. *If $s \succ^* t$, then $\mathsf{consts}\ s \supseteq \mathsf{consts}\ t$.*
2. *If $\Gamma \vdash_n s : A$ and $c \in \mathsf{consts}\ s$, then $\mathsf{ord}\ (\Omega c) \leq n+1$.*

To eliminate and introduce constants, we define an operation $s[\kappa]$, similar to substitution, replacing constants according to κ. Analogous to substitution this operation is capture avoiding. For example, for $\kappa\mathsf{a} = x$ we have $(\mathsf{g}\ x\ \mathsf{a})[\kappa] = \mathsf{g}\ x\ x$ and $(\lambda x.\mathsf{a})[\kappa] = \lambda y.x$.

$\boxed{s[\kappa]}$ $\boxed{\sigma[\kappa]}$

Definition 9.2 *For a map from constants to terms κ, we write $s[\kappa]$ for the result of replacing all constants of s according to κ. We write $\sigma[\kappa]$ for the substitution obtained by replacing the constants of σ according to κ.*

Lemma 9.3

1. *$\succ$, $\succ^*$, $\equiv$ are compatible with constant replacement.*
2. *If $\kappa c = c$ for all $c \in \mathsf{consts}\ s$, then $s[\kappa] = s$.*
3. *If $s[\kappa] = s$, then $s[\sigma[\kappa]] = s[\sigma][\kappa]$.*
4. *If $\Gamma \vdash s : A$ and $\Gamma \vdash \kappa c : \Omega c$ for all $x \in \mathsf{consts}\ s$, then $\Gamma \vdash s[\kappa] : A$.*
5. *If $\Gamma \vdash_n s : A$ and $\Gamma \vdash_n \kappa c : \Omega c$ for all $x \in \mathsf{consts}\ s$, then $\Gamma \vdash_n s[\kappa] : A$.*

Inhabiting Types When proving conservativity results about unification, we need to replace constants and variables not belonging to the nth-order fragment of the calculus. For example, the variable $y : (\alpha \to \beta) \to \beta \to \alpha$ does not belong to the second-order fragment but it may occur in a unifying substitution for a second-order equation. Its type $(\alpha \to \beta) \to \beta \to \alpha$ is not inhabited by any closed term in a language without constants. However, as substitutions may insert open terms, the variable y can be replaced by $\lambda y_1 y_2.x$ in the context $\Delta = (x : \alpha)$. In contrast to the variable y, $\lambda y_1 y_2.x$ is a first-order term. We generalise this idea with the term $\mathsf{inhab}_x\ A$ which inhabits type A whenever x is a variable of the target type of A.

Definition 9.4

$$\mathsf{target}\ \alpha = \alpha \qquad \mathsf{arity}\ \alpha = 0$$

$$\mathsf{target}\ (A \to B) = \mathsf{target}\ B \qquad \mathsf{arity}\ (A \to B) = 1 + \mathsf{arity}\ B$$

$$\mathsf{target}\ \Gamma = [(x, \mathsf{target}\ A) \mid (x, A) \in \Gamma]$$

$$\mathsf{inhab}_x\ A := \Lambda Y.x \quad \textit{where}\ Y = [y_1, \ldots, y_{\mathsf{arity}\ A}]\ \textit{and}\ x \notin Y$$

target A
target Γ
arity A
inhab$_x$ A

Using $\mathsf{inhab}_x\ A$ we can produce terms of arbitrary types A whenever we may choose the context.

Lemma 9.5 $\dfrac{(x, \mathsf{target}\ A) \in \Gamma}{\Gamma \vdash_1 \mathsf{inhab}_x\ A : A}$ *and* $\dfrac{(x, A) \in \Gamma}{\mathsf{target}\ \Gamma \vdash_1 \mathsf{inhab}_x\ A : A}$

Substitution Transformations Recall from Section 2.5 the three transformations we apply to to higher-order substitutions, unifying an equation $\Gamma \vdash_n s \stackrel{?}{=} t : A$, in order to obtain nth-order substitutions.

1. Whenever we encounter a free variable $x : A_1 \to \cdots \to A_k \to \alpha$ not of order n, we substitute it with the first-order term $\lambda x_1 \cdots x_k.z$ where $z : \alpha$.
2. Whenever we encounter a constant $c : A_1 \to \cdots \to A_k \to \alpha$ which does not appear in s or t, we replace it with the first-order term $\lambda x_1 \cdots x_k.z$ where $z : \alpha$ analogously to the previous technique.
3. We normalise all terms after applying the above transformations.

For the first transformation, we substitute all free variables by first-order inhabitants of the same type.

Lemma 9.6 *Let* $\Gamma \vdash_n s \stackrel{?}{=} t : A$. *If* $\Delta \vdash \sigma : \Gamma$ *and* $s[\sigma] \equiv t[\sigma]$, *then there exists a context* Σ *and a substitution* $\Sigma \vdash \tau : \Gamma$ *with* $s[\tau] \equiv t[\tau]$ *and* $\mathsf{ord}\ \Sigma \leq 1$.

Proof Define $\tau'\ x := \mathsf{inhab}_x\ A$ if $(x : A) \in \Delta$ and $\tau'\ x := x$ otherwise. Pick $\Sigma := \mathsf{target}\ \Delta$ and $\tau := \sigma[\tau']$. The claim follows with Fact 4.11 and Lemma 9.5 and $\equiv$ being compatible with substitution (Fact 4.7). □

For the second transformation, recall that the key observation is that s and t are unifiable if and only if they are unifiable with a substitution that only draws constants from $\mathsf{consts}\ s$ and $\mathsf{consts}\ t$. In the case where s and t are in the nth-order fragment of the language, all such constants are guaranteed to have an order of at most $n + 1$. While we require the transformation only for well-typed higher-order

substitutions, the same transformation can also be used to simplify the proof of conservativity of unification with respect to constants. In said proof it is essential that the order of the substitution is not increased in the transformation. Therefore, we show that the relevant constants can be removed from substitutions inserting terms of order $m > 0$ and deduce in a second step that this entails the case of well-typed higher-order substitutions in general.

Lemma 9.7 *Let* $m > 0$ *and* $\Gamma \vdash_n s \stackrel{?}{=} t : A$. *If* $\Delta \vdash_m \sigma : \Gamma$ *and* $s[\sigma] \equiv t[\sigma]$, *then there exists a substitution* $\Delta \cup \Sigma \vdash_m \tau : \Gamma$ *with* $s[\tau] \equiv t[\tau]$, $\mathsf{ord}\ \Sigma \leq 1$ *and* $\mathsf{consts}\ (\tau x) \subseteq \mathsf{consts}\ [s,t]$ *for all* x.

Proof We define the list $C := \mathsf{consts}\ [\sigma x \mid x \in \mathsf{dom}\ \Gamma]$ containing all constants that potentially have to be replaced and associate every constant $c \in C$ with a fresh variable x_c. We replace constants according to κ, where $\kappa c = c$ if $c \in \mathsf{consts}\ [s,t]$, $\kappa c = \mathsf{inhab}_{x_c}\ (\Omega c)$ if $c \in C$, and $\kappa c = x_0$ otherwise for some fixed variable x_0. We pick $\Sigma := \mathsf{target}\ [(x_c : \Omega c) \mid c \in C]$ and $\tau := \sigma[\kappa]$. Typing follows from Lemmas 9.3 and 9.5 and from Lemma 9.3 we know that $s[\kappa] = s$ and $t[\kappa] = t$. Hence, $s[\tau] = s[\sigma[\kappa]] = s[\sigma][\kappa] \equiv t[\sigma][\kappa] = t[\sigma[\kappa]] = t[\tau]$. □

Note that in the formalisation we do not associate every constant with a variable directly. Instead, the variable x_c is represented by the De Bruijn index $|\Delta| + i$ where i is the index of the first occurrence of c in C. The effect is the same — we are guaranteed that x_c does not occur in $\mathsf{dom}\ \Delta$ and there is at least one variable for every constant.

Corollary 9.8 *Let* $\Gamma \vdash_n s \stackrel{?}{=} t : A$. *If* $\Delta \vdash \sigma : \Gamma$ *and* $s[\sigma] \equiv t[\sigma]$, *then there exists a substitution* $\Delta \cup \Sigma \vdash \tau : \Gamma$ *with* $s[\tau] \equiv t[\tau]$, $\mathsf{ord}\ \Sigma \leq 1$ *and* $\mathsf{consts}\ (\tau x) \subseteq \mathsf{consts}\ [s,t]$ *for all* x.

In the proof of conservativity, we need to show that certain terms obtained after transformations 1, 2, and 3 are in the nth-order fragment of the calculus.

Lemma 9.9

$$\frac{\Gamma \vdash s : A \quad \mathsf{normal}\ s \quad \mathsf{ord}\ [\Omega c \mid c \in \mathsf{consts}\ s], \mathsf{ord}\ A \leq n+1 \quad \mathsf{ord}\ \Gamma \leq n}{\Gamma \vdash_n s : A}$$

Proof By induction on $\Gamma \vdash s : A$.

1. Case $\Gamma \vdash x : A$. Then $(x : A) \in \Gamma$, thus the claim follows with $\mathsf{ord}\ \Gamma \leq n$.
2. Case $\Gamma \vdash c : \Omega c$. The claim follows by assumption.
3. Case $\Gamma \vdash \lambda x.s : A \to B$. Then $\Gamma, (x : A) \vdash s : B$. We utilise $\mathsf{ord}\ (A \to B) \leq n+1$ to obtain $\mathsf{ord}\ A \leq n$ and the claim follows with the inductive hypothesis.

4. Case $\Gamma \vdash s\ t : B$. Then $\Gamma \vdash s : A \to B$ and $\Gamma \vdash t : A$ for some A. Since $s\ t$ is normal, the applicative head of s must be some atom a. As a consequence, using the head decomposition from Lemma 4.27 and the typing inversion for list application from the right (Lemma 4.26), we obtain that a has some type of the form $\overleftarrow{L} \to A \to B$. Regardless of whether a is a variable or a constant, we can prove that the order of its type is at most $n + 1$ by our assumptions. Thus $\mathsf{ord}\ A \leq n$ and the claim follows with the inductive hypotheses. □

Since Γ may also contain variables which neither occur in s nor in t, the constraint $\Gamma \vdash_n s \stackrel{?}{=} t : A$ does not imply that $\mathsf{ord}\ \Gamma \leq n$. As a consequence, there can be $(x : A) \in \Gamma$ where $\mathsf{ord}\ A > n$. Thus, if t is the term inserted by a substitution after transformations 1, 2, and 3, we cannot ensure that t is in the nth-order fragment of the language, since Lemma 9.9 is not applicable. However, as only the variables occurring in s and t are relevant for unifiability, we can address this complication with the following lemma:

Lemma 9.10 *Let $\Gamma \vdash_n s \stackrel{?}{=} t : A$. If $s[\sigma] \equiv t[\sigma]$ and $\Delta \vdash_n \sigma x : A$ for all $(x : A) \in \Gamma$ where $x \in \mathsf{vars}\ [s, t]$, then there exists a substitution $\Sigma \vdash_n \tau : \Gamma$ such that $s[\tau] \equiv t[\tau]$.*

Proof We associate with every $(x : A) \in \Gamma$ a fresh variable y_x. Pick $\Sigma := \Delta \cup \mathsf{target}\ [(y_x : A) \mid (x : A) \in \Gamma]$ and $\tau x := \sigma x$ if $x \in \mathsf{vars}\ [s, t]$, $\tau x := \mathsf{inhab}_{y_x}\ A$ if $(x : A) \in \Gamma$ and $\tau x := x$ otherwise. Typing follows with Lemma 9.3 and the equivalence with Fact 4.1. □

Conservativity We combine the transformation presented above to conclude the conservativity of unification and lift the result to systems of equations.

Fact 9.11

1. *Let $\Gamma \vdash_n s \stackrel{?}{=} t : A$. If $\Delta \vdash \sigma : \Gamma$ and $s[\sigma] \equiv t[\sigma]$, then there is a substitution $\Sigma \vdash_n \tau : \Gamma$ with $s[\tau] \equiv t[\tau]$ for some context Σ.*
2. *Let $\Gamma \vdash_n E : L$. If $\Delta \vdash \sigma : \Gamma$ and $E_{\mathsf{L}}[\sigma] \equiv E_{\mathsf{R}}[\sigma]$, then there is a substitution $\Sigma \vdash_n \tau : \Gamma$ with $E_{\mathsf{L}}[\tau] \equiv E_{\mathsf{R}}[\tau]$ for some context Σ.*

Proof

1. Subsequent use of Lemmas 9.6 and 9.10 and Corollary 9.8 as well as Lemmas 9.1 and 9.9.
2. For systems, we use a similar construction as in Section 5.2 and transform lists of terms $E_{\mathsf{L}}, E_{\mathsf{R}}$ into single terms $\lambda h.h\ E_{\mathsf{L}}, \lambda h.h\ E_{\mathsf{L}}$. The claim then follows analogously to Item 1. □

Theorem 9.12 (Conservativity) *Let $n \leq m$.*

1. $\mathbf{U}_n \preceq \mathbf{U}_{n+1}$
2. $\mathbf{U}_n \preceq \mathbf{U}_m$
3. $\mathbf{U}_n \preceq \mathbf{U}$
4. $\mathbf{SU}_n \preceq \mathbf{SU}_{n+1}$
5. $\mathbf{SU}_n \preceq \mathbf{SU}_m$
6. $\mathbf{SU}_n \preceq \mathbf{SU}$

Proof By picking the identity function for each reduction and Facts 4.15 and 9.11. □

None of the reduction functions have to transform Γ, s, t, or A. Thus, one can say higher-order unification subsumes nth-order unification. As a corollary, we conclude the undecidability of higher-order unification in general and establish the enumerability of nth-order unification, system unification and nth-order system unification.

Corollary 9.13

1. $\mathbf{MPCP} \preceq \mathbf{U}$ *and* $\mathbf{PCP} \preceq \mathbf{U}$.
2. $\mathbf{U}_n, \mathbf{SU}$ *and* $\mathbf{SU}_n$ *are enumerable.*

9.2 Constants

In this section we investigate the role of constants in unification. Goldfarb [25] proves the undecidability of second-order unification in the presence of a 2-ary function constant whereas Farmer [16] proves the decidability of second-order unification for all languages containing only constants of at most arity one. Huet [28] on the other hand shows that unification is undecidable in third-order languages regardless of the constants of the language. In the following, we present techniques how constants can be introduced and eliminated from a language in the context of unification. As a consequence, we obtain Huet's result as a corollary of Goldfarb's result.

$\boxed{\mathbf{U}^{\mathcal{C}}}$ $\boxed{\mathbf{U}_n^{\mathcal{C}}}$

In Chapter 4, we defined the simply-typed λ-calculus parametric over a type $\mathcal{C}$ of constants. We write $\mathbf{U}^{\mathcal{C}}$ to refer to the unification problem in the language where all constants are of type $\mathcal{C}$. Analogously, we write $\mathbf{U}_n^{\mathcal{C}}$ for the nth-order unification problem in the language where all constants are of type $\mathcal{C}$.

Introduction of Constants In the following we prove the how constants can be introduced without affecting nth-order unifiability. We assume two types of constants $\mathcal{C}, \mathcal{D}$ such that the pair (ι, ϱ) forms a tight retraction $\mathcal{C} \hookrightarrow \mathcal{D}$ and denote the constant signature of $\mathcal{C}$ by $\Omega_{\mathcal{C}}$ and the signature of $\mathcal{D}$ by $\Omega_{\mathcal{D}}$. Furthermore, we require $\Omega_{\mathcal{C}} c = \Omega_{\mathcal{D}}(\iota c)$ for all $c : \mathcal{C}$. We show the conservativity of nth-order unification with respect to constants, meaning $\mathbf{U}_n^{\mathcal{C}} \preceq \mathbf{U}_n^{\mathcal{D}}$. Formally, we transform instances of $\mathbf{U}_n^{\mathcal{C}}$ into instances of $\mathbf{U}_n^{\mathcal{D}}$ with the reduction function:

$$f(\Gamma \vdash_n s \stackrel{?}{=} t : A) = \Gamma \vdash_n s[\iota] \stackrel{?}{=} t[\iota] : A$$

Once more, the key observation is that two terms u, v are unifiable if and only if they are unifiable with a substitution that only draws constants from $\mathsf{consts}\, u$ and $\mathsf{consts}\, v$. As a consequence, when given a substitution unifying $s[\iota]$ and $t[\iota]$, we replace all constants that do not occur in $\mathsf{im}\ \iota$ with terms of the same type. We cannot simply replace constants by variables as they may have a type of order $n+1$ whereas the order of the types of variables is bounded by n.

For example, consider $\mathcal{C} = \emptyset$ and $\mathcal{D} = \{\mathsf{g}\}$ for $\mathsf{g} : \alpha \to \alpha \to \alpha$. For the sake of simplicity, we omit the retraction (ι, ϱ) and use set notation in this example. The equation $\Gamma \vdash f \stackrel{?}{=} g : \alpha \to \alpha \to \alpha$ for $\Gamma = (f, g : \alpha \to \alpha \to \alpha)$ is solved by $\sigma f = \sigma g = \mathsf{g}$ in a language with constants from $\mathcal{D}$. To obtain a solution with constants only from $\mathcal{C}$, we replace g by a term of the same type in σ. Explicitly, we replace g with the term $\lambda x_1 x_2 . x_{\mathsf{g}}$ where x_{g} is a fresh variable. The resulting substitution τ can be typed in the context $\Delta = (x_{\mathsf{g}} : \alpha)$ and we have $f[\tau] \equiv \lambda x_1 x_2 . x_{\mathsf{g}} \equiv g[\tau]$.

In general, we associate every constant $d : \mathcal{D}$ with a fresh variable x_d and replace constants according to κ:

$$\kappa\, d = c \ \text{ where } \varrho d = c \qquad\qquad \kappa\, d = \mathsf{inhab}_{x_d}\, (\Omega_{\mathcal{D}} d) \ \text{ where } \varrho d = \emptyset$$

Lemma 9.14 *Let* $D := \mathsf{consts}\, [\sigma x \mid x \in \mathsf{dom}\, \Gamma]$.

1. $s[\iota][\sigma[\iota]] = s[\sigma][\iota]$ *and* $s[\sigma[\kappa]] = s[\iota][\sigma][\kappa]$
2. $\dfrac{\Delta \vdash_n \sigma : \Gamma}{\Delta \vdash_n \sigma[\iota] : \Gamma}$ *and* $\dfrac{\Delta \vdash_n \sigma : \Gamma \quad \mathsf{target}\, [(x_d : \Omega_{\mathcal{D}} d) \mid d \in D] \subseteq \Delta}{\Delta \vdash_n \sigma[\kappa] : \Gamma}$

Fact 9.15 $\mathbf{U}_n^{\mathcal{C}} \preceq \mathbf{U}_n^{\mathcal{D}}$

Proof We pick the reduction function $f(\Gamma \vdash_n s \stackrel{?}{=} t : A) = \Gamma \vdash_n s[\iota] \stackrel{?}{=} t[\iota] : A$.

$\to$: Let $\Sigma \vdash_n \tau : \Gamma$ and $s[\tau] \equiv t[\tau]$. Pick $\Delta := \Sigma$ and $\sigma := \tau[\iota]$. Then by Lemma 9.14 $s[\iota][\sigma] = s[\iota][\tau[\iota]] = s[\tau][\iota] \equiv t[\tau][\iota] = t[\iota][\tau[\iota]] = t[\iota][\sigma]$ and $\Delta \vdash_n \sigma : \Gamma$.

$\leftarrow$: Let $\Sigma \vdash_n \tau : \Gamma$ and $s[\iota][\tau] \equiv t[\iota][\tau]$. We group all relevant constants in the list $D := \mathsf{consts}\, [\sigma x \mid x \in \mathsf{dom}\, \Gamma]$. Pick $\Delta := \Sigma \cup \mathsf{target}\, [(x_d : \Omega_{\mathcal{D}} d) \mid d \in D]$ and $\sigma := \tau[\kappa]$. Lemma 9.14 ensures $\Delta \vdash_n \sigma : \Gamma$ and in addition, $s[\sigma] = s[\tau[\kappa]] = s[\iota][\tau][\kappa] \equiv t[\iota][\tau][\kappa] = t[\tau[\kappa]] = t[\sigma]$. □

In the formalisation the variable x_d is obtained through a mapping h from constants to variables. Analogous to the proof of Lemma 9.7, we pick the assignment h such that $hd = |\Sigma| + i$ where i is the first occurrence of d in D if $d \in D$, and $hd = x_0$ if $d \notin D$ for some fixed x_0.

Elimination of Constants In the following we show how all constants with an order strictly smaller than n can be eliminated from the nth-order fragment without affecting unifiability. In particular, this result will allow us to derive the strongest version of Goldfarb's result [25]: Unification is undecidable in any second-order language with a single 2-ary function constant of order 2. As mentioned in Chapter 2, the decidability of monadic second-order unification [16] implies that we cannot hope to eliminate all constants of order n as well.

Recall from Section 2.7 that the main idea is to replace constants by bound variables. For example, the terms $\Gamma \vdash \mathsf{g}\ x \stackrel{?}{=} \mathsf{g}\ \mathsf{a} : \alpha \to \alpha$ where g, a are constants and $\Gamma = (x : \alpha)$ are transformed into the terms $\Gamma \vdash \lambda x_\mathsf{a}.\mathsf{g}\ (x\ x_a) \stackrel{?}{=} \lambda x_\mathsf{a}.\mathsf{g}\ x_a : \alpha \to \alpha \to \alpha$ where $\Delta = (x : \alpha \to \alpha)$, if a is eliminated. Instead of $\sigma x = \mathsf{a}$, both terms are unifiable under the substitution $\sigma x = \lambda x_a.x_a$ after the transformation.

Formally, we assume two types of constants $\mathcal{C}, \mathcal{D}$ and the pair (ι, ϱ) forming a tight retraction $\mathcal{C} \hookrightarrow \mathcal{D}$ and prove that $\mathbf{U}_n^\mathcal{D} \preceq \mathbf{U}_n^\mathcal{C}$ if $\mathsf{ord}\ (\Omega_\mathcal{D} d) < n$ for all $d \notin \mathsf{im}\ \iota$. Similar to the previous paragraph, we require $\Omega_\mathcal{C} c = \Omega_\mathcal{D}(\iota c)$ for all $c : \mathcal{C}$. The retract can be understood as a filter deciding which values of $\mathcal{D}$ are to be retained and which are to be replaced.

To simplify matters, we start by analysing how a fixed list of constants D can be replaced. For now, fix a list of constants D and assume $\mathsf{ord}\ (\Omega_\mathcal{C} d) < n$ for all $d \in D$.

We associate with every constant $d \in D$ a fresh variable x_d and group them together as $X = [x_d \mid d \in D]$. In the formalisation, this is realised similarly to the proof of 9.7. We encode variables with εx, constants by εd and lift the encoding to terms εs, types $\varepsilon(A)$, contexts $\varepsilon(\Gamma)$, and substitutions $\varepsilon\sigma$.

$$\varepsilon x := x\ X \qquad \varepsilon d := \begin{cases} c & \varrho d = c \\ x_d & d \in D \\ x_0 & \text{othw.} \end{cases} \qquad \varepsilon s := \Lambda X.s[\sigma_\varepsilon][\kappa_\varepsilon]$$

$$\varepsilon(A) := [\Omega_\mathcal{D} d \mid d \in D] \to A \qquad \varepsilon(\Gamma) := [\varepsilon(A) \mid A \in \Gamma] \qquad (\varepsilon\sigma)x := \varepsilon(\sigma x)$$

where $\sigma_\varepsilon x := \varepsilon x$, $\kappa_\varepsilon c := \varepsilon c$, and x_0 is some fixed variable. In the previous example, we would have $D = [\mathsf{a}]$, $X = [x_\mathsf{a}]$, $\sigma_\varepsilon x = x\ x_\mathsf{a}$, $\kappa_\varepsilon \mathsf{a} = x_\mathsf{a}$ and $\kappa_\varepsilon \mathsf{g} = \mathsf{g}$.

We observe $(\varepsilon(\mathsf{g}\ x))\ \mathsf{a} = (\lambda x_\mathsf{a}.\mathsf{g}\ (x\ x_a))\ \mathsf{a} \equiv \mathsf{g}\ (x\ \mathsf{a})$. Thus, applying the encoded term to a almost reverses the effect of the encoding. The only difference is that after reduction, the variable x is applied to a. This motivates the definition $\varepsilon^{-1} s := s[\iota]\ D$. The idea is that $\varepsilon^{-1}(\varepsilon s) \succ^* s[x \mapsto x\ D]$, meaning the term obtained by encoding and decoding almost reduces to the original except for the variables. The free variables are applied to all constants that were removed by the encoding. We lift the decoding to substitutions by $(\varepsilon^{-1}\sigma)\ x := \varepsilon^{-1}(\sigma x)$.

Lemma 9.16 *Let all constants $d \in \mathsf{consts}\ s$ which are not contained in $\mathsf{im}\ \iota$ be in D. In addition, let all constants $d \in \mathsf{consts}\ (\sigma x)$ which are not contained in $\mathsf{im}\ \iota$ be in D for all x.*

1. $\dfrac{\Gamma \vdash_n s : A}{\varepsilon(\Gamma) \vdash_n \varepsilon s : \varepsilon(A)}$ *and* $\dfrac{\Gamma \vdash_n t : \varepsilon(A)}{\Gamma \vdash_n \varepsilon^{-1} t : A}$
2. $\dfrac{\Delta \vdash_n \sigma : \Gamma}{\varepsilon\Delta \vdash_n \varepsilon\sigma : \varepsilon(\Gamma)}$ *and* $\dfrac{\Delta \vdash_n \tau : \varepsilon(\Gamma)}{\Delta \vdash_n \varepsilon^{-1}\tau : \Gamma}$
3. $\equiv$ *is compatible with ε and ε^{-1}.*
4. $(\varepsilon s)[\varepsilon\sigma] \succ^* \varepsilon(s[\sigma])$ *and* $\varepsilon^{-1}((\varepsilon s)[\tau]) \succ^* s[\varepsilon^{-1}\tau]$

Theorem 9.17 $\mathbf{U}_n^{\mathcal{D}} \preceq \mathbf{U}_n^{\mathcal{C}}$ *whenever* $\mathsf{ord}\ (\Omega_{\mathcal{D}} d) < n$ *for all* $d \notin \mathsf{im}\ \iota$.

Proof Pick $f(\Gamma \vdash_n s \stackrel{?}{=} t : A) = \varepsilon(\Gamma) \vdash_n \varepsilon s \stackrel{?}{=} \varepsilon t : \varepsilon(A)$. Note that since $\mathsf{ord}\ (\Omega_{\mathcal{D}} d) < n$ for all $d \notin \mathsf{im}\ \iota$, the encoding of s and t can be typed in the nth-order fragment of the calculus. It remains to show

$$\mathbf{U}_n^{\mathcal{D}}(\Gamma \vdash_n s \stackrel{?}{=} t : A) \text{ iff } \mathbf{U}_n^{\mathcal{C}}(f(\Gamma \vdash_n s \stackrel{?}{=} t : A))$$

We pick $D := [d \mid d \in \mathsf{consts}\ [s,t] \text{ where } d \notin \mathsf{im}\ \iota]$.

$\rightarrow$: Let $\Sigma' \vdash_n \tau' : \Gamma$ and $s[\tau'] \equiv t[\tau']$. Using Lemma 9.7 we obtain a substitution $\Sigma' \cup \Sigma \vdash_n \tau : \Gamma$ such that $s[\tau] \equiv t[\tau]$ and $\mathsf{consts}\ (\tau x) \subseteq \mathsf{consts}\ [s,t]$ for all x. We pick $\Delta := \varepsilon(\Sigma' \cup \Sigma)$ and $\sigma := \varepsilon\tau$. With Lemma 9.16 we know $\Delta \vdash_n \sigma : \varepsilon(\Gamma)$ and $(\varepsilon s)[\sigma] = (\varepsilon s)[\varepsilon\tau] \equiv \varepsilon(s[\tau]) \equiv \varepsilon(t[\tau]) \equiv (\varepsilon t)[\varepsilon\tau] = (\varepsilon t)[\sigma]$.

$\leftarrow$: Let $\Sigma \vdash_n \tau : \Gamma$ and $(\varepsilon s)[\tau] \equiv (\varepsilon t)[\tau]$. Pick $\Delta := \Sigma$ and $\sigma := \varepsilon^{-1}\tau$. With Lemma 9.16 we know $\Delta \vdash_n \sigma : \Gamma$ and $s[\sigma] = s[\varepsilon^{-1}\tau] \equiv \varepsilon^{-1}((\varepsilon s)[\tau]) \equiv \varepsilon^{-1}((\varepsilon t)[\tau]) \equiv t[\varepsilon^{-1}\tau] = t[\sigma]$. □

Using the techniques presented above we derive Goldfarb's strongest result and show how Huet's result can be obtained from Goldfarb's result.

Corollary 9.18

1. $\mathbf{H10} \preceq \mathbf{U}_2^{\{\mathsf{a},\mathsf{b},\mathsf{g}\}} \preceq \mathbf{U}_2^{\{\mathsf{g}\}}$
2. $\mathbf{U}_2^{\{\mathsf{g}\}} \preceq \mathbf{U}_3^{\{\mathsf{g}\}} \preceq \mathbf{U}_3^{\emptyset} \preceq \mathbf{U}_3$
3. *Let (ι, ϱ) be a retraction $\{\mathsf{g}\} \hookrightarrow \mathcal{C}$ with $\Omega_{\mathcal{C}}(\iota\mathsf{g}) = \alpha \to \alpha \to \alpha$. Then* $\mathbf{H10} \preceq \mathbf{U}_2^{\mathcal{C}}$.

Chapter 10

Formalisation

In this chapter we remark on interesting aspects of the formalisation, highlight important design choices, and discuss complications we encountered. All proofs presented in this thesis are formalised in the proof assistant Coq and the theorems, facts and lemmas of this thesis are hyperlinked with their formal counterparts. The thesis is accompanied by brief documentation of the Coq code available at:

`http://www.ps.uni-saarland.de/~spies/bachelor.php`

The formalisation is self-contained and follows the structure presented in this thesis. The only axiom assumed in the work is the widely accepted axiom of functional extensionality. Functional extensionality expresses that two Coq functions are equal whenever they agree on all arguments. In particular, we do not assume any classical axioms, rendering the formalisation fully constructive. The assumption of functional extensionality has no effect on the computability of Coq functions in a model of computation. Consequently all reduction functions presented in this thesis are computable.

Tools Working with substitutions is simplified by the Autosubst 2 [55] tool. We synthesise the term syntax and the substitution operation $s[\sigma]$ with this tool from a higher-order abstract syntax specification. Furthermore, we use the tool to synthesise several lemmas concerning substitutions and automation to use said lemmas. As a consequence, many claims regarding substitution can be proved automatically by use of the `asimpl` tactic. Frequently proving such claims involves repeated use of the substitution lemmas. In this work, we extended the automation mechanism with substitution lemmas concerning our metalevel operations. Doing so renders the automation powerful enough to automatically prove equalities such as $(\overline{n}((\lambda x.x)\ x))[\mathsf{a}/x] = \overline{n}((\lambda x.x)\ \mathsf{a})$.

We extensively used the rewriting mechanism [53] included with Coq. The relation $\equiv$ is registered as an equivalence relation and $\succ^*$ as a pre–order. The compatibility lem-

mas of both relations are added to the mechanism to simplify their use. For example, the compatibility of ε with $\equiv$ (Lemma 9.16) can be derived automatically from the compatibility of $\equiv$ with list abstractions, substitutions and constant replacement.

We used the Equations tool [54] to implement a unification algorithm for the λ-free fragment of the calculus.

External Code We build on the formalisation of abstract reduction systems contained in the folder `std/ars` which was developed in the lecture Semantics at Saarland University [50]. The code contained in the file `list_reduction.v` is our contribution. Furthermore, we build on the formalisation of enumerability and reductions by Forster et al. [21].

Variables We formalise the simply-typed λ-calculus presented in this thesis using a De Bruijn representation of variables. Recall that the variable n in De Bruijn notation indicates that n binders have to be skipped until the binder of the variable is reached. For example, the term $(\lambda xyz.z)\ (\lambda y.y)$ can be expressed as $(\lambda\lambda\lambda 2)\ (\lambda 0)$. For the sake of readability we choose named syntax on paper. As a consequence some notions differ on paper from their formal counterparts. For example, typing contexts are represented as lists of types in the formalisation and $(x : A) \in \Gamma$ corresponds to $\Gamma[x] = A$. We remark in the thesis whenever this is the case.

Overhead We believe that the overhead generated by formalising all results in a proof assistant is similar to the overhead produced by writing detailed paper proofs. Several proofs in this thesis, for example Theorem 6.12 and Lemma 7.22, require a considerable amount of bookkeeping which is greatly simplified by working in a proof assistant.

Traditionally, higher-order unification is considered in a Church-typed λ-calculus. In this thesis we use Curry-typing instead. While working with a Church-typed calculus can be more appealing on paper, in Coq Church-typing entails dependently typed syntax. Dependently typed syntax complicates statements, definitions and proofs since functions transforming terms have to transform their types as well. Furthermore, dependently typed syntax is currently not supported by the Autosubst 2 [55] tool. As a consequence of Curry-typing, well-typedness must be established at several places in the development and must be preserved while analysing terms.

10.1 Overview

Overall, the self-contained formalisation amounts to approximately 9500 lines of code in its entirety. A significant portion of this code is devoted to what we would call "preliminaries" which we do not consider relevant when it comes to the complexity of

our development. Thus, preliminaries are excluded in the following overview which summarises the size of the formalisation in lines of code.

We formalise the Curry-style simply-typed λ-calculus in the directory `calculus` and the unification problems in the directory `unification`. The undecidability of third-order unification is contained in the directory `third_order` and the undecidability of second-order unification in the directory `second_order`. The decidability of first-order unification is established in `firstorder.v`. Conservativity and the results regarding constants are contained in the directory `concon`.

Overview	**Specification**	**Proofs**
Simply-typed λ-calculus	790	1120
Higher-Order Unification	350	380
Third-Order Undecidability	190	400
Second-Order Undecidability	570	850
First-Order Decidability	290	510
Conservativity & Constants	480	890
Total	**2670**	**4150**

Third-Order Unification Our formalisation of the undecidability of third-order unification can be decomposed into a formal definition of the modified Post correspondence problem and the Post correspondence problem, a definition of the encoding of strings, the original reduction by Huet [28], and our simplified version.

Third-Order Unification	**Specification**	**Proofs**
PCP Problems	30	30
Shared Encoding	80	110
Original	50	170
Simplified	30	90
Total	**190**	**400**

Second-Order Unification Our formalisation of the undecidability of second-order unification can be decomposed into a formal definition of Hilbert's tenth problem, the explanation based on [13], the motivation for the encoding of multiplication, and the actual reduction.

Second-Order Unification	**Specification**	**Proofs**
Hilbert's Tenth Problem	30	10
Motivation Structure	180	260
Motivation Multiplication	30	20
Reduction	330	560
Total	**570**	**850**

Chapter 11

Conclusion

In this chapter we comment on related work and indicate directions into which this work could be expanded in the future. A comprehensive summary of the field of higher-order unification and related problems can be found in [13, 27]. Levy and Veanes [33] give a compact historical overview on higher-order unification problems.

11.1 Related Work

We give related work in the fields of synthetic undecidability, higher-order unification and formalisations of unification.

Synthetic Undecidability In the field of synthetic undecidability, most notably the work of Forster et al. [18, 20, 19, 21, 32] is to mention. In [21] they develop synthetic undecidability in the type theory of Coq and lay the foundations for our work. In particular, they provide a framework for establishing enumerability which we used in this work. In [32] the synthetic undecidability of Hilbert's tenth problem is proved and in [20] the synthetic undecidability of the Post correspondence problem. Thus, in the context of their work we obtain the following chain of reductions:

$$\mathsf{H} \preceq \mathsf{PCP} \preceq \mathsf{H10} \preceq \mathsf{U}_2^{\{\mathfrak{g}\}} \preceq \mathsf{U}_3 \preceq \mathsf{U}_n \preceq \mathsf{U}$$

where $3 \leq n$ and H refers to the halting problem on turing machines. This work is intended as a contribution to their library of undecidable problems formalised in the proof assistant Coq:

`https://github.com/uds-psl/coq-library-undecidability`

Higher-Order Unification The undecidability of higher-order unification was established independently by Huet [28] in 1973 and Lucchesi [34] in 1972. Both show the undecidability of third-order unification by a reduction from the Post correspondence problem. The reduction Lucchesi gives is more complicated than the one by Huet. In this work we simplified Huet's proof by reduction from the modified Post correspondence problem. For comparison, we provide a formalisation of the original proof by Huet as well.

Baxter [3] gives an undecidability proof of third-order unification where the arity of all terms is restricted to two. We conjecture that the proof we gave in Section 7.1 can be adapted to yield the same result. Instead of producing a system of equations of the shape $\lambda fa.s_1 \stackrel{?}{=} \lambda fa.t_1, \ldots, \lambda fa.s_n \stackrel{?}{=} \lambda fa.t_n$, we produce the single equation

$$\lambda fa.\mathsf{cons}\ s_1\ (\cdots(\mathsf{cons}\ s_n\ \mathsf{nil})\cdots) \stackrel{?}{=} \lambda fa.\mathsf{cons}\ t_1\ (\cdots(\mathsf{cons}\ t_n\ \mathsf{nil})\cdots)$$

where $\mathsf{cons} : \alpha \to \beta \to \beta$ and $\mathsf{nil} : \beta$ are constants. Narendran [38] shows that even monadic third-order unification, i.e. third-order unification where every term is at most of arity one is undecidable.

In 1981 Goldfarb [25] improved on the result that third-order unification is undecidable by proving the undecidability of second-order unification. Dowek [13] gives an explanation of the proof structure of Goldfarb's work and Narendran [38] gives an explanation of Goldfarb's multiplication equations in the context of Huet's higher-order unification procedure [29]. With this thesis we attempt to give an explanation of the equations outside of the context of unification in a more intuitive setting.

Levy and Veanes [33], Schubert [49], Ganzinger et al. [23], and Farmer [17] analyse undecidable fragments of second-order unification. Levy and Veanes, Schubert, and Ganzinger et al. give new reductions to establish the undecidability of second-order unification. Levy and Veanes reduce from simultaneous rigid E-unification, Schubert from the halting problem on two-counter automata, and Ganzinger et al. from rigid reachability.

From a computational perspective, our naive enumeration of high-order unification is terribly inefficient. In 1975 Huet introduced a computationally more efficient unification procedure [29]. Prior to the unification procedure of Huet, the enumerability had already been established [41, 31].

The notion of removing constants from the underlying language is already present in [56]. Working with normal substitutions without loss of generality is a well-established practice and used for example in [52, 34].

In this thesis we used named syntax on paper and a De Bruijn representation in the formalisation. In [11] and [14] formal representations of the problem are considered and related.

Formalisations Robinson was the first to give a unification algorithm for first-order unification [45]. The algorithm presented in this thesis is based on the procedure described in [51]. There are already a number of formalisations of first-order unification. We do not intend for this list to be exhaustive. Paulson [39] verifies first-order unification in the theorem prover LCF. McBride [36] verifies first-order unification in LEGO. Bove [4] verifies the problem in ALF. Coen [5] verifies it in CCL — a

logical theory developed in Isabelle. Avelar et al. [1] verify the problem in PVS. Ruiz-Reina et al. [48] verify a unification algorithm with quadratic running time in ACL2. Rouyer [47], Jaume [30], Smolka and Husson [51], and Ribeiro and Camarão [44] each verify some form of first-order unification in Coq. None of the above present unification in the setting of the λ-calculus. The version of the problem they use may be understood as unification in the λ-free fragment of our calculus where all variables are free. We are not aware of other formalisations of higher-order unification.

11.2 Future Work

While the results presented in this section have already been proved, they have yet to be formalised.

In this work we prove the decidability of first-order unification and the undecidability of second-order unification with at least one 2-ary function constant. The remaining gap — second-order unification with constants of at most arity one — is called monadic second-order unification and is decidable [16, 60].

While sufficient to establish enumerability, the naive enumeration we gave in Chapter 5 is computationally inefficient. Huet [29] proposes a procedure for higher-order unification which provides the foundation of the unification algorithms underlying modern proof assistants. However, we are not aware of a formalised correctness proof of said procedure.

While higher-order unification is undecidable in general, decidable and undecidable fragments have been found. One such decidable fragment of higher-order unification is pattern unification [37]. Pattern unification restricts the terms of the language by allowing only certain arguments of higher-order variables. Higher-order variables can only be applied to bound, distinct variables.

A different direction this work could be expanded into is the area of type systems. Dowek proves that typeability is undecidable in the $\lambda\Pi$-calculus [12] by reduction from the Post correspondence problem. Typeability, also known as type inference, asks whether a given term can be assigned a type. The proof is an adaptation of Huet's undecidability proof of third-order unification. In future work one could investigate whether the undecidability proof can be simplified by reduction from the modified Post correspondence problem. Type checking, i.e. verifying that a given term s has a given type A, is undecidable in a Curry-typed version of System F_ω. In the folklore a reduction from second-order unification to type checking in this calculus is known [59].

Bibliography

[1] Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura, and Mauricio Ayala-Rincón. First-order unification in the PVS proof assistant. *Logic Journal of the IGPL*, 22(5):758–789, 2014.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.

[3] Lewis D Baxter. The undecidability of the third order dyadic unification problem. *Information and Control*, 38(2):170–178, 1978.

[4] Ana Bove. Programming in Martin-Löf type theory: Unification-A non-trivial example. 1999.

[5] Martin David Coen. Interactive program derivation. Technical report, University of Cambridge, Computer Laboratory, 1992.

[6] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. North-Holland Publishing Company, 1972.

[7] Martin Davis. Arithmetical problems and recursively enumerable predicates 1. *The journal of symbolic logic*, 18(1):33–41, 1953.

[8] Martin Davis and Hilary Putnam. *A computational proof procedure; Axioms for number theory; Research on Hilbert's Tenth Problem*. Air Force Office of Scientific Research, Air Research and Development ..., 1959.

[9] Martin Davis, Hilary Putnam, and Julia Robinson. The decision problem for exponential Diophantine equations. *Annals of Mathematics*, pages 425–436, 1961.

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

[11] Flávio LC de Moura, Mauricio Ayala-Rincón, and Fairouz Kamareddine. Higher-order unification: A structural relation between Huet's method and the one based on explicit substitutions. *Journal of Applied Logic*, 6(1):72–108, 2008.

[12] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 139–145. Springer, 1993.

[13] Gilles Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, 2:1009–1062, 2001.

[14] Gilles Dowek, Thérese Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 366–374. IEEE, 1995.

[15] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems – lecture notes. 2018. URL `https://courses.ps.uni-saarland.de/sem_ws1718/3/Resources`.

[16] William M Farmer. A unification algorithm for second-order monadic terms. *Annals of Pure and applied Logic*, 39(2):131–174, 1988.

[17] William M Farmer. Simple second-order languages for which unification is undecidable. *Theoretical Computer Science*, 87(1):25–41, 1991.

[18] Yannick Forster and Dominique Larchey-Wendling. Towards a library of formalised undecidable problems in Coq: The undecidability of intuitionistic linear logic. In *Workshop on Syntax and Semantics of Low-level Languages, Oxford*, 2018.

[19] Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117. ACM, 2019.

[20] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. In *International Conference on Interactive Theorem Proving*, pages 253–269. Springer, 2018.

[21] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51. ACM, 2019.

[22] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. Call-by-push-value in Coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 118–131. ACM, 2019.

[23] Harald Ganzinger, Florent Jacquemard, and Margus Veanes. Rigid reachability. In *Annual Asian Computing Science Conference*, pages 4–21. Springer, 1998.

[24] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proof and types*. Cambridge University Press, 1989.

[25] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[26] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[27] Gérard Huet. Higher order unification 30 years later. In *International Conference on Theorem Proving in Higher Order Logics*, pages 3–12. Springer, 2002.

[28] Gérard P Huet. The undecidability of unification in third order logic. *Information and control*, 22(3):257–267, 1973.

[29] Gerard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[30] Mathieu Jaume. A full formalization of SLD-resolution in the Calculus of Inductive Constructions. *Journal of Automated Reasoning*, 23(3):347–371, 1999.

[31] Don C Jensen and Tomasz Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3(2):123–171, 1976.

[32] Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. Technical report, Feb 2019. URL `http://www.ps.uni-saarland.de/Publications/details/Larchey-WendlingForster:2019:H10_in_Coq.html`. To appear.

[33] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Information and Computation*, 159(1-2):125–150, 2000.

[34] CL Lucchesi. The undecidability of the unification problem for third order languages. *Report CSRR*, 2059:129–198, 1972.

[35] Yuri V. Matijasevivc. Enumerable sets are Diophantine. In *Soviet Mathematics: Doklady*, volume 11, pages 354–357, 1970.

[36] Conor McBride. First-order unification by structural recursion. *Journal of functional programming*, 13(6):1061–1075, 2003.

[37] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.

[38] Paliath Narendran. Some remarks on second order unification. 1989.

[39] Lawrence C Paulson. Verifying the unification algorithm in LCF. *Science of computer programming*, 5:143–169, 1985.

[40] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[41] Tomasz Pietrzykowski. A complete mechanization of second-order type theory. *Journal of the ACM (JACM)*, 20(2):333–364, 1973.

[42] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[43] John C Reynolds. Types, abstraction and parametric polymorphism. 1983.

[44] Rodrigo Ribeiro and Carlos Camarão. A mechanized textbook proof of a type unification algorithm. In *Brazilian Symposium on Formal Methods*, pages 127–141. Springer, 2015.

[45] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[46] Julia Robinson. Existential definability in arithmetic. *Transactions of the American Mathematical Society*, 72(3):437–449, 1952.

[47] Joseph Rouyer. *Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs*. PhD thesis, INRIA, 1992.

[48] José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso, and María-José Hidalgo. Formal correctness of a quadratic unification algorithm. *Journal of Automated Reasoning*, 37(1-2):67–92, 2006.

[49] Aleksy Schubert. Second-order unification and type inference for Church-style polymorphism. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–288. ACM, 1998.

[50] Gert Smolka. Semantics. 2018. URL `https://courses.ps.uni-saarland.de/sem_ws1718/3/Resources`.

[51] Gert Smolka and Adrien Husson. Introduction to Computational Logic. Unification, 2014. URL `https://courses.ps.uni-saarland.de/icl/2/Resources`.

[52] Wayne Snyder and Jean H Gallier. Higher order unification revisited: Complete sets of transformations. *Technical Reports (CIS)*, page 778, 1989.

[53] Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of formalized reasoning*, 2(1):41–62, 2010.

[54] Matthieu Sozeau and Cyprien Mangin. Equations reloaded. Technical report, 2018.

[55] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted De Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–180. ACM, 2019.

[56] Richard Statman. On the existence of closed terms in the typed λ calculus II: Transformations of unification problems. *Theoretical Computer Science*, 15(3): 329–338, 1981.

[57] Masako Takahashi. Parallel reductions in λ-calculus. *Journal of Symbolic Computation*, 7(2):113–123, 1989.

[58] The Coq Development Team. `coq.inria.fr`.

[59] Joe B Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.

[60] AP Zhezherun. Decidability of the unification problem for second-order languages with unary functional symbols. *Cybernetics and Systems Analysis*, 15(5): 735–741, 1979.