Mechanising Syntax with Binders in Coq

Kathrin Stark

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
an der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2019

Berichterstatter:

Prof. Dr. Gert Smolka, Universität des Saarlandes

Prof. Brigitte Pientka, PhD, McGill University

Dekan:

Univ.-Prof. Dr. Sebastian Hack

Prüfungsausschuss:

Prof. Dr. Markus Bläser (Vorsitz)

Prof. Dr. Gert Smolka

Prof. Brigitte Pientka, Ph.D.

Dr. Roland Leißa (Beisitzer)

Tag des Kolloquiums: 14. Februar 2020

Textfassung vom 11. Dezember 2019

Abstract

Mechanising binders in general-purpose proof assistants such as Coq is cumbersome and difficult. Yet binders, substitutions, and instantiation of terms with substitutions are a critical ingredient of many programming languages. Any practicable mechanisation of the meta-theory of the latter hence requires a lean formalisation of the former.

We investigate the topic from three angles: First, we realise formal systems with binders based on both pure and scoped de Bruijn algebras together with basic syntactic rewriting lemmas and automation. We automate this process in a compiler called Autosubst; our final tool supports many-sorted, variadic, and modular syntax.

Second, we justify our choice of realisation and mechanise a proof of convergence of the σ_{SP}-calculus, a calculus of explicit substitutions that is complete for equality of the de Bruijn algebra corresponding to the λ-calculus.

Third, to demonstrate the practical usefulness of our approach, we provide concise, transparent, and accessible mechanised proofs for a variety of case studies refined to de Bruijn substitutions.

Kurzzusammenfassung

Die Mechanisierung von Bindern in universellen Beweisassistenten wie Coq ist arbeitsaufwändig und schwierig. Binder, Substitutionen und die Instantiierung von Substitutionen sind jedoch kritischer Bestandteil vieler Programmiersprachen. Deshalb setzt eine praktikable Mechanisierung der Metatheorie von Programmiersprachen eine elegante Formalisierung von Bindern voraus.

Wir nähern uns dem Thema aus drei Richtungen an: Zuerst realisieren wir formale Systeme mit Bindern mit Hilfe von reinen und indizierten de Bruijn Algebren, zusammen mit grundlegenden syntaktischen Gleichungen und Automatisierung. Wir automatisieren diesen Prozess in einem Kompilierer namens Autosubst. Unser finaler Kompilierer unterstützt Sortenlogik, variadische Syntax und modulare Syntax.

Zweitens rechtfertigen wir unsere Repräsentation und mechanisieren einen Beweis der Konvergenz des σ_{SP}-Kalküls, einem Kalkül expliziter Substitutionen der bezüglich der Gleichheit der puren de Bruijn Algebra des λ-Kalküls vollständig ist.

Drittens entwickeln wir kurze, transparente und leicht zugängliche mechanisierte Beweise für diverse Fallstudien, die wir an de Bruijn Substitutionen angepasst haben. Wir weisen so die praktische Anwendbarkeit unseres Ansatzes nach.

Acknowledgements

This thesis would not have been possible without a variety of people. First, I want to thank my advisor Gert Smolka for both his invaluable advice and the great amount of freedom given in his group.

Next, I want to thank my colleagues: Christian Doczkal, Yannick Forster, Jonas Kaiser, Dominik Kirst, Moritz Lichter, Steven Schäfer, and Tobias Tebbi. Special thanks go to my office mates Jonas Kaiser and Fabian Kunze. The Programming Systems Lab would not have been the same without Ute.

Research without collaborators would be a dull business. I hence want to thank Andreas Abel, Guillaume Allais, Yannick Forster, Aliya Hameer, Jonas Kaiser, Alberto Momigliano, Brigitte Pientka, Steven Schäfer, Gert Smolka, and Simon Spies — I am grateful for everything I've learned while working with you. I particularly want to thank Jonas Kaiser and Steven Schäfer for our fruitful collaboration on Autosubst 2.

I am grateful to have been part of the engaging and thriving community of Saarland University and the Graduate School of Computer Science. I will miss the time as a tutor and, in particular, being part of the Mathematical Preparatory Course.

I want to thank Tobias Blaß, Yannick Forster, and Fabian Kunze for three new points of view on drafts of my thesis and priceless advice.

I want to thank my friends for countless evenings in the city, hiking and snow in August, shared meals, ICoffee breakfasts, (more or less voluntary) sports, girls' evenings, theatre, wine tastings – in short, all the time together: Thank you, Andreas, Caro, Clara, Fabian, Felix, Felix, Felix, Jana, Marc, Maike, Maxi, Nathalie, Nikolai, Noemi, Norine, Thomas, Tobi, and many others. Without you, Saarbrücken would have been a different town.

Finally, I want to thank my wonderful family and Tobi for their never-ending support.

Contents

Chapter 1

Introduction

Binders, substitutions, and the instantiation of terms with substitutions are a key ingredient of Church's λ-calculus [25]. In the λ-calculus, the term $(\lambda f.f\,x)\,g$ represents the application of a function $\lambda f.fx$ to its argument g and **reduces** to the term $(f\,x)[f/g]$ where each occurrence of f is **substituted** by g. Alternatively, we say that we **instantiate** $f\,x$ with the **substitution** $[f/g]$. We call "$\lambda f.$" a **binder** because f is a placeholder for the later argument. Church's λ-calculus was extremely influential and nowadays, binders and substitutions are inevitable when formalising the meta-theory of formal systems.

To convince an audience of the truth of a statement, we require an agreed-on set of assumptions and reasoning rules. An arbitrarily complex proof using only these rules is then indisputable.

Interactive proof assistants, such as Coq [111], allow and simplify the development of such proofs in an interplay between humans and computers. Collaboration with a computer has several advantages: The computer mechanically verifies that only the agreed-on rules are used, handles easy and repetitive cases on its own, and verifies that small changes in the definitions still yield a correct proof. We refer to such proofs as **mechanised**.

While they are guaranteed to be correct, mechanised proofs often involve a considerable overhead compared to traditional paper-based proofs: What is obvious for humans might be difficult for computers; and proofs are traditionally optimised for humans. If we want to close this gap, we have to catch up on the right definitions and best practices, possibly by providing tool support.

Mechanised meta-theory of formal systems with binders and substitutions is a typical example of this dilemma. On the one hand, proofs about the meta-theory of formal systems are usually technical, repetitive, and subject to frequent changes – a perfect target for mechanised proofs. However, at the same time, many practices concerning binders and substitutions that are common on paper lack a formal counterpart and hence require additional care [12].

For example, Church's proofs [25] "assume [...] an understanding of the operation of substitution". In this tradition, it is common to silently assume that **α-equivalent** terms — terms that only differ in the choice of bound names like $\lambda x.x$ and $\lambda y.y$ — are equal or to assume that the names of bound and free variables are always chosen from different sets, known as the Barendregt convention [14]. Both assumptions have to be made explicitly in mechanised proofs. Many proofs for formal systems are therefore difficult to mechanise, even though the paper-based proof requires only basic knowledge about binders. In fact, there is no consensus in the community on the best way to handle binders — even though the POPLMark challenge [12], which evaluated the state of the art of mechanised meta-theory, has received much attention and was posed almost 15 years ago.

In this thesis, our goal is to enable users to mechanise meta-theoretic proofs for formal systems with binders. Following Church's assumption, the user requires only a basic understanding of binders, substitutions, and instantiation. In particular, we want to unburden the user from understanding the implementation details of instantiation and from *manually* reasoning about syntax.

In practical terms, this imposes three requirements: First, we require a representation of binders, substitutions, and α-equivalence. Second, we need a definition of instantiation of terms with substitutions. Finally, we require a set of basic syntactic lemmas about instantiation and a **proof method** on how to handle **substitution equations**, i.e. equations between expressions with syntactic terms and instantiations.

Independent of the chosen approach, the realisation of a custom formal system is complex and requires tedious, technical, and repetitive code, in short: **substitution boilerplate**. This is a well-known problem in the community. For example, in their paper on a translation of ML modules into Fω using locally nameless syntax (LN) [10], Rossberg et al. [95] remark:

> "Our experience [...] was more painful than we had anticipated. Compared to the sample LN developments, ours was different in making use of various forms of derived n-ary (as well as basic unary binders) and in dealing with a larger number of syntactic categories. Out of a total of around 550 lemmas, approximately 400 were tedious "infrastructure" lemmas; only the remainder had direct relevance to the metatheory of Fω or elaboration. The number of required infrastructure lemmas appears to be quadratic in the number of variable classes[...], the number of "substitution" operations needed per class [...] and the arity (unary and n-ary) of binding constructs. So we cannot, hand-on-heart, recommend the vanilla LN style for anything but small, kernel language developments."

Even worse, the realisation for one formal system alone, e.g. Fω, is but a drop in the ocean: every new formal system requires a new realisation and hence involves new substitution boilerplate.

Apart from the substitution boilerplate, developing a practical realisation for non-standard formal systems is already a challenge. The design space for realisations that provide binders, substitutions, and instantiation is huge, yet only few choices work well in actual proofs.

Schäfer et al. [100] address this problem with Autosubst 1, a library which automatically generates a practical realisation for a restricted class of formal systems. Their approach is based on pure **de Bruijn algebras** and basic syntactic equational lemmas adapted from the σ_{SP}-calculus [31]. As these form a complete rewriting system for the de Bruijn algebra of the λ-calculus [99], many substitution lemmas can be solved via rewriting alone.

While Autosubst is great in what it does, it only supports a very restricted set of syntax. These restrictions are inherent to its implementation, and overcoming them requires a complete redesign. This prompted our design of Autosubst 2, or from now on only Autosubst. Autosubst inherits the focus on de Bruijn substitutions from its predecessor, but expands the range of supported syntax to scoped syntax, mutual and many-sorted syntax, functors, variadic syntax, and modular syntax. Each extension requires careful design decisions and new best practices. Autosubst hence subsumes Autosubst 1.

We approach the topic from three angles:

First, we justify the use of rewriting for the special case of the λ-calculus. To argue over substitutions, we use the σ_{SP}-calculus [56], a calculus of explicit substitutions whose convergent reduction was previously shown to be sound and complete for the pure de Bruijn algebra [99]. To show that rewriting is an adequate proof method, we extend this result by a mechanised proof that the σ_{SP}-calculus is confluent and terminating, simplifying a previous paper-based proof [30].

Second, we develop best practices for and automatically realise formal systems with binders. We present our solution to the substitution boilerplate problem, the Autosubst compiler. Autosubst takes a specification in a variant of higher-order abstract syntax and generates the corresponding de Bruijn algebra, syntactic substitution lemmas, and custom automation as human-readable Coq code. Autosubst supports scoped syntax, many-sorted syntax, first-class renamings, polyadic binders, external type constructors, variadic binders, and modular syntax.

Third, we demonstrate the practicality of our approach in a variety of mechanised case studies refined to de Bruijn substitutions. We provide concise, transparent, and accessible proofs of normalisation for the simply-typed λ-calculus, type safety of the variadic λ-calculus, modular proofs of weak head normalisation and strong normalisation of a λ-calculus with boolean and arithmetic expressions, as well as type safety for System F with records and subtyping (widely known as the POPLMark challenge [12]). All developments are concise, thanks to Autosubst's automatic boilerplate generation.

1.1 Related Work

In this section, we start with an overview on de Bruijn algebras as the basis for reasoning on syntax, continue with explicit calculi of substitutions which inspire the equational theory implemented, discuss possibilities to avoid the substitution boilerplate problem in a generic-purpose proof assistant, and finally turn to the mechanisation of meta-theory. Each part divides into an overview of the related work and the significance of this work for this thesis.

1.1.1 Binders and De Bruijn Algebra

So far, there is no consensus in the community on the best way to represent binders: There are approaches using unnamed syntax à la de Bruijn [33], named syntax [14, 25], locally nameless syntax [10, 53, 74], higher-order abstract syntax [83], and nominal syntax [87]. Even special-purpose proof assistants (e.g. Beluga [85], Twelf [84], and Abella [48]) have been developed to address the difficulties in reasoning with binders.

In this thesis, we opt for **de Bruijn indices** [33], also known as unnamed syntax: we avoid names altogether and instead refer to a variable by the distance to its enclosing binder. For example, $\lambda x.x\,(\lambda y.y\,x)$ is represented by $\lambda.0\,(\lambda.0\,1)$.

De Bruijn syntax has a variety of advantages: it directly implements α-equivalence, is implementable in a general-purpose type theory,[1] and can be generalised to a variety of different systems.

One difficulty with de Bruijn syntax is that if a substitution is propagated under a binder, all free variables change their meaning. We hence define instantiation with substitutions which replace all indices at once, also known as parallel substitutions or **de Bruijn substitutions**. Following an idea of Schäfer et al. [100], we use the **substitution primitives** of the σ-calculus [2], a calculus of explicit substitutions, as atomic substitution operations. These are expressive enough to define β- and η-reduction and can express the scope change under a binder. Together, terms in the λ-calculus and these substitution primitives form a two-sorted algebra, the corresponding **de Bruijn algebra** [99].

While de Bruijn syntax is in principle very well understood, it also contains very technical lemmas and requires carefully chosen definitions to avoid unmanageable overhead in later proofs. In general, all proofs using de Bruijn syntax include manual and technical statements about substitutions; and hence without automation require deep knowledge on the representation via de Bruijn syntax. Schäfer et al. [99] propose a way to solve these technical statements automatically: They provide a set of basic syntactic equation lemmas on de Bruijn algebra, inspired by reduction in the σ_{SP}-calculus [31], a calculus of explicit substitutions: These include preservation of identity, compositionality, and various interaction lemmas between the atomic primitives. Together these form

[1] In this thesis we use the Coq proof assistant, but the practices developed in this thesis extend to any other proof assistant supporting inductive types.

a complete and convergent rewriting system for the de Bruijn algebra. We know of no similar completeness results for any other binder representation.

Moreover, stating the lemmas correctly and getting all the technical detail right can be simplified using **scoped de Bruijn syntax** [18] which adopts the idea of unnamed references, but additionally carries an upper bound of free variables, called a **scope**. As a consequence, shifting errors are detected by type errors, see also [97] for an introduction. We see scoped syntax as closer to the ideal representation of syntax.

Significance for this thesis. Binders have been extended in various ways, see e.g. [23]. We show that the strategy to get automated equations and to avoid substitution boilerplate can be extended to more formal systems.

To preserve practicality, we have to be conservative and restrict ourselves to carefully selected design principles. We extend de Bruijn substitutions and in particular its reasoning to the following formal systems: **first-class renamings**, used in proofs with Kripke-style logical relations [77]; **polyadic binders** such as in a λ-calculus with pairs; **external sorts and sort constructors**, needed for record types; **many-sorted** and mutually inductive syntax in System F [50, 91]; **first-order binders** in first-order logic and the π-calculus [76]; **variadic binders**, used in a multivariate λ-calculus [90] and during pattern matching; and finally, **modular syntax** which is frequently used in textbook representations [86].

Three of these extensions deserve to be mentioned specifically: First, unlike previous approaches, we handle **many-sorted syntax** with **vector substitutions**, which parallelise the already parallel de Bruijn substitutions to a vector of substitutions. This results in a straightforward definition of instantiation, and also an equational theory similar to the one of the λ-calculus.

Second, we offer support for **variadic syntax** by introducing variadic substitution primitives which extend the primitives of the σ-calculus. We know of no other tool which supports automated proofs for variadic binders in a proof assistant. Among others, variadic binders are powerful enough to express reduction of patterns.

Third and finally, we introduce support for **modular syntax**. Although modular syntax is commonly used in paper-based proofs [86], we know of no other practical solution for proof assistants.

1.1.2 Calculi of Explicit Substitutions

Calculi of explicit substitutions [2, 31, 38] were introduced as an intermediate layer between the theory and implementation of the λ-calculus. They distinguish themselves from the λ-calculus by a syntactic representation of substitutions defined mutually inductive with terms. This allows us to examine the exact behaviour of substitutions [99].

There are various calculi of explicit substitutions for the untyped λ-calculus alone: The

σ-calculus as introduced by Abadi et al. [2] with identity substitution, extension, shifting, and composition as substitution primitives; the $\sigma_{\Uparrow}$-calculus, which first appeared in in Hardin and Lévy [56], is now available in [96], and adds a primitive for lifting; and last, the σ_{SP}-calculus, which contains additional reductions to make reduction in the σ-calculus confluent. Only the σ_{SP}-calculus is sound and complete for equality on the de Bruijn algebra corresponding to the λ-calculus [99].

Completeness implies that all equations between expressions can be proved using reduction in the σ_{SP}-calculus, given that reduction is convergent. Autosubst 1 uses the connection between the de Bruijn algebra corresponding to the λ-calculus and the σ_{SP}-calculus to define automation for substitution and generalise the σ_{SP}-calculus to custom syntax [100]. The σ_{SP}-calculus by Schäfer et al. [99] is proven to be sound and complete for the pure de Bruijn algebra of the λ-calculus in Coq.

Significance for this thesis. Convergence is necessary to justify the rewriting method [13], but Schäfer et al. [99] declare the formalisation and mechanisation of a paper-based proof of termination [30] to be future work. In this thesis, we accomplish this obligation and formalise and mechanise a proof of convergence for the rewriting system in use.

The proof of local confluence is straightforward. Compared to Saïbi [96] we omit the detour over a critical pair analysis [13, Chapter 6][59, 69], but simply use Coq's backtracking mechanism to explore the different derivations of reduction.

Termination for related calculi has been proven several times [30, 55, 121]. Kamareddine and Qiao [65] further provide a mechanised proof of termination for the original σ-calculus (and their own calculus of explicit substitutions) in ALF [72] strictly following Curien et al. [30]. Unfortunately, we were not able to recover the mechanisation. In this thesis, we give a mechanisation which simplifies the previous proofs, also based on [30].

1.1.3 Compiling Syntactic Specifications

Recall that in general-purpose proof assistants, the realisation of a syntactic system is complex and requires tedious boilerplate. Although it is technically possible to re-define each new syntactic system manually, this is only applicable if one understands all subtleties of the approach deeply: recall that small changes in the definition can have a big impact on the proofs. At the same time, syntactic systems resemble each other which brings up the question whether we can avoid repetitive, technical substitution boilerplate and realise the knowledge gained in mechanising formal systems in a more general fashion.

There are two fundamentally different ways how to handle this boilerplate in a general-purpose proof assistant: First, we can define a static type of formal systems, use signatures to describe a specific formal system, and then use generic programming to define

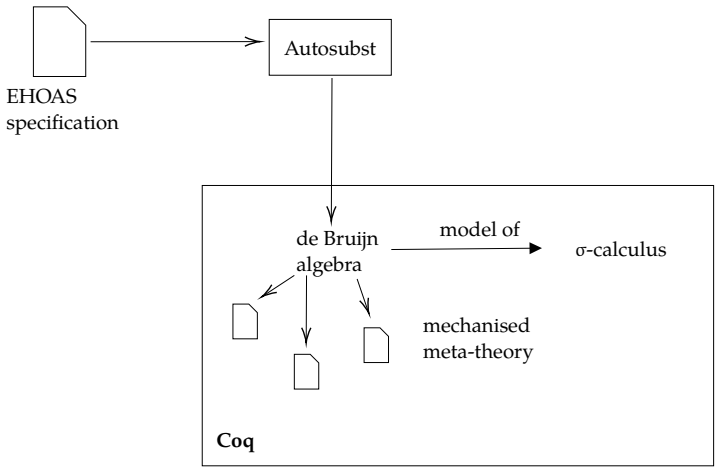

Figure 1.1: Autosubst design.

instantiation generically (see e.g. [7, 49]). Alternatively, we can generate the necessary structure via compilation, either internally with Ltac [36] or MetaCoq [106], or externally via a compiler [11, 67, 102].

Both approaches have different trade-offs: While an algebra-like approach is undoubtedly more elegant, it also introduces a level of indirectness caused by the intermediate presentation. We hence follow the approach by compilation.

Previous compilers for syntax with binders include Ott [102], LNGen [11], Needle&Knot [67, 68], and Autosubst 1 [100]. Ott generates code for a wealth of definitions in either LaTeX or a proof assistant of choice, e.g. Coq. However, except for statements on well-formedness, Ott provides no proofs. LNGen is based on Ott and further integrates definitions and lemmas for locally nameless syntax. Needle&Knot generates custom de Bruijn syntax and single-point syntax using a custom compiler in Haskell. Autosubst 1 uses the Ltac language with pure de Bruijn substitutions for single-sorted syntax, and heterogeneous substitutions for (non-mutual) many-sorted syntax. It further offers automation for substitution equations based on the σ_{SP}-calculus.

Significance for this thesis. The Autosubst compiler is realised as an external tool; its design is depicted in Figure 1.1.[2] Its input is a specification in a HOAS-like language [83] which offers the possibility to represent binders naturally with negative positions corresponding to binders. For example, the λ-calculus is specified as:

```
tm : sort
abs : (tm → tm) → tm
app : tm → tm → tm
```

[2]Note that Autosubst 1 and Autosubst have no common code base.

The output is the corresponding de Bruijn algebra (i.e. the definition of the corresponding inductive type, and instantiation of renamings and substitutions via the primitives of the σ-calculus), together with syntactic rewrite lemmas, suitable notations, and custom tactics. We output either pure or scoped Coq code.

1.1.4 Mechanised Meta-Theory

Mechanised meta-theory of formal systems has several advantages over paper-based proofs: It eliminates mistakes, changes definitions, and automates tedious proofs [4, 12].

Binders and substitutions are often used when establishing **global properties** of a formal system. While properties such as confluence and Church-Rosser properties are in large parts independent of binders, substitutions are key when establishing proofs in a syntactic way, e.g. during type safety à la Wright and Felleisen [120] (stating that typed programs do not get stuck) and normalisation (stating that typed programs either can reduce to or always reduce to a normal form).

De Bruijn Syntax requires slightly different statements than named syntax. A central notion are context renaming and context morphism lemmas [52, 63] which allow us to prove substitutivity of objects with contexts, i.e. during typing or subtyping.

There are two benchmarks which have influenced this thesis: The already-mentioned POPLMark challenge [12] and the POPLMark Reloaded challenge [3]. Both challenges give us the possibility to evaluate the Autosubst compiler against other tools.

The POPLMark challenge proposes to show type safety of System F with subtyping and proposes four key areas: binders, induction principles, experimentation, and component reuse. In this thesis, we focus on binders and component reuse only. Three of the four solutions to the full POPLMark challenge use de Bruijn syntax. All are shorter than the HOAS solution in Twelf with 4500 lines, with line counts ranging between 830 (for Needle&Knot) and 2500. Still, except for one solution, all solutions require users to generate manual substitution boilerplate, and even the one exception applies substitution lemmas manually. Save one exception, all solutions further use adapted definitions and define pattern matching to yield the result of an instantiation and not the instantiation itself; the solutions hence avoid variadic substitution altogether. In this thesis, our ambition is to show how to reason automatically with variadic syntax in a more direct manner.

The POPLMark Reloaded challenge [3, 4] extends the POPLMark challenge with proofs with Kripke-logical relations [77] of strong normalisation for the simply-typed λ-calculus. It poses several new challenges concerning anti-renaming lemmas and intrinsically typed syntax.

Both challenges highlight the need for modular solutions and the need for component reuse. Still, we are not aware of any solution to either challenge that does not have to

copy-paste definitions and proofs. Approaches to truly modular syntax either build in a layer of indirectness or require dramatically changed proofs [19, 37, 66, 79, 101].

Significance for this thesis. To evaluate the Autosubst compiler, we need both suitable evaluation criteria, and a suitable set of proofs. In an ideal case, this set contains standard proofs done with other approaches as well.

For suitable **evaluation criteria**, we follow the POPLMark challenge [12], which suggests three criteria to evaluate a formal development: *conciseness*, the cost of formalisation compared to a proof by hand; *transparency*, the deviation of a formalisation compared to a proof by hand; and *accessibility*, the entry cost of a formalisation.

For the proofs, we choose a mixture of substitution-heavy proofs and benchmark problems. The main focus is on the **substitutivity** of different type-theoretic constructions, i.e. how instantiation behaves on functions, inductive predicates, and inductive types. Our proofs include substitutivity of reduction, preservation, weak head normalisation, strong normalisation with Kripke-style logical relations [77] in two styles, decidability of declarative equivalence, and modular proofs of strong normalisation.

During the development of the POPLMark Reloaded challenge, the author of this thesis has developed the formal proofs for de Bruijn syntax and de Bruijn substitutions and helped to formalise soundness of Raamsdonk's characterisation of strong normalisation to the usual definition of strong normalisation.

In this thesis, we moreover present a practicable approach for modular syntax based on injections and fixpoints of functors. In a sense, this is an adaption of Swierstra's Data Types à la Carte [109] to a proof assistant restricted to positive types. This approach scales to modular proofs of strong normalisation for a λ-calculus with boolean and arithmetic expressions. Our compiler hence also offers support for modular syntax.

1.2 Overview

Except for the preliminaries on the Coq proof assistant in Chapter 2, this thesis is split into three parts: In the first part, we describe the theory behind Autosubst with formalised and mechanised results on the pure σ_{SP}-calculus (Chapters 3 – 4). In the second part, we then develop best practices and equational theories for a variety of formal systems. We integrate these into the Autosubst compiler which translates from an HOAS-like specification to a type-theoretic interpretation (Chapters 5 –8). Finally, we present a range of case studies adapted to de Bruijn substitutions (Chapters 9 – 11).

Part 1. In Chapter 3 we recall pure and scoped de Bruijn algebras on the example of the λ-calculus. This yields the base for a first discussion on the reasoning principles on syntax and the necessary substitution boilerplate.

For the particular case of the λ-calculus, we justify our choice: In Chapter 4 we recall

the pure σ_{SP}-calculus and that it is a sound and complete model for equality on the de Bruijn algebra of the λ-calculus (Schäfer et al. [99]). We complement these results with a mechanised proof of confluence and termination of the σ_{SP}-calculus, based on and simplifying a paper-based proof by Curien et al. [30].

Part 2. In this chapter, we describe the developed best practices for syntax with binders.

In Chapter 5, we present the specification language, a form of higher-order abstract syntax called EHOAS.

In Chapter 6, we show how to construct de Bruijn algebra together with the corresponding equational rewriting lemmas for syntax with first-class renamings, polyadic syntax, many-sorted syntax, syntax with first-order binders, syntax with functors, and variadic syntax. Each extension was carefully designed to satisfy the correct equational properties.

In Chapter 7, we introduce a way to handle modular syntax in Coq, based on injections and functors. We generate modular de Bruijn algebra and show how to automate this support.

In Chapter 8, we introduce a compiler called Autosubst, a generic way to define the corresponding de Bruijn algebra for a custom specification.

Part 3. Last, we evaluate our approach empirically. We present a set of substitution-heavy proofs and benchmarks, adapted to de Bruijn substitutions.

In Chapter 9 we start with simple proofs for substitutivity of reduction and type preservation for both the monadic and variadic λ-calculus. We then continue with weak head normalisation and two variants of strong normalisation proofs: one via Schäfer's expression relation, and one via Raamsdonk's characterisation of strong normalisation. We extend these proofs in a modular way to a λ-calculus with boolean and arithmetic expressions. We further give a mechanised proof that definitional equivalence and algorithmic equivalence are equivalent.

In Chapter 10 we present a concise, transparent, and accessible proof of type soundness of System F with subtyping and records, also known as the POPLMark challenge. For patterns, this proof assumes the existence of pattern typing and pattern matching, two substitution-independent definitions. These proofs require many-sorted syntax, functors, and variadic binders.

In Chapter 11 we give a concise overview of other developments using Autosubst, including a summary of a development of large parts of the operational and denotational semantics of call-by-push-value [71].

1.3 Supporting Publications

This thesis is based on the following previous publications:

[62] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '17, pages 10–14. ACM, 2017

[64] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de Bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 293–306, 2018

[98] Steven Schäfer and Kathrin Stark. Embedding higher-order abstract syntax in type theory. In *Abstract for Types Workshop*, June 18 – 21 2018

[108] 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, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180, 2019

[45] 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, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 118–131, 2019

[4] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark Reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29:e19, 2019

[43] Yannick Forster and Kathrin Stark. Coq à la carte - a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, USA, January 20–21, 2020*, January 2020. To appear

The central publications concerning this thesis are [62] and [108]. In [62], we sketch the redesign of Autosubst implemented as a one-phase compiler and introduce vector substitutions with a sketch of the equational theory online. The paper also introduces the idea to use higher-order abstract syntax as a specification language. In [108], we extend the previous results with an intermediate representation of abstract proof terms, which allows for scoped syntax [18] and proof assistants other than Coq. Section 6.4 is based on and reuses parts of the respective descriptions in [62, 108]. The motivation for first-class renamings in Section 6.7.1, the motivation for vector substitutions in Section 6.7.3, and Chapter 8 reuses large parts of [108]. Part 1 of the POPLMark challenge was first

presented in [108] and we reuse parts of this description in Section 10.1 and the evaluation in Section 10.4.1. The case study of algebraic equivalence was first mentioned in [108].

Another central publication is [43], in which we introduce modular syntax. The example specification for modular syntax in Chapter 5 is adapted from this publication, Chapter 7 and Section 8.4 largely reuse parts of the corresponding description in [43]. The case study on a modular proof of strong normalisation reuses the one of [43] and was developed by the author of this thesis.

The description of strong normalisation via Raamsdonk's characterisation in Section 9.6 and the adaption to de Bruijn substitutions is based on the respective description in the POPLMark Reloaded challenge originally posed by Abel, Momigliano, and Pientka [3]. Abel et al. [4] then extend the description and compare three solutions, of which the author of this thesis has contributed one. In contrast to the original description, in this thesis, we use a scoped (instead of intrinsically typed) representation.

Published results not appearing in the thesis. There are further published results which are not directly part of the thesis, but have influenced the design of Autosubst:

In [45], we formalise large parts of the meta-theory of call-by-push-value (CBPV). As CBPV is a subsuming paradigm for both call-by-name and call-by-value, many meta-theoretical results (weak and strong normalisation, confluence, a denotational semantics) can be transported to call-by-name and call-by-value calculi. This development was well-suited to test Autosubst's practicality for mutual inductive syntax and the interplay of different formal systems. It further shows that Autosubst suffices for many more areas of mechanising meta-theory. This extension is not yet part of Autosubst.

Coq's generated induction and recursion principles give the first-order reasoning principle only. However, the binding discipline is implicit in the first-order syntax. Our approach to binder-aware recursion in [64] is based on work of Allais et al. [7]. We extend this solution to mutual inductive sorts and to not only state but also *reason* about traversals. We introduce the notion of lifting, which simplifies and generalises previous proofs.

In a following abstract [64], we give a sketch how to support full higher-order abstract syntax [83] with substitutions. If implemented, this would provide, for example, context renaming and context morphisms for type systems.

Unpublished results. The mechanised proof of convergence is not published yet.

Since the above publications, EHOAS we extended with support for functors and variadic syntax. No previous publication introduced EHOAS in this grade of detail. The description of the dependency analysis and the generation of abstract proof terms has been thoroughly extended; it further covers the handling of the new syntactic systems.

Regarding the compiler, the published version of Autosubst [108] already handles polyadic syntax and first-class renamings, but we never explain its generation in detail. The handling of first-class renamings, functors, and variadic syntax are newly implemented in this thesis.

For modular syntax, the example was changed to the instantiation with renamings and substitutivity. The section describing the modular de Bruijn algebra corresponding to a λ-calculus with boolean and arithmetic expressions is entirely new.

Part B of the POPLMark challenge is developed newly for this thesis.

1.4 Contributions

The main contributions of this thesis are as follows:

- We give a formalised and mechanised proof that reduction in the σ_{SP}-calculus is terminating and confluent, following and simplifying a proof by Curien et al. [30]. We hence justify the use of the rewriting method in Autosubst.
- We introduce EHOAS, a natural specification language for syntax with binders.
- We present Autosubst, a compiler for specifications of syntactic systems which supports modular de Bruijn syntax in the Coq proof assistant – both for pure and scoped de Bruijn syntax.
- We show how to handle polyadic syntax and first-class renamings in a de Bruijn algebra.
- We present vector substitutions, a new way to reason about many-sorted syntax with various sorts of variables in a principled way.
- We give and automatically generate simplified de Bruijn algebras for syntax with first-order binders only.
- We introduce new primitives and reasoning principles for de Bruijn substitutions with variadic binders.
- We show how to combine Autosubst and modular syntax with a variant of Swierstra's Datatypes à la Carte approach [109]. Our approach works in a type theory requiring inductive data types to be strictly positive.
- Several Coq developments demonstrate the usefulness of our tool.

  Among others, we present concise, transparent, and accessible proofs of strong normalisation for the simply-typed λ-calculus (one via a variant of Girard et al. [51], one via Kripke-style logical relations [4] and via an intermediate inductive predicate following van Raamsdonk et al. [115]).

- We give the first truly modular proof of strong normalisation for a λ-calculus with boolean and arithmetic expressions.
- Under the assumption of pattern typing and pattern matching with suitable properties, we give a solution to the POPLMark challenge, which is to show type safety for System F with subtyping, records, and pattern matching. To our best knowledge, this is the first solution with de Bruijn substitutions.

1.4.1 Mechanisation in Coq

Unless stated otherwise, all statements are formalised in the Coq proof assistant [111].

Whenever applicable, the lemmas in the thesis are linked with the Coq code. The accompanying development is available online:

www.ps.uni-saarland.de/~kstark/thesis.

The Autosubst compiler can be found online at the same address:

www.ps.uni-saarland.de/~kstark/thesis.

Chapter 2

Preliminaries

We begin with a short description of the underlying type theory for the remainder of the thesis, the polymorphic cumulative Calculus of Inductive Constructions [26, 27, 82, 113] and the Coq proof assistant [111]. Certain axioms simplify our constructions, discussed in Section 2.2.

2.1 The Coq Proof Assistant

The Coq proof assistant [111] is our tool of choice to reason in this thesis. Coq is a mature and actively developed proof assistant with a solid user base.

Coq implements an intensional type theory with (mutual) inductive data types [27, 40, 82], dependent types, an infinite, cumulative hierarchy of types, Type, and an impredicative universe of propositions, Prop. It thus provides all building blocks for pure and scoped de Bruijn syntax. Its usage as a proof assistant is justified by the Curry-Howard Correspondence [58, 117], a deep connection between logic and programming languages: propositions correspond to types, proofs correspond to programs, and proof normalisation corresponds to the evaluation of programs. As a consequence, proof checking is reduced to type checking.

Coq supports a wide range of features to simplify interactive theorem proving that we use throughout this thesis: Notations, type classes [105], rewriting, and hint databases. Most importantly, Coq supports custom tactics [36]. Tactics dramatically simplify repetitive proofs, which are common in the mechanisation of programming languages.

Functions and their composition will play an important role. We write $\sigma \equiv \tau$ and say that σ and τ are **equivalent** if σ and τ are point-wise equal, i.e. $\forall x.\sigma\, x = \tau\, x$. Throughout this thesis, we write $\_ \circ \_ : (A \to B) \to (B \to C) \to (A \to C)$ to denote **forward composition** of functions, different to common use.

Similar to custom data types in programming languages, Coq allows **inductive types** for creating new types. For example, the inductive type of **natural numbers** describes

the smallest type which can be constructed with the following two rules:

$$n \in \mathbb{N} := 0 \mid 1 + n$$

indicating that a natural number can be either zero or the successor of another natural number. Coq provides elimination and induction principles to define functions and proofs over inductive types.

Coq supports the usual logical connectives via inductive predicates: Truth $\top$, falsity $\bot$, disjunction $X \vee Y$, conjunction $X \wedge Y$, and existential quantifiers $\exists x.X$. Note that universal quantification is an internal definition.

An **option** over a type X, written $\mathcal{O}(X)$, represents an optional value of type X, with either the empty option $\emptyset$ or $\lfloor x \rfloor$, the option with an element x of the original type X.

Coq also allows **type constructors**. For example, we represent **lists** $\mathcal{L} : \mathsf{Type} \to \mathsf{Type}$ as the smallest type satisfying:

$$\frac{}{\mathsf{nil} : \mathcal{L}(X)} \qquad \frac{x : X \quad xs : \mathcal{L}(X)}{x \cdot xs : \mathcal{L}(X)}$$

We can write structurally **recursive functions** over an inductive type. For example, we write a function $xs_n : \mathbb{N} \to \mathcal{L}(X) \to \mathcal{O}(X)$ that yields the nth element of a list, if available:

```
Fixpoint nth (n : ℕ) (xs : ℒ x) :=
  match n, xs with
  | 0, x · xs ⇒ ⌊x⌋
  | 1 + n, x · xs ⇒ nth n xs
  | _ ⇒ ∅
  end.
```

Note that as an over-approximation all functions must be structurally recursive to ensure consistency of the logic. This restriction will influence our definition of instantiation.

Coq allows dependent types, i.e. types indexed by values. Further, inductive types can be indexed by the value of another type, i.e. Coq allows families of inductive types. For example we can index lists with their lengths, i.e. define **vectors** $\mathbb{V}_n\,(X)$:

$$\frac{}{\mathsf{nil} : \mathbb{V}_0\,(X)} \qquad \frac{x : X \quad xs : \mathbb{V}_n\,(X)}{x \cdot xs : \mathbb{V}_{1+n}\,(X)}$$

Sets. We represent sets $\{x : X \mid P\,x\}$ as predicates $P : X \to \mathsf{Prop}$. This representation allows a direct definition of the set-theoretic connectives.

2.2 Axioms

Because Coq is a constructive type theory, certain statements assumed in classical mathematics are not provable, i.e. the **axiom of excluded middle**, $X \vee \neg X$. However, these

statements can be assumed without compromising the consistency of the logic. We outline the assumptions used throughout the thesis.

Functional Extensionality states that two functions $f, g : X \to Y$ are equal if and only if they coincide at all arguments, i.e. $\forall x. f\, x = g\, x$.

Functional extensionality does not hold in Coq. However, functional extensionality does not yield additional proof power, i.e. is conservative over a version without functional extensionality [57]. We assume functional extensionality in our later rewrite tactic (see Chapter 8).

Part I

De Bruijn Syntax and Sigma Calculi

Chapter 3

Lambda Calculus with de Bruijn Syntax

In this chapter, we recall how to realise the λ-calculus with de Bruijn syntax [33]. Most interesting is instantiation with substitutions and the syntactic rewriting lemmas on this representation. Besides an introduction to the handling of de Bruijn syntax, this chapter shows the substitution boilerplate a practical realisation of a formal system with binders requires in a general-purpose proof assistant. All lemmas proven in this chapter are also generated automatically by Autosubst.

Let us start with a named representation of the λ-calculus due to Church [25]:

$$s, t \in \mathsf{tm} \;:=\; \mathsf{var}\, x \mid \mathsf{app}\, s\, t \mid \lambda x.s$$

The λ-calculus consists of **variables** $\mathsf{var}\, x$ of an infinite namespace, **functions** or **abstractions** $\lambda x.s$, and **applications** $\mathsf{app}\, s\, t$. Abstractions **bind** a new variable x and proceed with a **function body** s in which this new variable might appear.

We usually want to identify terms which only differ in the chosen names of bound variables, i.e. are **α-equivalent**:

$$\lambda x.x\,(\lambda y.y\,x) =_\alpha \lambda y.y\,(\lambda z.z\,y)$$

This is crucial for the central notion of **instantiation** s_t^x, which states that each occurrence of the variable x in s is replaced by the term t. For the intended meaning, we have to avoid the **capture** of variables. This might require to rename variables, and hence implicitly uses α-equivalence. For example,

$$(\mathsf{app}\,(\lambda z.z\,x)\,(\lambda x.x))^x_{(\lambda x.z\,x)} = \mathsf{app}\,(\lambda y.y\,(\lambda x.z\,x))\,(\lambda x.x).$$

Instantiation is used in the central definition of reduction, called β-reduction:

$$\mathsf{app}\,(\lambda x.s)\,t \succ s_t^x$$

In Church's representation, he assumes a capture-avoiding definition of instantiation to be given, together with certain properties. In a proof assistant, we have to define these for a self-contained representation.

Unnamed syntax, introduced by de Bruijn [33], solves this problem with a simple idea: Instead of explicit names, variables are represented by natural numbers $n : \mathbb{N}$ and refer to the nth enclosing binder. Hence, α-equivalence collapses to syntactic equality: Both $\lambda x.x\,(\lambda y.y\,x)$ and $\lambda y.y\,(\lambda z.z\,y)$ are represented by $\lambda.0\,(\lambda.0\,1)$.

The design space to define instantiation is big, and the exact definition has an enormous impact on the practicality of the approach. In this thesis, we follow de Bruijn [33] and use **de Bruijn substitutions** σ, also called parallel substitutions, that replace all variables at once and hence are represented by functions $\sigma : \mathbb{N} \to \mathsf{tm}$. The exact definition uses the primitives of the σ-calculus [2] due to an idea of Schäfer et al. [99]. Together, terms and substitutions of the λ-calculus form a two-sorted algebra, called the **de Bruijn algebra** [99].

As Coq is restricted to structural recursion, we follow Adams [5] and first define instantiation with **renamings**, i.e. substitutions which only rename variables.

In the next step, we recall the syntactic rewriting lemmas to reason on the de Bruijn algebra of the λ-calculus. These include that identity substitutions preserve identity, that instantiation is composable, and that instantiation is extensional. As an example, we show that β-reduction is substitutive.

In the second part of this chapter, we change to a **scoped** variant of de Bruijn syntax due to Bird and Paterson [18] and previously used with the primitives of the σ-calculus in [97]. Intuitively, each term is indexed by its scope. Scoped syntax simplifies the usage of de Bruijn syntax, since it yields an additional layer of safety: As all terms are indexed by their scope, missing shiftings can be detected by Coq's type checker.

While the type of a term changes to a type family, i.e. $\mathsf{tm} : \mathbb{N} \to \mathsf{Type}$, almost everything else remains unchanged: We use elements of the canonical finite type instead of natural numbers, and a scoped counterpart of all primitives. Similar to the pure case, we define a *scoped de Bruijn algebra* corresponding to the scoped λ-calculus.

3.1 Pure de Bruijn Algebra

In **de Bruijn syntax** [33], variables are represented by natural numbers:

$$s, t \in \mathsf{tm} \;:=\; \mathsf{var}\, x \mid \mathsf{app}\, s\, t \mid \lambda.s \qquad x \in \mathbb{N}$$

A variable x implicitly refers to the xth enclosing binder. Each abstraction $\lambda.s$ implicitly binds a new term and changes previous references, i.e. **free variables** in $\lambda.s$ and s have a different interpretation:

$$\begin{array}{cccccc} s & & 0 & 1 & 2 & \\ & & | & | & | & \dots \\ \lambda.s & 0 & 1 & 2 & 3 & \end{array}$$

While in $\lambda.s$, 0 refers to the binder of the new abstraction, $1+n$ refers to the free variable which is referred to as n in only s. We use the point after a binder, e.g. $\lambda.s$ as a universal notation to highlight a **scope change**. Here are some examples of terms in de Bruijn syntax:

$$\lambda x.\lambda y.y\,(\lambda x.x) \sim \lambda.\lambda.0\,(\lambda.0)$$
$$\lambda x.z\,x \sim \lambda.(1+z)\,0$$

Note that in the second term, we had to lift z by 1 to avoid capture by the new binder.

The **scope change** of a binder such as abstraction will be the main challenge in the remaining chapter. Note that it affects all variables in the body of the function, and hence multiple variables in parallel.

3.1.1 Instantiation

Let us recall the example of the introduction,

$$(\mathsf{app}\,(\lambda\,z.z\,x)\,(\lambda\,x.x))^{x}_{(\lambda x.z\,x)} = \mathsf{app}\,(\lambda\,y.y\,(\lambda x.z\,x))\,(\lambda\,x.x)$$

with arbitrary free variables x and z. In de Bruijn syntax, we hence have:

$$(\mathsf{app}\,(\lambda.0\,(1+x))\,(\lambda.0))^{x}_{(\lambda.z\,0)} \overset{!}{=} \mathsf{app}\,(\lambda.0\,(\lambda.(2+z)\,0))\,(\lambda 0.0)$$

All free variables in the replaced term have to be increased each time we traverse a binder: we have to rename all variables at once. Moreover, previously bound terms have to remain unchanged.

For de Bruijn syntax, we hence generalise capture-avoiding instantiation with a single variable x, s^{x}_{t}, to an instantiation $s[\sigma]$ with a **de Bruijn substitution** σ acting on all free variables at once. A de Bruijn substitution is hence represented by a function $\sigma : \mathbb{N} \to \mathsf{tm}$. We can think of a de Bruijn substitution as an infinite stream of terms.

Instantiation makes the implicit binding discipline observable. Most importantly, to replace the correct variable and avoid capture, we require substitution primitives which enable us to account for the implicit scope change. Already de Bruijn uses such primitives in his seminal paper [33].

A **renaming** is a certain subclass of substitutions and replaces an index with indices only,[1] i.e. is represented by a function $\xi : \mathbb{N} \to \mathbb{N}$. Every renaming ξ can be transformed into a substitution by forward composition with the variable constructor, $\xi \circ \mathsf{var}$. If apparent from the context, we might omit this composition by abuse of notation, i.e. might write $[\xi]$. Renamings play an essential role both in the definition of instantiation and later proofs.

We use a range of canonical substitution and renaming primitives, inspired by the σ-calculus [2]:

[1] Note that in contrast to other representations, renamings have to be neither injective nor bijective.

1. **Identity**, $\mathsf{id} : \mathbb{N} \to \mathbb{N}$, a renaming defined by $\mathsf{id}\, n := n$.
2. **Shifting**, $\uparrow : \mathbb{N} \to \mathbb{N}$, a renaming defined by $\uparrow n := 1 + n$.
3. **Extension**, $s \cdot \sigma$, which extends a stream $\sigma : \mathbb{N} \to \mathsf{tm}$ with a new element $s : \mathsf{tm}$ at the first position:

$$(s \cdot \sigma)\, 0 := s$$
$$(s \cdot \sigma)\, (1 + n) := \sigma\, n$$

We also repeatedly use forward composition of functions, $f \circ g$. Note that this is *not* composition of substitutions, which would not even pass the type checker. Composition binds stronger than expansion and is left-associative.

Our next goal is **capture-avoiding instantiation** with a de Bruijn substitution, $s[\sigma]$. We define instantiation mutually recursive with the forward composition of substitutions, $\sigma \circ [\tau]$:

$$(\mathsf{var}\, x)[\sigma] = \sigma\, x \qquad (\sigma \circ [\tau])\, x = (\sigma\, x)[\tau]$$
$$(\mathsf{app}\, s\, t)[\sigma] = \mathsf{app}\, (s[\sigma])\, (t[\sigma])$$
$$(\lambda.s)[\sigma] = \lambda.s[\Uparrow \sigma] \qquad \text{with } \Uparrow \sigma = \mathsf{var}\, 0 \cdot \sigma \circ [\uparrow]$$

Instantiation traverses the term homomorphically. In the case of a variable, we apply the respective substitution; for both application and abstraction, instantiation with σ is pushed into the subterms. Moreover, for abstraction and hence the traversal of a binder, we have to adapt the substitution according to the implicit scope change via the **lifting** $\Uparrow \sigma$.

This lifting consists of two parts. First, $\Uparrow \sigma$ should not change any reference to 0, we hence extend σ with $\mathsf{var}\, 0$. Second, free variables change their meaning under binders and we have to avoid captured variables. We thus require a scope change, realised via the post-composition with shifting, $\sigma \circ [\uparrow]$. We also say that σ is **shifted**.

For example, instantiation with shifting yields the following equation:

$$(\mathsf{app}\, (\lambda.\mathsf{app}\, (\mathsf{var}\, 0)\, (\mathsf{var}\, 1))\, (\mathsf{var}\, 0))[\uparrow] = \mathsf{app}\, (\lambda.\mathsf{app}\, (\mathsf{var}\, 0)\, (\mathsf{var}\, 2))\, (\mathsf{var}\, 1)$$

Implementation. The definition of instantiation is intertwined with forward composition, and hence is not structurally recursive. As a consequence, Coq forbids the definition as-is. We follow Adams [5] and streamline instantiation with substitutions via instantiation with renamings, written $s\langle\xi\rangle$:

1. We define the composition of renamings $\xi \circ \langle\zeta\rangle := \xi \circ \zeta$, which is forward function composition. We can then define the **lifting of a renaming** as:

$$\Uparrow^* \xi := 0 \cdot \xi \circ \langle\uparrow\rangle$$

2. We define **instantiation with a renaming**, $s\langle\xi\rangle$, which uses $\Uparrow^*$ instead of $\Uparrow$.
3. We define the composition of a substitution and instantiation with a renaming:

$$(\sigma \circ \langle\xi\rangle)\, x := (\sigma\, x)\langle\xi\rangle.$$

   We then define the **lifting of a substitution** as $\Uparrow \sigma := \mathsf{var}\, 0 \cdot \sigma \circ \langle\uparrow\rangle$.
4. We define **instantiation with a substitution**, using $\Uparrow$ in the case of abstraction.
5. We define the composition of substitutions, $\sigma \circ [\tau]$, via the above equations.

Once we have instantiation, we can express β-reduction using only the previously declared primitives:

$$\mathsf{app}\, (\lambda.s)\, t \succ s[t \cdot \mathsf{var}] \tag{3.1}$$

We replace $\mathsf{var}\, 0$ with t and then leave the remaining variables unchanged by extension with the variable constructor var which acts as identity substitution. In the further thesis, we abbreviate $t \cdot \mathsf{var}$ by $t..$, i.e. Equation 3.1 is represented by $\mathsf{app}\, (\lambda.s)\, t \succ s[t..]$.

3.1.2 Equational Reasoning on de Bruijn Syntax

Once defined, our next goal is to reason about the equality of syntactic expressions. Assume for example, that we want to show that β-reduction is **substitutive**, i.e. is stable under instantiation with substitutions:

$$(\mathsf{app}\, (\lambda.s)\, t)[\sigma] \succ (s[t \cdot \mathsf{var}])[\sigma]$$

This requires us to solve an equation of the form:

$$s[\Uparrow \sigma][t[\sigma]..] = s[t..][\sigma]$$

or without notation:

$$s[\mathsf{var}\, 0 \cdot \sigma \circ [\uparrow]][t[\sigma] \cdot \mathsf{var}] = s[t \cdot \mathsf{var}][\sigma].$$

We require several substitution properties to prove this equation. In the following, we show that instantiation with the identity substitution yields the original terms, substitutions can be composed, instantiation with renamings and substitutions behave analogously, and instantiation is extensional. Together this forms a complete, convergent rewriting system.

Equational Theory of the Substitution Primitives

We start with equations on the substitution primitives, hereafter referred to as the **interaction laws**. Recall that we write $f \equiv g$ to say that two functions f and g are equivalent.

Fact 3.1 (Interaction Laws).

1. $\mathsf{id} \circ f \equiv f \equiv f \circ \mathsf{id}$ *identity*
2. $(f \circ g) \circ h \equiv f \circ (g \circ h)$ *associativity*
3. $(s \cdot \sigma) \circ f \equiv (f\, s) \cdot (\sigma \circ f)$ *distributivity*
4. $\uparrow \circ (s \cdot \sigma) \equiv \sigma$ *interaction*
5. $0 \cdot \uparrow \equiv \mathsf{id}$ η-*identity*
6. $(\sigma\, 0) \cdot (\uparrow \circ \sigma) \equiv \sigma$ η-*law*

Proof. The first two equations, identity and associativity, follow directly by the definition of forward composition. Distributivity, η-identity, and the η-law follow by case analysis on the function argument, while interaction follows directly by the definition of expansion. □

In our later implementation, all these laws are stated parametically in the sort. We can so omit the automatic generation of these primitives.

Monad Laws

The most difficult proof is to show that instantiation is a monad as shown by Altenkirch and Reus [8], i.e.

$$s[\mathsf{var}] = s \qquad \text{right identity}$$
$$s[\sigma][\tau] = s[\sigma \circ [\tau]] \qquad \text{compositionality}$$

We refer to these laws as the **monad laws**.

The proof of both properties will be by induction on s. Cases where we traverse a binder will require special care and a so-called **lifting lemma**.

Right Identity. We start with the proof that instantiation with the variable constructor yields the original term. In general, the proof follows the structure of the term: Instantiation on all subterms again yields the original subterm. Things get involved if we go under binders, since we have to show that the lifting of the variable constructor is still equivalent to the variable constructor. We therefore split the proof into the following two lemmas:

Fact 3.2 (Identity Lifting). $\Uparrow \mathsf{var} \equiv \mathsf{var}$.

Proof. We have to show that $(\Uparrow \mathsf{var})\, n = \mathsf{var}\, n = n$. This follows directly by a case analysis on n. □

Lemma 3.3 (Right Identity). $s[\mathsf{var}] = s$.

Proof. By induction on s. In the variable case, the claim follows directly by the definition of identity; for application, it follows using the inductive hypothesis. To apply the inductive hypothesis for abstraction, we have to show that $\Uparrow \mathsf{id} = \mathsf{id}$ which follows with functional extensionality and Lemma 3.2. □

Note that as given, we assumed functional extensionality. We can omit functional extensionality by strengthening the statement to:

$$\forall \sigma. \sigma \equiv \mathsf{var} \to s[\sigma] = s$$

in which case Lemma 3.2 is identical to the required statement for abstraction. Lemma 3.3 still holds choosing $\sigma := \mathsf{var}$. The same works for the remaining proofs (Lemma 3.4, Lemma 3.7, Lemma 3.8), where we implicitly assume that the statement is given as above, but state (and prove) it in its simplified form. Autosubst proves the statement in its strengthened form.

Compositionality. We continue with compositionality. Compositionality requires a total of 8 lemmas.

Lemma 3.4 (Compositionality).

1. $(\Uparrow^* \xi) \circ (\Uparrow^* \zeta) \equiv \Uparrow^* (\xi \circ \zeta)$
2. $s\langle\xi\rangle\langle\zeta\rangle = s\langle\xi \circ \langle\zeta\rangle\rangle$
3. $(\Uparrow^* \xi) \circ (\Uparrow \tau) \equiv \Uparrow (\xi \circ \tau)$
4. $s\langle\xi\rangle[\tau] = s[\xi \circ \tau]$
5. $(\Uparrow \sigma) \circ (\Uparrow^* \zeta) \equiv \Uparrow (\sigma \circ \langle\zeta\rangle)$
6. $s[\sigma]\langle\zeta\rangle = s[\sigma \circ \langle\zeta\rangle]$
7. $(\Uparrow \sigma) \circ [\Uparrow \tau] \equiv \Uparrow (\sigma \circ [\tau])$
8. $s[\sigma][\tau] \equiv s[\sigma \circ [\tau]]$.

Proof. (2), (4), (6), and (8) are all proven by induction on s, using the respective lifting lemma (1), (3), (5), and (7) in the case of abstraction.

The lifting lemmas are where the action happens: (1) and (3) follow directly by case analysis on the argument. (5) and (7) require a case analysis on the argument as well; in each case, both sides reduce to $\mathsf{var}\, 0$ for $n = 0$. The case where $n = 1 + n'$ gets more involved. We here cover only (7), (5) is similar.

We show that

$$\begin{aligned}
((\Uparrow \sigma) \circ [\Uparrow \tau])(1+n) &= ((\mathsf{var}\,0 \cdot (\sigma \circ \langle\uparrow\rangle) \circ [\mathsf{var}\,0 \cdot (\tau \circ \langle\uparrow\rangle)])(1+n) && \Uparrow \\
&= ((\sigma \circ \langle\uparrow\rangle)\, n)[\mathsf{var}\,0 \cdot (\tau \circ \langle\uparrow\rangle)] && \cdot \\
&= (\sigma\, n)\langle\uparrow\rangle[\mathsf{var}\,0 \cdot (\tau \circ \langle\uparrow\rangle)]) && \circ \\
&= (\sigma\, n)[\uparrow \circ (\mathsf{var}\,0 \cdot (\tau \circ \langle\uparrow\rangle))]) && (4) \\
&= (\sigma\, n)[\tau \circ \langle\uparrow\rangle]) && \text{interaction} \\
&= (\sigma\, n)[\tau]\langle\uparrow\rangle && (6) \\
&= ((\sigma \circ [\tau]) \circ \langle\uparrow\rangle)\, n && \circ \\
&= (\mathsf{var}\,0 \cdot (\sigma \circ [\tau]) \circ \langle\uparrow\rangle)(1+n) && \cdot \\
&= (\Uparrow (\sigma \circ [\tau]))(1+n) && \Uparrow
\end{aligned}$$

Note that the proof required both (4) and (6). In (5), we require (2) instead of (4). □

Even if the proof needs several equations, it follows an easy structure. This structure can be adapted to more complex systems.

Equational Theory of λ

The above equations do not suffice to solve all equations. We additionally prove the following statements, called the **supplementary laws**:

Fact 3.5 (Supplementary Laws).

1. $\mathsf{var} \circ [\sigma] \equiv \sigma$ — *left identity*
2. $(\sigma \circ [\tau]) \circ [\theta] \equiv \sigma \circ [\tau \circ [\theta]]$ — *compositionality*
3. $\sigma \circ [\mathsf{var}] \equiv \sigma$ — *right identity*

Proof. Left identity follows directly by the definition of instantiation and composition, while the functional variants of compositionality and right identity follow with the monad laws (Lemma 3.3 and Lemma 3.4). □

We will abbreviate these laws by left identity, right identity, and compositionality. Note that all these are equivalences. To rewrite with these equivalences with the standard `rewrite` tactic of Coq, we then require functional extensionality.

See Figure 3.1 for a full overview of the **equational theory** of the untyped λ-calculus. Together, these syntactic equation lemmas form a convergent rewriting system, see Chapter 4.

Let us now return to our proof that β-reduction is substitutive:

Lemma 3.6. *β-reduction is substitutive.*

$$
\begin{aligned}
(\mathsf{var}\, x)[\sigma] &= \sigma\, x \\
(\mathsf{app}\, s\, t)[\sigma] &= \mathsf{app}\, (s[\sigma])\, (t[\sigma]) \\
(\lambda.s)[\sigma] &= \lambda.s[\mathsf{var}\, 0 \cdot \sigma \circ [\uparrow]] \\
s[\mathsf{var}] &= s \\
s[\sigma][\tau] &= s[\sigma \circ [\tau]] \\
\mathsf{var} \circ [\sigma] &\equiv \sigma \\
(\sigma \circ [\tau]) \circ [\theta] &\equiv \sigma \circ [\tau \circ [\theta]] \\
\sigma \circ [\mathsf{var}] &\equiv \sigma
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathsf{id} \circ f &\equiv f \equiv f \circ \mathsf{id} \\
(f \circ g) \circ h &\equiv f \circ (g \circ h) \\
(s \cdot \sigma) \circ f &\equiv (f\, s) \cdot (\sigma \circ f) \\
\uparrow \circ\, (s \cdot \sigma) &\equiv \sigma \\
0 \cdot \uparrow &\equiv \mathsf{id} \\
(\sigma\, 0) \cdot (\uparrow \circ\, \sigma) &\equiv \sigma
\end{aligned}
$$

Figure 3.1: Equational theory of λ.

Proof. We have to show:

$$
\begin{aligned}
s[\Uparrow \sigma][t[\sigma]..] &= s[\mathsf{var}\, 0 \cdot \sigma \circ \langle\uparrow\rangle][t[\sigma] \cdot \mathsf{var}] && \Uparrow \\
&= s[(\mathsf{var}\, 0 \cdot \sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathsf{var})]] && \text{compositionality} \\
&= s[(\mathsf{var}\, 0)[t[\sigma] \cdot \mathsf{var}] \cdot (\sigma \circ \langle\uparrow\rangle) \circ [(t[\sigma] \cdot \mathsf{var})]] && \text{distributivity} \\
&= s[(\mathsf{var}\, 0)[t[\sigma] \cdot \mathsf{var}] \cdot \sigma \circ (\langle\uparrow\rangle \circ [t[\sigma] \cdot \mathsf{var}])] && \text{associativity} \\
&= s[(t[\sigma] \cdot \mathsf{var})\, 0 \cdot (\sigma \circ [\uparrow \circ\, (t[\sigma] \cdot \mathsf{var})])] && \text{compositionality} \\
&= s[t[\sigma] \cdot (\sigma \circ [\mathsf{var}])] && \cdot, \text{ interaction} \\
&= s[t[\sigma] \cdot \sigma] && \text{right identity} \\
&= s[t[\sigma] \cdot (\mathsf{var} \circ [\sigma])] && \text{left identity} \\
&= s[(t \cdot \mathsf{var}) \circ [\sigma]] && \text{distributivity} \\
&= s[t \cdot \mathsf{var}][\sigma]. && \text{compositionality}
\end{aligned}
$$

□

While this proof might seem daunting, note that in all steps we used the equations in Figure 3.1 from left to right. Autosubst will hence prove the above equation automatically. In the next chapter, we explain why this is enough.

Extensionality, Renamings, and Substitutions

We additionally prove that instantiation with pointwise equivalent substitutions leads to equal terms.

Lemma 3.7 (Extensionality).

1. *If* $\sigma \equiv \tau$, *then* $\Uparrow \sigma \equiv \Uparrow \tau$.

2. *If* $\sigma \equiv \tau$, *then* $s[\sigma] = s[\tau]$.

Proof. (2) follows by induction on s, using the lifting lemmas in the case of abstraction. The lifting lemma holds by case analysis on the respective argument. Note that extensionality for instantiation with substitutions does not rely on extensionality for instantiation with renamings. □

We claimed that every renaming could be translated into a substitution. We also prove this claim formally:

Lemma 3.8 (Coincidence).

1. $\Uparrow^* \xi \circ \mathsf{var} \equiv \Uparrow (\xi \circ \mathsf{var})$
2. $s[\xi \circ \mathsf{var}] = s\langle\xi\rangle$

Proof. The first statement, (1), is proven by case analysis on the function argument. Then, (2) follows by induction on s using (1) in the case of abstraction. □

The Autosubst compiler will automatically generate all these lemmas.

3.1.3 De Bruijn Algebra

Together terms and substitutions with the suggested substitution primitives form a two-sorted algebra, the **de Bruijn algebra** [99].

De Bruijn algebras will provide the semantics for our later EHOAS specifications. We will see how to extend the de Bruijn algebra of the λ-calculus to more complex syntactic sorts in Chapter 6.

One might ask whether all equations can be proven via a systematic proof method, e.g. whether there exists a finite set of equations and equality is **complete** for this set of equations [13]. Our reasoning in the next chapter will mainly revolve around this question.

3.2 Scoped de Bruijn Algebra

While pure de Bruijn syntax offers a lean and elegant representation of binders, definitions and lemmas have to be adapted and might require technical adaptions. The common problem is that we have to take care in which context we talk about a de Bruijn term, and hence sometimes have to add shiftings to lift a term to the right context.

However, scoped de Bruijn syntax [18] offers a simple solution: In scoped syntax, we annotate every term with its scope: For example, the type of terms, tm^k, is indexed by the upper bound k of free variables. We can hence use the type checking mechanism of

Coq, which yields type errors in the case of forgotten shiftings. Additionally, we have a type of closed terms tm^0 and can easily represent a context with n variables in it.

Consider the scoped representation of de Bruijn syntax, where we annotate subterms with their **scope**, i.e. an upper bound of the free variable:

$$s, t \in \mathsf{tm}^k \ := \ \mathsf{var}\, x \mid \mathsf{app}\, s^k\, t^k \mid \lambda.s^{1+k} \qquad x \in \mathbb{I}^k$$

Variables are taken from within this scope, here the **finite type** $\mathbb{I}^k$ with k elements defined below. We will frequently omit the scope if a definition is parametric in the scope. While application consists of two subterms with k free variables each, the type of abstraction allows an additional variable bound by abstraction: Here, the upper bound of free variables is increased to $1 + k$.

We define a canonical **finite type** $\mathbb{I} : \mathbb{N} \to \mathsf{Type}$ with n elements by recursion on n:

```
Fixpoint 𝕀 (n : ℕ): Type :=
match n with
| 0 ⇒ ⊥
| 1+n ⇒ 𝟘 (𝕀 n)
end.
```

The definition by recursion is non-standard, but useful to recover certain definitional equations.

We write $0_{\mathbb{I}} : \mathbb{I}^{1+n}$ to denote the finite element representing the natural number 0 (actually the empty option $\emptyset$ in a type $\mathbb{I}^{1+n}$), and $1_{\mathbb{I}} +_{\mathbb{I}} x : \mathbb{I}^{1+n}$ for the successor of $x : \mathbb{I}^n$ (realised with $\lfloor x \rfloor$). We also directly write $n_{\mathbb{I}}$ for the n-ary shifting of the empty option.

Again, a **de Bruijn substitution** simultaneously replaces all free variables; this time it is represented by a function $\sigma : \mathbb{I}^m \to \mathsf{tm}^n$. We can think of a substitution as a stream of terms with length m. We define all the previous primitives using scoped variants:

- **Identity**, $\mathsf{id}_n : \mathbb{I}^n \to \mathbb{I}^n$, the renaming with $\mathsf{id}\, x := x$.
- **Shifting**, $\uparrow_n : \mathbb{I}^n \to \mathbb{I}^{1+n}$, the renaming with $\uparrow x := 1_{\mathbb{I}} +_{\mathbb{I}} x$.
- **Extension**, $s \cdot_n \sigma$, which extends a stream $\sigma : \mathbb{I}^n \to \mathsf{tm}^k$ with a new element $s : \mathsf{tm}^k$ at the first position and thus yields a stream of type $\mathbb{I}^{1+n} \to \mathsf{tm}^k$:

$$(s \cdot \sigma)\, 0_{\mathbb{I}} = s$$
$$(s \cdot \sigma)\, (1_{\mathbb{I}} +_{\mathbb{I}} x) = \sigma\, x$$

In the future, we omit the indices if irrelevant or clear from the context.

See Figure 3.2 for instantiation $\_[\_] : (\mathbb{I}^m \to \mathsf{tm}^n) \to \mathsf{tm}^m \to \mathsf{tm}^n$ on the scoped λ-calculus. Its definition is entirely analogous to its pure counterpart, it only differs in that we use the scoped primitives. For example, shifting has the following type:

$$\Uparrow \_ : (\mathbb{I}^m \to \mathsf{tm}^n) \to (\mathbb{I}^{1+m} \to \mathsf{tm}^{1+n})$$

$$
\begin{aligned}
(\mathsf{var}\ x)[\sigma] &= \sigma\, x & (\sigma \circ [\tau])\, x &= (\sigma\, x)[\tau] \\
(\mathsf{app}\ s\ t)[\sigma] &= \mathsf{app}\ (s[\sigma])\ (t[\sigma]) & & \\
(\lambda.s)[\sigma] &= \lambda.s[\Uparrow \sigma] & \text{with } \Uparrow \sigma &= \mathsf{var}\ 0_{\mathbb{I}} \cdot \sigma \circ [\uparrow]
\end{aligned}
$$

Figure 3.2: Instantiation for the scoped λ-calculus.

As before, we have to first define instantiation with renamings, where a renaming is a function $\xi : \mathbb{I}^m \to \mathbb{I}^n$.

The scoped primitives satisfy the same laws as their pure counterpart in Figure 3.1, given the respective scope constraints. Additionally we can show:

Lemma 3.9 (Expansion). *Let σ and τ be arbitrary substitutions on the empty domain, i.e. of type $\mathbb{I}^0 \to \mathsf{tm}^n$. Then, $\sigma \equiv \tau$.*

Proof. By case analysis on the argument. As $\mathbb{I}^0$ is the empty type, the claim follows immediately. □

In the future, we denote the empty substitution, **expansion**, by $!_m$ where m denotes the scope of the codomain.

Together, scoped terms and the substitution primitives given above form the **scoped de Bruijn algebra**. Again, constructors and primitives will be the operations of this algebra.

Scoped syntax [16] thus offer the same support for substitutions as pure syntax. Its great advantage is that it offers type safety, and so simplifies the entry-level for new users. We will talk about this more explicitly in later sections.

It moreover offers the possibility to talk explicitly about **closed terms**, i.e. terms without any variables. We can simply express closedness with the type, tm^0.

3.3 Discussion

Even for de Bruijn syntax, instantiation can be defined in many ways. In particular, there are **single-point substitutions** which do not replace all variables at once but replace one variable by a new term. This complicates equational reasoning because a normal form would require to determine the order of variables.

The scope change can be handled at two places: Either directly during the binder as we do, or we can introduce an additional lifting function which traverses the whole term until a variable. Small differences will have a big impact on the quality of equational reasoning. We are not aware of any completeness result of any other approach for reasoning on de Bruijn syntax or named syntax.

We see scoped syntax as the sweet spot between de Bruijn syntax and **intrinsically typed syntax** [16], where the type of a syntactic sort is indexed with its semantic type. It is powerful enough to detect errors if something is wrong related to bindings. Intrinsically typed syntax is difficult to implement for more complex syntactic systems, as it requires inductive-recursive types [42] (which are not implemented in Coq).

Chapter 4

Pure Sigma Calculus

In the last chapter, we have introduced the pure and scoped de Bruijn algebra corresponding to the λ-calculus. To prove that β-reduction is substitutive (Lemma 3.6), it sufficed to rewrite with the equations of Figure 3.1. We now present a proof method for solving equations on the pure de Bruijn algebra based on rewriting due to Schäfer et al. [99]. This strategy can be generalised to arbitrary assumption-free equations between terms.

For the remainder of the chapter, we assume that the reader is familiar with the usual language and constructions of abstract rewriting systems. We refer to Appendix A for an overview of the definitions, and to Baader and Nipkow [13] for a gentle introduction.

To make the above claim of a proof strategy for assumption-free equations rigorous, we first recall calculi of explicit substitutions [2, 31, 56] which offer a syntactic representation of instantiation, achieved by mutual inductive definition with expressions. The σ-calculus [2], the first calculus of explicit substitutions, was initially developed as an intermediate representation between the λ-calculus and its actual implementation.

A calculus of explicit substitutions consists of two parts: first, a mutual inductive type of expressions and substitution expressions, and second, a notion of reduction, $s \rightsquigarrow t$. Reduction contains both the reduction behaviour of instantiation and the interaction laws between the different substitution primitives. For example, the σ-calculus by Abadi et al. [2], the first calculus of explicit substitutions, contains expressions for λ-terms and a constructor for applying an instantiation. Note that we do *not* talk about reduction in the λσ-calculus which additionally incorporates β-reduction; we are interested the behaviour of instantiation only.

A range of different calculi of explicit substitutions has been developed for the λ-calculus alone, all with varying properties of confluence and termination. The remainder of this chapter will be based on the σ_{SP}-calculus by Curien et al. [31], a successor of the σ-calculus which is designed to be confluent.

This calculus is particularly well-suited as a base for automation on de Bruijn algebras

since Schäfer et al. [99] show that reduction in the σ_{SP}-calculus is sound and complete for the de Bruijn algebra of the λ-calculus. Using only reduction on the σ_{SP}-calculus, we can hence prove all solvable equations between expressions. We recall the exact definition of the σ_{SP}-calculus, reduction, and the model construction in Section 4.1 and Section 4.2. Different to Schäfer et al. we use identity substitution as a primitive.

To use the rewriting method for substitution equations, one additionally requires convergence, i.e. confluence and termination [13]. Concerning a mechanisation, Schäfer et al. [99, p. 2] remark:

> "While the verification of our decision method is not difficult (even in Coq), a verification of the rewriting method is surprisingly complex since the existing termination proof [...] is far from straightforward. We did not succeed in simplifying this proof and think that a formalization with a proof assistant is a substantial enterprise."

Formalisation and mechanisation of both the confluence proof and the termination proof are the focus of this chapter. As is standard, we use Newman's Lemma [81] (see Lemma A.4 for its Coq formalisation), which states that confluence and the simpler local confluence, i.e. joinability after doing two single steps only, are equivalent under the assumption of termination.

Proving *local confluence* is straightforward. Traditionally, local confluence is shown over the convergence of critical pairs [13, Chapter 6][59, 69]. Such an analysis reduces the number of reductions to examine for local confluence by removing trivial instances and ending up with only **critical pairs**, i.e. overlapping instances of two rewriting rules.

The convergence of critical pairs is also the proof method chosen in the formalisation of another calculus of explicit substitutions by Saïbi [96]. Lucky for us, we do not need to formalise the notion of trivial and critical pairs but can use Coq's depth-first search with backtracking which investigates all possible derivations of reduction. Implicitly, Coq examines the same derivations of reduction as both a critical pair analysis and the proof for a critical pair analysis, only automatically.

For *termination*, the main challenge is the rule corresponding to *abstraction reduction*:

$$(\lambda.s)[\sigma] \succ \lambda.(s[0 \cdot \sigma \circ S]).$$

This rule introduces new symbols and thus prevents a proof by either induction on the term structure or the usual (simplification) termination strategies, such as recursive path orderings, Knuth-Bendix [60], or polynomial interpretations [24].

Several proofs exist in the literature. The first proof for a similar system was established by Hardin and Laville [55] for CCL, Categorical Combinators, where the authors prove convergence over a function that counts the appearances of possible abstraction reductions. Compositions double the number of abstractions and create new reduc-

tions, which complicates the definition of this function and the resulting termination proof.

The next proof by Curien et al. [30] uses the original σ-calculus as a base and follows another strategy: It proves a generalisation of the theorem that for every strongly normalising substitution σ, also $\sigma \circ S$ is strongly normalising, a result called **preservation**. The proof requires several non-trivial generalisations. A third proof by Zantema [121] interprets λ-terms into algebras.

Notably, there is a mechanisation in ALF closely following Curien et al. [65], but unfortunately we were unable to retrieve this proof. The authors report that many theorems needed more details than their paper-based counterpart. We base our formalisation on the proof idea by Curien et al., but simplify the proof by adding another intermediate level and then simplifying the proof of the core calculus. The proof is still very technical, but requires only 700 lines of code. See Section 4.8 for a detailed discussion of the differences to both the original proof and the mechanised proof in ALF.

Proof outline of termination. Termination is split into three parts. In each part, we reduce termination of one notion of reduction to termination of a simplified notion of reduction. We start with the usual definition of reduction on the σ_{SP}-calculus.

In a first step, we note that certain operations behave similarly to reduction, e.g. applications and extension, composition and instantiation. We simplify our original expressions to a *unified syntax* where expressions and substitution expressions are unified (Section 4.4). As every step in the original system can be simulated, strong normalisation of unified expressions implies strong normalisation of reduction.

In the next step, we separate our set of rules into projection rules, which reduce the size of expressions, and distribution rules, propagating instantiation with substitution into subterms (Section 4.5). Strong normalisation of both projection and distribution rules is shown to be equivalent to strong normalisation of the subset of distribution rules alone.

Proving termination of the distribution calculus is then our final proof obligation and at the same time, the most involved one (Section 4.6). The central notion is a preservation theorem which states that we can add renamings at several positions without inflicting strong normalisation. This preservation theorem is very similar to the one of Curien et al. [30].

Finally, using Newman's Lemma [81], we show confluence via local confluence and termination. We end with a more detailed comparison of termination proofs in the literature (Section 4.8).

4.1 Syntax and Reduction

We recall the syntax of the σ_{SP}-calculus. The σ_{SP}-calculus consists of two mutual inductive sorts: **expressions**, $s, t \in \mathsf{exp}$, and **substitution expressions**, $\sigma, \tau \in \mathsf{subst}$:

$$\begin{aligned} s, t \in \mathsf{exp} &:= 0 \mid \mathsf{app}\, s\, t \mid \lambda.s \mid s[\sigma] \mid \nu^{\mathsf{exp}} && \nu^{\mathsf{exp}} \in \mathbb{N} \\ \sigma, \tau \in \mathsf{subst} &:= I \mid S \mid s \cdot \sigma \mid \sigma \circ \tau \mid \nu^{\mathsf{subst}} && \nu^{\mathsf{subst}} \in \mathbb{N} \end{aligned}$$

Expressions closely follow the syntax of the λ-calculus: We consider the variable 0, applications $\mathsf{app}\, s\, t$, abstractions $\lambda.s$, and **expression parameters** ν^{exp} ranging over expressions. These parameters are simply placeholders for arbitrary terms and remain unchanged and should not be replaced with variables. They are unaffected by instantiation. Additionally, an expression can be instantiated with a substitution expression $s[\sigma]$.

For substitution expressions, we consider the identity substitution I, shifting S corresponding to the semantic $\uparrow$, extension $s \cdot \sigma$, substitution composition $\sigma \circ \tau$ and **substitution parameters** ν^{subst}. As for the corresponding de Bruijn algebra, composition binds stronger than expansion and is left-associative.

The most notable difference to the de Bruijn algebra is that substitutions have a syntactic representation. Every variable $\mathsf{var}\, n$ can be represented by the n-ary instantiation of the variable 0 with S.

The σ_{SP}-calculus comes with a notion of **reduction**, $s \rightsquigarrow t$. For simplicity, we use the same notation for reduction on expressions and substitution expressions. See Figure 4.1 for the rules, which can be applied in an arbitrary context.

The reduction rules should seem familiar from the previous chapter: the first three rules describe the propagation of instantiation with substitutions; from now on called variable reduction, abstraction reduction, and application reduction. We moreover have reduction rules corresponding to the monad laws and the interaction lemmas.

Axiomatic equivalence, written $s \equiv t$ and $\sigma \equiv \tau$, describes the equivalence closure of reduction. As the reduction rules can be applied in an arbitrary context, the following lemma follows directly:

Lemma 4.1 (Congruence). *Axiomatic equivalence is a congruence w.r.t. the constructors of expressions and substitution expressions.*

4.2 De Bruijn Algebra as a Model of the Sigma Calculus

As established by Schäfer et al. [99], the pure de Bruijn algebra is a semantic model of the σ_{SP}-calculus. We here recall the exact model construction and the proofs of soundness and completeness.

The denotation of expressions and substitution expressions relies on an **expression assignment** $\alpha : \mathbb{N} \to \mathsf{tm}$ and a **substitution assignment** $\beta : \mathbb{N} \to (\mathbb{N} \to \mathsf{tm})$ which give an

$$
\begin{aligned}
0[s \cdot \sigma] &\rightsquigarrow s \\
(\mathsf{app}\ s\ t)[\sigma] &\rightsquigarrow \mathsf{app}\ s[\sigma]\ t[\sigma] \\
(\lambda.s)[\sigma] &\rightsquigarrow \lambda.(s[0 \cdot (\sigma \circ S)]) \\
I \circ \sigma &\rightsquigarrow \sigma \\
\sigma \circ I &\rightsquigarrow \sigma \\
(\sigma \circ \tau) \circ \theta &\rightsquigarrow \sigma \circ (\tau \circ \theta) \\
S \circ (s \cdot \sigma) &\rightsquigarrow \sigma \\
s[I] &\rightsquigarrow s \\
s[\sigma][\tau] &\rightsquigarrow s[\sigma \circ \tau] \\
(s \cdot \sigma) \circ \tau &\rightsquigarrow (s[\tau]) \cdot (\sigma \circ \tau) \\
0 \cdot S &\rightsquigarrow I \\
0[\sigma] \cdot S \circ \sigma &\rightsquigarrow \sigma
\end{aligned}
$$

Figure 4.1: Reduction in the σ_{SP}-calculus, with the congruence rules omitted.

interpretation to expression and substitution parameters respectively. We then define a **denotation**

$$
\begin{aligned}
[\![\_]\!]_{\alpha,\beta} &: \mathsf{exp} \to \mathsf{tm} \\
[\![\_]\!]_{\alpha,\beta} &: \mathsf{subst} \to (\mathbb{N} \to \mathsf{tm}).
\end{aligned}
$$

which is depicted in Figure 4.2.

Two expressions s and t are said to be **denotationally equivalent** if $[\![s]\!]_{\alpha,\beta} = [\![t]\!]_{\alpha,\beta}$ for all assignments α and β. Similarly, two substitution expressions σ and τ are denotationally equivalent if $[\![\sigma]\!]_{\alpha,\beta} \equiv [\![\tau]\!]_{\alpha,\beta}$ for all assignments α and β. Note that we require functional equivalence and not equality.

The de Bruijn algebra is a **sound** model of axiomatic equivalence, if two axiomatically equivalent (substitution) expressions are also denotationally equivalent. In contrast, axiomatic equivalence is said to be **complete** for equality on the de Bruijn algebra, if denotational equivalence implies axiomatic equivalence.

Proving soundness is straightforward and follows by a mutual induction on reduction for expressions and substitutions expressions using the previously established laws in Figure 3.1:

Lemma 4.2 (Soundness). *The de Bruijn algebra is a sound semantic model for the σ_{SP}-calculus.*

$$
\begin{aligned}
[\![0]\!]_{\alpha,\beta} &= \mathsf{var}\ 0 & [\![\mathrm{I}]\!]_{\alpha,\beta} &= \mathsf{var} \\
[\![\mathsf{app}\ s\ t]\!]_{\alpha,\beta} &= \mathsf{app}\ ([\![s]\!]_{\alpha,\beta})\ ([\![t]\!]_{\alpha,\beta}) & [\![\mathrm{S}]\!]_{\alpha,\beta} &= \uparrow \circ \mathsf{var} \\
[\![\lambda.s]\!]_{\alpha,\beta} &= \lambda.([\![s]\!]_{\alpha,\beta}) & [\![s \cdot \sigma]\!]_{\alpha,\beta} &= ([\![s]\!]_{\alpha,\beta}) \cdot ([\![\sigma]\!]_{\alpha,\beta}) \\
[\![s[\sigma]]\!]_{\alpha,\beta} &= ([\![s]\!]_{\alpha,\beta})\ [[\![\sigma]\!]_{\alpha,\beta}] & [\![\sigma \circ \tau]\!]_{\alpha,\beta} &= ([\![\sigma]\!]_{\alpha,\beta}) \circ ([\![\tau]\!]_{\alpha,\beta}) \\
[\![\nu^{\mathsf{exp}}]\!]_{\alpha,\beta} &= \alpha\, \nu^{\mathsf{exp}} & [\![\nu^{\mathsf{subst}}]\!]_{\alpha,\beta} &= \beta\, \nu^{\mathsf{subst}}
\end{aligned}
$$

Figure 4.2: Translation from expressions (substitution expressions) to de Bruijn terms (substitutions).

$$
\frac{s \rightsquigarrow s'}{\mathsf{app}\ s\ t \rightsquigarrow \mathsf{app}\ s'\ t} \qquad \frac{t \rightsquigarrow t'}{\mathsf{app}\ s\ t \rightsquigarrow \mathsf{app}\ s\ t'} \qquad \frac{s \rightsquigarrow s'}{\lambda.s \rightsquigarrow \lambda.s'} \qquad \frac{s \rightsquigarrow s'}{s[\sigma] \rightsquigarrow s'[\sigma]} \qquad \frac{\sigma \rightsquigarrow \sigma'}{s[\sigma] \rightsquigarrow s[\sigma']}
$$

$$
\frac{s \rightsquigarrow s'}{s \cdot \sigma \rightsquigarrow s' \cdot \sigma} \qquad \frac{\sigma \rightsquigarrow \sigma'}{s \cdot \sigma \rightsquigarrow s \cdot \sigma'} \qquad \frac{\sigma \rightsquigarrow \sigma'}{\sigma \circ \tau \rightsquigarrow \sigma' \circ \tau} \qquad \frac{\tau \rightsquigarrow \tau'}{\sigma \circ \tau \rightsquigarrow \sigma \circ \tau'}
$$

Figure 4.3: Implementation of the congruence rules of reduction in Coq.

Proving completeness is considerably harder and relies on a translation into a simplified form of expressions called normal forms. We omit this theorem here but refer to the formalised completeness result of Schäfer et al. [99].

Theorem 4.3 (Schäfer et al. [99]). *Reduction in the σ_{SP}-calculus is complete for equality in the de Bruijn algebra.*

Due to this correspondence, the meta-theoretic properties of reduction in the σ_{SP}-calculus established in the next sections have a direct influence on reasoning on mechanised syntax in the de Bruijn algebra.

4.3 Local Confluence

We show that reduction is locally confluent, i.e. if $s \rightsquigarrow t$ and $s \rightsquigarrow t'$, then t and t' are joinable by reduction. Recall that for the proof we naively explore all derivations of $s \rightsquigarrow t$ and $s \rightsquigarrow t'$.

Hence the exact definition of the congruence closure of reduction is relevant; see Figure 4.3 for the definition as an inductive predicate as implemented in Coq.

To show that t and t' are joinable, we proceed by a nested inversion on the derivation of $s \rightsquigarrow t$ and $s \rightsquigarrow t'$. We repeat this process for all appearing reductions until all derivations of the form $s \rightsquigarrow t$ are trivial in the sense that s is an atomic, not further decomposable expression. This analysis leads to 141 cases, of which 130 can be solved automatically using backward reasoning (`eauto`) until depth 20. Coq needs to know that axiomatic equivalence is a congruence (Lemma 4.1).

The remaining instances are not the critical pairs, but rather instances of the "trivial" occurrences. If a subterm diverges to two different redexes, backwards reasoning does not suffice. For example, we could have the case of an application $\mathsf{app}\ s\ t$ where

$$\frac{s \rightsquigarrow s'}{\mathsf{app}\ s\ t \rightsquigarrow \mathsf{app}\ s'\ t} \qquad \frac{s \rightsquigarrow s''}{\mathsf{app}\ s\ t \rightsquigarrow \mathsf{app}\ s''\ t}$$

where we want to unify $\mathsf{app}\ s'\ t$ and $\mathsf{app}\ s''\ t$ via the inductive hypothesis. Backwards reasoning does not suffice, but a two-line Ltac script automatically solves the remaining 11 cases.

Lemma 4.4 (Local Confluence). *Reduction for the σ_{SP}-calculus is locally confluent.*

Proof. By induction on one of the paths, case analysis on the second paths, using congruence of axiomatic equivalence (Lemma 4.1). □

Note that implicitly our proof follows the same line as a critical pairs analysis: A critical pair analysis only reduces the number of pairs to join and removes all trivial proof obligations. In our case, we let Coq automatically solve these cases.

4.4 Reduction to Unified Expressions

We now turn to the proof of termination of reduction. In a first step, we unify expressions and substitution expressions. We hence consider the following syntax of **unified expressions** uexp:

$$s, t \in \mathsf{uexp} \ := \ c \mid \lambda.s \mid (s, t) \mid s \gg t$$

This representation unifies operations which behave similarly to reduction, i.e. application and extension are unified by the pair operator, instantiation and composition are unified by unified composition $\gg$. Parameters, the identity substitution, and the shift substitution behave all similarly to reduction and hence will all be represented by the **constant** c.

We split the rules of reduction into **projection rules**, written $s \supset t$, and **distribution rules**, short $s \succ t$; see Figure 4.4 for a definition. Both kinds of rules may be applied in an arbitrary context. **Projection** projects a sub-expression s to a subexpression s_i and hence decreases the size of an expression. On the other hand, **distribution** propagates substitution instantiation through abstractions, pairs, and compositions. By slight abuse of language, we usually refer to the different rules by the corresponding rule of the original σ_{SP}-calculus.

Unified reduction is the union of these two closures:

$$s \rightsquigarrow_u t := s \supset t \vee s \succ t$$

We write $\rightsquigarrow_u^*$ to denote its reflexive-transitive closure, and $\rightsquigarrow_u^+$ to denote its transitive closure. Both notions preserve congruence:

Projection $(s \supset t)$

$$\begin{aligned} \lambda.s &\supset s \\ (s,t) &\supset s \\ (s,t) &\supset t \\ s \gg t &\supset s \\ s \gg t &\supset t \end{aligned}$$

Distribution $(s \succ t)$

$$\begin{aligned} (s,t) \gg u &\succ (s \gg u, t \gg u) \\ (s \gg t) \gg u &\succ s \gg t \gg u \\ (\lambda.s) \gg u &\succ \lambda.(s \gg (c, u \gg c)) \end{aligned}$$

Figure 4.4: Unified reduction for unified expressions, congruence rules omitted.

$$\begin{aligned} \lceil 0 \rceil &= c & \lceil I \rceil &= c \\ \lceil \mathsf{app}\, s\, t \rceil &= (\lceil s \rceil, \lceil t \rceil) & \lceil \uparrow \rceil &= c \\ \lceil \lambda.s \rceil &= \lambda.\lceil s \rceil & \lceil s \cdot \sigma \rceil &= (\lceil s \rceil, \lceil \sigma \rceil) \\ \lceil s[\sigma] \rceil &= \lceil s \rceil \gg \lceil \sigma \rceil & \lceil \sigma \circ \tau \rceil &= \lceil \sigma \rceil \gg \lceil \tau \rceil \\ \lceil \nu^{\mathsf{exp}} \rceil &= c & \lceil \nu^{\mathsf{subst}} \rceil &= c \end{aligned}$$

Figure 4.5: Translation $\lceil \_ \rceil : \mathsf{exp} \to \mathsf{uexp}$ from expressions to unified expressions.

Fact 4.5 (Congruence). *$\rightsquigarrow_u^*$ and $\rightsquigarrow_u^+$ are congruences.*

Translation functions, depicted in Figure 4.5,

$$\begin{aligned} \lceil \_ \rceil &: \mathsf{exp} \to \mathsf{uexp} \\ \lceil \_ \rceil &: \mathsf{subst} \to \mathsf{uexp} \end{aligned}$$

translate the original expressions into the new unified syntax. For example,

$$\lceil (\mathsf{app}\, s\, 0)[\sigma] \rceil = (\lceil s \rceil, c) \gg \lceil \sigma \rceil.$$

The original reduction can be simulated with unified reduction.

Fact 4.6 (Simulation). *If $s \rightsquigarrow t$, then $\lceil s \rceil \rightsquigarrow_u^+ \lceil t \rceil$, and if $\sigma \rightsquigarrow \tau$, then $\lceil \sigma \rceil \rightsquigarrow_u^+ \lceil \tau \rceil$.*

Proof. By a mutual induction on $s \rightsquigarrow t$ and $\sigma \rightsquigarrow \tau$, using congruence of the transitive closure. □

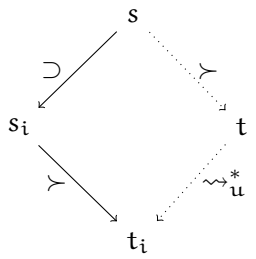

Figure 4.6: Exchangeability.

Note that unified reduction is strictly more general than reduction as whole parts of a term may be omitted — even without an identity substitution on the left or right side.

Moreover, in the above proof of simulation, we need the transitive closure: the statement does not hold for one step of unified reduction and the equivalence closure does not suffice for Corollary 4.7.

Strong normalisation for unified expressions hence suffices to show strong normalisation of the original reduction.

Corollary 4.7. *If* $\mathsf{sn}_{\rightsquigarrow}(\lceil s \rceil)$, *then* $\mathsf{sn}_{\rightsquigarrow_u}(s)$ *and if* $\mathsf{sn}_{\rightsquigarrow}(\lceil \sigma \rceil)$, *then* $\mathsf{sn}_{\rightsquigarrow_u}(\sigma)$.

Proof. Using the morphism lemma (Fact A.2) and simulation (Fact 4.6). □

In the following, we prove termination for unified expressions. We from now on refer to unified expressions as expressions.

4.5 Reduction to Distribution Termination

We simplify our reduction relation even further. For strong normalisation of unified reduction, strong normalisation of the distribution calculus alone suffices:

$$\mathsf{sn}_{\rightsquigarrow_u}(s) \leftrightarrow \mathsf{sn}_{\succ}(s).$$

The implication from right to left directly follows as we can simulate distribution in the full reduction relation (Fact A.3). The reverse intuitively holds as projections reduce the size of a term and hence reduce the number of distribution steps even further. We make this claim rigorous in the following.

First, note that projections always reduce the size of an expression and projection reduction is hence terminating:

Fact 4.8. *Assume that* $\mathsf{sn}_{\supset}(s)$ *and* $\mathsf{sn}_{\supset}(t)$. *Then* $\mathsf{sn}_{\supset}(\lambda.s)$, $\mathsf{sn}_{\supset}(s, t)$, *and* $\mathsf{sn}_{\supset}(s \gg t)$.

Proof. By a nested induction on $\mathsf{sn}_{\supset}(s)$ and $\mathsf{sn}_{\supset}(t)$. □

Corollary 4.9. *For all expressions* s, $\mathsf{sn}_{\supset}(s)$.

Proof. By induction on s, using Fact 4.8. □

We then require the following restricted form of confluence, called **exchangeability** and depicted in Figure 4.6: If we do a distribution step in a subterm s_i of s to t_i, then this distribution step is already possible in s, and moreover, we can afterwards reduce to this redex.

Lemma 4.10 (Exchangeability). *If* $s \supset s_i$ *and* $s_i \succ t_i$, *then there exists an expression* t *such that* $s \succ t$ *and* $t \rightsquigarrow_u^* t_i$.

Proof. By induction on $s \supset s_i$, and a subsequent case analysis on $s_i \succ t_i$, using congruence of $\rightsquigarrow_u$ (Fact 4.5). □

The reduction to strong normalisation of $\mathsf{sn}_{\succ}(s)$ is then no longer hard:

Lemma 4.11. *If* $\mathsf{sn}_{\succ}(s)$, *then* $\mathsf{sn}_{\rightsquigarrow_u}(s)$.

Proof. By induction on $\mathsf{sn}_{\succ}(s)$. The inductive hypothesis states that $\mathsf{sn}_{\rightsquigarrow_u}(s')$ for every s' with $s \succ s'$. We further do a second, nested induction on the fact that $\mathsf{sn}_{\supset}(s)$ (possible with Lemma 4.9).

To show that $\mathsf{sn}_{\rightsquigarrow_u}(s)$, we assume some s' with $s \rightsquigarrow_u s'$ and show that $\mathsf{sn}_{\rightsquigarrow_u}(s')$. Case analysis on $s \rightsquigarrow_u s'$. If $s \succ s'$, the assumption follows directly with the outer inductive hypothesis.

If $s \supset s'$, we would like to use the inductive hypothesis for $\mathsf{sn}_{\supset}(s)$. We thus have to show that for all s'' with $s' \succ s''$, also $\mathsf{sn}_{\rightsquigarrow_u}(s'')$. Assume such an s''. By exchangeability (Lemma 4.10), there exists some t with $s \succ t$ and $t \rightsquigarrow_u^* s''$. Using the outer inductive hypothesis, we thus know that $\mathsf{sn}_{\rightsquigarrow_u}(t)$ and also $\mathsf{sn}_{\rightsquigarrow_u}(s'')$ by forward propagation of strong normalsiation (Fact A.2), and thus our claim is shown. □

We hence finally get our desired equivalence of strong normalisation:

Corollary 4.12. $\mathsf{sn}_{\rightsquigarrow_u}(s)$ *iff* $\mathsf{sn}_{\succ}(s)$.

4.6 Termination of the Distribution Calculus

We turn to the core of the termination proof, showing strong normalisation of the distribution rules, $\mathsf{sn}_{\succ}(s)$. We start with an intuitive overview of the proof and then give a formal description. Although we show strong normalisation of the distribution rules, we will here outline the proof for whole unified reduction – this will give us stronger inductive hypotheses, and we will later swap out the statement with Lemma 4.12 when we actually have to examine reduction paths.

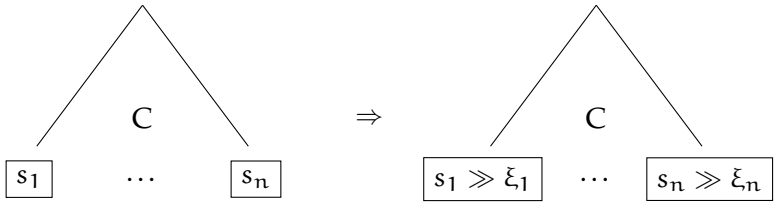

Figure 4.7: Extension of a term in context C with subterms s_i by renamings ξ_i.

With the usual approach by induction on s, we require that strong normalisation is preserved throughout various syntactic operations. In many cases, this is in fact true:

Lemma 4.13. *Assume that* $\mathsf{sn}_{\rightsquigarrow_u}(s)$ *and* $\mathsf{sn}_{\rightsquigarrow_u}(t)$. *Then* $\mathsf{sn}_{\rightsquigarrow_u}(c)$, $\mathsf{sn}_{\rightsquigarrow_u}(\lambda.s)$ *and* $\mathsf{sn}_{\rightsquigarrow_u}(s,t)$.

Proof. By a nested induction on $\mathsf{sn}_{\rightsquigarrow_u}(s)$ and $\mathsf{sn}_{\rightsquigarrow_u}(t)$. □

A similar statement for composition — if $\mathsf{sn}_{\rightsquigarrow_u}(s)$ and $\mathsf{sn}_{\rightsquigarrow_u}(t)$, then $\mathsf{sn}_{\rightsquigarrow_u}(s \gg t)$ — cannot be proven directly. The culprit is the distribution rule corresponding to abstraction reduction: To show that $\mathsf{sn}_{\rightsquigarrow_u}(\lambda.(c, s \gg c))$, we need to know that $\mathsf{sn}_{\rightsquigarrow_u}(s \gg c)$ — nothing an inductive hypothesis covers. Although this claim *seems* to be covered by the inductive hypothesis (after all, c is a restricted term), the proof requires several generalisations before it goes through. We call the generalisation of this theorem **preservation**.

As first generalisation, under an abstraction, we require that also $s \gg ((c, c) \gg c)$ is strongly normalising. We thus have to generalise c to arbitrary **renaming expressions** ξ, which here are expressions without abstraction. From now on we will use ξ and ζ to range over renaming expressions.

This is still not general enough, consider associativity:

$$(s_1 \gg s_2) \gg \xi \succ s_1 \gg (s_2 \gg \xi)$$

Although $s_2 \gg \xi$ is strongly normalising by the inductive hypothesis, we cannot conclude the same for the whole term $s_1 \gg (s_2 \gg \xi)$ as $s_2 \gg \xi$ is not necessarily a renaming. However, Curien et al. [30] come to the rescue: The authors observe that the term ξ is applied to s_2 at a *deeper position* of the term. It is thus possible (and necessary) to generalise our above claim again. See Figure 4.7 for a visualisation of this extions: we **extend** a term s with arbitrary subterms s_i in a **context** C by renamings ξ_i. Contexts C describe the positions renamings can be pushed into; we will make them precise in the following.

The proof of preservation then proceeds by a nested induction, decreasing on: 1.) the strong normalisation of the original term s, 2.) the depth of the applications of the renamings, or equivalently the size of the terms s_i, 3.) the strong normalisation of the extensions ξ_i. In the following, we collect the necessary building blocks to make this claim precise.

$$\frac{}{\mathsf{Ren}\ c} \qquad \frac{\mathsf{Ren}\ s \quad \mathsf{Ren}\ t}{\mathsf{Ren}\ (s,t)} \qquad \frac{\mathsf{Ren}\ s \quad \mathsf{Ren}\ t}{\mathsf{Ren}\ s \gg t}$$

Figure 4.8: Renaming expressions.

4.6.1 Renaming Expressions

As outlined before, the right side of a composition has to describe a renaming expression, short $\mathsf{Ren}\ s$. See Figure 4.8 for a definition. To continue with the proof, we require a range of properties. First, renamings are closed under unified reduction.

Fact 4.14. *If* $\mathsf{Ren}\ s$ *and* $s \rightsquigarrow_u t$, *then* $\mathsf{Ren}\ t$.

In contrast to full expressions, reduction on renaming expressions is terminating:

Fact 4.15 (SN of Renaming Expressions). *All renaming expressions* s *are strongly normalising w.r.t. unified reduction.*

Proof. By induction on $\mathsf{Ren}\ s$. The previously sketched proof goes through using the facts in Lemma 4.13, as we never encounter abstraction reduction. In the inductive cases, we require that renamings are closed under unified reduction (Fact 4.14). □

Contexts. Second, our proof relies on the possibility to describe subterms and positions of the original term. Contexts allow us to describe the position i into which our renamings ξ_i can be pushed:

$$C \in \mathsf{ctx} \ := \ [] \mid c \mid \lambda.C \mid (C_1, C_2) \mid s \gg C \mid C \gg \xi$$

The position of a renaming will be denoted with a hole $[]$. Renamings can be propagated into both components of a pair, (C_1, C_2). Compositions have to be treated specially: Note that the distribution rule only shifts renamings from left to right:

$$(s \gg t) \gg u \succ s \gg t \gg u.$$

Hence, we only have $s \gg C$. For abstraction reduction, we may push a context on the left side of a renaming, $C \gg \xi$.[1]

4.6.2 Patterns

Contexts define the skeleton of a term; **patterns** describe the subterms s_i. A pattern can be either a term s, also called a **leaf**, a **singleton pattern** $\langle p \rangle$ or a **pair pattern** $\langle p_1, p_2 \rangle$:

$$p \in \mathsf{pat} \ := \ s \mid \langle p \rangle \mid \langle p_1, p_2 \rangle$$

Renaming Patterns $(\mathsf{Ren}_\mathsf{p}\ p)$

$$\frac{\mathsf{Ren}\ s}{\mathsf{Ren}_\mathsf{p}\ s} \qquad \frac{\mathsf{Ren}_\mathsf{p}\ p}{\mathsf{Ren}_\mathsf{p}\ \langle p\rangle} \qquad \frac{\mathsf{Ren}_\mathsf{p}\ p \quad \mathsf{Ren}_\mathsf{p}\ q}{\mathsf{Ren}_\mathsf{p}\ \langle p, q\rangle}$$

Figure 4.9: Renaming patterns.

$$\begin{aligned}
[][s]_C &= \lfloor s\rfloor \\
c[_]_C &= \lfloor c\rfloor \\
(C_1, C_2)[\langle p, q\rangle]_C &= \lfloor (s,t)\rfloor && \text{if } C_1[p]_C = \lfloor s\rfloor \text{ and } C_2[q]_C = \lfloor t\rfloor \\
\lambda.C[\langle p\rangle]_C &= \lfloor \lambda.s\rfloor && \text{if } C[p]_C = \lfloor s\rfloor \\
(\sigma \gg C)[\langle p\rangle]_C &= \lfloor \sigma \circ \tau\rfloor && \text{if } C[p]_C = \lfloor \tau\rfloor \\
(C \gg \xi)[\langle p\rangle]_C &= \lfloor \sigma \circ \xi\rfloor && \text{if } C[p]_C = \lfloor \sigma\rfloor \\
C[p]_C &= \emptyset && \text{otherwise}
\end{aligned}$$

Figure 4.10: Filling of a context.

A **renaming pattern** (see Figure 4.9 for the definition) contains only leaves with renaming expressions.

The partial function $\_[\_]_C :: \mathsf{ctx} \to \mathsf{pat} \to \mathcal{O}(\mathsf{uexp})$ describes how a context C and pattern p_s combine to a term $C[p_s]_C$. The function is partial in case the pattern does not fit. See Figure 4.10 for its definition. If $C[p_s]_C$ results in a term s, we say that p_s is a **pattern for** s **in context** C.

To state the extension of subterms by renamings, tentatively denoted by $s_i \gg \xi_i$ in the introduction of this section, we require component-wise compositions of patterns, written $\mathsf{comp}\ p_s\ p_\xi\ p_{s\gg\xi}$ (Figure 4.11). Each leaf s_i of a pattern p_s is extended by a respective leaf ξ_i of a renaming pattern p_ξ. Composition is a partial relation, which only yields a result if the pattern p_s and p_ξ match. We also say that p_s and p_ξ **compose to** $p_{s\gg\xi}$.

[1] We will see that it is essential that we know that ξ is a renaming.

$$\frac{}{\mathsf{comp}\ s\ t\ (s \gg t)} \qquad \frac{\mathsf{comp}\ p_1\ p_2\ p_3}{\mathsf{comp}\ \langle p_1\rangle\ \langle p_2\rangle\ \langle p_3\rangle} \qquad \frac{\mathsf{comp}\ p_1\ p_2\ p_3 \quad \mathsf{comp}\ q_1\ q_2\ q_3}{\mathsf{comp}\ \langle p_1, q_1\rangle\ \langle p_2, q_2\rangle\ \langle p_3, q_3\rangle}$$

Figure 4.11: Pattern composition.

Pattern Reduction $(p \rightsquigarrow_p q)$

$$\frac{s \rightsquigarrow_u t}{s \rightsquigarrow_p t}$$

Split Reduction $(p \rightsquigarrow_{\mathsf{split}} q)$

$$\begin{aligned}(s, t) &\rightsquigarrow_{\mathsf{split}} \langle s, t\rangle\\ \lambda\, s &\rightsquigarrow_{\mathsf{split}} \langle s\rangle\\ s \gg t &\rightsquigarrow_{\mathsf{split}} \langle t\rangle\end{aligned}$$

Figure 4.12: Pattern reduction, congruence rules omitted.

4.6.3 Reduction on Patterns

Last, we need a possibility to express that the depth of application increases and reduction in the extensions ξ_i terminate.

We start with the termination of the extensions ξ_i. We define **pattern reduction** as reduction in the leaves of a pattern (Figure 4.12); as always all rules can be applied at all positions and the congruence rules are omitted.

A pattern is strongly normalising w.r.t. pattern reduction if all of its leaves are strongly normalising w.r.t. unified reduction, and as a consequence every renaming pattern is strongly normalising w.r.t. pattern reduction:

Fact 4.16. *Assume that* $\mathsf{sn}_{\rightsquigarrow_u}(s)$, $\mathsf{sn}_{\rightsquigarrow_p}(p)$, *and* $\mathsf{sn}_{\rightsquigarrow_p}(p')$. *Then also* $\mathsf{sn}_{\rightsquigarrow_p}(s)$, $\mathsf{sn}_{\rightsquigarrow_p}\langle p\rangle$, *and* $\mathsf{sn}_{\rightsquigarrow_p}\langle p, p'\rangle$.

Lemma 4.17. *If* p *is a renaming pattern, then* $\mathsf{sn}_{\rightsquigarrow_p}(p)$.

Proof. By induction on being a renaming pattern, using Lemma 4.16. □

The second notion of size is concerned with the depth of the context a renaming is applied to. For a fixed term, this is equivalent to the size of the terms s_i extended with renamings ξ_i. We use **split reduction** $p \rightsquigarrow_{\mathsf{split}} p'$ to split up these terms (Figure 4.12). Split reduction can be applied in an arbitrary context. Each reduction splits up the terms a renaming is applied to. For example, a leaf with a pair (s_1, s_2) can be split up into a pattern pair of two leaves $\langle s_1, s_2\rangle$.

As each term is finite, this process has to terminate at some time:

Fact 4.18. *Assume that* $\mathsf{sn}_{\rightsquigarrow_u}(s)$, $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}(p)$, *and* $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}(p')$. *Then also* $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}(s)$, $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}\langle p\rangle$, *and* $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}\langle p, p'\rangle$.

Corollary 4.19. *For all patterns* p, $\mathsf{sn}_{\rightsquigarrow_{\mathsf{split}}}(p)$.

4.6.4 Preservation

We have now finally assembled all parts to prove the generalisation of preservation.

Theorem 4.20 (Preservation). *Let s be a strongly normalising term w.r.t. distribution, let p_s be a pattern for s in context C. Let further p_ξ be a renaming pattern, $p_{s \gg \xi}$ be a pattern, such that p_s and p_ξ compose to $p_{s \gg \xi}$, and let $p_{s \gg \xi}$ be a pattern for t in context C. Then t is strongly normalising w.r.t. distribution.*

Proof. By a nested induction on 1.) the strong normalisation of s, 2.) split normalisation of p_s (Lemma 4.19), and 3.) unified reduction of p_ξ (Lemma 4.17). For strong normalisation of t, we assume some t' with $t \succ t'$ and show that t' is strongly normalising.

Depending on $t \succ t'$ there are three possibilities how we get smaller, each resulting in an appropriate term s, context C, patterns p_s, p_ξ, and $p_{s \gg \xi}$, satisfying the above conditions, and additionally that $p_{s \gg \xi}$ is a pattern for t' in context C such that we can use the actual inductive hypothesis.

(C1) There is some s' such that $s \rightsquigarrow_u s'$, with new C, p_s, p_ξ, and $p_{s \gg \xi}$.

(C2) There is some p_s with $p_s \rightsquigarrow_{\mathsf{split}} p_{s'}$, and new C, p_ξ, and $p_{s \gg \xi}$.

(C3) There is some $p_{\xi'}$ with $p_\xi \rightsquigarrow_p p'_\xi$ and new $p_{s \gg \xi}$.

Note that (C1) is actually by an induction on the statement of strong normalisation with unified reduction, but we later change this to strong induction by distribution only.

In each case, strong normalisation follows with the appropriate inductive hypothesis. The remainder of the proof shows that always one of these cases holds by induction on the context C.

Case $C = []$. We have $p_s = s$, $p_\xi = \xi$, and $p_t = s \gg \xi$, i.e. p_t is an actual expression. We thus assume that $s \gg \xi \succ t'$. Case analysis. If $s \succ s'$ or $\xi \succ \xi'$, we are immediately done with condition (1) or (3) respectively.

Otherwise, $s \gg \xi$ does a step. In each case, we can split the pattern, and (2) holds. We demonstrate this process on abstraction reduction, $(\lambda.s) \gg \xi \succ \lambda.(s \gg (c, \xi \gg c)) =: t$. We split up the pattern from $\lambda.s$ to $p_s := \langle s \rangle$ and choose $p_\xi := \langle (c, \xi \gg c) \rangle$ as our renaming pattern. The two patterns compose to $p_t = \langle s \gg (c, \xi \gg c) \rangle$. Further, p_s is a pattern for $\lambda.s$ in the context $C := \lambda.[]$, $C[\langle s \rangle]_C = \lambda.s$, and so is p_t for t, $C'[p_{t'}]_C = \lambda.(s \gg (c, \xi \gg c))$.

The other cases follow analogously.

Case $C = (C_1, C_2)$. We thus have $p_s = \langle p_{s_1}, p_{s_2} \rangle$, $p_\xi = \langle p_{\xi_1}, p_{\xi_2} \rangle$, and $p_t = \langle p_{t_1}, p_{t_2} \rangle$. The only way $(C_1[p_{t_1}]_C, C_2[p_{t_2}]_C)$ reduces is if $C_i[p_{t_i}]_C$ steps for $i \in \{1, 2\}$.

In each case, we use the appropriate inductive hypothesis for C_i and thus know that the step of $C_i[p_{t_i}]_C$ comes with condition (C1), (C2), (C3). Each of this steps and side conditions can be propagated into the larger context of C_1 and C_2 (Facts 4.134.164.18).

Case $C = \lambda.C$. Analogous to pairs of contexts.

Case $C = u \gg C$. Thus $p_s = \langle p_s \rangle$, $p_\xi = \langle p_\xi \rangle$, and $p_t = \langle p_t \rangle$. Case analysis on $u \gg C[p_t]_C \succ t'$. If $u \succ u'$, we are immediately done with (C1); if the step happens in $C[p_t]_C$ we use the inductive hypothesis as above.

It remains to prove that our claim holds in the case of a step. Note that for compositions, the possibility of a step only depends on the syntactic structure of u, and so if $u \gg C[p_t]_C$ steps, also $u \gg C[p_s]_C$ does, and we are done with (1).

Case $C = C \gg \zeta$. Again, $p_s = \langle p_s \rangle$, $p_\xi = \langle p_\xi \rangle$, and $p_t = \langle p_t \rangle$, and thus we consider $C[p_t]_C \gg \zeta \succ t'$. If either $C[p_t]_C$ or ζ steps, our claim holds by the IH or (C1), and we remain again with reduction.

We do a case analysis on C. In most cases, C dictates the structure of $C[p_t]_C$ and $C[p_s]_C$, in which case already $C[p_s]_C \gg \zeta$ did a step. There is one exception: If $C = []$ it can happen that $[][p_t]_C = (s \gg \xi) \gg \zeta$ and $[][p_s]_C = s \gg \zeta$. So, $[][p_t]_C$ does a step, which s cannot simulate directly. However, we *can* step[2] from $s \gg \zeta$ to s and show that our claim follows for term s and the renaming pattern (here, it is crucial that ζ is indeed a renaming!) $\xi \gg \zeta$. □

4.6.5 Termination

With this result the remainder of our proof is straightforward:

Corollary 4.21. *If* $\mathsf{sn}_{\rightsquigarrow_u}(s)$, *then* $\mathsf{sn}_{\rightsquigarrow_u}(s \gg c)$.

Proof. Using preservation with context $C := []$, the term pattern s, and renaming pattern c. □

Corollary 4.22. *If* $\mathsf{sn}_{\rightsquigarrow_u}(s)$ *and* $\mathsf{sn}_{\rightsquigarrow_u}(t)$, *then* $\mathsf{sn}_{\rightsquigarrow_u}(s \gg t)$.

Proof. By nested induction on $\mathsf{sn}_{\rightsquigarrow_u}(s)$ and $\mathsf{sn}_{\rightsquigarrow_u}(t)$, using the above Corollary 4.21. □

Theorem 4.23 (Termination of Unified Reduction). *Every unified expression is strongly normalising w.r.t. unified reduction.*

Proof. By induction on s, using Lemma 4.13 and the above Corollary 4.22. □

Corollary 4.24 (Termination). *For all expressions* s *and substitution expressions* σ, $\mathsf{sn}_{\rightsquigarrow}(s)$ *and* $\mathsf{sn}_{\rightsquigarrow}(\sigma)$.

Proof. With Lemma 4.7 and strong normalisation of unified expressions (Theorem 4.23). □

[2] This requires that we did our original induction on $\mathsf{sn}_{\rightsquigarrow_u}(s)$ which yields a stronger induction hypothesis. We can later change this condition because both statements are equivalent.

4.7 Convergence

Once we know that strong normalisation holds, the σ_{SP}-calculus is confluent using Newman's lemma:

Corollary 4.25 (Confluence). *Reduction on the σ_{SP}-calculus is confluent.*

Proof. Directly with local confluence (Lemma 4.4) and termination (Corollary 4.24), using Newman's Lemma (Lemma A.4). □

4.8 Discussion

4.8.1 De Bruijn Algebra as Models for Sigma Calculi

The σ_{SP}-calculus was first proposed as a base for reasoning on the de Bruijn algebra by Schäfer et al. [99]. Schäfer et al. also first show that the de Bruijn algebra is a semantic model of the σ_{SP}-calculus. This forms the basis for the Autosubst 1 tool by Schäfer, Tebbi, and Smolka [100]. The tactic `asimpl` normalises terms according to the rules of the σ_{SP}-calculus using rewriting.

Completeness is what makes this approach for reasoning about syntax unique. We know of no similar result for other reasoning methods on syntax.

4.8.2 Calculi of Explicit Substitutions

Calculi of explicit substitutions describe a whole family of calculi:

The original σ-calculus was proposed by Abadi et al. [2]. It is sound but not complete for the de Bruijn algebra as no rules correspond to the η-laws. It is further not confluent, as the supplementary laws such as right identity are missing. For instance, the ground equation

$$0[\uparrow] \cdot (\uparrow \circ \uparrow) = \uparrow$$

cannot be shown.

The $\sigma_{\Uparrow}$-calculus (also: the λ_{Env}-calculus) was introduced by Hardin and Lévy [56] and requires the additional $\Uparrow$-operator, representing lifting. Again, this calculus is sound but not complete for de Bruijn algebra, mainly because the interference of lifting and other substitution primitives is not specified enough. Still, there are no η-reduction laws. The $\sigma_{\Uparrow}\eta$-calculus of Hardin [54] has lifting and the η-rules; but again interference is not strong enough to be complete.

The σ_{SP}-calculus of Curien et al. [31] was introduced as a confluent "repairing" calculus of the original σ-calculus. It is sound and complete for the de Bruijn algebra as shown in [99].

For all calculi, e.g. the σ-calculus, we precompose a λ to denote the addition of β-reduction, e.g. λσ. Different to the standard literature, in this thesis, we always consider the properties of a theory *without* β-reduction, which is an entirely different (and much easier) story. For example, σ is terminating but λσ is not [75].

There are three formalisations of the meta-theory of calculi of explicit substitutions: First, there is a formalisation of termination for the $\sigma_{\Uparrow}$-calculus in Coq by Saïbi [96] (this proof is significantly easier than for the σ-calculus). The development further contains a proof of confluence of $\lambda\sigma_{\Uparrow}$ using Yokouchis lemma, following Curien et al. [31]. Second, there is an ALF development formalising strong normalisation of the σ-calculus, see below [65]; and third, there is a formalisation of a named calculus of explicit substitutions [34].

4.8.3 Termination

Different variants of the σ-calculus have a different difficulty in the proofs of termination. For example, the $\sigma_{\Uparrow}$-calculus has a straightforward proof which requires just a natural termination function, see Curien et al. [31]. This proof was mechanised by Saïbi [96]. Note that we cannot use this calculus as it is not complete for the de Bruijn algebra, and the proof is impossible for the stronger σ_{SP}-calculus. Even on the simpler subsystem, the termination function does not work because of abstraction.

We hence need a more involved proof, in our case following the one of Curien et al. [30] for the σ-calculus. The proof follows the following lines: The authors declare general contexts C and later show that if a term is strongly normalising, also an inflation of this context — depending on the structure of the subterm, a renaming is added — is strongly normalising. However, this approach does not only require to define a rather arbitrary notion of inflation but also several restrictions on the syntactic form of these contexts to so-called good contexts and very good contexts.

Our proof is based on the same central idea, but simplifies the proof in several steps. There are two central simplifications: First, we built in an additional stage of the distribution calculus; second, we simplified the representation of contexts, inflation, and composition. For example, we always extend by a renaming and do not let this depend on the fact whether the leaf is a renaming. For this part, it is crucial that we only handle the distribution rules. A similar lemma to exchangeability already turns up in Curien et al. [30], there to extend termination of the σ-calculus to termination of the σ_{SP}-calculus.

There are further small improvements: Our unified syntax is slightly more general and unifies more syntactic objects. Moreover, during preservation, we show that each step fulfils the properties. In contrast, Curien et al.' s proof relies on a lemma based on good and very good contexts.

In total, termination alone required only 750 lines of code. A substantial fraction of time

was spent in simplifying the proof to its main idea. It could be that the last part could be simplified with the proof by Zantema [121]; this is left for future work.

Termination of the σ-calculus has been proven in ALF [72] by Kamareddine and Qiao [65]. In contrast to us, the proof follows precisely the outline of Curien et al. [30].

Part II

From HOAS to de Bruijn Syntax

Chapter 5

EHOAS Specifications

In this second part of the thesis, we introduce the Autosubst compiler, a tool which generates substitution support for custom syntax. This part is split into four chapters:

First, we need a specification language to describe syntactic systems with binders, presented in this chapter. We continue with the interpretation into a suitable de Bruijn algebra; in Chapter 6 we show how this de Bruijn algebra can be established for wide classes of syntax. Modular syntax is handled via injections and is established in Chapter 7. Last, we show how to compile a specification to a de Bruijn algebra, established in Chapter 8 and hence solve the boilerplate problem.

So let us start with our specification language. One of the earliest specification languages is the one of Ott [102], a tool designed for quick prototyping. Ott takes a specification of a syntactic system and generates textual output of the implemented system in both LaTeX and various proof assistants. Variables and binders are described with names. For example, the specification of the untyped λ-calculus given in [102] looks as follows:

```
term, t :: ''t_ ::=|
 x           ::   :: var
 | \ x . t     ::   :: lam (+ bind x in t +)
 | t 't       ::   :: app
```

Other custom specification languages for code generation such as the one for Autosubst 1 [100], Knot of Needle&Knot, or the custom language of Binders Unbound [119] support a sub- or superset of Ott, but follow the line of Ott with explicit variable constructors.

In this chapter, we choose another way based on higher-order abstract syntax, short HOAS [83]. The term HOAS mainly occurs in logical frameworks [84], where it refers to the technique to *implement* binders and instantiation with substitutions of the object logic using functions and applications of the meta-logic. See for example the type of abstraction in the λ-calculus with the negative occurrence highlighted in grey:

```
abs : (tm → tm) → tm
```

HOAS is hence a generalisation of abstract syntax trees in which negative occurrences of a sort represent binders.

However, HOAS is also suitable as a mere specification language with a natural description of binders. Interesting enough, we know of no specification language using input in higher-order abstract syntax. We appreciate that HOAS is very easy to understand and further reveals no information about the underlying binders: As binders are represented in an abstract form, they are independent of the actual realisation. However, HOAS is incompatible with a general-purpose type theory [32] and we cannot use it as an interpretation. Instead, we interpret HOAS into a first-order representation using the already introduced de Bruijn algebra.

To be more precise, in this thesis, we use a variant of HOAS, customised to the expressiveness of Autosubst: **EHOAS**, short for **extended higher-order abstract syntax**, restricts higher-order abstract syntax to second-order and non-dependent binders. At the same time, it offers language primitives for new language constructs, here, variadic and modular syntax.

The EHOAS specification language gives us the chance to describe the various syntactic systems used throughout this thesis, here, presented in increasing complexity. We start with the simply-typed λ-calculus and fix notation used in the further thesis and required to make the interpretation precise. We then continue with polyadic syntax, used in a $\lambda-$calculus with pairs and elimination of a pair via pattern matching; then present many-sorted syntax, needed in the π-calculus [76] and (call-by-value) System F [50, 91]. Next, we consider how to handle external sorts and sort constructors, appearing in a λ-calculus with pairs where eliminations are implemented via projections. EHOAS further allows external parameters, e.g. needed for parametric first-order logic [44].

There are binding constructs which go beyond usual HOAS. First, we support the binding of variadic binders, i.e. the binding of (finitely) arbitrarily many binders, required for nested lets and pattern matching, e.g. in System F with records [21]. Last, EHOAS supports the specification of modular syntax. Users can define and combine so-called features, which describe subparts of a syntax. We know of no other specification language supporting modular syntax, although Ott offers a simple form of modularity in the generation of syntax via separate compilation. However, this notion of modularity does neither imply *true modularity* as introduced in Chapter 7 nor does it support users in sharing proofs between different formal systems.

We give an overview of the whole EHOAS grammar in the last section. In the next chapters, we then implement EHOAS as the specification language for Autosubst and interpret the here-described syntactic systems by their corresponding de Bruijn algebras.

```
tm: Sort
app : tm → tm → tm
lam : (tm → tm) → tm
```

Figure 5.1: EHOAS specification for λ.

5.1 EHOAS

To make our description precise, we start and fix terms and notations. Naturally and intended, this language is similar to the one for data types or inductive types.

We start with the EHOAS specification of the untyped λ-calculus, λ, in Figure 5.1. The specification consists of two simple parts: A **sort declaration** `tm : Sort`, which declares a new syntactic **sort** `tm`, and a list of **constructor declarations** of `tm`, containing both `app : tm → tm → tm` and `lam : (tm → tm) → tm`.

As in functional languages, each **constructor declaration** of a sort x consists of the **constructor name** (here: `app` and `lam`), an **argument list**, and the **result sort** x.

Arguments are the part where we deviate from constructors in Coq. Each **argument** consists of a (possibly empty) list of **binders** in the form of syntactic sorts, appearing in negative positions, and an **argument head**. All binders are said to be **bound** in their argument head. Binders will need a translation into a first-order structure.

Different to what we have seen in the de Bruijn representation of the untyped λ-calculus, in the EHOAS specification we do not need an explicit constructor for variables. This information will be inferred automatically in the *interpretation* of the EHOAS specification. We will call a sort with a variable constructor **open**, a sort without a variable constructor is said to be **closed**.

For example, in the interpretation to scoped de Bruijn syntax, `tm` will be indeed equipped with a variable constructor `var`:

```
Inductive tm : ℕ → Type :=
| var n : 𝕀 n → tm n
| app n : tm n → tm n → tm n
| lam n : tm (1+n) → tm n.
```

Further, binders in the EHOAS specification leave us with an increased scope, here `tm (1+n)`.

5.2 EHOAS by Example

We consider the EHOAS representation of the syntactic systems used in this thesis, presented in increasing complexity. Autosubst will be able to generate a suitable interpretation in de Bruijn syntax for each of them.

```
tm: Sort

app : tm → tm → tm
lam : (tm → tm) → tm

pair : tm → tm → tm
matchpair : tm → (tm → tm → tm) → tm
```

Figure 5.2: EHOAS specification for $\lambda_\times$.

```
chan, proc : Sort

par : proc → proc → proc   -- P | Q
input : chan → (chan → proc) → proc   -- c(x).P
output : chan → chan → proc → proc   -- c̄⟨y⟩.P
repl : proc → proc   -- !P
restr : (chan → proc) → proc   - νx.P
nil : proc   -- 0
```

Figure 5.3: EHOAS specification for the π-calculus.

We start with polyadic binders. See Figure 5.2 for the EHOAS specification of the **λ-calculus with pairs**, $\lambda_\times$. Pairs require an introduction rule and an elimination rule using pattern matching. Note that the `matchpair` constructor uses two binders of sort `tm` in the argument head. We call such a repeated binding **polyadic**, in contrast to a monadic binder. The binders of a polyadic binder can be of different sorts.

EHOAS also allows to define several, possibly interdependent sorts of syntax at once. See Figure 5.3 for the EHOAS representation of the **π-calculus** [76], a process calculus which allows the communication of different processes via channels. It hence consists of both channels `chan` and processes `proc`. We have constructors for concurrency, input prefixing, output prefixing, replication, restriction, and the nil process. In this variant of the π-calculus, only channels can be bound, either during the input (where a channel is received) or during restriction (when the name of a channel is restricted).

Concerning binders, many-sorted syntax gets more interesting if the different sorts are **mutually inductive** and require several sorts of binders, short: are **many-sorted**. Figure 5.4 specifies a call-by-value variant of **System F** (short: $\mathrm{F}_{\mathsf{CBV}}$). System F, also known as the polymorphic λ-calculus, introduces parametric polymorphism into a programming language during type abstraction $\Lambda.s$. As we can see from its sort specification, it is a syntactic system with three sorts: types (`ty`) and the mutual inductive definition of terms (`tm`) and values (`vl`).

Types consist of the function type `arr` and the universal quantification `all`, which binds

```
ty, tm, vl : Sort

arr : ty → ty → ty
all : (ty → ty) → ty

app  : tm → tm → tm
tapp : tm → ty → tm
vt   : vl → tm

lam  : ty → (vl → tm) → vl
tlam : (ty → tm) → vl
```

Figure 5.4: EHOAS specification for $\mathsf{F}_{\mathsf{CBV}}$.

```
𝔹 : Sort
ℕ : Sort
Funcs : Sort
Preds : Sort

ℒ : Sort → Sort
× : Sort → Sort → Sort
𝕍 : ℕ → Sort → Sort
ar_f : Funcs → ℕ
ar_p : Preds → ℕ
```

Figure 5.5: External sorts and sort constructors.

a type within a type. Additional to the known primitives of the λ-calculus, terms offer the possibility of a type application (`tapp`) and a type abstraction (`tabs`), binding a type in a term. Note that in the call-by-value variant of System F only values, not terms, are bound during abstraction, and hence in the interpretation only values will be open. Terms may be embedded into values via the embedding constructor `vt`.

When specifying a syntactic system, we want to redefine neither sorts nor sort constructors independent of binders; sometimes we want to use **external sorts**. See Figure 5.5 for the type signature of external sorts used in future specifications: We consider booleans, natural numbers, and last (abstract types for) function symbols (`Funcs`), and predicate symbols (`Preds`). The list also contains the type signature of **external sort constructors** and functions: lists, products, vectors, and arity functions for the function and predicate symbols. In the implementation, external sorts will be essential to reuse functions from the standard library. We assume that each external sort is closed and does not depend on a current definition.

See Figure 5.6 for a specification of $\lambda_\times$ with an alternative elimination rule via projections. We can eliminate a projection using a boolean value which determines which component of a pair to choose. Note that in the specification, we did not define the constructors of booleans, although we mentioned them in the sort declaration.

```
𝔹: Sort
tm: Sort

app : tm → tm → tm
lam : (tm → tm) → tm

pair : tm → tm → tm
proj : 𝔹 → tm → tm
```

Figure 5.6: EHOAS specification for $\lambda_\times$.

```
ℕ : Sort                    ⊤ : ty
ty : Sort                   arr : ty → ty → ty
ℒ : Functor                 all : (ty → ty) → ty
× : Functor                 recty : ℒ (ℕ× ty) → ty
```

Figure 5.7: EHOAS specification for record types.

External sort constructors, also called **functors**,[1] work similarly. See Figure 5.7 for the example of record types which extends the types of System F with a maximum type $\top$, and a record type `recty`. A record type is a list of record type positions, each consisting of a label (represented by a natural number) and again a type. Functors need to be declared in a **functor declaration**, e.g. $\mathcal{L}$: `Functor` for lists. They may only appear in the argument head, not as a binder.

Functors may be arbitrarily nested, but have to terminate with primitive sorts which we call the **arguments** of the functors. For example, for record types, $\mathbb{N}$ and `ty` are the arguments of $\mathcal{L}$ $(\mathbb{N} \times \texttt{ty})$. If the argument head is a functor with several arguments, the binders are binders for all arguments.

EHOAS further allows **substitution-independent parameters**, needed for example for **parametric first-order logic**, FOL (Figure 5.8). Parametric first-order logic consists of two sorts, terms (term) and formulas (form). We characterise the allowed functions and predicates via parametric types `Funcs` and `Preds`. Both kinds of symbols come with an arity, which can be accessed via the functions ar_f: `Funcs` $\to \mathbb{N}$ and ar_p: `Preds` $\to \mathbb{N}$. We allow their application in the constructors `func` and `pred`, which take a vector of the corresponding arity in each case. In first-order logic, we can quantify over terms with either a universal ($\forall$) or an existential ($\exists$) quantifier. In each case, we bind a term in a formula and return a formula.

For example, we could define a signature which contains a constant and binary operation, and one nullary and one unary predicate. See Figure 5.9 for the specialised

[1] The term "functor" is induced by the later implementation for external sort constructors, see Section 6.3.

```
Funcs, Preds, term, form : Sort
𝕍 : Functor

func (f : Funcs) : 𝕍 (ar_f f) term → term
pred (P : Preds) : 𝕍 (ar_p P) term → form
∀, ∃ : (term → form) → form
```

Figure 5.8: EHOAS specification for parameterised FOL.

```
term, form : Sort

c : term
binop : term → term → term

P_0 : form
P_1 : term → form
∀, ∃ : (term → form) → form
```

Figure 5.9: EHOAS specification for FOL*.

EHOAS specification, which we call simplified first-order logic, or short FOL*. In its parameterised form, we would have Funcs := Preds := $\mathbb{B}$. In the case of predicates, true represents the nullary and false represents the unary predicate. Then the arity function returns 0 for true and 1 otherwise; analogously for function symbols.

We start with **variadic syntax**. Consider the EHOAS specification of a λ-calculus which does not take only one argument during abstraction, but an arbitrary number of p arguments. We call this system the **multivariate λ-calculus**, λ_ν [90]. See Figure 5.10 for the corresponding EHOAS specification. In the case of abstraction, we bind p binders of sort x at once, $\mathbb{V}\,p\,x$. Note that p is unknown during the definition. Different to the usual λ-calculus, an application may take not only a single argument for application but a whole vector of terms. Note that despite our use of vector notation, each variable is still bound separately. Parameters may then be used in other definitions, e.g. in functors. For a variadic binder $\mathbb{V}\,p\,x$, x is the binder head.

Variadic variables are also usable in another context: consider the input syntax of

```
tm : Sort
app p : tm → 𝕍p tm → tm
lam p: (𝕍 p tm → tm) → tm
```

Figure 5.10: EHOAS specification for λ_ν.

```
ty, tm, pat, ℕ : Sort
ℒ, × : Functor

⊤ : ty
arr : ty → ty → ty
all : (ty → ty) → ty
recty : ℒ (ℕ × ty) → ty

patvar : ty → pat
patℒ : ℒ (label × pat) → pat

app  : tm → tm → tm
tapp : tm → ty → tm
abs  : ty → (tm → tm) → tm
tabs : ty → (ty → tm) → tm
rectm : ℒ (ℕ × tm) → tm
proj : tm → ℕ → tm
letpat p : pat → tm → (𝕍 p tm → tm) → tm
```

Figure 5.11: EHOAS specification for $\mathsf{F}_{<:}$.

```
exp, 𝔹, ℕ : Type

begin lam
  lam : (exp → exp) → exp
  app : exp → exp → exp
end lam

begin booleans
  constBool : 𝔹 → exp
  if : exp → exp → exp → exp
end booleans

begin arith
  constNat : ℕ → exp
  plus : ℕ → ℕ → exp
end arith

compose lambdas := lam
compose booleans := lam :+: bool
compose arith := lam :+: arith
compose all := lam :+: bool :+: arith
```

Figure 5.12: EHOAS specification for modular syntax.

System F with records, $\mathsf{F}_{<:}$ (Figure 5.11), with types and terms as described by the POPLMark challenge [12]. We have already seen the definition of record types before. We enrich the terms with an additional constructor for record terms, consisting of a list of element labels and their respective terms. We can project from a record element and a label to a term (`proj`). A let constructor takes a pattern and two terms, one terms binds a variadic number p of variables, where p is a natural number used for counting.

As we do not know in beforehand how many variables will be bound during the pattern matching process, we require a variadic binder. With the p bound terms, we have a vector of p types, annotating the types of the matched terms, and p lists of labels, describing the path to a subcomponent of the record.

5.3 Modular Syntax

Last, EHOAS allows the specification of **modular syntax**. For this example, assume that we start with the EHOAS specification of the untyped λ-calculus as in Figure 5.1.

We then want to extend expression independently with booleans and natural numbers;

$$
\begin{aligned}
\mathsf{decls} &:= \overline{\mathsf{decl}} \\
\mathsf{decl} &:= \mathsf{sortDecl} \mid \mathsf{constrDecl} \mid \mathsf{functorDecl} \mid \mathsf{featureDecl} \mid \mathsf{composeDecl} \\
\mathsf{sortDecl} &:= x : \mathsf{Sort} \\
\mathsf{functorDecl} &:= F : \mathsf{Functor} \\
\mathsf{constrDecl} &:= \mathsf{cname}[\mathsf{param}] : \overline{\mathsf{arg}} \to x \\
\mathsf{featureDecl} &:= \mathsf{begin}\ f\ \overline{\mathsf{constrDecl}}\ \mathsf{end}\ f \\
\mathsf{composeDecl} &:= \mathsf{compose}\ E := \overline{f} : + : f \\
\mathsf{param} &:= \overline{(c : c')} \\
\mathsf{arg} &:= \overline{\mathsf{binder} \to \mathsf{arghead}} \\
\mathsf{binder} &:= x \mid \mathbb{V}\, p\, x \\
s \in \mathsf{arghead} &:= x \mid F\, c\, (s_1, \ldots, s_n)
\end{aligned}
$$

Figure 5.13: Grammar for EHOAS specifications.

for example, to obtain proofs for different variants of this calculus. On paper, we would define these extensions as follows:

$$
\begin{aligned}
s, t, u : \mathsf{exp}_{\mathbb{B}} &::= \cdots \mid b \mid \mathsf{if}\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \\
s, t : \mathsf{exp}_{\mathbb{N}} &::= \cdots \mid n \mid s + t \\
s, t, u : \mathsf{exp}_{\mathbb{B},\mathbb{N}} &::= \cdots \mid b \mid \mathsf{if}\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \mid n \mid s + t
\end{aligned}
$$

We mirror this situation and allow users to define syntax modularly for mechanised proofs.[2]

See Figure 5.12 for the corresponding input file in EHOAS. From the above explanation, we introduce two new syntactic constructions: **features**, which introduce a partial specification of a list of sorts; and **variants**, which combine several features to different variants of the appearing sorts.

Each feature (here: `lam`, `booleans`, and `arith`) is surrounded by a `begin`... `end` block. In this case, we require four different variants, each introduced by `compose` and a list of features: One which consists of only the feature for λ-expressions, one with booleans, one with arithmetic expressions, and one containing all sorts of expressions.

5.4 A Grammar for EHOAS

Figure 5.13 contains the complete grammar of EHOAS. Each EHOAS specification consists of a list of declarations, where each declaration declares a sort, `sortDecl`, a con-

[2] In Chapter 7, we further explain how to construct modular proofs over the generated syntax.

structor, `constrDecl`, a functor, `functorDecl`, a feature `featureDecl`, or a composed sort, `composeDecl`.

In the following, `x` ranges over sorts, `cname` ranges over constructor names, `F` ranges over functors, `f` ranges over features, `S` ranges over composed sorts, and `c` ranges over arbitrary external terms and types. A sort declaration defines a new sort `x`; a functor declaration defines a new functor `F`. On the technical side, let us remark that there is no need to sort constructor declarations according to their sorts; Autosubst automatically detects the dependencies of mutual inductive sorts. A constructor declaration takes a constructor name `cname`, a list of parameters `param` (which is a list of sort declarations with external sorts), and then maps a list of arguments `arg` to a sort `x`.

Again, each argument consists of a list of binders mapped to an argument head. Each binder can be either a simple binder (denoted via a sort `x`) or a variadic binder, `𝕍 p x` which variadically binds `p` parameters of sort `x`. Note that we do not allow to bind any functor expressions or arbitrary types. Last, the argument head can be either a simple sort or the functor application on arbitrarily many external arguments and a list of again argument heads, separated by commas and collected in parentheses.

We turn to the EHOAS specification of modular syntax. As described before, each feature declaration is embedded in a `begin` ... `end` block with the name of the feature `f`. It then consists of a list of constructor declaration, of not necessarily the same main argument. Last, for a composed sort, `composeDecl`, we have a (non-empty) list of features combine by :+:.

The above specification can describe all the example we have seen in the previous sections.

Chapter 6

Extended Calculi with de Bruijn Syntax

In the last chapter, we presented EHOAS, a specification language for syntax with binders. However, higher-order abstract syntax is inherently incompatible with type theory [32] and we hence use a first-order realisation of the previous systems based on de Bruijn syntax. In Chapter 3, we have seen this realisation for the untyped λ-calculus. In this chapter, we give a first look at the definition of custom generalised de Bruijn algebras.

Before we can tackle the generalisation properly, we briefly recap the design principles for the untyped λ-calculus. First, we restricted ourselves to a finite set of primitives which accounted for the scope change of a binder. These primitives satisfied a finite set of equations. In our extensions, we might require new primitives and interaction lemmas to account for more complex scope changes; but we still restrict ourselves to a finite set of primitives.

Second, we used *one* operation of instantiation with substitutions, which accounted for the whole scope change and replaced all variables at once. We define instantiation with one operation for each sort only, even in the presence of multiple variable sorts.

Third, we provided a range of equational substitution lemmas and obtained a sound and complete proof strategy. We hence have to re-prove the monad laws and adapt the proof method in each case. Different from the λ-calculus, we focus only on the practical aspects and provide neither proofs of termination and confluence, nor proofs of soundness and completeness for a corresponding calculus of explicit substitutions.

More specifically, we generalise the substitution primitives and reasoning principles of the λ-calculus to first-class renamings, polyadic syntax, many-sorted syntax, external sorts and sort constructors, variadic syntax, and, in Chapter 7, modular syntax. We illustrate each of these generalisations with one specific formal system as described below.

Together, this chapter forms the base of the Autosubst tool. In all cases, the definition of instantiation and the proof of the substitution laws are (and have to be) so regular

that we can generate the corresponding code. We highlight these regularities in this chapter. The implementation of Autosubst is handled in Chapter 8 and requires additional information on the structure of the syntactic systems we here assume to be readily available.

In this chapter, we use scoped syntax as introduced in Chapter 3. All extensions except variadic syntax can be (and, in fact, in Autosubst are) implemented in pure de Bruijn syntax as well. All substitution code linked in this chapter is generated by Autosubst.

The part on vector substitutions reuses parts of [62, 108].

Organisation of the Chapter. In Section 6.1, we start with *first-class renamings* and their representation in the untyped λ-calculus. Recall that in the σ_{SP}-calculus, there is no distinction between substitutions and renamings. Hence renamings are second-class. Different to that, in actual developments, it is helpful to access renamings directly (see Section 6.7.1). First-class renamings require us to extend the previous proofs to renamings, also in the equational theory. The need for an explicit representation of renamings first appeared in proofs using Kripke-style logical relations [4].

In Section 6.2, we continue and extend reasoning to *polyadic binders* on the example of a λ-calculus with pairs and elimination of pairs via pattern matching. Polyadic binders were first used in a proof development of call-by-push-value [45]. The implementation is straightforward to realise and only requires the repeated use of the lifting operations and lemmas.

External sorts and sort constructors (Section 6.3) are illustrated on the example of record types. External sort constructors are among others used for record types and hence first appeared when tackling Part 2 of the POPLMark challenge. Schäfer [97, Chapter 9.2] presents the idea of traversable containers in his thesis; we here show how proof code is generated for Autosubst.

We continue with *many-sorted syntax* as in System F, where two sorts of variables appear (Section 6.4). Instantiation hence has to substitute both type and value variables. To satisfy our requirement of one instantiation operation, we *parallelise* the previously already parallel de Bruijn substitutions to so-called **vector substitutions** [108]. Vector substitutions offer a possibility to extend our design principles to many-sorted syntax. Most interesting, we need more involved lifting lemmas than in the univariate case. This section reuses parts of a previous publication [108].

Binders in first-order logic, also known as quantifiers, only use variables of a previously known term sort. Instantiation with substitutions on terms hence collapses to parameter instantiation only. In Section 6.5, we hence consider a simplified class of binders and show how to omit the previous detour over instantiation with renamings. The need for such binders came first up in a formalisation if first-order logic by the colleagues of the author of this thesis in [46], and was then generalised and adapted to Autosubst. As far

as we know, no other system treats first-order syntax specifically.

Last, in Section 6.6, we consider *variadic syntax* on the example of the multivariate λ-calculus [90]. Variadic binders bind a previously unknown number of variables and usually appear during nested let constructions and (possibly nested) pattern matching. Because of this generalised scope change, we will — for the first time — require new substitution primitives and new reasoning principles on the primitives themselves. We further have to prove new lifting lemmas. The author of this thesis first got interested in this problem when proving Part 2 of the POPLMark challenge.

6.1 First-Class Renamings in the Lambda Calculus

In this section, we outline the additional equational laws for renamings in the de Bruijn algebra of the λ-calculus (Chapter 3, based on the EHOAS specification in Figure 5.1). Unless stated otherwise, we omit laws for renamings in all further descriptions. However, Autosubst proves and uses them in the implementation.

We have already seen the definition of instantiation with renamings, $s\langle\xi\rangle$, in Chapter 3. If renamings are-first-class, the resulting rewriting system has to hold equational laws for both instantiations with renamings and substitutions. Naturally, we start with adding the reduction equations for instantiation with renamings.

Next, we have already encountered several laws as an intermediate result for compositionality (Lemma 3.4):

$$\begin{aligned} s\langle\xi\rangle\langle\zeta\rangle &= s\langle\xi \circ \langle\zeta\rangle\rangle \\ s\langle\xi\rangle[\tau] &= s[\xi \circ \tau] \\ s[\sigma]\langle\zeta\rangle &= s[\sigma \circ \langle\zeta\rangle] \end{aligned}$$

We add these to our equational theory. Additionally, we require counterparts to the identity law and the supplementary laws:

Lemma 6.1 (Renaming Identity). $s\langle\mathsf{id}\rangle = s$.

Lemma 6.2 (Supplementary Laws for Renamings).

1. $\mathsf{var} \circ \langle\xi\rangle \equiv \xi$
2. $\sigma \circ \langle\mathsf{id}\rangle \equiv \sigma$
3. $(\sigma \circ \langle\xi\rangle) \circ \langle\zeta\rangle \equiv \sigma \circ \langle\xi \circ \zeta\rangle$
4. $(\sigma \circ \langle\xi\rangle) \circ [\theta] \equiv \sigma \circ [\xi \circ \theta]$
5. $(\sigma \circ [\tau]) \circ \langle\zeta\rangle \equiv \sigma \circ [\tau \circ \langle\theta\rangle]$

All these are proven similar to the respective substitution laws.

Last, we use the coincidence law between instantiation with renamings and substitutions, not only as a statement but also in our equational theory:

$$s[\xi \circ \mathsf{var}] = s\langle\xi\rangle$$

As we rewrite from left to right, renamings are preserved unless the user explicitly decides to do otherwise (see Section 8.3.2).

6.2 Polyadic Binders in the Lambda Calculus with Pairs

We continue with an example for polyadic binders. Recall the λ-calculus with pairs, $\lambda_\times$, as specified in Figure 5.2. Elimination of a pair via pattern matching binds two variables and hence produced a polyadic binder. In the scoped representation, the scope of the term t is thus increased by 2:

$$s, t \in \mathsf{tm}^k \;:=\; \mathsf{var}\, x \mid \mathsf{app}\, s^k\, t^k \mid \lambda x.s^{1+k} \mid (s^k, t^k) \mid \mathsf{match}\, s^k\, \mathsf{in}\, t^{2+k} \qquad x \in \mathbb{I}^k$$

We can depict this scope change with the following picture:

$$\begin{array}{lccccccc} t & & & 0 & 1 & 2 & & \\ & & & \vert & \vert & \vert & \dots & \\ \mathsf{match}\, s\, \mathsf{in}\, t & 0 & 1 & 2 & 3 & 4 & & \end{array}$$

We will have to account for this scope change both during instantiation and the proof of the monad laws.

We start with instantiation. Instantiation, $\_[\_] : (\mathbb{I}^m \to \mathsf{tm}^n) \to \mathsf{tm}^m \to \mathsf{tm}^n$, is defined analogous to the λ-calculus and still traverses the term homomorphically (Figure 6.1). The only notable difference is that in the case of the polyadic scope change, we apply the lifting operation $\Uparrow$ twice. In t, both $0_{\mathbb{I}}$ and $1_{\mathbb{I}}$ remain unchanged, while all expressions in σ are shifted by 2. Thus $\Uparrow (\Uparrow \sigma)$ could be equivalently expressed by the following custom operation:

$$\Uparrow^2 := \mathsf{var}\, 0_{\mathbb{I}} \cdot \mathsf{var}\, 1_{\mathbb{I}} \cdot \sigma \circ [\uparrow \circ \uparrow]$$

Using unary lifting instead of a custom binary lifting, however, simplifies the proofs and avoids repetitions. The original primitives for the λ-calculus thus suffice for the new scope change as described above.

We define β-reduction on pairs as:

$$\mathsf{match}\, (s_1, s_2)\, \mathsf{in}\, t \succ t[s_1 \cdot s_2 \cdot \mathsf{var}]$$

or, with notation, $\mathsf{match}\, (s_1, s_2)\, \mathsf{in}\, t \succ t[s_1, s_2..]$.

We now turn to the equational laws; more specifically, the monad laws, the supplementary laws, coincidence, and extensionality from Chapter 3.

$$
\begin{aligned}
(\mathsf{var}\, x)[\sigma] &= \sigma\, x \\
(\mathsf{app}\, s\, t)[\sigma] &= \mathsf{app}\, (s[\sigma])\, (t[\sigma]) \\
(\lambda.s)[\sigma] &= \lambda.s[\Uparrow \sigma] \\
(\mathsf{match}\, s\, \mathsf{in}\, t)[\sigma] &= \mathsf{match}\, s[\sigma]\, \mathsf{in}\, t[\Uparrow (\Uparrow \sigma)] \\
(s, t)[\sigma] &= (s[\sigma], t[\sigma])
\end{aligned}
\qquad
\begin{aligned}
(\sigma \circ [\tau])\, x &= (\sigma\, x)[\tau] \\
&\\
\text{with } \Uparrow \sigma &= \mathsf{var}\, 0_{\mathbb{I}} \cdot \sigma \circ [\uparrow]
\end{aligned}
$$

Figure 6.1: Instantiation for $\lambda_\times$.

The lifting lemmas and their proofs remain unchanged, all that changes is the number of applications of the lifting lemma. For example, when showing the right identity law,

$$s[\mathsf{var}] = s,$$

we use Lemma 3.2 twice in the case of pair elimination :

$$
\begin{aligned}
(\mathsf{match}\, s\, \mathsf{in}\, t)[\mathsf{var}_{\mathsf{tm}}] &= \mathsf{match}\, s[\mathsf{var}]\, \mathsf{in}\, t[\Uparrow (\Uparrow \mathsf{var}_{\mathsf{tm}})] \\
&= \mathsf{match}\, s\, \mathsf{in}\, t[\Uparrow\ \mathsf{var}_{\mathsf{tm}}] \\
&= \mathsf{match}\, s\, \mathsf{in}\, t[\mathsf{var}_{\mathsf{tm}}] \\
&= \mathsf{match}\, s\, \mathsf{in}\, t
\end{aligned}
$$

The same holds for all other recursive laws requiring lifting lemmas.

To prove an equation between terms with polyadic binders, we repeatedly rewrite the previous interaction lemmas (Lemma 3.1) and the usual monad equations.

For example, we can show that β-reduction is substitutive:

Lemma 6.3. *β-reduction is substitutive.*

Proof. We have to show that

$$t[s_1 \cdot s_2..][\sigma] = t[\Uparrow (\Uparrow \sigma)][s_1[\sigma] \cdot s_2[\sigma]..].$$

Using repeatedly the defining equations of instantiation, the monad laws, and the interference laws, we reach the point with $t[s_1[\sigma] \cdot s_2[\sigma] \cdot \sigma]$ at both sides. □

6.3 External Sorts and Sort Constructors in Record Types

We recall record types specified in Figure 5.11. In their type-theoretic representation, we define them as:

$$A, B \in \mathsf{ty}^k \ ::= \ \mathsf{var}_{\mathsf{ty}}\, X \mid \top \mid A^k \to B^k \mid \forall.A^{1+k} \mid \{l_1 : A_1^k..l_n : A_n^k\} \qquad X \in \mathbb{I}^k; l_1..l_n \in \mathbb{N}$$

$$
\begin{aligned}
(\mathsf{var}_{\mathsf{ty}}\, x)[\sigma] &= \sigma\, x & (\sigma \circ [\tau])\, x &= (\sigma\, x)[\tau] \\
\top[\sigma] &= \top \\
(A \to B)[\sigma] &= A[\sigma] \to B[\sigma] \\
(\forall.A)[\sigma] &= \forall.A[\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \sigma] & \text{with } \Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \sigma &= \mathsf{var}_{\mathsf{ty}}\, 0_{\mathbb{I}} \cdot \sigma \circ [\uparrow] \\
\{l_i : A_i\}[\sigma] &= \mathsf{map}\,(\mathsf{id} \times [\sigma])\,\{l_i : A_i\}
\end{aligned}
$$

Figure 6.2: Instantiation for record types.

$$
\begin{aligned}
(\mathsf{id} \times \mathsf{id})(p) &= p \\
(f_1 \times f_2)((g_1 \times g_2)(p)) &= (g_1 \circ f_1 \times g_2 \circ f_2)(p) \\
f_1 \equiv g_1 \to f_2 \equiv g_2 \to (f_1 \times f_2)(p) &= (g_1 \times g_2)(p)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{map}\,\mathsf{id}\,xs &= xs \\
\mathsf{map}\,f\,(\mathsf{map}\,g\,xs) &= \mathsf{map}\,(g \circ f)\,xs \\
f \equiv g \to \mathsf{map}\,f\,xs &= \mathsf{map}\,g\,xs
\end{aligned}
$$

Figure 6.3: Functor laws for pairs and lists.

Record types hence consist of a list of labels (represented each by a natural number) and types.

Instantiation propagates into all components of the records. On record types, this propagation corresponds to a nested mapping on lists, written $\mathsf{map}\,f$, and mapping on a pair, written $(f \times g)$. See the definition of instantiation in Figure 6.2. Each occurring type constructor requires such a mapping function, defined before calling the Autosubst compiler.

For the monad laws to hold, the mapping functions must preserve the laws of an extensional functor: identity, composition, and extensionality. See Figure 6.3 for the functor laws for pairs and lists.

Then, record types fulfil indeed the monad laws. As before, left identity follows directly by the definition of instantiation, right identity and compositionality follow by induction on s using the already existing lifting lemmas, and the supplementary laws follow directly with the previous monad laws. For record types, we require the functor laws.

$$\begin{aligned}
\mathsf{map}\,\mathsf{id} &\equiv \mathsf{id} \\
\mathsf{map}\,f \circ \mathsf{map}\,g &\equiv \mathsf{map}\,(f \circ g) \\
(\mathsf{id} \times \mathsf{id}) &\equiv \mathsf{id} \\
(f_1 \times f_2) \circ (g_1 \times g_2) &\equiv (f_1 \circ g_1 \times f_2 \circ g_2)
\end{aligned}$$

Figure 6.4: Supplementary laws for functor laws.

For example, for identity preservation of record types we have:

$$\begin{aligned}
\{l_i : A_i\}[\mathsf{var}] &= \mathsf{map}\,(\mathsf{id} \times [\mathsf{var}])\,\{l_i : A_i\} && \text{instantiation} \\
&= \mathsf{map}\,(\mathsf{id} \times \mathsf{id})\,\{l_i : A_i\} && \text{inductive hypothesis} \\
&= \mathsf{map}\,\mathsf{id}\,\{l_i : A_i\} && \text{functor identity} \\
&= \{l_i : A_i\} && \text{functor identity}
\end{aligned}$$

Note that we use the inductive hypothesis on all elements in the list in the second step. We thus require an advanced property of the Coq termination checker, which allows us to propagate an inductive hypothesis into a list. Proofs further may not be opaque, which implies that we cannot use the definitions of the standard library.

Further, instantiation is still extensional:

Lemma 6.4. *If* $\sigma \equiv \tau$, *then* $s[\sigma] = s[\tau]$.

Proof. Similar to before, using the extensionality instance of the corresponding map functions. The proof again requires that the corresponding extensionality proofs for lists and pairs are not opaque. □

Coincidence remains unchanged and also requires no new properties of functors.

Last, to prove an equation, we add the functor laws of all occurring functors. We further require the supplementary laws for functors in Figure 6.4, again defined by the user before calling Autosubst. See Section 6.6 for an example of equational reasoning in a formal system with functors.

6.4 Many-Sorted Syntax in Call-by-Value System F

We now turn to many-sorted syntax. This will be a major extension and requires a new concept called **vector substitutions**.

We recall the call-by-value variant of System F, F_{CBV}, as described in Figure 5.4. Recall that we have two kinds of bindings: In a value abstraction, $\lambda A.s$, we bind values; in a type abstraction $\Lambda.s$, we bind types. In a first-order representation, we hence need both

$$
\begin{aligned}
(\mathsf{var}_{\mathsf{vl}}\, x)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{vl}} &= \sigma_{\mathsf{vl}}\, x \\
(\Lambda.s)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{vl}} &= \Lambda.(s[\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \sigma_{\mathsf{ty}}; \Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}}]_{\mathsf{tm}}) \\
(\lambda_A.s)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{vl}} &= \lambda_{(A[\sigma_{\mathsf{ty}}]_{\mathsf{ty}})}.(s[\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ \sigma_{\mathsf{ty}}; \Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}}]_{\mathsf{tm}})
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{app}\, s\, t)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}} &= \mathsf{app}\, (s[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}})\, (t[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}}) \\
(s\, A)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}} &= (s[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}})\, (A[\sigma]_{\mathsf{ty}}) \\
(\mathsf{vt}\, v)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}} &= \mathsf{vt}\, (v[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{vl}})
\end{aligned}
$$

$$
(\sigma_{\mathsf{vl}} \circ [\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{vl}})(x) := (\sigma_{\mathsf{vl}}\, x)[\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{vl}}
$$

$$
\begin{aligned}
\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \sigma_{\mathsf{ty}} &:= \mathsf{var}_{\mathsf{ty}}\, 0_{\mathbb{I}} \cdot \sigma_{\mathsf{ty}} \circ [\uparrow]_{\mathsf{ty}} & \qquad \Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}} &:= \sigma_{\mathsf{vl}} \circ [\uparrow; \mathsf{var}_{\mathsf{vl}}]_{\mathsf{vl}} \\
\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ \sigma_{\mathsf{ty}} &:= \sigma_{\mathsf{ty}} \circ [\mathsf{var}_{\mathsf{ty}}]_{\mathsf{ty}} & \qquad \Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}} &:= \mathsf{var}_{\mathsf{vl}}\, 0_{\mathbb{I}} \cdot \sigma_{\mathsf{vl}} \circ [\mathsf{var}_{\mathsf{ty}}; \uparrow]_{\mathsf{vl}}
\end{aligned}
$$

Figure 6.5: Term and value instantiation for F_{CBV}.

type variables $\mathsf{var}_{\mathsf{ty}}\, X$ and value variables $\mathsf{var}_{\mathsf{vl}}\, x$. While the sort of types contains only one type of variables, terms s and values v are many-sorted, i.e. contain both type and value variables. In scoped syntax, terms and values are hence indexed by the upper bound of both type and value variables:

$$
\begin{aligned}
A^k, B^k \in \mathsf{ty}^k &:= \mathsf{var}_{\mathsf{ty}}\, X \mid A^k \to B^k \mid \forall A^{1+k} && X \in \mathbb{I}^k \\
s^{k;l}, t^{k;l} \in \mathsf{tm}^{k;l} &:= s^{k;l}\, t^{k;l} \mid s^{k;l}\, A^k \mid \mathsf{vt}\, v^{k;l} \\
u^{k;l}, v^{k;l} \in \mathsf{vl}^{k;l} &:= \mathsf{var}_{\mathsf{vl}}\, x \mid \lambda_{A^k}.s^{k;1+l} \mid \Lambda.s^{1+k;l} && x \in \mathbb{I}^l
\end{aligned}
$$

Note how the different kinds of binding induce different scope changes.

Instantiation on terms and values will replace both type and value variables. A substitution has hence to account for both bindings. As a consequence, we will require new lifting operations and lifting lemmas.

In the following, we assume that type instantiation $\_\,[\_]_{\mathsf{ty}} : \mathsf{ty}^m \to (\mathbb{I}^m \to \mathsf{ty}^{m'}) \to \mathsf{ty}^{m'}$ already exists and satisfies the expected properties.

6.4.1 Instantiation

We start with the definition of term and value instantiation. Instantiation on terms $\mathsf{tm}^{m;n}$ and values $\mathsf{vl}^{m;n}$ has to replace both type and value variables. We hence require one component for type variables, $\sigma_{\mathsf{ty}} : \mathbb{I}^m \to \mathsf{ty}^{m'}$, and one component for value variables, $\sigma_{\mathsf{vl}} : \mathbb{I}^n \to \mathsf{vl}^{m';n'}$. We combine multiple de Bruijn substitutions

into a single *vector* of de Bruijn substitutions, short called a **vector substitution**. We write $s[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}}$ for a term s where all type variables are replaced according to σ_{ty} and all value variables according to σ_{vl}, and similarly for values. Hence the full type of instantiation will be:

$$\_[\_;\_]_{\mathsf{tm}} : \mathsf{tm}^{\,m;n} \to (\mathbb{I}^m \to \mathsf{ty}^{m'}) \to (\mathbb{I}^n \to \mathsf{vl}^{\,m';n'}) \to \mathsf{tm}^{\,m';n'}$$
$$\_[\_;\_]_{\mathsf{vl}} : \mathsf{vl}^{\,m;n} \to (\mathbb{I}^m \to \mathsf{ty}^{m'}) \to (\mathbb{I}^n \to \mathsf{vl}^{\,m';n'}) \to \mathsf{vl}^{\,m';n'}$$

Note that the indices of the co-domains have to match index-wise.

Vector substitutions lead to similar definitions as before. See Figure 6.5 for the mutual instantiation of terms and values for scoped $\mathrm{F}_{\mathsf{CBV}}$. As we already know, instantiation is defined mutually recursive with forward composition of substitutions. We point out the following aspects:

First, whenever we reach a variable, we have to project the correct substitution component, e.g.

$$(\mathsf{var}_{\mathsf{vl}}\, x)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{vl}} = \sigma_{\mathsf{vl}}\, x$$

for value variables.

Second, when a given subterm is of a different sort, we have to select the correct instantiation function and subvector. Take for example $(s\, A)[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}]_{\mathsf{tm}}$, where we use type instantiation for instantiating the subterm A is $A[\sigma_{\mathsf{ty}}]_{\mathsf{ty}}$ and the correct subvector is σ_{ty}. We hence need to define instantiation for terms and values mutually recursive. The same holds for substitution composition: for example, $\sigma_{\mathsf{ty}} \circ [\tau_{\mathsf{ty}}]_{\mathsf{ty}}$ and $\sigma_{\mathsf{vl}} \circ [\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{vl}}$, require the appropriate subvector.

Last, the traversal of binders changes the interpretation of the indices in scope. We adjust each component of the substitution vector via a customised *lifting operation*. For our example, these are the four operations $\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}$, $\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}$, $\Uparrow^{\mathsf{ty}}_{\mathsf{vl}}$, and $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}$: one for each combination of substitution sort (indexed below) and lifted variable (indexed above).

There are two cases: Either the component corresponds to the sort of binders we just traversed ($\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}$ and $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}$), or it does not ($\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}$ and $\Uparrow^{\mathsf{ty}}_{\mathsf{vl}}$). In the former case, say $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}}$, the component that corresponds to the sort of the binder we just traversed, — here, σ_{vl} — is modified almost as before. The index $0_{\mathbb{I}}$ is mapped to $\mathsf{var}_{\mathsf{vl}}\, 0_{\mathbb{I}}$ as capture-avoiding substitution should not replace the newly bound variable. We then have to ensure that $(\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \sigma_{\mathsf{vl}})(1+x)$ is first mapped to $\sigma_{\mathsf{vl}}\, x$ and then adjusted to bypass the new binder. In the λ-calculus, this was achieved by post-composing the scope change $\uparrow$ to σ_{vl}. Here, the scope change requires one component for type variable and one for value variable: as it increases the value scope and leaves the type scope unchanged, it can be expressed by the vector $[\mathsf{var}_{\mathsf{ty}}; \uparrow]$. In total, we have the equation in Figure 6.5.

If the components do not match, say $\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ \sigma_{\mathsf{ty}}$, we do not have to extend the substitution with a new variable. The substitution still has to incorporate the fitting scope change,

in the case of abstraction, $[\mathsf{var}_{\mathsf{ty}}; \uparrow]$. Adjusted to the relevant scope for types we post-compose σ_{ty} with $[\mathsf{var}_{\mathsf{ty}}]$.

Implementation. Recall that the mutual recursion between instantiation and composition is still not structural and we hence first define instantiation for renamings, written $s\langle \xi; \zeta \rangle_{\mathsf{tm}}$ and $v\langle \xi; \zeta \rangle_{\mathsf{vl}}$. For example, $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}$ is in fact defined as

$$\Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \ \sigma_{\mathsf{vl}} := \mathsf{var}_{\mathsf{vl}}\, 0_{\mathbb{I}} \cdot \sigma_{\mathsf{vl}} \circ \langle \mathsf{id}; \uparrow \rangle_{\mathsf{vl}}.$$

There might be significant simplifications. For example, composition of renamings degenerates to function composition, and hence does not need a full vector:

$$\xi_{\mathsf{vl}} \circ \langle \zeta_{\mathsf{ty}}; \zeta_{\mathsf{vl}} \rangle_{\mathsf{vl}} := \xi_{\mathsf{vl}} \circ \zeta_{\mathsf{vl}}.$$

Reduction. For $\mathrm{F}_{\mathsf{CBV}}$, there are two kinds of β-reduction, one for value abstraction and one for type abstraction:

$$\begin{aligned} \mathsf{app}\ (\mathsf{vt}\ (\lambda A.s))\ (\mathsf{vt}\, v) &\succ s[\mathsf{var}_{\mathsf{ty}}; v \cdot \mathsf{var}_{\mathsf{vl}}]_{\mathsf{tm}} \\ (\mathsf{vt}\ (\Lambda.s))\ A &\succ s[A \cdot \mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}]_{\mathsf{tm}} \end{aligned}$$

We require the embedding operator for the reduction to type-check. In the reduction, we use the corresponding substitution component to replace a variable.

In this section, we indexed type, term, and value instantiation accordingly. We omit this index in the future if clear from the context.

6.4.2 Equational Reasoning

We now extend the previous reasoning principles to vector substitutions: The interference laws remain unchanged (there are no new primitives) but the monad and supplementary laws hold in a generalised form.

We start with the identity law, which already highlights many of the things we have to take care of. Identity substitution extends to a vector of identities and we show that

$$s[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = s \text{ and } v[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = v.$$

We require the following lifting laws, one for each lifting operation:

Lemma 6.5 (Identity Lifting).

1. $\Uparrow^{\mathsf{ty}}_{\mathsf{ty}} \ \mathsf{var}_{\mathsf{ty}} \equiv \mathsf{var}_{\mathsf{ty}}$
2. $\Uparrow^{\mathsf{ty}}_{\mathsf{vl}} \ \mathsf{var}_{\mathsf{vl}} \equiv \mathsf{var}_{\mathsf{vl}}$
3. $\Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \ \mathsf{var}_{\mathsf{ty}} \equiv \mathsf{var}_{\mathsf{ty}}$

4. $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \mathsf{var}_{\mathsf{vl}} \equiv \mathsf{var}_{\mathsf{vl}}$

Proof. If the lifted variable does *not* correspond to the newly bound variable (e.g. for $\Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ \mathsf{var}_{\mathsf{vl}}$), this follows immediately. Otherwise, e.g. for $\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \mathsf{var}_{\mathsf{ty}}$, we have to take a case analysis on the examined variable; again, in both cases the statement follows similar to previous identity lifting lemmas. □

With the lifting lemmas, the main proof is routine and only requires a mutual induction. We examine the following law in detail to show how Autosubst will generated the corresponding proof terms.

Lemma 6.6 (Identity). $s[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = s$ *and* $v[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = v$.

Proof. By a mutual induction on s and v. In the case of the value variable constructor, the goal holds directly by definition of $\mathsf{var}_{\mathsf{vl}}$.

Otherwise, in each case, the statement holds by the inductive hypotheses. E.g. in the case of type application, $s\ A$, we show that $s[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = s$ and $A[\mathsf{var}_{\mathsf{ty}}] = A$. For A, the statement relies on the respective proofs for ty.

If we traverse a binder, we need to account for the scope change. For example, for type abstraction, we show that $\lambda.s[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = \lambda.s[\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \mathsf{var}_{\mathsf{ty}}; \Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ \mathsf{var}_{\mathsf{vl}}] = \lambda.s[\mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}] = \lambda.s$ using identity lifting (Lemma 6.5) and the inductive hypothesis for s. □

We note that we have hence to watch out for similar details as in instantiation: First, whenever we reach a variable, the law has to hold as-is. Second, when a given subterm is of a different sort, we have to select the correct law for the corresponding sort with the correct subvector and equations. Hence, we have to prove the laws for terms and values mutually inductive and require the substitution lemmas for types as assumed before. Last, the new lifting operations require new lifting laws. These lifting laws are a super-set of the ones for types. As in our definition of a lifting $\Uparrow^{x}_{y}$, we do a case analysis on whether the components x and y match.

We similarly prove compositionality. As before, we have to show the statement for all combinations of renamings and substitutions. Additionally, we require new lifting lemmas for each case, in total 16.

For example, for composition of substitutions and substitutions only we require the following four lifting lemmas:

Lemma 6.7 (Vector Substitution-Substitution-Compositionality Lifting).

1. $(\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \sigma) \circ [\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \tau_{\mathsf{ty}}]_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ (\sigma \circ [\tau_{\mathsf{ty}}]_{\mathsf{ty}})$
2. $(\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ \sigma) \circ [\Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ \tau_{\mathsf{ty}}]_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{vl}}_{\mathsf{ty}}\ (\sigma \circ [\tau_{\mathsf{ty}}]_{\mathsf{ty}})$
3. $(\Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ \sigma) \circ [\Uparrow^{\mathsf{ty}}_{\mathsf{ty}}\ \tau_{\mathsf{ty}}; \Uparrow^{\mathsf{vl}}_{\mathsf{vl}}\ \tau_{\mathsf{vl}}]_{\mathsf{vl}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{vl}}\ (\sigma \circ [\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{vl}})$

4. $(\Uparrow^{vl}_{vl}\ \sigma) \circ [\Uparrow^{vl}_{ty}\ \tau_{ty}; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \equiv \Uparrow^{vl}_{vl}\ (\sigma \circ [\tau_{ty}; \tau_{vl}]_{vl})$

Proof. We do a case analysis on whether the added scope and the scope itself are equal.

If they are unequal, the proof follows without the previous case analysis. The proof still requires the instances for the composition of renaming and substitution, and substitution and renaming. For example, for 3. we have:

$$\begin{aligned}
(\Uparrow^{ty}_{vl}\ \sigma) \circ [\Uparrow^{ty}_{ty}\ \tau_{ty}; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl}(x) &= (\Uparrow^{ty}_{vl}\ \sigma)(x)[\Uparrow^{ty}_{ty}\ \tau_{ty}; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= ((\sigma \circ \langle \uparrow; \mathsf{var}_{vl} \rangle)(x))[\Uparrow^{ty}_{ty}\ \tau_{ty}; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= (\sigma\, x)\langle \uparrow; \mathsf{var}_{vl} \rangle [\Uparrow^{ty}_{vl}\ \tau_{ty}; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= (\sigma\, x)[\uparrow \circ \Uparrow^{ty}_{ty}\ \tau_{ty}; \mathsf{var}_{vl} \circ \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= (\sigma\, x)[\tau_{ty} \circ \langle \uparrow \rangle; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= (\sigma\, x)[\tau_{ty} \circ \langle \uparrow \rangle; \Uparrow^{vl}_{vl}\ \tau_{vl}]_{vl} \\
&= ((\sigma \circ [\tau_{ty}; \tau_{vl}]_{vl}) \circ \langle \uparrow; \mathsf{var}_{vl} \rangle)(x) \\
&= \Uparrow^{ty}_{vl}\ (\sigma \circ [\tau_{ty}; \tau_{vl}]_{vl})(x)
\end{aligned}$$

Note that this proof uses the same steps as in Lemma 3.4, but without the case analysis. □

In the statement of the lifting lemmas, τ is not always the full vector but might simplify. Similarly, whenever σ is a renaming, τ always simplifies to just the respective component. The same is valid for the actual statement of compositionality.

We can then show the last statement similar to before:

Lemma 6.8 (Compositionality).

1. $s[\sigma_{ty}; \sigma_{vl}][\tau_{ty}; \tau_{vl}] = s[\sigma_{ty} \circ [\tau_{ty}]; \sigma_{vl} \circ [\tau_{ty}; \tau_{vl}]]$
2. $v[\sigma_{ty}; \sigma_{vl}][\tau_{ty}; \tau_{vl}] = v[\sigma_{ty} \circ [\tau_{ty}]; \sigma_{vl} \circ [\tau_{ty}; \tau_{vl}]]$

Proof. By a mutual induction on s and v, using the lifting lemmas (Lemma 6.7) in case of type and value abstraction. □

Adaption of the supplementary laws is routine, but again requires us to get the vectors right:

Lemma 6.9 (Supplementary Laws).

1. $\mathsf{var}_{vl} \circ [\sigma_{ty}; \sigma_{vl}] \equiv \sigma_{vl}$
2. $[\mathsf{var}_{ty}; \mathsf{var}_{vl}]_{tm} \equiv \mathsf{id}$
3. $[\mathsf{var}_{ty}; \mathsf{var}_{vl}]_{vl} \equiv \mathsf{id}$

4. $(\sigma_{\mathsf{tm}} \circ [\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{tm}}) \circ [\theta_{\mathsf{ty}}; \theta_{\mathsf{vl}}]_{\mathsf{tm}} \equiv \sigma_{\mathsf{tm}} \circ [\tau_{\mathsf{ty}} \circ [\theta_{\mathsf{ty}}]_{\mathsf{ty}}; \tau_{\mathsf{vl}} \circ [\theta_{\mathsf{ty}}; \theta_{\mathsf{vl}}]_{\mathsf{vl}}]_{\mathsf{tm}}$
5. $(\sigma_{\mathsf{vl}} \circ [\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]_{\mathsf{vl}}) \circ [\theta_{\mathsf{ty}}; \theta_{\mathsf{vl}}]_{\mathsf{vl}} \equiv \sigma_{\mathsf{vl}} \circ [\tau_{\mathsf{ty}} \circ [\theta_{\mathsf{ty}}]_{\mathsf{ty}}; \tau_{\mathsf{vl}} \circ [\theta_{\mathsf{ty}}; \theta_{\mathsf{vl}}]_{\mathsf{vl}}]_{\mathsf{vl}}$

Proof. The first statement follows directly by definition of instantiation. Note that a whole component of the substitution disappear. The remaining laws follow directly from the monad laws. □

Note that we only need a left identity law for each sort with variables, while compositionality has to be proven for each substitution sort.

Similarly, extensionality holds, and requires again the following four lifting lemmas:

Lemma 6.10 (Extensionality Lifting). *Assume that* $\sigma_{\mathsf{ty}} \equiv \tau_{\mathsf{ty}}$ *and* $\sigma_{\mathsf{vl}} \equiv \tau_{\mathsf{vl}}$. *Then*

1. $\Uparrow^{\mathsf{ty}}_{\mathsf{ty}} \sigma_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{ty}} \tau_{\mathsf{ty}}$
2. $\Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \sigma_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \tau_{\mathsf{ty}}$
3. $\Uparrow^{\mathsf{ty}}_{\mathsf{vl}} \sigma_{\mathsf{vl}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{vl}} \tau_{\mathsf{vl}}$
4. $\Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \sigma_{\mathsf{vl}} \equiv \Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \tau_{\mathsf{vl}}$

Proof. If the upper and lower index match, we do a case analysis on the argument x: If $x = 0_{\mathbb{I}}$, the claim follows directly; otherwise, we use the assumption. Otherwise, the claim follows directly with the assumption. □

Lemma 6.11 (Extensionality).

$$\frac{\sigma_{\mathsf{ty}} \equiv \tau_{\mathsf{ty}} \quad \sigma_{\mathsf{vl}} \equiv \tau_{\mathsf{vl}}}{s[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}] = s[\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]} \qquad \frac{\sigma_{\mathsf{ty}} \equiv \tau_{\mathsf{ty}} \quad \sigma_{\mathsf{vl}} \equiv \tau_{\mathsf{vl}}}{v[\sigma_{\mathsf{ty}}; \sigma_{\mathsf{vl}}] = v[\tau_{\mathsf{ty}}; \tau_{\mathsf{vl}}]}$$

Proof. By mutual induction on s and v, using the lifting lemmas (Lemma 6.10). □

Lemma 6.12 (Coincidence Lifting). *Assume that* $\xi \circ \mathsf{var}_{\mathsf{ty}} \equiv \sigma_{\mathsf{ty}}$ *and* $\xi \circ \mathsf{var}_{\mathsf{vl}} \equiv \sigma_{\mathsf{vl}}$. *Then*

1. $(\Uparrow^{*\mathsf{ty}}_{\mathsf{ty}} \xi) \circ \mathsf{var}_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{ty}} \sigma_{\mathsf{ty}}$
2. $(\Uparrow^{*\mathsf{vl}}_{\mathsf{ty}} \xi) \circ \mathsf{var}_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \sigma_{\mathsf{ty}}$
3. $(\Uparrow^{*\mathsf{ty}}_{\mathsf{vl}} \xi) \circ \mathsf{var}_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{ty}}_{\mathsf{vl}} \sigma_{\mathsf{vl}}$
4. $(\Uparrow^{*\mathsf{vl}}_{\mathsf{vl}} \xi) \circ \mathsf{var}_{\mathsf{ty}} \equiv \Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \sigma_{\mathsf{vl}}$

Proof. Case analysis. If the bound and environmental sort are equal, we do a case analysis on the argument; otherwise, the claim follows directly. □

Lemma 6.13 (Coincidence).

1. $s[\xi_{\mathsf{ty}} \circ \mathsf{var}_{\mathsf{ty}}, \xi_{\mathsf{vl}} \circ \mathsf{var}_{\mathsf{vl}}]_{\mathsf{tm}} = s\langle \xi_{\mathsf{ty}}, \xi_{\mathsf{vl}} \rangle_{\mathsf{tm}}$

2. $\nu[\xi_{\mathsf{ty}} \circ \mathsf{var}_{\mathsf{ty}}, \xi_{\mathsf{vl}} \circ \mathsf{var}_{\mathsf{vl}}]_{\mathsf{vl}} = \nu\langle \xi_{\mathsf{ty}}, \xi_{\mathsf{vl}} \rangle_{\mathsf{vl}}$

Proof. By mutual induction on s and ν, requiring the corresponding lifting lemmas in the case of abstraction (Lemma 6.12). □

As we can already see, with vector substitutions, all these laws are not especially hard to prove and follow regular patterns, but defining and establishing the statements manually is highly repetitive.

The corresponding equational theory then uses the definitional laws for evaluation, the defining equations for interaction, the interaction laws, the equational theory and supplementary laws of the types of $\mathrm{F}_{\mathsf{CBV}}$, and the equational theory and supplementary laws of the terms and values of $\mathrm{F}_{\mathsf{CBV}}$.

We use these equations to show that reduction is substitutive:

Lemma 6.14. *Reduction is substitutive.*

Proof. There are two equations to show.

First, we have to show that:

$$s[\mathsf{var}; \nu..][\sigma; \tau] = s[\Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \ \sigma; \Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \ \tau][\mathsf{var}; \nu[\sigma; \tau]..]$$

and then for the second equation that:

$$s[A..; \mathsf{var}][\sigma; \tau] = s[\Uparrow^{\mathsf{vl}}_{\mathsf{ty}} \ \sigma; \Uparrow^{\mathsf{vl}}_{\mathsf{vl}} \ \tau][A[\sigma]..; \mathsf{var}]$$

Both equations follow by a reduction of the equations with the respective equations of the equational theory for types, terms, and values. □

6.5 First-Order Binders in First-Order Logic and the Pi Calculus

In general, Coq's condition of structural termination forces us to first define instantiation with renamings, then instantiation with substitutions (see Section 3.1). In restricted classes of syntax, either instantiation with renamings or substitutions are redundant, and we can hence omit the detour. For this section, we revoke our previous assumption that we omit all renaming primitives and explicitly write down all intermediate lemmas.

6.5.1 First-Order Logic

Recall the specification of simplified first-order logic, short FOL* (EHOAS specification in Figure 5.9). Recall that it consists of two sorts, terms term with a constant symbol c and a binary function symbol $*$, and formulas form with a nullary and unary predicate,

$$
\begin{aligned}
(\mathsf{var}\, x)[\sigma]_{\mathsf{tm}} &= \sigma\, x & (\sigma \circ [\tau]_{\mathsf{tm}})\, x &= (\sigma\, x)[\tau]_{\mathsf{tm}} \\
(c)[\sigma]_{\mathsf{tm}} &= c \\
(s + t)[\sigma]_{\mathsf{tm}} &= s[\sigma]_{\mathsf{tm}} + t[\sigma]_{\mathsf{tm}} \\
\\
\bot[\sigma]_{\mathsf{form}} &= \bot \\
(\forall.p)[\sigma]_{\mathsf{form}} &= \forall.p[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma]_{\mathsf{form}} & \text{with } \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma &:= \mathsf{var}_{\mathsf{tm}}\, 0_{\mathbb{I}} \cdot (\sigma \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})
\end{aligned}
$$

Figure 6.6: Instantiation on FOL.

in which formulas depend on terms, but not vice versa. As before, we can define its scoped representations as follows:

$$
\begin{aligned}
s, t \in \mathsf{tm}^k &:= \mathsf{var}\, x \mid c \mid s^k * t^k & x &\in \mathbb{I}^k \\
p \in \mathsf{form}^k &:= P_0 \mid P_1\, s^k \mid \forall.p^{1+k}
\end{aligned}
$$

As terms occur negatively in formulas, both terms and formulas can contain term variables. However, variables are only used when terms and term instantiation are already defined. We say that term is a **first-order binder** and call a substitution sort which only contains first-order binders a **first-order sort**. As a consequence, lifting is independent of instantiation with renamings, and we can simplify our development (instantiation and proofs) significantly by postponing the definition. This is possible for all first-order sorts.

See Figure 6.6 for the simplified instantiation on terms and formulas. The order is slightly different from before: We (1) generate instantiation of terms, (2) define substitution composition and the lifting operation $\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma$, and (3) define formula instantiation, which uses lifting in the case of the universal quantifier.

These simple changes allow us to simplify the proof of the monad laws significantly. Terms are independent of lifting, and hence we prove directly:

Lemma 6.15 (Equational Theory on Terms of FOL).

1. $\mathsf{var} \circ [\sigma]_{\mathsf{tm}} \equiv \sigma$
2. $s[\mathsf{var}]_{\mathsf{tm}} = s$
3. $s[\sigma]_{\mathsf{tm}}[\tau]_{\mathsf{tm}} = s[\sigma \circ [\tau]_{\mathsf{tm}}]_{\mathsf{tm}}$
4. $[\mathsf{var}]_{\mathsf{tm}} \equiv \mathsf{id}$
5. $[\sigma]_{\mathsf{tm}} \circ [\tau]_{\mathsf{tm}} \equiv [\sigma \circ [\tau]_{\mathsf{tm}}]_{\mathsf{tm}}$
6. *If* $\sigma \equiv \tau$, *then* $s[\sigma]_{\mathsf{tm}} = s[\tau]_{\mathsf{tm}}$.

Proof. The first statement follows by definition; the next three statements are proven by induction on s; the remaining laws follow directly by the definition of instantiation or using the previous monad laws. □

Note that all these changes simplify the laws for terms.

Afterwards, we prove the lifting lemmas:

Lemma 6.16 (Lifting Lemmas).

1. $\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \mathsf{var}_{\mathsf{tm}} \equiv \mathsf{var}_{\mathsf{tm}}$
2. $(\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma) \circ [\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau]_{\mathsf{tm}} \equiv \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} (\sigma \circ [\tau]_{\mathsf{tm}})$
3. *If* $\sigma \equiv \tau$, *then* $\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma \equiv \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau$.

Proof. 1. follows directly by case analysis on the argument; similarly for 3.. We show the exact reasoning for 2., as here we required renaming instances before. Case analysis on n. For $n = 0$, the claim follows directly. Otherwise:

$$
\begin{aligned}
&((\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma) \circ [\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau]_{\mathsf{tm}})(1+n) && \\
&= ((\mathsf{var}\, 0_{\mathbb{I}} \cdot (\sigma \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})) \circ [\mathsf{var}\, 0 \cdot (\tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})]_{\mathsf{tm}})(1+n) && \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \\
&= ((\sigma \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})(n))[\mathsf{var}\, 0_{\mathbb{I}} \cdot \tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}]_{\mathsf{tm}}) && \cdot \\
&= (\sigma\, n)[\uparrow \circ \mathsf{var}]_{\mathsf{tm}}[\mathsf{var}\, 0_{\mathbb{I}} \cdot \tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}]_{\mathsf{tm}}) && \circ \\
&= (\sigma\, n)[(\uparrow \circ \mathsf{var}) \circ [(\mathsf{var}\, 0_{\mathbb{I}} \cdot (\tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}))]_{\mathsf{tm}}]_{\mathsf{tm}}) && \text{Lemma 6.15} \\
&= (\sigma\, n)[\uparrow \circ \mathsf{var} \circ [(\mathsf{var}\, 0_{\mathbb{I}} \cdot (\tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}))]]_{\mathsf{tm}}) && \text{associativity} \\
&= (\sigma\, n)[\uparrow \circ (\mathsf{var}\, 0_{\mathbb{I}} \cdot (\tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}))]_{\mathsf{tm}}) && \text{Lemma 6.15} \\
&= (\sigma\, n)[\tau \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}}]_{\mathsf{tm}} && \text{interaction} \\
&= (\sigma\, n)[\tau]_{\mathsf{tm}}[\uparrow \circ \mathsf{var}]_{\mathsf{tm}} && \text{Lemma 6.15} \\
&= ((\sigma \circ [\tau]_{\mathsf{tm}}) \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})\, n && \text{associativity} \\
&= (\mathsf{var}\, 0_{\mathbb{I}} \cdot (\sigma \circ [\tau]_{\mathsf{tm}}) \circ [\uparrow \circ \mathsf{var}]_{\mathsf{tm}})(1+n) && \cdot \\
&= (\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} (\sigma \circ [\tau]_{\mathsf{tm}}))(1+n) && \Uparrow^{\mathsf{tm}}_{\mathsf{tm}}
\end{aligned}
$$

We use directly the composition law for terms, and hence do not require intermediate lemmas for composition with renamings. □

The equational theory for formulas follows using the above lifting laws:

Lemma 6.17 (Equational Theory on Formulas of FOL).

1. $p[\mathsf{var}]_{\mathsf{form}} = s$
2. $p[\sigma]_{\mathsf{form}}[\tau]_{\mathsf{form}} = p[\sigma \circ [\tau]]_{\mathsf{form}}$
3. $[\mathsf{var}]_{\mathsf{form}} \equiv \mathsf{id}$

4. $[\sigma]_{\mathsf{form}} \circ [\tau]_{\mathsf{form}} \equiv [\sigma \circ [\tau]_{\mathsf{form}}]_{\mathsf{form}}$
5. *If* $\sigma \equiv \tau$, *then* $p[\sigma]_{\mathsf{form}} = p[\tau]_{\mathsf{form}}$.

Proof. Using the lifting lemmas (Lemma 6.16). □

We can omit all interaction laws between renamings and substitutions.

6.5.2 Pi Calculus

In some syntactic systems, even only renamings suffice. Consider the syntax of the π-calculus and the corresponding EHOAS specification from Figure 5.3, in scoped syntax depicted as:

$$\begin{aligned} c \in \mathsf{chan}^k &:= \mathsf{var}\, x \qquad x \in \mathbb{I}^k \\ P, Q \in \mathsf{proc}^k &:= P^k | Q^k \mid c^k.P^{1+k} \mid \overline{c^k}\langle c'^k \rangle.P^k \mid !P^k \mid \nu.P^{1+k} \mid 0 \end{aligned}$$

with concurrency, input prefixing, output prefixing, replication, restriction, and the nil process. Both in the case of input prefixing and restriction, we bind a new channel name x.

However, a channel describes nothing but a name. As a consequence, the sort will later on consist of only variables.

It thus suffices if we define only instantiation with renamings. The equational theory will be similar to the one for first-order logic, but instead, we only define instantiation with renamings. A variable sort is always also a first-order sort.

6.6 Variadic Binders in the Multivariate Lambda Calculus

We now turn to the last system in this chapter and implement variadic binders in the multivariate λ-calculus. It is also the first system, which will require us to implement new primitives, accounting for a variadic scope change. As will be seen, the implementation still follows similar regularities as before.

We start with recalling the multivariate λ-calculus, λ_ν, where abstraction binds a variadic number of n elements as described in Figure 5.10. Its scoped interpretation looks as follows:

$$s, t \in \mathsf{tm}^k := \mathsf{var}\, x \mid \mathsf{app}\, s^k \{t_1^k .. t_n^k\} \mid \lambda_n.s^{n+k} \qquad\qquad x \in \mathbb{I}^k$$

where $\{t_1^k .. t_n^k\}$ is represented by a functional vector. Note that for scoped syntax we have to increase the scope not only by 1 but by n. We call this a **variadic scope change**.

$$
\begin{aligned}
(\mathsf{var}\, x)[\sigma] &= \sigma\, x & (\sigma_1 \circ [\sigma_2])\, x &= (\sigma_1\, x)[\sigma_2] \\
(s\, \bar{t}_n)[\sigma] &= (s[\sigma])(\mathsf{map}[\sigma]\bar{t}_n) & & \\
(\lambda_p.s)[\sigma] &= \lambda_p.(s[\Uparrow^{tm,p}_{tm}\ \sigma]) & \text{with } \Uparrow^{tm,p}_{tm}\ \sigma &= (\mathsf{hd}_p \circ \mathsf{var}) \cdot_p (\sigma \circ [\uparrow^p])
\end{aligned}
$$

Figure 6.7: Instantiation for the multivariate λ-calculus.

The change of meaning for free variables can be depicted by the following picture:

s		..	0	1	2	
			|	|	|	...
$\lambda_n.s$	0	..	n	1 + n	2 + n	

Instantiation. The new variadic scope change requires us to extend the primitive building blocks for instantiation, declared in Chapter 3. We require the following variadic operations:

1. **Variadic shifting** $\uparrow^m : \mathbb{I}^n \to \mathbb{I}^{m+n}$, a renaming with $\uparrow^m(x : \mathbb{I}^n) := m_{\mathbb{I}} +_{\mathbb{I}} x$.
2. **Variadic head**, $\mathsf{hd}_m : \mathbb{I}^m \to \mathbb{I}^{m+n}$, a renaming with

$$
\begin{aligned}
\mathsf{hd}_0 &:= \mathsf{id}_0 \\
\mathsf{hd}_{1+m} &:= 0_{\mathbb{I}} \cdot (\mathsf{hd}_m \circ \uparrow)
\end{aligned}
$$

3. **Variadic extension** $\_ \cdot_m \_ : (\mathbb{I}^m \to X) \to (\mathbb{I}^n \to X) \to (\mathbb{I}^{m+n} \to X)$, which precedes an arbitrary stream $\tau : \mathbb{I}^n \to X$ with a new stream $\sigma : \mathbb{I}^m \to X$:

$$
\begin{aligned}
(\sigma \cdot_0 \tau)\, x &:= \tau\, x \\
(\sigma \cdot_{1+m} \tau)\, (\emptyset) &:= \sigma\, \emptyset \\
(\sigma \cdot_{1+m} \tau)\, (\lfloor x \rfloor) &:= \sigma \cdot_m \tau(x)
\end{aligned}
$$

We still allow composition of the primitives and the identity renaming. As highlighted in the discussion section of this chapter, all these are real generalisations of the previous primitives. Using these primitives will further require new interference laws; we postpone these until we have shown that these primitives indeed suffice for instantiation.

See Figure 6.7 for the definition of instantiation. Unchanged, we traverse the term homomorphically, and we know all components expect the handling of variadic abstraction. Similar to monadic abstraction, we handle the scope change with a lifting operation, only this time a variadic one:

$$\Uparrow^{tm,p}_{tm} : (\mathbb{I}^k \to \mathsf{tm}^m) \to (\mathbb{I}^{p+k} \to \mathsf{tm}^{p+m})$$

The first p binders are bound by variadic abstraction and hence remain unaffected, we skip them with an extension by hd_p. Afterwards, if a variable is replaced, the result has to be lifted by p according to the scope change $\uparrow^p$, we achieve this by a post-composition. This corresponds precisely to the structure of a monadic lifting function.

Analogous to the monadic case, the definition is not structurally recursive, and we require the intermediate notions of a renaming, $s\langle\xi\rangle$, and a lifting operation for renamings:

$$\Uparrow^{tm,p}_{tm*} \ \xi := \mathsf{hd}_p \cdot_p (\xi \circ \langle\uparrow^p\rangle)$$

Given instantiation, we can define β-reduction for the multivariate λ-calculus:

$$\mathsf{app}\ (\lambda_p.s)\ \vec{p}_t \succ s[\vec{p}_t \cdot_p \mathsf{var}]$$

We abbreviate $t \cdot_p \mathsf{var}$ to $t..$, similar to the monadic case.

Equational theory. Now that we have defined instantiation let us turn to the equational theory. First, note that a simple pattern matching on an index $x : \mathbb{I}^{m+n}$ is no longer possible. However, we can state the following case analysis using the disjoint sum and the previously defined primitives:

Lemma 6.18. *Let* $x : \mathbb{I}^{m+n}$. *Then* $\{x' \mid x = \mathsf{hd}_n x'\} + \{x' \mid x = \uparrow^m x'\}$.

Proof. By induction on m. For $m = 0$, the right side holds for $x' = x$, and the claim follows directly. Otherwise, if $m = 1 + m'$, case analysis on x. If $x = \emptyset$, then choose the left side and $x' = \emptyset$. Otherwise, if $x = \lfloor x' \rfloor$ use the inductive hypothesis on x' and propagate the result. □

Further note that extension fulfils the following congruence law:

Lemma 6.19 (Congruence). *If* $\sigma \equiv \sigma'$ *and* $\tau \equiv \tau'$, *then* $\sigma \cdot_p \tau \equiv \sigma' \cdot_p \tau'$.

Proof. By induction on p, and a subsequent case analysis on the argument. □

Both these laws will turn out to be very helpful in the following.

Next, we turn to the interaction lemmas. Not surprising, we can show equations similar to their monadic counterparts:

Lemma 6.20 (Variadic Interaction).

1. $(\sigma \cdot_m \tau)(\mathsf{hd}_m\, x) = \sigma\, x$ *head*
2. $(\sigma \cdot_m \tau)(\uparrow^m x) = \tau\, x$ *tail*
3. $f((\sigma \cdot_m \tau)(x)) = ((\sigma \circ f) \cdot_m (\tau \circ f))(x)$ *distributivity*
4. $\mathsf{hd}_m \circ (\sigma \cdot_m \tau) \equiv \sigma$ *functional head*

5. $\uparrow^{m} \circ (\sigma \cdot_{m} \tau) \equiv \tau$ *functional tail*
6. $(\sigma \cdot_{m} \tau) \circ f \equiv (\sigma \circ f) \cdot_{m} (\tau \circ f)$ *functional distributivity*
7. $\mathsf{hd}_{p} \cdot_{p} \uparrow^{p} \equiv \mathsf{id}$ *η-law*

Proof. 2. holds by induction on m, 1. by induction on m and a subsequent case analysis on the argument. 6. and 7. then hold by a case analysis on the argument using Lemma 6.18, and subsequent usage of 2. and 1. Note that we need a functional variant of the head and tail laws. □

We now have all the components to show the monad laws. However, the monad laws need new lifting lemmas, accounting for the variadic scope change. Lucky for us, as the primitives for unary and variadic binders resemble each other, also the lifting lemmas are very similar. However, many equations do no longer hold definitionally, and hence the following proof steps have to be adapted:

1. The previous case distinction on a natural number or a finite type no longer works. Instead, we require the η-law which does a case analysis on a variable $x : \mathbb{I}^{m+n}$ and rewrite with the resulting equations.
2. Often, we require the η-law which states that two extensions are equal if both sides of an extension are pair-wise equal. We use Lemma 6.19 for this part of the proof.
3. The equation $(s \cdot \sigma)\, 0_{\mathbb{I}} = s$ holds definitionally, for variadic syntax we have to explicitly invoke the variadic head laws.
4. The equation $\uparrow \circ (s \cdot \sigma) x = \sigma\, x$ holds definitionally, for variadic syntax we have to explicitly invoke the variadic tail law.

These are the only cases where our lemmas differ from the ones in Chapter 3.

Let us start and bring these observations into practice: In the following we prove variadic variants of all lifting lemmas.

Lemma 6.21 (Identity Lifting). $\Uparrow^{tm,p}_{tm}\ \mathsf{var} \equiv \mathsf{var}$.

Proof. Using a case analysis via Lemma 6.18, we have two cases: For $x = \mathsf{hd}_{p} x'$, reflexivity suffices; while for $x = \uparrow^{p} x'$ the claim follows with congruence. □

Lemma 6.22 (Compositionality Lifting).

1. $(\Uparrow^{tm,p}_{tm*}\ \xi) \circ (\Uparrow^{tm,p}_{tm*}\ \zeta) \equiv \Uparrow^{tm,p}_{tm*}\ (\xi \circ \zeta)$
2. $(\Uparrow^{tm,p}_{tm*}\ \xi) \circ (\Uparrow^{tm,p}_{tm}\ \tau) \equiv \Uparrow^{tm,p}_{tm}\ (\xi \circ \tau)$
3. $(\Uparrow^{tm,p}_{tm}\ \sigma) \circ (\Uparrow^{tm,p}_{tm*}\ \zeta) \equiv \Uparrow^{tm,p}_{tm}\ (\sigma \circ \langle \xi \rangle)$
4. $(\Uparrow^{tm,p}_{tm}\ \sigma) \circ [\Uparrow^{tm,p}_{tm}\ \tau] \equiv \Uparrow^{tm,p}_{tm}\ (\sigma \circ [\tau])$

Proof. As before, we prove these lifting laws interleaved with the full compositionality statements. We show 4., similar to the monadic λ-calculus.

First, $(\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \sigma) \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau] = ((\mathsf{hd}_{\mathsf{p}} \cdot_{\mathsf{p}} (\sigma \circ \langle \uparrow^{\mathsf{p}} \rangle)) \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau]$, and with distribution of variadic extension, $(\mathsf{hd}_{\mathsf{p}} \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau]) \cdot_{\mathsf{p}} ((\sigma \circ \langle \uparrow^{\mathsf{p}} \rangle) \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau])$. We use the congruence law (Lemma 6.19), and the definition of $\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}$, such that it suffices to show that:

1. $\mathsf{hd}_{\mathsf{p}} \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau] \equiv \mathsf{hd}_{\mathsf{p}}$; and
2. $(\sigma \circ \langle \uparrow^{\mathsf{p}} \rangle) \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau] \equiv (\sigma \circ [\tau]) \circ \langle \uparrow^{\mathsf{p}} \rangle$.

For the first claim, note that the left side corresponds to $\mathsf{hd}_{\mathsf{p}} \circ [\mathsf{hd}_{\mathsf{p}} \cdot_{\mathsf{p}} \tau \circ \langle \uparrow^{\mathsf{p}} \rangle]$, and hence with the functional head law the two sides coincide.

For the second part, we have:

$$
\begin{aligned}
&((\sigma \circ \langle \uparrow^{\mathsf{p}} \rangle) \circ [\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau])\, x && \\
&= ((\sigma \circ \langle \uparrow^{\mathsf{p}} \rangle) \circ [\mathsf{hd}_{\mathsf{p}} \cdot_{\mathsf{p}} \tau \circ \langle \uparrow^{\mathsf{p}} \rangle])\, x && \Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}} \\
&= (\sigma\, x)\langle \uparrow^{\mathsf{p}} \rangle [\mathsf{hd}_{\mathsf{p}} \cdot_{\mathsf{p}} \tau \circ \langle \uparrow^{\mathsf{p}} \rangle] && \text{associativity} \\
&= (\sigma\, x)[\uparrow^{\mathsf{p}} \circ (\mathsf{hd}_{\mathsf{p}} \cdot_{\mathsf{p}} \tau \circ \langle \uparrow^{\mathsf{p}} \rangle)] && \text{compositionality} \\
&= (\sigma\, x)[\tau \circ \langle \uparrow^{\mathsf{p}} \rangle] && \text{head} \\
&= (\sigma\, x)[\tau] \langle \uparrow^{\mathsf{p}} \rangle && \text{compositionality} \\
&= ((\sigma \circ [\tau]) \circ \langle \uparrow^{\mathsf{p}} \rangle) x &&
\end{aligned}
$$

Again we require compositionality for each a renaming and a substitution, analogous to the monadic case. □

Lemma 6.23 (Extensionality Lifting). *If* $\sigma \equiv \tau$, *then* $\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \sigma \equiv \Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \tau$.

Proof. Using the definition of $\Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}$ and Lemma 6.19, it suffices that each part of the extension is equivalent. For the first part, this follows directly, while for the second part, this follows with our assumption. □

Lemma 6.24 (Coincidence Lifting). *If* $\xi \circ \mathsf{var} \equiv \sigma$, *then* $(\Uparrow_{\mathsf{tm}*}^{\mathsf{tm},\mathsf{p}}\ \xi) \circ \mathsf{var} \equiv \Uparrow_{\mathsf{tm}}^{\mathsf{tm},\mathsf{p}}\ \sigma$.

Proof. We have $(\Uparrow_{\mathsf{tm}*}^{\mathsf{tm},\mathsf{p}}\ \xi) \circ \mathsf{var} \equiv ((\mathsf{hd}_{\mathsf{p}} \circ \mathsf{var}) \cdot_{\mathsf{p}} (\xi \circ \langle \uparrow^{\mathsf{p}} \rangle)) \circ \mathsf{var}$ by the distribution law. With extension congruence (Lemma 6.19), it suffices that both parts of extension coincide, what they do in the first part by definition, and for the second part using the assumption. □

Proving the lifting lemmas was the main work, the remaining proofs remain unchanged:

Lemma 6.25 (Equational Theory of λ_v).

1. $\mathsf{var} \circ [\sigma] \equiv \sigma$
2. $s[\mathsf{var}] = s$

3. $s[\sigma][\tau] = s[\sigma \circ [\tau]]$
4. $(\sigma \circ [\tau]) \circ [\theta] \equiv \sigma \circ [\tau \circ [\theta]]$
5. $\sigma \circ [\mathsf{var}] \equiv \sigma$

Proof. The first statement follows directly by the definition of instantiation, while the next two statements follow as before by induction on s, using the variadic lifting lemmas (Lemma 6.21 and Lemma 6.22). As in Lemma 3.5, the last two statements follow directly from the first. □

Lemma 6.26 (Extensionality). *If* $\sigma \equiv \tau$, *then* $s[\sigma] = s[\tau]$.

Proof. By induction on s, using the variadic lifting lemma (Lemma 6.23). □

Lemma 6.27 (Coincidence). $s[\xi \circ \mathsf{var}] = s\langle\xi\rangle$.

Proof. By induction on s, using the variadic lifting lemma (Lemma 6.24). □

To solve an equation between expressions with variadic binders, we use the equational theory of λ_ν together with the functor laws for lists, and the variadic interaction laws. Let us prove that β-reduction is substitutive:

Lemma 6.28. *β-reduction is substitutive.*

Proof. We have to show that

$$s[t \cdot_p \mathsf{var}][\sigma] = s[(\mathsf{hd}_p \circ \mathsf{var}_{\mathsf{vl}}) \cdot_p \sigma \circ \langle\uparrow^p\rangle][\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}].$$

Both sides reduce to $s[\mathsf{map}\,[\sigma]\,t \cdot_p \sigma]$, the first equation with composition of substitutions, distribution of extension, and left identity. For the right side, we have:

$$\begin{aligned}
&s[(\mathsf{hd}_p \circ \mathsf{var}_{\mathsf{vl}}) \cdot_p \sigma \circ \langle\uparrow^p\rangle][\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}] \\
&= s[((\mathsf{hd}_p \circ \mathsf{var}_{\mathsf{vl}}) \cdot_p \sigma \circ \langle\uparrow^p\rangle) \circ [\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}]] && \text{compositionality} \\
&= s[((\mathsf{hd}_p \circ \mathsf{var}_{\mathsf{vl}}) \circ [\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}] \cdot_p \sigma \circ \langle\uparrow^p\rangle \circ [\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}])] && \text{distributivity} \\
&= s[(\mathsf{hd}_p \circ \mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var} \cdot_p \sigma \circ \langle\uparrow^p\rangle \circ [\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}])] && \text{left identity} \\
&= s[(\mathsf{map}\,[\sigma]\,t \cdot_p \sigma \circ \langle\uparrow^p\rangle \circ [\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var}])] && \text{head} \\
&= s[(\mathsf{map}\,[\sigma]\,t \cdot_p \sigma \circ [\uparrow^p \circ (\mathsf{map}\,[\sigma]\,t \cdot_p \mathsf{var})])] && \text{compositionality} \\
&= s[\mathsf{map}\,[\sigma]\,t \cdot_p \sigma \circ [\mathsf{var}]] && \text{tail} \\
&= s[\mathsf{map}\,[\sigma]\,t \cdot_p \sigma] && \text{right identity}
\end{aligned}$$

□

Note that we required the functor laws to show that this is substitutive.

6.7 Discussion

There are many options on how to interpret formal systems in de Bruijn syntax. We hence discuss related work connected to the various extensions in this chapter.

6.7.1 First-Class Renamings

We start with first-class renamings. As mentioned in the introduction of this chapter, we required first-class renamings in proofs using Kripke-style logical relations [4].

Even if instantiation could be realised immediately, we require renamings to state some laws; the second-class view obscures the proofs. First, substitution properties might require proving the corresponding instance for renamings. Second, certain statements only hold for renamings. The special handling of renamings is especially relevant for many-sorted syntax, where whole parts of the substitution fall away, or the definition of composition simplifies significantly. We give two more detailed examples.

Reason 1. A well-known example for the first reason are context morphism lemmas as appeared in work by Goguen and McKinna [52] and Kaiser et al. [63], stating for example that typing of $\mathsf{F}_{\mathsf{CBV}}$ is substitutive:

$$\frac{\Gamma \vdash s : A \qquad \forall x.\Delta \vdash \tau\, x : (\Gamma\, x)[\sigma]}{\Delta \vdash s[\sigma;\tau] : A[\sigma]}$$

The proof proceeds by induction on $\Gamma \vdash s : A$. In the case of abstraction the context changes to $\Gamma \cdot A$ and we need to show that the precondition is still preserved, i.e.,

$$A \cdot \Delta \vdash (\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau)\, x : ((A \cdot \Gamma)\, x)[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma]$$

for all x. For an arbitrary x, this statement is no longer covered by the inductive hypothesis and we thus first require a proof for renamings of the form

$$\frac{\Gamma \vdash s : A \qquad \forall x.(\Gamma\, x)\langle\xi\rangle = \Delta\,(\zeta\, x)}{\Delta \vdash s\langle\xi;\zeta\rangle : A\langle\xi\rangle}$$

Reason 2. As already stated, some statements only hold for renaming. Well-known are anti-renaming lemma,

$$\text{injective } \xi \to s\langle\xi\rangle \succ t\langle\xi\rangle \to s \succ t.$$

6.7.2 Syntax with Functors

Recall record types, where we defined records as lists of again types. It is well-known (see e.g. the implementation of [67, 86, 116]) that one could also define records and a specialised type of lists mutually recursive:

```
Inductive ty (k : ℕ) :=
 | var_ty : 𝕀 k → ty k
 | All : ty (1 + k) → ty k
 | base : ty k
 | recty : ty_ℒ k → ty
with ty_ℒ (k : ℕ) :=
 | nil : ty_ℒ k
 | cons : ty k → ty_ℒ k → ty_ℒ k.
```

Using many-sorted syntax (Section 6.4), we would so gain instantiation and a proof of the monad laws. However, this means that we cannot use external primitives of lists and hence does not scale to larger developments.We hence prefer the approach as previously appeared in the thesis of Schäfer [97, Section 9.2].

There are many other examples of functors, for example, vectors and the constant functor $C : X \to Y$ with $C_x\, s := x$ has the mapping function $\mathsf{map}_C\, f\, x := x$. Note that all external sorts can be seen as an instance of a constant functor.

6.7.3 Many-Sorted Syntax

Recall call-by-value System F, F_{CBV}, from Figure 5.4; a syntactic system with two separate sorts of variables (Section 6.4). We have several choices on how to interpret the different variables: First, we can define one instantiation of terms with substitutions, as done in our case. Another possibility, done in Autosubst 1, is to define separate instantiation operations for all variables, which Schäfer et al. [100] call heterogeneous substitutions.

This resulted in distinct instantiation operations for type and value variables, e.g. for values:

$$\_[\_]_{\mathsf{ty}} : \mathsf{vl}^{\,m\,n} \to (\mathbb{I}^m \to \mathsf{ty}^{m'}) \to \mathsf{vl}^{\,m'\,n}$$
$$\_[\_]_{\mathsf{vl}} : \mathsf{vl}^{\,m\,n} \to (\mathbb{I}^n \to \mathsf{vl}^{m\,n'}) \to \mathsf{vl}^{\,m\,n'}$$

In total, Schäfer et al. [100] would require five instantiation operations.

We face the problem that the various instantiation operations interfere and become challenging to permute. Take for examples

$$s[\sigma]_{\mathsf{vl}}[\tau]_{\mathsf{ty}} = s[\sigma]_{\mathsf{ty}}[\sigma \circ [\tau]_{\mathsf{ty}}]_{\mathsf{vl}}$$

where permuting the two instantiations requires us to replace types in substituted values. Even more important, this made Autosubst fail to scale to mutual inductive sorts like those of our example F_{CBV}: It is impossible to define instantiation of terms and values separately, as each requires the other definition.

Vector substitutions have several practical advantages: First, users have just one notation for instantiation, improving *accessibility*. As users do not have to solve equations

for interacting instantiation operation manually, our tool improves in *accessibility* and *conciseness*. The same holds for the following lemmas, where often vector substitutions simplify the lemma statements and allow to combine several lemmas. We also think that the various instantiation operation impact *transparency*.

6.7.4 Variadic Syntax

Note that variadic syntax generalises the monadic operations to variadic operations:

Fact 6.29 (Coincidence of Monadic and Variadic Primitives).

1. $\uparrow \equiv \uparrow^1$
2. $s \cdot \tau \equiv (s \cdot \mathsf{id}_0) \cdot_1 \tau$
3. $0 \cdot \mathsf{id}_0 \equiv \mathsf{hd}_1$

Proof. Directly by definition. □

For extension and the head renaming, we used the extension with the empty substitution to transform a term to a substitution, e.g. $s \cdot \mathsf{id}_0$.

Chapter 7

Modular Syntax

In the last chapter, we have seen how to define and reason about de Bruijn algebras for a variety of formal systems. In this chapter, we extend de Bruijn algebras to modular syntax. This extension is different from the previous ones in that it does not just cover more complex syntactic systems and primitives to reason about syntax with binders, but also primitives to reason about the modular composition of proofs.

The practical benefit of modular syntax and component reuse in a proof assistant is undoubted: Despite all efforts, 15 years after the POPLMark challenge [12], mechanising proofs concerning syntax with binders in proof assistants is still considered hard and one does not want to duplicate these efforts. Besides the treatment of binders, both the already mentioned POPLMark and the POPLMark Reloaded [4] challenge hence mention *component reuse* as one of the evaluation criteria for a practical development. Component reuse covers both reusing definitions and parts of proofs. However, to the best of our knowledge, all submitted solutions to either challenge follow a copy-paste approach and do not actually reuse proofs.

Copy-pasting proofs results in inelegant and hard-to-maintain developments, but so far there is no convenient alternative. Although suggestions how to use modular syntax for proof assistants like Coq and Agda exist [19, 37, 66, 79, 101], we failed to locate a development based on one of the proposed solutions, apart from the case studies contained in the publications. Most solutions suffer from much more complex definitions and proofs than their non-modular counterparts and hence defeat the point.

We propose a solution based on injections and functors. This surprisingly simple approach indeed scales up to complex proofs, like the strong normalisation of a λ-calculus extended with boolean and arithmetic expressions, which we present in Section 9.6.

The first steps are very similar to Swierstra's Data Types à la Carte approach [109]. We define **features**, which specify a subpart of the formal system as functors parameterised by the overall type. For example, arithmetic, boolean, and lambda features are encoded as:

```
Arith X = (X, X) + ℕ    -- addition and natural constants
```

```
Booleans X = (X, X, X) + 𝔹 -- if and boolean constants
Lambda X = ℕ + (X, X) + X -- variables, application, abstraction
```

We then define **variants** of data types which combine several of these features via the fixpoint of a coproduct. Different from Swierstra, these types are defined dynamically and not an instantiation of a more general type. See Section 7.5 for a more thorough comparison.

In the next step, we define **smart constructors**, which lift constructors to variants, modular recursive functions, and modular proofs. Our first attempt of stating proofs requires that termination is re-checked and is hence not fully modular. In Section 7.2, we therefore develop (and generate) alternative induction principles.

In Section 7.3 we define custom substitution support for modular syntax on the example of a λ-calculus and extensions, i.e. we use the previously developed design principles to construct a full **modular de Bruijn algebra**.

The whole approach is extended to modular dependent predicates. Similar to the previous sections, we provide a modular proof that reduction is substitutive.

In the last section of this chapter, we compare our approach to related work. In a case study proving monotonicity and type soundness of a big presentation of mini-ML, our approach required around a fifth of the line of code of comparable approaches.

Unlike the previous chapters, this chapter uses only *pure* de-Bruijn syntax since, as of now, Autosubst does not support scoped syntax yet. This is not an inherent flaw in the approach and future work.

This chapter is based in significant parts on [43].

7.1 Modular Syntax

We give a high-level overview of modular syntax via our adaption of the Data Types à la Carte approach [109] to Coq and compilation via Autosubst .We base our account on the EHOAS specification in Figure 5.12. This section largely uses Coq syntax opposed to mathematical syntax.

7.1.1 Modular Inductive Data Types

We start with exemplary **feature functors**, defined in Figure 7.1. We treat variables as a separate feature because we want several features to access variables. Each feature is parameterised by the full type exp, which is only defined at the very end. As a convention, feature functors always have a symbol as a subscript, e.g. exp_λ.

We can combine different features to a **variant**, as depicted in Figure 7.2. The user will be able to declare these in a specification. We write exp_1, exp_2, and exp_3 for different **variants** of exp. In our later proof development, each variant will be generated in a

```
Inductive var (exp : Type) :=
| var : ℕ → var exp.

Inductive exp_λ (exp : Type) :=
| app : exp → exp → exp_λ exp
| abs : exp → exp_λ exp.

Inductive exp_𝔹 (exp : Type) :=
| const_𝔹 : 𝔹 → exp_𝔹 exp
| if : exp → exp → exp → exp_𝔹 exp.

Inductive exp_+ (exp : Type) : Type :=
| plus : exp → exp → exp_+ exp
| constNat : ℕ → exp_+ exp.
```

Figure 7.1: Feature functors.

```
Inductive exp_1 :=
| inj_var : var exp_1 → exp_1
| inj_λ : exp_λ exp_1 → exp_1.

Inductive exp_2 :=
| inj_var : var exp_2 → exp_2
| inj_λ : exp_λ exp_2 → exp_2
| inj_𝔹 : exp_𝔹 exp_2 → exp_2.

Inductive exp_3 :=
| inj_var : var exp_3 → exp_3
| inj_λ : exp_λ exp_3 → exp_3
| inj_ℕ : exp_+ exp_3 → exp_3.
```

Figure 7.2: Generated instantiations.

separate file and we can hence use the original names. We will speak of **feature sorts** for the specialised sorts $\mathsf{exp}_\lambda\mathsf{exp}$ and $\mathsf{exp}_\mathbb{B}\mathsf{exp}$, and of a **variant sort** for a variant of exp.

7.1.2 Modular Constructors

We want to lift the constructors from features, e.g. app, to constructors for an instantiation, e.g. exp_2. **Smart constructors** [109] combine the constructors of the modular type $\mathsf{exp}_\mathbb{B}$ with the actual constructors of exp_2, i.e.

```
Definition app_B s t := inj_B (app s t).
```

However, more variants would require a new definition each and again lead to code duplication. This is even a problem for generated code, as we might want to define a type parameterised by the variant.

We hence mirror the approach of Swierstra and define **tight retracts** between types [104], using Coq's type classes [105]:

```
Class X <: Y := { inj : X → Y;  retr : Y → 𝒪(X);
                  retract_works : ∀x, retr (inj x) = Some x;
                  retract_tight : ∀x y, retr y = Some x → inj x = y}.
```

The function inj of a retract is injective, i.e. if inj x = inj y, then also $x = y$.

Let us give an exemplary proof that previous definitions are indeed retracts (later, we will generate the retract proofs automatically):

Lemma 7.1. $\mathsf{exp}_\mathbb{B}\mathsf{exp} <: \mathsf{exp}$.

Proof. We define inj to be $\mathsf{inj}_\mathbb{B}$, retr to be the function which maps $\mathsf{inj}_\mathbb{B}(s)$ to $\lfloor s \rfloor$, and every other expression to $\emptyset$. It follows directly that this is a retract. For tightness, we do a case analysis on y. If $y = \mathsf{inj}_\mathbb{B}(y')$, the equality follows as $\mathsf{inj}_\mathbb{B}$ is injective. Otherwise, $\lfloor s \rfloor$ and $\emptyset$ cannot be equal and we have a contradiction. □

We hence define the following instance of the type class:

```
Instance exp_retract_B : exp_B exp <: exp.
```

Using the retract typeclass, we define a more general version of application, e.g.:

```
Definition app_< {exp} (exp_λ exp <: exp) s t := inj (app s t).
```

Similarly, we define constructors $\mathsf{if}_<$ and $\mathsf{var}_<$ and $\mathsf{c}_<$ to use in arbitrary contexts:

```
Check (app_< (if_< (c_< true) then var_< 1 else var_< 2) t).
```

Autosubst automatically generates the proofs for `exp_retract_λ`, `exp_retract_B`, and the definition of smart constructors, e.g. $\mathsf{app}_<$, $\mathsf{var}_<$, and $\mathsf{if}_<$ _ then _ else _.

```
Section 𝔹.
Variable exp: Type.
Variable _⟨_⟩ : (ℕ → ℕ) → exp → exp.

Definition _⟨_⟩𝔹 : (ℕ → ℕ) → exp𝔹 exp → exp :=
 fun s ⇒ match e with
 | const𝔹 ⇒ const𝔹<
 | if s then t else u ⇒ if< s⟨ξ⟩ then t⟨ξ⟩ else u⟨ξ⟩
 end.

End 𝔹.
```

Figure 7.3: Definition of $\_\langle\_\rangle_{\mathbb{B}}$.

```
...
Variable ⇑tm* : (ℕ → ℕ) → ℕ→ ℕ.

Definition _⟨_⟩λ : (ℕ → ℕ) → expλ exp → exp :=
 fun s ⇒ match e with
 | λ.s ⇒ λ<.s⟨⇑tm* ξ⟩
 | app s t ⇒ app< s⟨ξ⟩ t⟨ξ⟩
 end.
```

Figure 7.4: Definition of $\_\langle\_\rangle_{\lambda}$.

7.1.3 Recursive Functions on Modular Syntax

Our modular definition of expressions allows us to define **modular functions**. In this section, we define a modular version of instantiation with renamings,

$$\_\langle\_\rangle : (\mathbb{N} \to \mathbb{N}) \to \mathsf{exp} \to \mathsf{exp}.$$

We first focus on exp_2 and modularly add the definitions for exp_3 later on.

The definition consists of two steps: First, the definition of the feature functions, then their composition.

For each **feature function**, we parameterise over a type `exp` and a function $\_\langle\_\rangle : (\mathbb{N} \to \mathbb{N}) \to \mathsf{exp} \to \mathsf{exp}$, and define functions

$$\begin{aligned}\_\langle\_\rangle_{\mathsf{var}} &: (\mathbb{N} \to \mathbb{N}) \to \mathsf{var}\ \mathsf{exp} \to \mathsf{exp}\\ \_\langle\_\rangle_{\lambda} &: (\mathbb{N} \to \mathbb{N}) \to \mathsf{exp}_{\lambda}\mathsf{exp} \to \mathsf{exp}\\ \_\langle\_\rangle_{\mathbb{B}} &: (\mathbb{N} \to \mathbb{N}) \to \mathsf{exp}_{\mathbb{B}}\mathsf{exp} \to \mathsf{exp}\end{aligned}$$

for all features contained in exp_2. In Coq, we set parameters using the `Variable` command in a section.

```
Fixpoint _⟨ξ⟩ (s : exp₂) : exp :=
 match s with
 | inj_var  s ⇒ s⟨ξ⟩^⟨_⟩_var
 | inj_λ  s ⇒ s⟨ξ⟩^⟨_⟩;⇑^tm_λ
 | inj_𝔹  s ⇒ s⟨ξ⟩^⟨_⟩_𝔹
 end.
```

Figure 7.5: Definition of instantiation with renamings.

Figure 7.3 shows the exemplary definition of $\_\langle\_\rangle_{\mathbb{B}}$. Note that the result type is (and has to be) the variant sort exp. This is a practice we keep up: *Feature sorts occur only in the recursive argument.* As a consequence, we have to use smart constructors.

The feature function for lambda, see Figure 7.4, requires the additional assumption that lifting of renamings exists. This function will only be defined for the variant sort. We again add the additional parameter with the `Variable` command.

We now turn to the definition of the **variant function**. After closing the section, the parameters appear in the definition and we obtain e.g. a function parametrised in the type `exp` and the function $\_\langle\_\rangle$,

$$\_\langle\_\rangle_{\mathbb{B}} : \forall \mathtt{exp}, ((\mathbb{N} \to \mathbb{N}) \to \mathtt{exp} \to \mathtt{exp}) \to (\mathbb{N} \to \mathbb{N}) \to \mathtt{exp}_{\mathbb{B}}\ \mathtt{exp} \to \mathtt{exp}.$$

We write applications to a function f and an expression s and renaming ξ as $s\langle\xi\rangle^{f}_{\mathbb{B}}$.

Instantiation for $\mathtt{exp}_2$ can then be obtained, as shown in Figure 7.5, by a simple case analysis calling the respective feature functions. Note that the lambda variant requires a previously defined lifting operation for renamings. Since e.g. $\_\langle\_\rangle_{\mathbb{B}}$ only uses $\_\langle\_\rangle$ on structurally smaller arguments, this definition is terminating.

At this point, we are finished with instantiation with renamings. However, for later proofs, we need to assume the following connection between the feature and variant function (see e.g. Section 7.3):

$$(\mathsf{inj}\ s)\langle\xi\rangle = s\langle\xi\rangle_{\mathbb{B}} \tag{7.1}$$

We call this property **feature coincidence**. For the individual variants, the proof is trivial as the terms are equal by definition.

7.1.4 Proofs on Modular Syntax

We now turn to proofs over modular syntax. Since proofs are just dependently typed functions, the principles of the last section remain unchanged. As an example, we show that instantiation with the identity renaming preserves identity. Again, we split the proof into **feature proofs** and **variant proofs**.

For *feature proofs*, we add the following parameter to the section:

```
Variable ren_id : ∀ s : exp, s⟨id⟩ = e.
```

We then show the statement for the separate features:

```
Lemma ren_id_var :
 ∀ s : var exp, s⟨id⟩_var = s. Proof. (* ... *) Defined.
Lemma ren_id_𝔹 :
 ∀ s : exp_𝔹 exp, s⟨id⟩_𝔹 = s. Proof. (* ... *) Defined.
```

All proofs are by an easy case analysis on `e`. For example, in the `if` case, we have to prove that

$$\text{if } (s\langle \text{id}\rangle) \text{ then } (t\langle \text{id}\rangle) \text{ else } u\langle \text{id}\rangle = \text{if } s \text{ then } t \text{ else } u$$

where $s\langle \text{id}\rangle = s$, $t\langle \text{id}\rangle = t$, and $u\langle \text{id}\rangle = u$ by the assumption `ren_id`, and so the whole claim follows.

In the case of abstraction, we require an additional assumption. Recall that we used the lifting operation. We hence have to assume that lifting will preserve identity, which will be proven when proving a variant correct:

```
Variable up_ren_id_tm_tm : ⇑^tm_tm var ≡ var.
Lemma ren_id_λ :
 ∀ s : exp_λ exp, s⟨id⟩_λ = s. Proof. (* ... *) Defined.
```

In the proof the following equation will turn up:

$$\lambda_<.s\langle \text{var}\rangle_\lambda = \lambda_<.s\langle \Uparrow^{*\text{tm}}_{\text{tm}} \text{var}\rangle \lambda_<.s\langle \text{var}\rangle = \lambda_<.s$$

The **variant lemma** for e.g. exp_2 follows from the respective lemmas for v^{exp}, exp_λ, and $\text{exp}_\mathbb{B}$. To actually prove the lemma, we first have to prove the corresponding law for the lifting lemma:

```
Definition up_ren_id_exp_exp : ...

Fixpoint ren_id (s : exp_2) : s⟨id⟩ = s.
Proof.
 destruct s; cbn;
 [ apply ren_id_var
 | apply ren_id_λ with (⇑^tm_tm :=⇑^tm_tm) (up_ren_id_exp_exp := up_ren_id_exp_exp)
 | apply ren_id_𝔹 ];
 eauto.
Qed.
```

Since Coq's induction principle for exp_2 is too weak, we do the proof by direct recursion[1] on the expression `e` rather than induction. We fix this deficiency in Section 7.2 and introduce modular induction principles.

[1] In the Coq proofs, this requires that `ren_id_λ` is closed via the `Defined` and not the `Qed` keyword.

7.1.5 Introduction of New Features

When we extend our definitions to the type exp_3, we have to define $\_\langle\_\rangle_+$ and prove that it preserves identity.

For this, we need a new section. We prove:

```
Section Arith.
Variable exp : Type.
Variable _⟨_⟩ : (ℕ → ℕ) → exp → exp.

Definition _⟨_⟩₊ : (ℕ → ℕ) → exp₊ exp → exp :=
 fun s ⇒ match s with
 | const_ℕ ⇒ const_ℕ<
 | s + t ⇒ s⟨ξ⟩ +< t⟨ξ⟩
 end.

Lemma ren_id_var :
∀ s : exp₊ exp, s⟨id⟩₊ = s. Proof. (* ... *) Defined.
```

The combining function and lemma are then defined identically to before, with just one further case. Note that for arithmetic expression we neither use lifting of renamings, nor the identity law for lifting.

Until now, our definitions offer the same power as the dynamically extensible types of Data Types à la Carte [109]. We essentially defined simple modular data types, modular functions over them and extended the approach to proofs. Autosubst supports the automatic definition of feature functors and combined types as we have already seen in a previous chapter, and we provide commands to define and combine modular functions and lemmas directly.

7.2 Modular Induction Principles

The induction principle Coq generates for e.g. the type exp_2 reads as follows:

```
exp₂_ind :
∀ P : exp₂ → Prop,
(∀ s : var exp₂, P (inj_var s)) →
(∀ s : exp_λ exp₂, P (inj_λ s)) →
(∀ s : exp_𝔹 exp₂, P (inj_𝔹 s)) → ∀s : exp₂, P s.
```

Note that there are no induction hypotheses available. We fix this problem by automatically generating a stronger induction principle as part of the output of Autosubst, called the **modular induction principle**. Using the induction principle avoids re-checking of termination in instantiated proofs like `ren_id`.

The strong induction principles are built on a notion of syntactic **subexpressions**, defined in Figure 7.6). For example, s is a subexpresion of app s t, written $s \in_\lambda$ app s t.

$$
\begin{aligned}
e \in_{\mathsf{var}} \mathsf{var}\ x &:= \bot \\
e \in_{\lambda} \mathsf{app}\ s\ t &:= e = s \vee e = t \\
e \in_{\lambda} \lambda.s &:= e = s \\
e \in_{\mathbb{B}} \mathsf{const}_{\mathbb{B}}\ b &:= \bot \\
e \in_{\mathbb{B}} (\mathsf{if}\ s\ \mathsf{then}\ t\ \mathsf{else}\ u) &:= e = s \vee e = t \vee e = u
\end{aligned}
$$

Figure 7.6: Containment for expressions.

Theorem 7.2. `exp₂_induction :`
`∀ P : exp₂ → Prop,`
`(∀ s : var exp₂, (∀ s' ∈var s. P s') → P (inj_var s)) →`
`(∀ s : exp_λ exp₂, (∀ s' ∈λ s. P s') → P (inj_𝔹 s)) →`
`(∀ s : exp_𝔹 exp₂, (∀ s' ∈𝔹 s. P s') → P (inj_𝔹 s)) →`
`∀ s : exp₂, P s.`

Proof. By induction on s. □

Using the modular induction principle, we can obtain an alternative modular proof that instantiation with the identity renaming yields the original term by proving the following lemma first, which can now be defined opaquely using `Qed` in Coq:

Lemma 7.3.

1. *If* $\forall s' \in_{\mathsf{var}} s.\ s'\langle \mathsf{id} \rangle = s'$, *then* $s\langle \mathsf{id} \rangle_{\mathsf{var}} = s$.
2. *If* $\forall s' \in_{\lambda} s.\ s'\langle \mathsf{id} \rangle = s'$, *then* $s\langle \mathsf{id} \rangle_{\lambda} = s$.
3. *If* $\forall s' \in_{\mathbb{B}} s.\ s'\langle \mathsf{id} \rangle = s'$, *then* $s\langle \mathsf{id} \rangle_{\mathbb{B}} = s$.

Lemma 7.4. *For all* $e : \mathsf{exp}_2$, $e\langle \mathsf{id} \rangle = e$.

Proof. By the induction principle from Theorem 7.2 and Lemma 7.3. □

7.3 Modular de Bruijn Algebras

We continue and generalise the whole definition of a de Bruijn algebra from the previous sections to modular syntax. We sketch how this transformation works for the EHOAS specification from Figure 5.12. We take the lambda feature as an example.

The structure is entirely analogous to Chapter 3. We hence continue with the instantiation with substitutions. For example, feature instantiation for lambda terms assumes

a function $[\_] : (\mathbb{N} \to \mathsf{exp}) \to \mathsf{exp} \to \mathsf{exp}$, a lifting operation $\Uparrow^{\mathsf{tm}}_{\mathsf{tm}}$, and has the following type:

$$[\_]_\lambda : (\mathbb{N} \to \mathsf{exp}) \to \mathsf{exp}_\lambda \mathsf{exp} \to \mathsf{exp}$$

Note that according to our design principles instantiation takes the variant sort exp as result, and not the feature sort. Again, we assume that the feature function and variant function coincide on lambda expressions:

$$(\mathsf{inj}\ s)[\sigma] = s[\sigma]_\lambda.$$

To prove that $s[\mathsf{var}]_\lambda = s$ we need to parameterise the definition by a variable constructor; and again that the lifting operation preserves identities. Otherwise, we parameterise by the full statement as before. The same holds for coincidence and extensionality.

The definition of composition requires special care. Let us start with the definition of composition of renamings. For the lambda feature, we want to prove the following law:

$$s\langle\xi\rangle_\lambda\langle\zeta\rangle = s\langle\xi \circ \zeta\rangle_\lambda.$$

Recall the type of feature instantiation for renamings and note that for the composition only the inner definition uses feature instantiation, while the outer instantiation is variant instantiation. This is necessary to even type-check and will be the point where we first need feature coincidence (Equation 7.1).

We now prove compositionality (cf. Lemma 3.4); as always parameterising the proof with the fact that variant instantiation already is compositional for the full sort. For example, in the case of application, we show that:

$$\begin{aligned}
(\mathsf{app}\ s\ t)\langle\xi\rangle_\lambda\langle\zeta\rangle &= (\mathsf{app}_<\ (s\langle\xi\rangle)\ (t\langle\xi\rangle))\langle\zeta\rangle \\
&= (\mathsf{app}\ (s\langle\xi\rangle)\ (t\langle\xi\rangle))\langle\zeta\rangle_\lambda && \text{Equation 7.1} \\
&= \mathsf{app}_<\ (s\langle\xi\rangle\langle\zeta\rangle)\ (t\langle\xi\rangle\langle\zeta\rangle) \\
&= \mathsf{app}_<\ (s\langle\xi \circ \zeta\rangle)\ (t\langle\xi \circ \zeta\rangle) && \text{IH} \\
&= (\mathsf{app}\ s\ t)\langle\xi \circ \zeta\rangle_\lambda
\end{aligned}$$

To prove the statement, we first reduce instantiation with renamings. In the next step, we do not reduce anymore, as at this point we technically do not know how full instantiation with renamings reduces, except in the case of an injection (hence Equation 7.1). We rewrite with this equation and then continue with the usual proof. The same procedure holds for all substitution components.

Last, let us observe that for the variant proofs, we need to know all parameters the definition depends on. For example, for instantiation with substitutions, we have to insert all the dependencies for instantiation with renamings. These dependencies are regular and can (and will) be deduced automatically.

For the *equational theory* of a modular λ-calculus, we add the usual laws. Further, our rewriting method registers the lemmas of feature coincidence for renamings and substitutions.

Note that to later use the substitution support of Autosubst for a feature in a modular manner, we have to assume all the intermediate statements from this section.

7.4 Modular Inductive Predicates

We extend our approach to modular inductive predicates with dependent types over modular syntax. This covers reduction and a simple proof of substitutivity of reduction to showcase the substitution support. From now on, we switch to a more mathematical representation.

We start with the definition of reduction. In total, we provide three **feature relations**, depicted in Figure 7.7:

$$\_ \succ_{\mathbb{B}} \_ : \mathsf{exp} \to \mathsf{exp} \to \mathsf{Prop}$$
$$\_ \succ_{\lambda} \_ : \mathsf{exp} \to \mathsf{exp} \to \mathsf{Prop}$$
$$\_ \succ_{+} \_ : \mathsf{exp} \to \mathsf{exp} \to \mathsf{Prop}$$

As before, the relation is parameterised by a type `exp`, retracts like $\mathtt{exp}_{\mathbb{B}}$ `exp` <: `exp`, and a relation $\_ \succ \_ :$ `exp` → `exp` → `Prop`. For both the definitions and the later proofs, we further have to assume that a modular de Bruijn algebra exists.[2] Note that we use smart constructors everywhere; otherwise, the definition is identical to a non-modular one.

Variant reduction, in our case for the type with all features, then combines the different predicates, see Figure 7.8.

Similar to the assumption of retracts between types, we have to assume that the modular versions of predicates can be embedded into the full predicates. We do not require the equations of a retract, because as long as there is a proof for predicates the proof itself is irrelevant:

$$s \succ_i t \to s \succ t \tag{7.2}$$

In the case of later inversions (which will be needed in Part 3 of the thesis), we also require the corresponding law for tight retracts:

$$\mathsf{inj}\, s \succ t \to \mathsf{inj}\, s \succ_i t \tag{7.3}$$

We now give a modular proof of substitutivity for this language:

$$s \succ t \to s[\sigma] \succ t[\sigma]$$

[2] At the moment, this has unfortunately to be assumed via explicit `Variable` commands. Autosubst provides the necessary assumptions to copy-paste. We have posted a feature request for the import of variables.

$$\frac{}{\mathsf{if}\ \mathsf{const}_{\mathbb{B}}\, b\ \mathsf{then}\ t\ \mathsf{else}\ u \succ_{\mathbb{B}} \mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ u}$$

$$\frac{s \succ s'}{\mathsf{if}_<\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \succ_{\mathbb{B}} \mathsf{if}_<\ s'\ \mathsf{then}\ t\ \mathsf{else}\ u}$$

$$\frac{t \succ t'}{\mathsf{if}_<\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \succ_{\mathbb{B}} \mathsf{if}_<\ s\ \mathsf{then}\ t'\ \mathsf{else}\ u}$$

$$\frac{u \succ u'}{\mathsf{if}_<\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \succ_{\mathbb{B}} \mathsf{if}_<\ s\ \mathsf{then}\ t\ \mathsf{else}\ u'}$$

$$\frac{}{\mathsf{app}_<\ (\lambda_{<\mathcal{A}}.s)\ t \succ_\lambda s[t..]_<} \qquad \frac{s \succ s'}{\mathsf{app}_<\ s\ t \succ_\lambda \mathsf{app}_<\ s'\ t}$$

$$\frac{t \succ t'}{\mathsf{app}_<\ s\ t \succ_\lambda \mathsf{app}_<\ s\ t'} \qquad \frac{s \succ s'}{\lambda_{<\mathcal{A}}.s \succ_\lambda \lambda_{<\mathcal{A}}.s'}$$

$$\frac{s \succ s'}{s +_< t \succ_+ s' +_< t} \qquad \frac{t \succ t'}{s +_< t \succ_+ s +_< t'} \qquad \frac{}{\mathsf{atom}_<\ m +_< \mathsf{atom}_<\ n \succ_+ \mathsf{atom}_<\ (m+n)}$$

Figure 7.7: Feature reduction.

$$\frac{s \succ_{\mathbb{B}} s'}{s \succ s'} \qquad \frac{s \succ_\lambda s'}{s \succ s'} \qquad \frac{s \succ_+ s'}{s \succ s'}$$

Figure 7.8: Variant reduction.

This will give us the possibility to see how substitution proofs on a modular de Bruijn algebra work.

We start with the **feature proofs**. Similar to before, we use the modular versions $\succ_i$ in the modular statements for arguments we want to do induction on:

Lemma 7.5. *Assume that if* $s \succ t$, *then* $s[\sigma] \succ t[\sigma]$. *Then the following three implications hold:*

1. *If* $s \succ_{\mathbb{B}} t$, *then* $s[\sigma] \succ t[\sigma]$.
2. *If* $s \succ_{\lambda} t$, *then* $s[\sigma] \succ t[\sigma]$.
3. *If* $s \succ_{+} t$, *then* $s[\sigma] \succ t[\sigma]$.

Proof. For both boolean and arithmetic expressions, the proof is straightforward. We do induction on the first statement, and then show that we can imitate these proofs together with the substitution.

Let us give the example for the following rule:

$$\frac{s \succ s'}{\mathsf{if}_{<}\, s\, \mathsf{then}\, t\, \mathsf{else}\, u \succ_{\mathbb{B}} \mathsf{if}_{<}\, s'\, \mathsf{then}\, t\, \mathsf{else}\, u}$$

We have to show that $\mathsf{if}_{<}\, s\, \mathsf{then}\, t\, \mathsf{else}\, u[\sigma] \succ \mathsf{if}_{<}\, s'\, \mathsf{then}\, t\, \mathsf{else}\, u[\sigma]$. To even apply the corresponding inductive rule, we have to apply Equation 7.2 to prove the corresponding statement for $\succ_{<}$. Applying the respective rule, we have to show that $s[\sigma] \succ s'[\sigma]$ which follows with our global inductive hypothesis. All cases except β-reduction follow exactly this pattern.

Here, following the same pattern as before, we are left with the following equation:

$$s[(t \cdot \mathsf{var}) \circ [\sigma]] = s[\mathsf{var}\, 0_{\mathbb{I}} \cdot \sigma \circ \langle\uparrow\rangle][t[\sigma] \cdot \mathsf{var}]$$

Note that this statement contains no feature definitions. We hence have the same proof for substitutivity as before, only that we use the assumed properties. In the Coq proof, this part will be automated by an invocation of our simplification tactic. □

The only part in which this proof deviates from a non-modular one is the intermediate step of calling the retract property, and the additional equations for feature coincidence. We will provide automation support for both.

The rest of the proof is then very similar to the proofs in the last section:

Theorem 7.6. *If* $s \succ t$, *then* $s[\sigma] \succ t[\sigma]$.

Proof. By induction on $s \succ t$ and Lemma 7.5. As we have assumed several substitution properties, we now have to provide them: This can easily be solved by registering the lemmas with `eauto` (which Autosubst will do automatically). In fact, there is a simple MetaCoq tactic which will solve this statement on its own. □

Concerning the proofs of substitution properties, we are now in a similar position as in the last chapter. In the third part of this thesis, we give a modular proof of strong normalisation using the methods presented in this chapter.

In the third part of this thesis, we then provide more extensive proofs showing strong normalisation of the above calculus with extensions to show that this approach indeed scales.

7.5 Related Work

We compare our approach with the recent literature with a special focus on approaches that adapt Data Types à la Carte to proof assistants. All these approaches fulfil the criterion of true modularity.

Data Types à la Carte. The basic idea is very similar to Swierstra's Data Types à la Carte approach [109]. Swierstra proposes a solution in Haskell, where expressions are defined as a supertype parameterised by a functor `F` which is used to instantiate the type with so-called features.

Instead of our full type Swierstra then defines a general type for variants:

```
data Exp F = In (F (Exp F))
```

With these definitions and the pointwise coproduct on functors :+:, Swierstra would define different **variants** of exp as:

```
Definition exp1 := Exp (expλ :+: var ).
Definition exp2 := Exp (expλ :+: var :+: exp𝔹).
Definition exp3 := Exp (expλ :+: var :+: exp+).
```

Although Swierstra's development makes heavy use of type classes, it fulfils the POPLMark criteria of being concise, transparent, accessible, and truly modular.

However, Coq's positivity checker rejects the definition of Exp, because it would introduce a logical inconsistency via the negation functor $\mathsf{F}(\mathsf{T}) := \mathsf{T} \to \bot$.[3] Of course, in later instantiations we only want to insert strictly positive functors F, but there is no way to encode this invariant in Coq.

To the best of our knowledge, all adaptions of Data Types à la Carte to proof assistants have to add a layer of indirectness to circumvent this problem: Schwaab and Siek [101] formalise the syntax of (some) strictly positive functors and only allow such instances,[4] other approaches work with Church encodings [37] or containers and proof algebras [66].

[3] $\mathsf{Exp}\mathsf{F} \leftrightarrow (\mathsf{Exp}\mathsf{F} \to \bot)$ would be provable, but is contradictory.

[4] This is impossible in Coq, see Section 7.5.

```
Inductive Functor := Id : Functor | Const : Type → Functor.
Fixpoint eval (X : Type) (F : Functor) : Type :=
  match F with | Id ⇒ X | Const Y ⇒ X end.
Inductive mu (F : Functor) := inn : eval (mu F) F → mu F.
(* Error: Non strictly positive occurrence of "mu" *)
(*        in "eval (mu F) F → mu F". *)
```

Figure 7.9: Failing definition in Coq.

Instead of defining one general type, we generate the necessary instances on demand and hence retain all previous advantages. For function definitions, we do not rely on algebras but directly use Coq's built-in functions. We believe that this improves both the transparency and accessibility of our code. In Haskell, definitions are restricted to polymorphic, non-dependent types, and hence neither dependent functions nor dependent predicates are handled. However, the ideas scale, as demonstrated by our case studies.

Data Types à la Carte works with injections. Every injection in Haskell morally corresponds to a tight retract in Coq, which we require for our proofs.

Modular Type Safety Proofs in Agda. Schwaab and Siek [101] adapt the Data Types à la Carte approach to Agda and syntactically define a class of strictly positive functors which includes the identity functor, constant functors, products, and coproducts. A function `eval : Functor → Type → Type` is used to evaluate a functor and enables the definition of the least fixed point over strictly positive functors. Due to a more restricted checker for strict positivity, the approach is inapplicable to Coq (see Figure 7.9).

We were unable to obtain the source code for the paper, which makes a comparison difficult. However, in [43] we were able to implement the case study, proving preservation for a language with natural numbers, arrays and options (but no means for case analysis and no abstractions of binders) in about 150 lines of code.Schwaab and Siek mention that their final proof which composes all features is rejected in Agda because termination cannot be verified. In our setting, this does not pose a problem and Coq checks termination instantaneously.

Meta-Theory à la Carte. Delaware, Oliveira, and Schrijvers [37] adapt the Data Types à la Carte approach to Coq via Mendler-style Church encodings. Church-encodings rely on Coq's impredicative sets option and are used as a replacement of inductive data types. Their framework is implemented entirely in Coq and consists of 2500 lines.

The indirectness induced by Church encodings replacing inductive types, algebras replacing of functions and proof algebras replacing proofs impairs the readability of definitions for non-experts impacting transparency and accessibility.

Furthermore, Coq's impredicative `Set` option is known to be inconsistent with classical

logic (excluded middle plus unique choice[5]) and makes constructors of some inductive types lose injectivity.

Generic Datatypes à la Carte. Keuchel and Schrijvers [66] present a solution with binders based on a universe of containers. Containers consist of a type of codes and an interpretation function mapping codes to types. Their framework needs about 3500 lines of code.

From a theoretical perspective, the usage of containers seems to be the most satisfying approach, since it subsumes, for example, the strictly positive functors used by Schwaab and Siek [101]. From a practical perspective, using codes is unsatisfying, since definitions become even harder to read.

Comparison. As a case study, both Delaware et al. [37] and Keuchel and Schrijvers [66] prove monotonicity and type soundness of a big-step presentation of mini-ML. One key challenge in the case study is feature interaction, surfacing as the need to assume inversion properties. For each feature, typing, evaluation, monotonicity, and type preservation both approaches require about 1100 lines of code. We implemented the same case study for comparison in [43]. With our approach, all five features together need about 625 lines of code, i.e. we need about 125 lines per feature while obtaining transparent statements.[6] Big parts of this line difference are in the generation of preliminary code (which seems to be around 1/5th of the code needed in Generic Data Types à la Carte) and the directness of our approach, resulting in fewer lines for function definitions, proofs and tactics. For a more detailed discussion of this big line difference, see [43].

Proof Reuse. A variety of approaches has investigated proof reuse in general [15, 19, 41, 61, 88, 93]. Approaches in the literature span from implementing dedicated proof assistants [41], via extensions of type theory [15] to automated approaches to generalising statements as much as possible [88]. An exhaustive historical overview is available in Section 6.4 of Ringer et al.'s survey on proof engineering [92].

[5] https://github.com/coq/coq/wiki/Impredicative-Set
[6] See directory GDTC.

Chapter 8

The Autosubst Compiler

So far, we have approached syntax from two perspectives: In Chapter 5, we have declared EHOAS, a specification language for syntax with binders. In Chapter 6 and Chapter 7, we have reasoned on specific syntactic systems with binders using de Bruijn syntax. The definitions were feasible but repetitive to establish and generated huge amounts of substitution boilerplate.

In this chapter, we combine the two approaches and compile from EHOAS to de Bruijn syntax. We generalise the examples of the last chapters and automatically generate the substitution boilerplate for a custom EHOAS specification. Our tool is called Autosubst.

Given an EHOAS specification, Autosubst yields Coq code in the form of textual output which can be embedded into (and verified by) the Coq proof assistant. This output contains proof terms, notation commands, and (very simple) custom Ltac scripts establishing the rewriting system of the corresponding σ-calculus. For historical reasons, the Autosubst compiler is realised in Haskell; it does need no specific features of Haskell.

Compilation is split into three phases (Figure 8.1): Given an EHOAS input, Autosubst

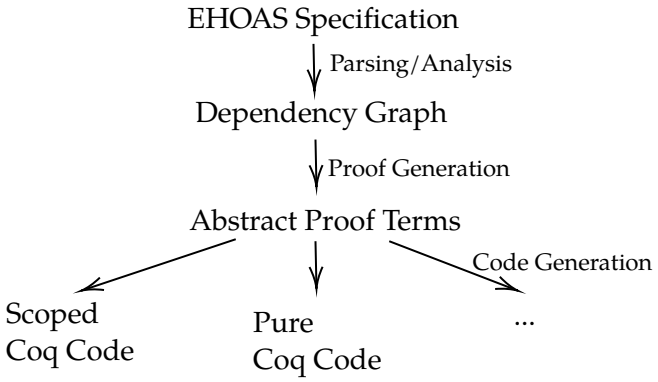

Figure 8.1: Phases of the Autosubst compiler.

first generates a dependency graph with additional information about the free variables and dependencies on other sorts. Dependency graphs with similar information are used in Twelf [84] and Beluga [85]. In the second step, Autosubst derives an abstract type-theoretic representation of the corresponding de Bruijn algebra. In the last step, Autosubst converts the abstract representation to pure or scoped Coq code. This whole process generalises the definitions of the last chapter.

Autosubst equips a sort with the necessary variable constructors only. A minimal solution simplifies the representation of syntax; further, we require minimality to achieve completeness of the rewriting system.

Note that we have no guarantee that the generated Coq code always type-checks or the generated substitution equations are complete for the corresponding de Bruijn algebra.[1] However, this is only a small restriction: Soundness of the generated code is still ensured as all code is verified by the Coq proof assistant, and the automation sufficed in a large number of case studies. Moreover, we have generated the code in a way such that the definition of inductive types, instantiation, and the statements of the substitution laws are readable by a human user.

In the generation of the abstract proof terms, we want to prevent generalisation problems and problems specific due to proof-assistant-specific behaviour. It is hence crucial to be as precise as possible. Precision includes that Autosubst uses as few implicit arguments as possible, does not use any notation during the definition, and finally uses *proof terms* instead of tactics. The last point further ensures that we can extend Autosubst to proof assistants without tactics, e.g. Lean.

We now look at the three compilation phases in detail. For clarity, we first describe the process for non-modular syntax. In Section 8.4, we then extend the previous design with modular syntax. Modular syntax changes compilation, as Autosubst additionally has to respect the dependency structure of features and composed sorts. Moreover, Autosubst generates one instance of a dependency graph for each composed sort (but does not use separate compilation).

The description of the Autosubst compiler in this section is an extended and revised version of the corresponding sections in [108]. The section on tool support for modular syntax is based on the corresponding description in [43].

8.1 Dependency Analysis on EHOAS

EHOAS allows a natural representation of binders, but many properties are left implicit: Which sorts contain variables? Which variables appear (transitively) in which sorts, what are the correct substitution vectors? Which sorts need to be mutually inductive?

Autosubst tracks this information in a dependency graph. For example, the specification

[1] In fact, they are not, consider the type with a single constant which would require a type-driven rule.

of F_{CBV} in Figure 5.4 would be represented as shown in Figure 8.2. Each sort appears as a node of the graph, together with its equipped constructors. Open and closed sorts are marked and annotated with the **substitution vector**, a vector of all necessary substitutions in its header (e.g. `[ty,vl]` for `tm`). The edges of the graph describe the **occurrence relation**, which defines the order in which the graph is processed.

In the following, we first outline how the occurrence relation is constructed, in the second part, how Autosubst handles custom syntax.

Occurrence. Occurrence plays a central role in Autosubst, as it defines the order in which components are processed. First, we say that a sort `y` **occurs directly** in sort `x` if and only if it appears as an argument head in one of `x`'s constructors. We refer to the transitive closure of direct occurrence as **occurrence**.

Occurrence naturally entails a notion of strongly connected components on the sorts. These will be the components that will have to be defined simultaneously. More specific, the corresponding inductive term sorts are declared mutually inductive, instantiation operations are defined mutually recursive, and the equational rules of the affected sorts are proven mutually inductive.

Autosubst then traverses this graph according to its topological ordering, preserving the input order of sorts where possible. The Autosubst compiler then establishes instantiation and substitution lemmas according to this topological ordering. For example, in Section 6.4 F_{CBV} types have to be defined before F_{CBV} terms and values.

Autosubst is now ready to define (1) which sorts are open, i.e. require a variable constructor, and (2) how to determine the correct substitution vector.

Occurrence suffices to characterise (1): A sort `x` is **open** if and only if `x` is bound in a sort `y` and also occurs in `y`. For F_{CBV}, this applies to `ty` and `vl`, but not to `tm`. If `x` does not occur in `y`, the binding is **vacuous**, as we will never see a variable in the respective sort. In this case, Autosubst produces a warning.

We can now state which sorts require an instantiation operation with which variables, i.e. specify the **substitution vector**. A sort `x` requires an instantiation with a substitution for all open sorts `y` which occur in `x`.

External sorts, first-order sorts, and variable sorts. *First-order sorts* and *variable sorts* were introduced in Section 6.5 and allowed a simplified definition of instantiation and the substitution lemmas – short, better code. Autosubst hence retains this information in the dependency graph.

Recall that external sorts are sorts which are defined before the invocation of Autosubst, e.g. booleans and natural numbers; we hence do not want to generate any substitution boilerplate. Autosubst assumes a sort to be external, if it appears in the sort declaration,

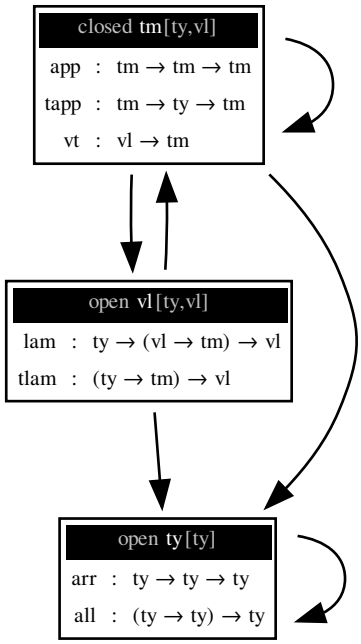

Figure 8.2: Dependency graph of F_{CBV}.

but neither appears in a binder nor occurs as result sort of a constructor.[2] External sorts are never substitution sorts.

For first-order sorts, let us consider an example. Recall that first-order logic as defined in Figure 5.9 consists of two sorts, terms term and formulas form, and generates a dependency graph in which formulas depend on terms, but not vice versa. See Figure 8.3 for the corresponding dependency graph, in which we can see that the term component is not bound in the same strongly connected component as its sort: This is what defines a first-order binder.

A *variable sort* is then a sort with no constructors, but a variable constructor. Again, Autosubst will check the corresponding condition.

8.2 Generation of Abstract Proof Terms

In the next section, we show how Autosubst uses the gathered information to generate the corresponding de Bruijn algebra. The dependency graph yields all information to define custom de Bruijn algebras. Our goal is an abstract representation of the Coq proof terms describing de Bruijn algebra. These components are so general that they could also be interpreted in another type-theoretic proof assistant.

[2]The condition to appear in a binder is necessary to yield the correct output for renaming sorts as in the π-calculus – without this condition, channels would be assumed to be external. The only case in which Autosubst generates the wrong code is if the user wants to define a custom empty sort.

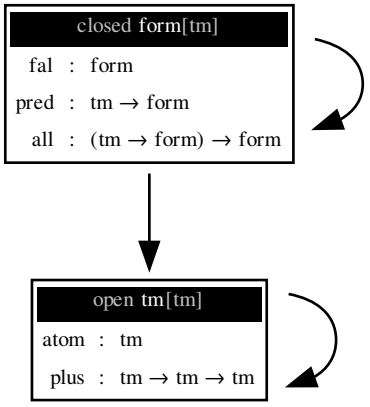

Figure 8.3: Dependency graph of FOL*.

Lifting lemmas might be required in different components. For example, in call-by-value System F, terms require the lifting operation $\Uparrow^{ty}_{ty}$ and corresponding lifting lemmas, but already its types provided these operations (see Section 6.4). During the generation of the abstract proof terms, Autosubst hence remembers which lifting lemmas are already generated. Autosubst generates all possible pairs of lifting lemmas, independent of their actual usage.

To simplify the generation of abstract proof terms, the Autosubst compiler internally uses a distinct abstract type for substitution objects. This type includes predefined instances of scope vectors, renaming vectors, substitution vectors, and equation vectors. Together with this data type, there are specific functions to select subvectors and to lift objects into a new scope. For example, for a renaming vector $[\xi_{ty}; \xi_{vl}]$, adaption to the sort ty reduces the vector to $[\xi_{ty}]$; and lifting by the type component changes the vector to $[\Uparrow^{*ty}_{ty} \ \xi_{ty}; \xi_{vl}]$.

8.2.1 Inductive Sorts

We start with the definition of the inductive sorts.

Sorts in the same strongly connected component have to be declared mutually inductive; each sort corresponds to one inductive type. See Figure 8.4 for the (pretty-printed) output of F_{CBV} terms and values.

Each sort s requires a vector of scope indices $n_1 ... n_k$, one for each element in its substitution vector. For the generation of the inductive term sorts, Autosubst aggregates the constructors, strips binders and, if necessary, adds a variable constructor var_s . The variable constructor takes a finite type with n_s elements as an argument.

```
Inductive tm (n_ty n_vl : ℕ) : Type :=
  | app : tm n_ty n_vl → tm n_ty n_vl → tm n_ty n_vl
  | tapp : tm n_ty n_vl → ty n_ty → tm n_ty n_vl
  | vt : vl n_ty n_vl → tm n_ty n_vl
with vl (n_ty n_vl : ℕ) : Type :=
  | var_vl : 𝕀 n_vl → vl n_ty n_vl
  | lam : ty n_ty → tm n_ty (1 + n_vl) → vl n_ty n_vl
  | tlam : tm (1 + n_ty) n_vl → vl n_ty n_vl.
```

Figure 8.4: Autosubst output for the inductive sorts of F_{CBV}. We removed parentheses and introduced indexed notation for the sake of readability.

If an argument requires a different sort, Autosubst might have to adapt the scope vector to the correct components. For example, the type scope of `lam` consists of only one component, `ty` n_{ty}. If a sort is bound, the respective component of the scope vector is increased by 1 for a monadic binder, by the respective parameter for a variadic binder. In case of a polyadic binder, the scope is increased repeatedly. For example, in `lam` the value scope increases, while in `tlam` the type scope increases.

8.2.2 Instantiation and Substitution Lemmas

The definition of instantiations and the substitution lemmas is split into three phases: Autosubst starts with the instantiation with renamings and substitutions. Autosubst then continues with the inductive lemmas and proves identity for renamings and substitutions, extensionality, compositionality, and the coincidence of renamings and substitutions, as explained in Chapter 6. In the last step, Autosubst proves the supplementary laws.

In each phase, the lifting operations are defined before the actual recursive definition. We will start by explaining the generation of recursive definitions.

Recursive definitions. In Chapter 6, we have already seen the definition of instantiation in great detail. See Figure 8.5 for the (printed) Autosubst output of the identity law for the terms and values of F_{CBV}. We now explain how this law is automatically generated.

First, note that to even *state* instantiation and the substitution laws correctly, Autosubst needs to express vectors of scopes, substitutions, and equations. For example, for $\mathtt{idSubst}_{\mathtt{tm}}$ Autosubst generates the binder vector $[n_{ty} : \mathbb{N}, n_{vl} : \mathbb{N}]$ and the object vector $\mathtt{n} := [n_{ty}, n_{vl}]$, accessible in later definitions. Some objects depend on previous vectors; in this case, the binder vector for substitutions σ depends on the scope $\mathtt{n}$ in both its domain and codomain. Note that the scope of the codomain of σ_{ty} has to be reduced from $[n_{ty}, n_{vl}]$ to $[n_{ty}]$.

```
Fixpoint idSubst_tm { n_ty n_vl : ℕ} (σ_ty : 𝕀 n_ty → ty n_ty) (σ_vl : 𝕀 n_vl → vl n_ty n_vl)
(Eq_ty : ∀x,  σ_ty x = var_ty n_ty x) (Eq_vl : ∀x, σ_vl x = var_vl n_ty n_vl x) (s : tm n_ty n_vl)
: s[σ_ty; σ_vl ] = s :=
  match s with
  | app _ _ s_0 s_1 ⇒ congr_app (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_0) (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_1)
  | tapp _ _ s_0 s_1 ⇒ congr_tapp (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_0) (idSubst_ty σ_ty Eq_ty s_1)
  | vt _ _ s_0 ⇒ congr_vt (idSubst_vl σ_ty σ_vl Eq_ty Eq_vl s_0)
  end
with idSubst_vl { n_ty n_vl : ℕ} (σ_ty : 𝕀 n_ty → ty n_ty) (σ_vl : 𝕀 n_vl → vl n_ty n_vl)
 (Eq_ty : ∀x,  σ_ty x = var_ty n_ty x) (Eq_vl : ∀x, σ_vl x = var_vl n_ty n_vl x) (s : vl n_ty n_vl)
  : s[σ_ty; σ_vl ] = s :=
  match s with
  | var_vl _ _ s ⇒ Eq_vl s
  | lam _ _ s_0 s_1 ⇒ congr_lam (idSubst_ty σ_ty Eq_ty s_0)
    (idSubst_tm (up_vl_ty σ_ty) (up_vl_vl σ_vl) (upId_vl_ty _ Eq_ty) (upId_vl_vl _ Eq_vl) s_1)
  | tlam _ _ s_0 ⇒ congr_tlam (idSubst_tm (up_ty_ty σ_ty)
   (up_ty_vl σ_vl) (upId_ty_ty _ Eq_ty) (upId_ty_vl _ Eq_vl) s_0)
  end.
```

Figure 8.5: Autosubst output for the identity monad law for F_{CBV}.

Next, recall that Autosubst uses a generalised form of the lemmas (Chapter 3). For example, the identity law requires explicit assumptions Eq_{ty} and Eq_{vl} stating that the substitutions σ_{ty} and σ_{vl} behave as the identity for each appearing sort. As expected, the return type states that instantiation with σ returns the original term indeed.

We proceed with the generated proof term. In this case, Autosubst does an induction, and hence in the proof term a recursion on s: `match s with ... end`. Autosubst proceeds with proving the claim for each constructor; Autosubst calls the ith argument of a constructor s_i. In case the respective sort contains a variable constructor, this case has to be handled separately. Here, Autosubst simply invokes the equation Eq_{vl} for value variables. (In the case of instantiation, Autosubst invokes the respective renaming or substitution.)

In the next step, Autosubst has to handle each argument separately with a recursive call and then combine the different proof terms. It is hard to handle equational reasoning with proof terms. Autosubst hence uses custom congruence statement for the inductive definition; for example for term application, Autosubst generates the following congruence law:

$$\mathsf{congr}_{app} : s = s' \to t = t' \to \mathsf{app}\, s\, t = \mathsf{app}\, s'\, t'$$

Autosubst uses these congruence laws in all recursive statements.

Last, let us examine how each argument is handled. Each argument requires again a call of the respective lemma, not necessarily of the same sort. For example, for `tapp`, the proof term calls $idSubst_{tm}$ for s_0 (which is a term) and $idSubst_{ty}$ for s_1 (which is a type).

```
Definition upId_vl_vl { n_ty n_vl : ℕ} (σ : 𝕀 n_vl → vl n_ty n_vl) (Eq : ∀x, σx = var_vl n_ty n_vl x)
: ∀ x, (up_vl_vl σ) x = (var_vl n_ty (1 + n_vl)) x :=
fun n ⇒ match n with
| Some x ⇒ f_equal <id; ↑> (Eq x)
| None ⇒ eq_refl
end.
```

Figure 8.6: Autosubst output for the identity lifting for values on values F_{CBV}.

The vectors used in the definition have to be adapted accordingly (see the invocation of $\texttt{idSubst}_{ty}$). Note that the previous lemma for types already exists because of Autosubst's proceeding via the topological order of occurrence. As terms and values appear in the same strongly connected component, the identity law for terms and values has to be defined mutually recursive.

Last, if an argument has binders, Autosubst requires the correct lifting operation or lifting lemma, depending on the sort handled and the binding argument (both sort and arity). In this case, for example for `lam`, Autosubst lifts all substitution components and equation components correspondingly. The equation components will require lifting lemmas, which we explain in the following. For polyadic binders, we repeatedly apply a lifting operation, as sketched in Section 6.2.

Functors deserve an additional remark. For functors, Autosubst implicitly requires that users provide appropriate mapping functions with appropriate names (for example: $\mathcal{L}$`_map`) and non-opaque lemmas of the functor equations, extensionality, $\mathcal{L}$`_id` and $\mathcal{L}$`_comp`). Autosubst applies the corresponding operation or lemma for each statement.

Autosubst uses a traversal-like function to simplify the generation of both instantiation and the recursive substitution lemmas. The function implements traversals as described by Allais et al. [6] and Kaiser et al. [64], traversing a term and changing the scope if necessary.

Definition of lifting operations. As we have already seen, the initial recursive definitions have a lot to take care of, but in conclusion, are very regular. Liftings are a little different: Most extensions require us to adapt exactly these proofs. In the current situation, Autosubst generates both a monadic and a variadic lifting operation for each lifting pair (x, y).

We pick an easy example to show how establishing lifting lemmas works. See Figure 8.6 for the definition of identity lifting on $\Uparrow^{vl}_{vl}$. Recall that the definition of a lifting $\Uparrow^{x}_{y}$ depends on the fact whether x and y are of the same sort, and the substitution vector of x and y.

The lifting operation does a case analysis on the argument with `match` (in the variadic

case, Autosubst has to call an additional lemma), and handles each case appropriately.

In the proofs in Chapter 6 we have seen, that many laws require long chains of rewriting. These proofs are the hardest part. In the proof term, you can see an application statement, which simplifies proofs:

$$\mathsf{f\_equal} : s = s' \to f\, s = f\, s'$$

More complicated are the composition statements, where Autosubst has to invoke symmetry and transitivity statements for equality explicitly (see Lemma 3.4 for the detailed proof).

For variadic reasoning, proofs get even harder. Autosubst can no longer use pattern matching but has to use the explicit lemmas for a case analysis, which requires additional levels of transitivity, as explained in Section 6.6.

Supplementary laws. For the supplementary laws, the main challenge is to *state* the correct laws: both which laws to state and how to state them. For example, Autosubst requires left identity only for all open sorts.

Otherwise, the proofs follow directly with reflexivity, the respective laws, and functional extensionality.

First-order sorts. Recall that in the case of first-order sorts, the substitution boilerplate (1) consists of slightly different statements and proofs, and (2) the type-theoretic statements have to be stated in an adapted order: Autosubst hence first defines instantiation, and then the lifting operations; analogous for the monad laws (Section 6.5).

8.3 Code Generation

In the last step, Autosubst transforms the abstract proof terms into plain text, readable (and verifiable) by Coq.

This stage profits from the separate treatment of substitution objects: The scoped and pure variant of the generated Coq code only differs in the `Show` instance of scoped objects. See for example Figure 8.7 for the pure representation of our previous definitions: Autosubst replaces the finite type $\mathbb{I}^n$ with a natural number, and removes all scoping information.

Porting instantiation and the substitution laws to a new proof assistant (with the corresponding support, for example for mutually inductive types) is straightforward. See, for example, Mameche [73] for a port of the Autosubst backend to Lean as part of her Bachelor thesis [35].

```
Inductive tm : Type :=
  | app : tm → tm → tm
  | tapp : tm → ty → tm
  | vt : vl → tm
with vl : Type :=
  | var_vl : ℕ → vl
  | lam : ty → tm → vl
  | tlam : tm → vl.

Fixpoint idSubst_tm (σ_ty : ℕ → ty) (σ_vl : ℕ → vl)
(Eq_ty : ∀x, σ_ty x = var_ty n_ty x) (Eq_vl : ∀x, σ_vl x = var_vl x) (s : tm) : s[σ_ty; σ_vl ] = s :=
  match s with
  | app s_0 s_1 ⇒ congr_app (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_0) (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_1)
  | tapp s_0 s_1 ⇒ congr_tapp (idSubst_tm σ_ty σ_vl Eq_ty Eq_vl s_0) (idSubst_ty σ_ty Eq_ty s_1)
  | vt s_0 ⇒ congr_vt (idSubst_vl σ_ty σ_vl Eq_ty Eq_vl s_0)
  end
with idSubst_vl (σ_ty : ℕ → ty) (σ_vl : ℕ → vl)
 (Eq_ty : ∀x, σ_ty x = var_ty x) (Eq_vl : ∀x, σ_vl x = var_vl x) (s : vl) : s[σ_ty; σ_vl ] = s :=
  match s with
  | var_vl s ⇒ Eq_vl s
  | lam s_0 s_1 ⇒ congr_lam (idSubst_ty σ_ty Eq_ty s_0)
    (idSubst_tm (up_vl_ty σ_ty) (up_vl_vl σ_vl) (upId_vl_ty _ Eq_ty) (upId_vl_vl _ Eq_vl) s_1)
  | tlam s_0 ⇒ congr_tlam (idSubst_tm (up_ty_ty σ_ty)
   (up_ty_vl σ_vl) (upId_ty_ty _ Eq_ty) (upId_ty_vl _ Eq_vl) s_0)
  end.
```

Figure 8.7: Pure output of F_{CBV} and the identity monad law.

```
(* Type Class for the Notation *)
Class Subst1 (X1 : Type) (Y Z: Type) :=
  subst1 : X1 → Y → Z.

Notation "s [ σ]" := (subst1 σs) (...).

(* Declared instance of the notation. *)
Instance Subst_ty { n_ty n_ty : ℕ} :
Subst1 (𝕀 n_ty → ty n_ty) (ty n_ty) (ty n_ty)
:= subst_ty n_ty n_ty .

(* Additional Printing Notation. *)
Notation "s [ σ ]" := (subst_ty σ s) (..., only printing).
```

Figure 8.8: Instantiation notation F_{CBV} types.

8.3.1 Automation for Substitutions

Recall that every EHOAS specification should also provide the corresponding variant of the σ-calculus and hence also provides a rewriting system. Under the assumption of functional extensionality, we can rewrite with the corresponding lemmas using the rewrite tactic of Ltac. The tactic `asimpl` rewrites the corresponding lemmas in the goal, respectively for `asimpl in *` in all hypotheses and the goal. Equations which hold definitionally, i.e. for reduction, are not rewritten, but directly reduced using Coq's evaluation tactic to reduce proof term size. Notations (Section 8.3.2) have to be unfolded.

All versions of `asimpl` invoke the custom Ltac tactic `fsimpl` that normalises terms with substitution primitives of the σ-calculus. This corresponds to simplification with (both monadic and variadic) interaction lemmas.

The `asimpl` tactic further accounts for the functor laws. It does so via lemmas registered via `autorewrite`.

A tactic without functional extensionality is possible with our restricted syntax, but has to traverse the term. This requires the (already automatically generated) extensionality lemma (Lemma 3.7).

8.3.2 Notation

We aim for a univariate syntax for instantiation, i.e. users should be able to write $s[\sigma]$ or $s\langle\xi\rangle$ without knowing the exact name of the specific instantiation operation.

Autosubst uses a type class instance to overload the parsing of notation in Coq (see Figure 8.8). Autosubst generates the required instances together with the remaining code. Each instance is unique because of its result sort. As automation works on terms without notation, `asimpl` will need to unfold the type class instances.

The folding of (dependent) instances is difficult, and Autosubst hence defines notations a second time for printing (Figure 8.8). As such, Autosubst will never fold *to* a type class instance.

Similarly, Autosubst allows a univariate syntax for instantiation with a vector substitution: Both for terms and values, users can write:

```
s[σ; τ]
v[σ; τ]
```

Due to technical limitations in Coq, Autosubst requires one type class instance for each vector size; by standard, Autosubst supports notation up to size 5.

Autosubst introduces similar notation for renamings ($A\langle\xi\rangle$ and $s\langle\xi;\zeta\rangle$), variable constructors (`ids`), and the lifting of variables. For example, a scope change that lifts one type variable van be written as $\uparrow^{ty}\ \sigma$, independent of the underlying type of σ.

Renamings and Substitutions. Recall that both renamings and substitutions may appear in a term and that Autosubst provides the following lemma connecting instantiation with renamings and substitutions:

$$s[\xi,\circ\ \mathsf{var}_{ty}] = s\langle\xi\rangle$$

There are of course more equations which hold between instantiation with renamings and substitutions. At the moment, Autosubst's automation tactic does not automatically transform between renamings and substitutions. Autosubst offers tactics which allow users to automatically transform renamings to substitutions (`substify`) and vice versa (`renamify`).

The first direction is the easy one, merely rewriting with the above equation from left to right. On the other side, `renamify` requires several transformations: A substitution of the form $\xi,\circ\ \mathsf{var}_{ty}$ can be directly transformed to a renaming, otherwise, we have to re-parenthesise to the left and then re-try. Moreover, we might have to fold the stream cons, e.g. from $\mathsf{var}_{ty}\ x\cdot(\xi,\circ\ \mathsf{var}_{ty})$ to $(x\cdot\xi)\circ\ \mathsf{var}_{ty}$. Rewriting up to associativity would solve the problem [20].

In general, users should only require the `substify` tactic, in case they want to apply a generalised lemma for instantiation with a renaming.

8.4 Tool Support for Modular Syntax

We now turn to tool support for modular syntax. Recall Figure 5.12 for an example input. To make modular syntax more convenient to use, we implement three kinds of tool support for modular syntax:

1. We extend the automation of Autosubst to support modular syntax (see also Section 7.3). Users can then use instantiation and the `asimpl` tactic that simplifies substitution goals also on modular syntax.

2. Based on the EHOAS input, we implement static code generation of feature functors and variants together with retracts, smart constructors, and modular induction principles.

3. Independent of this thesis, Forster implemented simple dynamic code generation based on MetaCoq [106] to ease the statement of modular fixpoints and lemmas and fully automate the composition of this fixpoint and lemmas. We refer to [43] for more details.

Both (1) and (2) require a strengthened dependency analysis, which we explain in the following. Compared to substitution boilerplate, the generation of the described laws was very straightforward.

8.4.1 Dependency Analysis for Modular Syntax

We now turn to the dependency analysis for modular syntax. We start with the dependency graph for one variant only.

First note that as-is, the specification describes different sorts (the feature sorts and the variants sorts) with the same names. Autosubst resolves this dependency by introducing new names: For each feature f and for every sort s in f with constructors $c_1, \ldots, c_n$ Autosubst generates a functor s_f with constructors $c_1, \ldots, c_n$. The result sort in each constructor is changed from s to s_f. For each specified variant I and all types $T_1, \ldots, T_m$ defined in a feature f of I, Autosubst generates the types $T_1, \ldots, T_m$ combining all specified features in a file I. The constructors of T are called inj_T_f. These constructors are also already added in the description of the dependency graph.

As given, in the dependency graph, there is a mutual dependency between the feature sort and the variant sort; and they would hence have to be defined mutually inductive.[3] This dependency is only resolved as Autosubst assumes all assumptions in the feature sort.

Next, Autosubst has to take care of necessary variables. Recall that a variable feature is added automatically if a sort requires a negative occurrence. The required variables have to stay constant in all features, as otherwise, the mechanisation of variants cannot reuse previous files. As Autosubst resolved the dependency in the last step, we have to omit the additional check that vacuous binders are forbidden.

In the definition of a proof term, Autosubst needs additional information which is collected during the analysis: Is a sort a feature sort, is it an overall sort, or is it completely independent of features? What is the feature a sort belongs to? Which is the overall sort of a feature sort? Does the feature contain binders, and hence require variables? What

[3] This is indeed the right observation once a combination is fixed – however, to keep the definition modular Autosubst axiomatises the behaviour of the parent sort, and still get the correct statements.

are its substitution properties? What are the features of a variant? Autosubst remembers all this information in the dependency graph.

Last, we are ready to generalise the above construction to several variants. For this, for each variant, Autosubst repeats the above process and hence generates several dependency graphs.

Let us conclude with the restrictions of our dependency analysis: First, as it is not clear which variant should be used, nothing may depend on sorts in a variant. Next, variants depend on only features. Last, if a sort is part of a variant, we forbid to include additional constructors. While this is a technical restriction, it does not restrict expressiveness: We could define these constructors in a separate feature.

8.4.2 Modular Syntax with Binders

Once the dependency graph exists, this phase is similar to before. Recall that there are small changes in a modular de Bruijn algebra, as highlighted in Section 7.3. During generation, Autosubst hence frequently has to determine whether the current code generation takes place in a feature or sort of a variant, and adapt code generation accordingly.

Autosubst still uses proof terms for all parts of the proofs; in particular, the generation does not depend on MetaCoq. Recall from Section 7.3 that there are hence various dependencies Autosubst has to take care of: For example, instantiation in the lambda feature depends on a lifting operation defined only for the variant sort. However, in the remainder of the code, these dependencies are very regular. Autosubst remembers which dependencies exist and automatically inserts them in the case the handled sort is indeed a variant sort.

Also the generation of necessary assumptions requires additional care. We define a function which turns a previous definition into a variable; we can hence reuse previous definitions.

8.4.3 Static Code Generation for Modular Syntax

We extend Autosubst's [108] interface to modular types. Autosubst hence also generates custom support for modular syntax, independent of substitution boilerplate: retractions, smart constructors, and induction principles based on this input.

More specifically, these are the helpers Autosubst generates:

1. For each specified instantiation $\mathtt{I}$ and all types $\mathtt{T}_1, \ldots, \mathtt{T}_m$ defined in a feature $\mathtt{f}$ of $\mathtt{I}$, Autosubst proves $\mathtt{T\_F\ T} <: \mathtt{T}$.
2. For each constructor $\mathtt{C}$, Autosubst automatically defines the respective smart constructor called $\mathtt{C\_}$ via injections.

```
Inductive term : Type :=
| Var (x : var)
| App (s t : term)
| Lam (s : {bind term}).
```

Figure 8.9: Specification of λ in Autosubst 1.

3. For every feature `f` defining type `T`, Autosubst generates the predicate `In_T_F`. For every instantiation `I` with instantiated type `T`, Autosubst generates the modular induction principle `induction_T` for `T`.

8.5 Restrictions

During lexing and parsing, Autosubst conducts certain sanity checks: Autosubst ensures that Coq identifiers do not appear as custom-defined sorts and only predefined sorts are used at all. It moreover ensures that the syntax satisfies our restrictions: Autosubst forbids vacuous binders, third-order constructors, bound functors, and negative occurrence of anything but a pure sort or a vector of pure sorts. Autosubst does not check quoted Coq code.

8.6 Comparison to Autosubst 1

Autosubst 1 by Schäfer et al. [99; 100] is the predecessor of the version of Autosubst developed in this thesis. In this section, we speak of Autosubst 1 and Autosubst 2 to make the difference precise. It was the first tool to recognise and exploit the connection between the de Bruijn algebra and the σ-calculus. Autosubst 1 takes as input annotated Coq code (see Figure 8.9 for the specification of the untyped λ-calculus) and generates a custom de Bruijn algebra using Ltac, Coq's internal tactic language [36]. Autosubst 1 was used for a variety of mechanisations [63, 78, 89, 112, 114, 118].

Autosubst 2 follows the general spirit of Autosubst 1 in its usage of de Bruijn substitutions, However, it supports a substantially larger class of syntax which was impossible to achieve in Autosubst 1's design. For example, Autosubst 1 required neither EHOAS nor a dependency analysis.

Most apparent change is the switch of the implementation: Autosubst 1 was developed inside Coq using the Ltac tactic language [36], while Autosubst 2 is an external tool implemented in Haskell. Although useful as a prototype, Ltac handicapped and practically even prevented the extension of Autosubst 1 to broader classes of syntax: First, it is impossible to insert intermediate stages to examine the global state of the syntax. Developing, for example, the dependency analysis with Ltac or generally inside Coq is probably much more difficult and requires global knowledge on the syntactic sorts defined. Further, Ltac supports neither mutual syntax nor custom notations. Moreover,

Ltac's error messages are often obscure, making debugging more difficult for users.

Although code generation requires more work in the beginning, the investment pays off quickly once extensions are considered. It is further more transparent, as users can examine the code that Autosubst generates. This code is human-readable. As an additional advantage, the separation into the type-theoretic interpretation and the printing simplifies the extension to other proof assistants, such as done by Mameche [73].

The different phases allow us to extend our tool with the new interpretation of scoped syntax. Scoped syntax first appeared during the proof of the POPLMark Reloaded challenge [3]. Scoped syntax offers a choice to write scope-safe statements with substitutions (see Section 3.2). As Coq yields a typing error if one forgets the lifting operation, it dramatically improves the accessibility of new users of de Bruijn syntax in Autosubst. Further, many statements can omit previous conditions on the context, improving conciseness. Last, as all terms are annotated by the respective scope, transparency is improved.

In conclusion, Autosubst 2 as-is supports substitution support for first-class renamings, vector substitutions, mutual inductive syntax, functors, variadic syntax, a simplified representation of first-order syntax, and modular syntax.

Part III

Case Studies

Chapter 9

Simply-Typed Lambda Calculus

In the last chapters, we have seen how the Autosubst compiler automatically generates pure and scoped de Bruijn algebras corresponding to syntactic specifications. In the following part, we switch to a more practical point of view and showcase how to use this output to mechanise the meta-theory of programming languages.

The goal of this chapter is threefold: First, we showcase that many standard textbook proofs indeed work with parallel de Bruijn substitutions. Second, we show how to reason with de Bruijn substitutions. Finally, these proofs give us a chance to evaluate the Autosubst compiler against other approaches.

We mainly remain on a mathematical level of presentation, but frequently show the actual implementation and how to handle a specific problem using Autosubst. We assume that the reader is familiar with the standard paper-based proofs of preservation, weak head normalisation, and strong normalisation for the simply-typed λ-calculus but give references to the techniques used.

The recurring topic will be the **monotonicity** (the special case for renamings) and **substitutivity** of different type-theoretic constructions with de Bruijn substitutions. We frequently omit proofs unrelated to this problem, even if they are interesting from a technical perspective. All proofs are fully mechanised, and the Coq development is available online. We present them in increasing order of complexity.

Organisation of the Chapter. We start with our first syntactic system, the λ-calculus. Despite its simplicity, it already showcases many of the practices with de Bruijn substitutions: How to state β- and η-reduction with the primitives of the σ-calculus, how to represent **contexts**, reorderings of contexts, and context morphisms, and the need for first-class renamings.

We start with reduction, $s \succ t$, an inductive predicate with only positive occurrences of open sorts and without variable constructors. If the scoping is respected, substitutivity of such predicates follows directly:

$$s \succ t \to s[\sigma] \succ t[\sigma]$$

The reverse direction of the above implication works for injective renamings only and is also much harder to establish. We typically call such a lemma an **anti-renaming lemma**.

Substitutivity gets more involved for predicates where open terms occur at negative positions; for example, for example, the typing context of a typing judgment. Substitutivity hence requires additional information on the position of variables in the context, for which we follow the ideas by Goguen and McKinna [52] and later by Kaiser et al. [63]: Context renamings generalise renamings to contexts and require a **context renaming lemmas**; context morphisms generalise substitutions to contexts and require a **context morphism lemmas**. These proofs repeatedly require us to solve substitution equations. Substitutivity of typing is necessary to establish **preservation** of typing during reduction, which we do in the next step.

In Section 9.3, we adapt preservation (and hence also context renaming and context morphism lemmas) to the **multivariate λ-calculus** [90]. The proof structure remains unchanged, but the proof requires us to reason about variadic binders. With the right definitions, this extension is trivial.

In the next step, we give semantics to the typing predicate. It is folklore that proofs of **weak head** and **strong normalisation** build a model of terms of the typed λ-calculus. Similar to context morphism lemmas, the semantics of a typed term requires a proper interpretation of the freely appearing variables.

All proofs use **logical relations**, following [50, 51, 110] and hence require an intermediate notion of **candidates of reducibility** (here defined as a logical relation recursive over the type) which are all strongly normalising. The proof then mainly resolves around 1.) proving closure properties of this relation (some of which have to be shown in a mutual induction) and 2.) verifying that all typed terms are included in this relation.

There are different possibilities on how to define this relation both for weak and strong normalisation. In all cases, we use **Kripke-style logical relations** [77], which extend the context in the case of abstraction and hence require a proper handling of renamings.

For weak normalisation, Dreyer et al. [39] divide the logical relation into a value relation defined by recursion on the type and an expression relation independent of the type, hence simplifying the above proof. Our proof of weak normalisation combines this idea with Kripke-style logical relations.

For strong normalisation, we showcase two proofs, both based on Kripke-style logical relation and Girard's method but with a different definition of the logical relation. Both proofs further require different substitution properties.

We first follow a technique due to Schäfer [97], first used in [45]. Similar to the proof of weak normalisation of Dreyer et al. [39], the logical relation is divided into two parts. As strong normalisation is more involved, the expression relation has to be defined via an inductive predicate. We from now on refer to this representation of the expression

$$\frac{}{\mathsf{app}\ (\lambda_{\mathcal{A}}.s)\ t \succ s[t..]} \qquad \frac{s \succ s'}{\mathsf{app}\ s\ t \succ \mathsf{app}\ s'\ t} \qquad \frac{t \succ t'}{\mathsf{app}\ s\ t \succ \mathsf{app}\ s\ t'} \qquad \frac{s \succ s'}{\lambda_{\mathcal{A}}.s \succ \lambda_{\mathcal{A}}.s'}$$

Figure 9.1: Reduction in STLC.

relation as **Schäfer's expression relation**. We can prove general properties about this expression relations, and hence the proof for a specific reduction strategy is relatively straightforward.

We then give a second proof proposed by the **POPLMark Reloaded challenge** [3, 4], which uses **Raamsdonk's characterisation of strong normalisation** from [115] as an intermediate notion. This gives us the possibility to evaluate our solution against other approaches.

We then turn to modular syntax. We continue with where we have left off in Chapter 7 and provide fully modular proofs of preservation, weak head normalisation, and strong normalisation via Schäfer's expression relation. For this, we extend the proofs from the λ-calculus to a λ-calculus with boolean and arithmetic expressions. We know on no similar-sized development for modular proofs.

No definition used η-equivalence so far. In our final case study, we follow a proof by Crary [28] and show that $\beta\eta$-equivalence for a simply-typed λ-calculus implies algorithmic equivalence, needed to show decidability of $\beta\eta$-equivalence (which we do not prove in this thesis). This proof was further mechanised in Beluga [22], which we use as a possibility to compare the proofs. In the proof, we need binary logical relations. To the best of our knowledge, this is the first mechanised proof following Crary [28] using de Bruijn substitutions.

In total, all developments allow concise, transparent, and accessible proofs (Section 9.9).

9.1 Reduction and Values

Recall the EHOAS specification of λ (Figure 5.1). Autosubst creates the definitions as outlined in Chapter 3, together with a custom tactic `asimpl` using the convergent rewriting system. We access instantiation with renamings and substitutions via notation, i.e. `s<ξ>` and `s[σ]`, respectively.

We start with **full reduction**, $s \succ t$, an inductive predicate depicted in Figure 9.1. Open sorts only occur at positive positions and variables do not appear at all. This makes proofs such as substitutivity easy, as the `asimpl` tactic suffices.

We have already seen in Chapter 3 that β-reduction is substitutive. Extension to full reduction is straightforward.

Lemma 9.1 (Substitutivity). *If* $s \succ t$, *then* $s[\sigma] \succ t[\sigma]$.

Proof. By induction on $s \succ s'$. The case of β-reduction is solved with `asimpl`. □

We further write $s \succ^* t$ to denote the **reflexive-transitive closure of reduction**. Substitutivity directly follows from reduction:

Corollary 9.2. *If* $s \succ^* s'$, *then* $s[\sigma] \succ^* s'[\sigma]$.

Similar results hold for restricted forms of reduction, i.e. weak head reduction as defined in Section 9.8.

Reduction can be lifted to substitutions in a natural fashion: we write $\sigma \succ^* \tau$ and say that σ reduces to τ, if for all i, $\sigma\, i \succ^* \tau\, i$.

Fact 9.3 (Congruence). *Assume that* $s \succ^* s'$ *and* $t \succ^* t'$. *Then* $\mathsf{app}\, s\, t \succ^* \mathsf{app}\, s'\, t$, $\mathsf{app}\, s\, t \succ^* \mathsf{app}\, s\, t'$, *and* $\lambda.s \succ^* \lambda.s'$.

Lemma 9.4 (Substitutivity). *If* $\sigma \succ^* \tau$, *then* $s[\sigma] \succ^* s[\tau]$.

Proof. By induction on s, using congruence of reduction (Lemma 9.3). In the case of abstraction, we have to show that $\forall x.\ \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} (\sigma\, x) \succ^* \Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau\, x$. Case analysis on x. If $x = 0$, this follows directly using reflexivity of the reflexive-transitive closure. Otherwise, we have a post-composition with instantiation with $\uparrow$ and hence require substitutivity (Lemma 9.2) and the `substify`-tactic, which transforms instantiation with a renaming into instantiation with the corresponding substitution. □

While substitutivity is easy, anti-substitutivity does not even hold.[1] It does hold for renamings, although the proof is not straightforward, even with Autosubst:

Lemma 9.5 (Anti-Renaming of Reduction). *If* $s'\langle\xi\rangle \succ t$, *then there exists a* t' *such that* $t = t'\langle\xi\rangle$ *and* $s' \succ t'$.

Proof. We start with a dependent induction on $s'\langle\xi\rangle \succ t$, then do a repeated case analysis on the terms in the corresponding equation until we reach a constructor, repeatedly use inversion, and apply the inductive hypothesis. In the case of β-reduction we have to show that:

$$s\langle\xi\rangle[t\langle\xi\rangle..] = s[t..]\langle\xi\rangle$$

which is shown using Autosubst. □

We require the above lemma in the proof of strong normalisation.

[1] Consider x, which is replaced by a reducible term.

$$\frac{}{\Gamma \vdash \mathsf{var}\, x : \Gamma\, x} \qquad \frac{\Gamma \vdash s : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{app}\, s\, t : B} \qquad \frac{A \cdot \Gamma \vdash s : B}{\Gamma \vdash \lambda_A.s : A \to B}$$

Figure 9.2: Typing of λ.

We turn to **substitutivity of functions**. We say that a term is a **value** if it is an abstraction. We define values as a predicate over expressions:

$$\begin{aligned} \mathsf{value}\ (\lambda\_.\_) &= \top \\ \mathsf{value}\ \_ &= \bot \end{aligned}$$

As inverse definition, a term is **neutral** if it is either a variable or an application. Neutral terms are preserved by reduction. Values and neutral terms are directly substitutive.

Again, anti-renaming does not follow with the equations Autosubst provides:

Lemma 9.6 (Anti-Renaming). *If* $\mathsf{value}\ (s\langle\xi\rangle)$, *then* $\mathsf{value}\ s$.

9.2 Typing, Context Morphism Lemmas, and Preservation

In this section, we introduce typing with a **context** containing the following types:

$$A, B \in \mathsf{ty} \ := \ A \to B \mid \mathsf{Base}$$

Contexts implicitly introduce open terms at negative positions, such that the notion of instantiation on a context gets more complex. This is where context renaming and context morphism lemmas come in play.

We start with contexts and their representation in both pure and scoped syntax. Both definitions implicitly access term variables at negative positions. We continue with substitutivity of typing and its application in the proof of preservation.

Contexts. The representation of a context depends on the choice of scoped versus pure syntax. In pure de Bruijn syntax, a context is represented by a list of types. Hence, an empty context is represented by the empty list nil and we extend a context Γ by a type A via list extension, $A \cdot \Gamma$. Accessing the nth element in a pure context uses the function which accesses the nth element of a list, Γ_n. Note that this operation returns an option type.

In scoped de Bruijn syntax, a typing context is represented by a function with the number of free variables as domain, $\mathbb{I}^m \to \mathsf{ty}$. Hence, an empty context is represented by the function with the empty domain, $! : \mathbb{I}^0 \to \mathsf{ty}$, and we extend a context Γ by a type A via extension, $A \cdot \Gamma$. Accessing the nth element in a scoped context is done via function application, $\Gamma\, n$.

We say that x is a **valid position** in a context Γ if x indeed appears in Γ. In pure code, this is the explicit condition that x is smaller than the length of Γ, $x < |\Gamma|$. In scoped code, this information is implicit in the type of x.

In the following, we use the notation of scoped syntax, but all proofs work similarly for pure syntax apart from the additional scopedness conditions.

We turn to the definition of typing. A variable accesses its type in the context, $\Gamma \vdash \mathsf{var}\, x : \Gamma\, x$. For an abstraction $\lambda_A.s$, we look up the type of s in the context Γ extended with A.

Substitutivity. The definition of contexts as given implicitly has open terms in a negative position. We hence need a definition of instantiation on contexts which makes respects this scope. We hence now introduce two counter parts to ordinary renamings and substitutions: **context renamings** and **context morphisms** for the typing predicate [52, 63]. There is a particular way to do this for de Bruijn substitutions.

In pure syntax, a context Δ is a **reordering** of Γ via the renaming ξ, $\Gamma \leqslant_\xi \Delta$,

$$\forall x.x < |\Gamma| \to \Delta(\xi\, x) = \Gamma\, x.$$

This corresponds to the two conditions:

1. If x is a valid position in Γ, then $\xi\, x$ is a valid position in Δ.
2. For every valid position x in Γ, the element at position x in Γ equals the element at position $\xi\, x$ in Δ.

Note that as in the above definition contexts return an option type, the first condition implicitly holds.

In scoped syntax, we can simplify the above condition to:

$$\forall x.\Delta(\xi\, x) = \Gamma\, x.$$

Typing is stable under the reordering of a context:

Lemma 9.7 (Context Renaming Lemma). *If* $\Gamma \vdash s : A$ *and* $\Gamma \leqslant_\xi \Delta$, *then* $\Delta \vdash s\langle\xi\rangle : A$.

Proof. By induction on $\Gamma \vdash s : A$. The variable case follows directly with the reordering property; typing of an application follows imitating the rule with the inductive hypotheses. Last, for abstraction, we have to show that also $A \cdot \Gamma \leqslant_{0 \cdot (\xi \circ \uparrow)} A \cdot \Delta$, i.e. $(A \cdot \Delta)((0 \cdot \xi \circ \uparrow)\, x) = (A \cdot \Gamma)\, x$, which follows with a case analysis, `asimpl`, and $\Gamma \leqslant_\xi \Delta$. □

A reordering is a special case of a context morphism on substitutions, where $\Gamma \leqslant_\sigma \Delta$ if $\Gamma \vdash x : A$ implies that $\Delta \vdash \sigma\, x : A$.

Lemma 9.8 (Context Morphism Lemma). *If* $\Gamma \vdash s : A$ *and* $\Gamma \leqslant_\sigma \Delta$, *then* $\Delta \vdash s[\sigma] : A$.

$$\frac{}{\mathsf{app}\ (\lambda_n.s)\ \bar{p}_t \succ s[t \cdot_p \mathsf{var}]} \qquad \frac{s \succ s'}{\mathsf{app}\ s\ t \succ \mathsf{app}\ s'\ t}$$

Figure 9.3: Reduction in λ_ν.

Proof. Analogous to Lemma 9.7. In the case of abstraction, we have to show that $\forall x : \mathbb{I}^{1+n}.A \cdot \Delta \vdash (\mathsf{var}\ 0 \cdot \sigma \circ \langle\uparrow\rangle)\ x : (A \cdot \Gamma)\ x$. This requires a case analysis on x, where the case for $x = 0_{\mathbb{I}}$ follows directly with the variable rule, and the case for $x = 1_{\mathbb{I}} +_{\mathbb{I}} x$, requires showing that $A \cdot \Delta \vdash (\sigma\, x)\langle\uparrow\rangle : \Gamma\, i$ and so the context renaming lemma (Lemma 9.7). □

Our automation solves all these goals.

We need substitutivity of typing, when we do an induction on the typing predicate and the context plays a role. See the following standard proof of preservation, which requires the context morphism lemma in the case of β-reduction and the typing of an abstraction:

Lemma 9.9 (Preservation). *If* $\Gamma \vdash s : A$ *and* $s \succ t$, *then* $\Gamma \vdash t : A$.

Proof. By induction on $\Gamma \vdash s : A$, and a subsequent case analysis on $s \succ t$.

We only reduce if s is an abstraction or an application. In the case of abstraction, the claim follows directly with the inductive hypothesis.

For an application $\mathsf{app}\ s_1\ s_2$, there are three possibilities: β-reduction, reduction of s_1, and reduction of s_2. In the last two cases, the claim follows from the inductive hypothesis. For β-reduction, we have to show that $\Gamma \vdash s[t..] : B$ knowing that $\Gamma \vdash t : A$ and $A \cdot \Gamma \vdash s : B$ and thus require the context morphism lemma (Lemma 9.8). □

9.3 Preservation in the Multivariate Lambda Calculus

We mirror the previous proof for preservation in the multivariate λ-calculus, λ_ν (see Section 6.6 for its introduction). For the multivariate λ-calculus, we require both a new definition of reduction (established already in), and of typing.

See Figure 9.1 for a full definition of reduction, $s \succ t$. As before, we write $s \succ^* t$ to denote its reflexive-transitive closure and establish congruence w.r.t. the constructors of λ_ν. We define typing with a new kind of types, accounting for a whole vector of types. See Figure 9.4 for a definition.

The proof follows the same lines as before: We start with substitutivity, continue with context renaming and context morphism lemmas, and finally prove preservation itself.

Fact 9.10 (Substitutivity). *If* $s \succ t$, *then* $s[\sigma] \succ t[\sigma]$.

$$\mathsf{ty} := \bar{\mathsf{p}}_A \to B \mid \mathsf{Base}$$

$$\frac{}{\Gamma \vdash \mathsf{var}\, x : \Gamma\, x} \qquad \frac{\bar{\mathsf{p}}_A \cdot_{\mathsf{p}} \Gamma \vdash s : B}{\Gamma \vdash \lambda_{\mathsf{p}}.s : \bar{\mathsf{p}}_A \to B} \qquad \frac{\Gamma \vdash s : \bar{\mathsf{p}}_A \to B \quad \Gamma \vdash t : \bar{\mathsf{p}}_A}{\Gamma \vdash \mathsf{app}\, s\, t : B}$$

Figure 9.4: Typing for λ_ν.

Proof. By induction on $s \succ t$. The equation which turns up in the case of β-reduction,

$$s[t \cdot_{\mathsf{p}} \mathsf{var}][\sigma] = s[(\mathsf{hd}_{\mathsf{p}} \circ \mathsf{var}_{\mathsf{vl}}) \cdot_{\mathsf{p}} \sigma \circ \langle \uparrow^{\mathsf{p}} \rangle][\mathsf{map}\,[\sigma]\, t \cdot_{\mathsf{p}} \mathsf{var}],$$

can be solved using `asimpl`. □

The notion of a reordering and a context morphism remain unchanged.

Lemma 9.11 (Context Morphism Lemma).

1. *If* $\Gamma \vdash s : A$ *and* $\Gamma \leqslant_\xi \Delta$, *then* $\Delta \vdash s\langle \xi \rangle : A$.
2. *If* $\Gamma \vdash s : A$ *and* $\Gamma \leqslant_\sigma \Delta$, *then* $\Delta \vdash s[\sigma] : A$.

Proof. As before, the case for abstraction is most interesting. We have to solve the following equation:

$$\Gamma \cdot_{\mathsf{p}} \Delta \Uparrow^{\mathsf{tm},\mathsf{p}}_{\mathsf{tm}*} \ \xi\, x = \Gamma \cdot_{\mathsf{p}} \Delta x$$

in the case for renamings and that

$$\Gamma \cdot_{\mathsf{p}} \Delta \vdash (\mathsf{hd}_{\mathsf{p}} \circ \mathsf{var} \cdot_{\mathsf{p}} \sigma \circ \langle \uparrow^{\mathsf{p}} \rangle)\, x : (\Gamma \cdot_{\mathsf{p}} \Delta)\, x$$

for substitutions. The first equation can be proven with `asimpl`, while for the second statement we have to first do a case analysis on x using Lemma 6.18, and reuse the context renaming lemmas in the case that x is of the form $\uparrow^{\mathsf{p}} x'$. All this is entirely analogous to the monadic case. □

Also the statement of preservation remains unchanged:

Lemma 9.12 (Preservation). *If* $\Gamma \vdash s : A$ *and* $s \succ t$, *then also* $\Gamma \vdash t : A$.

Proof. Most interesting, for β-reduction we use the context morphism lemma and then have to show that

$$\Gamma \vdash (\bar{t} \cdot_{\mathsf{p}} \mathsf{var})\, x : (\bar{T} \cdot_{\mathsf{p}} \Gamma)\, x$$

knowing only that $\bar{t} \cdot_{\mathsf{p}} \Gamma \vdash s : A$ and $\forall x. \Gamma \vdash \bar{t}\, x : \bar{T}\, x$. □

In total, the proof requires only 41 proofs of specification and 46 lines of proof code. Except for the changes in the definitions and slight changes according to the p-ary binders, the proof scripts are almost identical to the monadic case.

9.4 Weak Head Normalisation

We now turn to a new proof and show that the (monadic) simply-typed λ-calculus is weakly head normalising, i.e. for every term s there exists a reduction sequence such that $s \succ^* v$ and v is a value. This requires an induction on the typing statement. We follow the proof by Dreyer et al. [39] using logical relations and a distinct value and expression relation.

Logical relations are defined by recursion on the type, and we define $\mathsf{R}\,(A) : \mathsf{ty} \to \mathsf{tm} \to \mathsf{Prop}$ as a function, where we represent a set over terms as a function $\mathsf{tm} \to \mathsf{Prop}$:

$$\mathsf{R}\,(\mathsf{Base}) := \{s \mid \exists v.s \succ^* v \wedge \mathsf{value}\,v\}$$
$$\mathsf{R}\,(A \to B) := \{\lambda_B.s \mid \forall \xi\, v.v \in \mathsf{R}\,(A) \to \exists v'.\mathsf{value}\,v' \wedge s[v \cdot \xi] \succ^* v' \wedge v' \in \mathsf{R}\,(B)\}$$

This definition is monotone.

Lemma 9.13 (Monotonicity). *If* $s \in \mathsf{R}\,(A)$, *then* $s\langle\xi\rangle \in \mathsf{R}\,(A)$.

Proof. By induction on A. We do a case analysis on A and in the case of a function need to know that the instantiation with substitutions and renamings composes, $s\langle \Uparrow^{*\mathsf{tm}}_{\mathsf{tm}} \xi\rangle[v \cdot \zeta \circ \mathsf{var}]$ reduces to $s[v \cdot \xi \circ (\zeta \circ \mathsf{var})]$. □

We define the **expression relation** over a logical relation as:

$$\mathcal{E}\,(A) := \{s \mid \exists v.s \succ^* v \wedge v \in \mathsf{R}\,(A)\}$$

Fact 9.14 (Value Inclusion). *If* $s \in \mathsf{R}\,(A)$, *then* $s \in \mathcal{E}\,(A)$.

Logical relations can be lifted to contexts and substitutions:

$$\mathcal{G}\,(\Gamma) := \{\sigma \mid \forall x.(\sigma\, x) \in \mathsf{R}\,(\Gamma\, x)\}$$

It requires that all variables in the context, substituted with σ, are already in the value relation. **Semantic typing**, $\Gamma \vDash s : A$, then lifts the original definition to the logical relation:

$$\Gamma \vDash s : A := \forall \sigma.\sigma \in \mathcal{G}\,(\Gamma) \to s[\sigma] \in \mathcal{E}\,(A).$$

The fundamental lemma is then proven as follows:

Lemma 9.15 (Fundamental Lemma). *If* $\Gamma \vdash s : A$, *then* $\Gamma \vDash s : A$.

Proof. By induction on $\Gamma \vdash s : A$. The proof requires congruence of reduction in the case of non-neutral terms (Lemma 9.3). In the case of abstraction, we need both value

inclusion (Lemma 9.14) and monotonicity of the context (Lemma 9.13). For abstraction, we encounter the situation where we have to simplify the term

$$s[\mathsf{var}\ 0 \cdot \sigma \circ (\_\langle\uparrow\rangle)][\nu, \xi \circ \mathsf{var}]$$

to $s[\nu \cdot \_\langle\xi\rangle]$ which is done automatically using `asimpl`. □

The last statement of weak normalisation then requires preservation of the identity substitution:

Lemma 9.16 (Weak Normalisation). *If* $\emptyset \vdash s : A$, *then there exists a value* v *such that* $s \succ^* v$.

Proof. If $\emptyset \vdash s : A$, we know that also $\emptyset \vDash s : A$. As $\mathsf{var} \in \mathcal{G}(\emptyset)$, also $s[\mathsf{var}] \in \mathcal{E}(A)$, and $s \in \mathcal{E}(A)$ by the substitution laws, and the claim holds. □

Note that scoped syntax simplifies the notion of an empty context and the statement of semantic typing significantly.

9.5 Schäfer's Expression Relation

We present a proof of strong normalisation of reduction in the simply-typed λ-calculus which uses Schäfer's expression relation. From now on, we write $\mathsf{sn}(s)$ to say that s is strongly normalising w.r.t. full reduction.

The proof follows the structure of weak normalisation and splits the logical relation into a **value** and **expression relation**. Closure then requires an inductive definition. We use this definition because it (1) is the shortest variant of strong normalisation, and (2) allows composable proofs of strong normalisation.

We start with the definition of the logical relation. This definition is different in that it uses a more general definition of **closure**, $\mathcal{E}(B)$, which we explain in the following:

$$\begin{aligned} \mathsf{R}(\mathsf{Base}) &:= \{s \mid \top\} \\ \mathsf{R}(A \to B) &:= \{\lambda_B.s \mid \forall \xi\, v.v \in \mathcal{E}(A) \to v'[v \cdot \xi] \in \mathcal{E}(B)\} \end{aligned}$$

The logical relation is clearly monotone.

Lemma 9.17 (Monotonicity). *If* $s \in \mathsf{R}(A)$, *then* $s\langle\xi\rangle \in \mathsf{R}(A)$.

Proof. Analogous to Lemma 9.13. □

We also need a general definition of closure (Figure 9.5). This closure fulfils several properties:

Lemma 9.18 (Closure Properties).

$$\frac{\mathsf{value}\ s \to s \in \mathsf{R}\,(A) \quad \forall s'.s \succ s' \to s' \in \mathcal{E}\,(A)}{s \in \mathcal{E}\,(A)}$$

Figure 9.5: Closure of a relation.

1. *If* $s \in \mathcal{E}\,(A)$, *then* $\mathsf{sn}\,(s)$.
2. *If* $s \in \mathcal{E}\,(A)$ *and* $s \succ s'$, *then* $s' \in \mathcal{E}\,(A)$.
3. *If* s *is a value and* $s \in \mathcal{E}\,(A)$, *then* $s \in \mathsf{R}\,(A)$.
4. *If the relation is monotone, values are backward-closed, and reduction is backward-closed, so is the closure.*

Proof. The first three statements follow directly. For the fourth statement, we need to know that both values and reduction are stable under anti-renaming (see Lemma 9.6). □

Note that these correspond to the usual closure properties of Girard.

The variable constructor is contained in the closure:

Fact 9.19. $\mathsf{var}\ x \in \mathcal{E}\,(A)$.

We define the context relation and semantic typing analogous to weak normalisation, but instead using Schäfer's expression closure. Again, we thus obtain the fundamental lemma:

Lemma 9.20 (Fundamental Lemma). *If* $\Gamma \vdash s : A$, *then* $\Gamma \vDash s : A$.

Proof. By induction on $\Gamma \vdash s : A$, similar to before. □

As before, the proof of strong normalisation itself is easy, once we have the fundamental lemma:

Lemma 9.21 (Strong Normalisation). *If* $\Gamma \vdash s : A$, *then* $\mathsf{sn}\,(s)$.

Proof. Using the fundamental lemma with Γ and the identity substitution, simplifying the goal using `asimpl`. □

9.6 Raamsdonk's Characterisation

In this section, we show strong normalisation of the simply-typed λ-calculus with sums via an inductive, syntax-dependent variation of strong normalisation à la van Raamsdonk et al. [115] (see Figure 9.6). This split results in elegant, compositional proofs of

Neutral strongly normalising terms $(\mathsf{SN}_\nu\ s)$

$$\frac{}{\mathsf{SN}_\nu\ (\mathsf{var}\ x)} \qquad \frac{\mathsf{SN}_\nu\ s \quad \mathsf{SN}\ t}{\mathsf{SN}_\nu\ (\mathsf{app}\ s\ t)}$$

Strongly normalising terms $(\mathsf{SN}\ s)$

$$\frac{\mathsf{SN}\ s}{\mathsf{SN}\ (\lambda_A.s)} \qquad \frac{\mathsf{SN}_\nu\ s}{\mathsf{SN}\ s} \qquad \frac{s \to_{\mathsf{SN}} s' \qquad \mathsf{SN}\ s'}{\mathsf{SN}\ s}$$

Strong head reduction $(s \to_{\mathsf{SN}} t)$

$$\frac{\mathsf{SN}\ t}{(\lambda_A.s)\ t \to_{\mathsf{SN}} s[t..]} \qquad \frac{s \to_{\mathsf{SN}} s'}{\mathsf{app}\ s\ t \to_{\mathsf{SN}} \mathsf{app}\ s'\ t}$$

Figure 9.6: Raamsdonk's characterisation of strong normalisation. Adapted from [4].

strong normalisation. This approach extends well to more complicated syntactic systems.

The proof was recently proposed as a successor of the POPLMark challenge [3, 4], and the author of this thesis has participated in the second paper where three solutions are suggested (one with de Bruijn syntax and in Coq, submitted by the author). It thus offers a possibility to compare our approach with other solutions for syntax with binders.

The main difference to the previous proofs is the organisation of the proof. Recall that in the previous proofs, we showed that every typable term s is in the logical relation, and every term in the logical relation is strongly normalising. This time, we have the intermediate structure of Raamsdonks's characterisation, $\mathsf{SN}\ s$.

Raamsdonk's characterisation only describes strongly normalising terms. It is a predicate on terms.

The proof then splits up in the following two parts:

1. Raamsdonk's Characterisation indeed describes strongly normalising terms, i.e. if $\mathsf{SN}\ s$, then $\mathsf{sn}\ (s)$ (also known as **soundness**).

2. We show that every term with $\Gamma \vdash s : A$, is contained in Raamsdonk's characterisation, i.e. $\mathsf{SN}\ s$. This part of the proof requires a fundamental lemma similar to before.

Most interesting for us, this separation means that we no longer require the anti-substitutivity and renaming lemma for the full definition of strong normalisation,

but instead an anti-renaming lemma for the custom inductive definition of strong normalisation. In contrast, the first part only requires the usual substitutivity results of reduction.

In the following, we sketch the corresponding proofs. We omit soundness, as this part is largely orthogonal to substitution properties. For a detailed discussion of the proofs and background, see [4].

Strong Normalisation of Raamsdonk's Characterisation

Every typed term $\Gamma \vdash s : A$ is also part of Raamsdonk's characterisation. The proof is via a Kripke-style logical relation, and thus requires various new substitution-related statements on a mutual inductive, but non-negative definition of strong normalisation.

The first statement we need to know is that Raamsdonk's characterisation is substitutive and anti-substitutive under renamings.

Lemma 9.22 (Renaming for Raamsdonk Normalisation).

1. *If* $\mathsf{SN}\ s$, *then* $\mathsf{SN}\ (s\langle\xi\rangle)$.
2. *If* $\mathsf{SN}_\nu\ s$, *then* $\mathsf{SN}_\nu\ (s\langle\xi\rangle)$.
3. *If* $s \to_{\mathsf{SN}} s'$, *then* $s\langle\xi\rangle \to_{\mathsf{SN}} s'\langle\xi\rangle$.

Proof. By mutual induction, similar to for example substitutivity of reduction. □

Lemma 9.23 (Anti-Renaming for Raamsdonk Normalisation).

1. *If* $\mathsf{SN}\ (s\langle\xi\rangle)$, *then* $\mathsf{SN}\ s$.
2. *If* $\mathsf{SN}_\nu\ (s\langle\xi\rangle)$, *then* $\mathsf{SN}_\nu\ s$.
3. *If* $s\langle\xi\rangle \to_{\mathsf{SN}} t$, *then there is* t' *such that* $t = t'\langle\xi\rangle$ *and* $s' \to_{\mathsf{SN}} t$.

Proof. This proof is the main challenge. We show the proof by a mutual induction on the relation. Again we repeatedly need to examine the term and invert equations. In the case of β-reduction for strong head reduction, we solve a similar equation as before. □

Again, this only holds for renamings.

Lemma 9.24. *If* $\mathsf{SN}\ (\mathsf{app}\ s\ (\mathsf{var}\ x))$, *then* $\mathsf{SN}\ s$.

Proof. By induction on $\mathsf{SN}\ (\mathsf{app}\ s\ (\mathsf{var}\ x))$. In the neutral case, the claim follows directly by case analysis. In the case of reduction, we have that $\mathsf{app}\ s\ (\mathsf{var}\ x) \to_{\mathsf{SN}} s'$ and know that $\mathsf{SN}\ s'$. We do a case analysis on the reduction. For β-reduction, $s = \lambda_A.s$, $s' = s[\mathsf{var}..]$ and thus $\mathsf{SN}\ s[\mathsf{var}..]$, and we have to show that $\mathsf{SN}\ (\lambda_A.s)$. This follows with the abstraction rule, the anti-renaming lemma (Lemma 9.23), and the fact that we can transform the above substitution into a renaming via `renamify`. □

We define the logical relation by a recursive function over the type:

$$\begin{aligned} R\,(\mathsf{Base}) &:= \{s \mid \mathsf{SN}\,s\} \\ R\,(A \to B) &:= \{s \mid \forall \xi\, v.v \in R\,(A) \to (\mathsf{app}\,(s\langle\xi\rangle)\,v) \in R\,(B)\} \end{aligned}$$

Lemma 9.25. *The logical relation is monotone.*

Proof. By induction on A, using compositionality of renamings in the function case. □

We require similar conditions as in the original proof of strong normalisation by Girard (Lemma B.5), and similar these lemmas have to be proven simultaneously:

Lemma 9.26 (Properties of the Logical Relation).

CR1 *If* $s \in R\,(A)$, *then* $\mathsf{SN}\,s$.

CR2 *If* $\mathsf{SN}_v\,s$, *then* $s \in R\,(A)$.

CR3 *If* $s \to_{\mathsf{SN}} s'$ *and* $s' \in R\,(A)$, *then* $s \in R\,(A)$.

Proof. Similar to before. The first property requires the anti-renaming lemma, Lemma 9.23, in the function case. The two other cases require the renaming lemma Lemma 9.22. □

Corollary 9.27. $\mathsf{var}\,x \in R\,(A)$.

Proof. Directly with Lemma 9.26. □

Here is the first point that we require the typing predicate.

Theorem 9.28 (Fundamental Lemma). *If* $\Gamma \vdash s : A$ *and* $\sigma \in R\,(\Gamma)$, *then* $s[\sigma] \in R\,(A)$.

Proof. By induction on typing, similar to the fundamental lemma using Schäfer's characterisation, but using the properties of the different logical relation (Lemma 9.26). □

Lemma 9.29 (Strong normalisation of Raamsdonk's characterisation). *If* $\Gamma \vdash s : A$, *then* $\Gamma \vdash \mathsf{SN}\,s$.

Proof. Using the fundamental lemma (Lemma 9.28) and the identity substitution, we get that $s[\mathsf{var}] \in R\,(A)$, i.e. $s \in R\,(A)$ using `asimpl`. With Lemma 9.26, also $\mathsf{SN}\,s$. □

With soundness of Raamsdonk's characterisation, it then follows that also every typed term is strongly normalising according to the accessibility version of strong normalisation.

9.7 Modular Strong Normalisation

We have seen proofs of several standard results for the λ-calculus. We use this section to show how to prove the same results in a modular way, continuing from our goal in Chapter 7. We transfer the previous proofs – preservation, weak head, and strong normalisation using Schäfer's expression relation – to modular proofs. The proof strategy itself remains unchanged.

We first translate the proofs for the λ-calculus to a modular variant. We then extend it by booleans and arithmetic expressions. We use the design principles encountered in Chapter 7.

In general, lifting the definitions is straightforward. It is more surprising that almost all statements can be defined in a modular way. In the following, we highlight the few differences compared to a direct proof.

Stating of the definitions and lemmas. First, most obvious, we have a modular definition of types with modular components for function types $A \to B$, boolean types TBool, and arithmetic types $\mathbb{N}$. We moreover require modular definitions of typing and reduction, see Figure 9.7 for the new definitions.

Not all parts can be defined in a modular manner. We distinguish between *modular definitions* which are proven once for each feature, *parameterised definitions* which are stated before all definitions in a parameterised fashion, and last, *global definitions*, which require global knowledge. See Figure 9.8 for an overview on which definitions and lemmas are proven in which manner.

Most lemmas and specifically those that are by induction on a respective modular predicate can be proven modularly. Parameterised definitions could in principle be proven once for each feature, but this would contain repetitive code. These are definitions which are independent of the particular feature, e.g. the lifting of the logical relation to expressions or contexts. Last, there is one proof for which we were unable to find a modular proof: This proof is the anti-renaming lemma for reduction, which already caused problems during substitution automation. This proof requires full knowledge of reduction which goes beyond the properties of a tight retract.

Substitution boilerplate. Several of the lemmas require the full substitution boilerplate, as we have seen already in the previous proofs. To state all proofs in a modular manner, we have to assume that the parameterised full type satisfies the respective substitution properties. This boilerplate is generated by Autosubst, but as Coq does not allow to reopen a section, we have to copy the around 100 lines. In particular, we also have to parameterise over the retract properties of instantiation with renamings and substitutions.

$$\frac{}{\Gamma \vdash_{\mathsf{var}} \mathsf{var}\, x : \Gamma\, x}$$

$$\frac{\Gamma \vdash s : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash_\lambda \mathsf{app}\, s\, t : B} \qquad \frac{A, \Gamma \vdash s : B}{\Gamma \vdash_\lambda \lambda_A.s : A \to B}$$

$$\frac{}{\Gamma \vdash_+ \mathsf{atom}_< n : \mathbb{N}} \qquad \frac{\Gamma \vdash s : \mathbb{N} \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash_+ s +_< t : \mathbb{N}} \qquad \frac{}{\Gamma \vdash_{\mathbb{B}} \mathsf{const}_{\mathbb{B}<}\, b : \mathsf{TBool}}$$

$$\frac{\Gamma \vdash b : \mathsf{TBool} \qquad \Gamma \vdash e_1 : A \qquad e_2 \vdash A :}{\Gamma \vdash_{\mathbb{B}} \mathsf{if}_<\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : A}$$

$$\frac{\Gamma \vdash_\lambda s : A}{\Gamma \vdash s : A} \qquad \frac{\Gamma \vdash_+ s : A}{\Gamma \vdash s : A} \qquad \frac{\Gamma \vdash_{\mathbb{B}} s : A}{\Gamma \vdash s : A}$$

$$\frac{}{\mathsf{app}_<\, (\lambda_{<A}.s)\, t \succ_\lambda s[t..]_<} \qquad \frac{s \succ s'}{\mathsf{app}_<\, s\, t \succ_\lambda \mathsf{app}_<\, s'\, t}$$

$$\frac{t \succ t'}{\mathsf{app}_<\, s\, t \succ_\lambda \mathsf{app}_<\, s\, t'} \qquad \frac{s \succ s'}{\lambda_{<A}.s \succ_\lambda \lambda_{<A}.s'}$$

$$\frac{s \succ s'}{s +_< t \succ_+ s' +_< t} \qquad \frac{t \succ t'}{s +_< t \succ_+ s +_< t'} \qquad \frac{}{\mathsf{atom}_<\, m +_< \mathsf{atom}_<\, n \succ_+ \mathsf{atom}_<\, (m+n)}$$

$$\frac{}{\mathsf{if}_<\, \mathsf{const}_{\mathbb{B}<}\mathtt{true}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \succ_{\mathbb{B}} e_1} \qquad \frac{}{\mathsf{if}_<\, \mathsf{const}_{\mathbb{B}<}\mathtt{false}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \succ_{\mathbb{B}} e_2}$$

$$\frac{e_1 \succ e_1'}{\mathsf{if}_<\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \succ_{\mathbb{B}} \mathsf{if}_<\, e_1'\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3}$$

$$\frac{e_2 \succ e_2'}{\mathsf{if}_<\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \succ_{\mathbb{B}} \mathsf{if}_<\, e_1\, \mathsf{then}\, e_2'\, \mathsf{else}\, e_3}$$

$$\frac{e_3 \succ e_3'}{\mathsf{if}_<\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \succ_{\mathbb{B}} \mathsf{if}_<\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3'}$$

$$\frac{s \succ_\lambda s'}{s \succ s'} \qquad \frac{s \succ_+ s'}{s \succ s'} \qquad \frac{s \succ_{\mathbb{B}} s'}{s \succ s'}$$

Figure 9.7: Typing and reduction for modular expressions.

What	Modular	Parameterised	Global
Substitution boilerplate	x	-	-
Typing	x	-	-
Reduction	x	-	-
CRL	x	-	-
CML	x	-	-
Preservation	x	-	-
LR for WN	x	-	-
Monotonicity LR	x	-	-
Lifting of LR	-	x	-
Value inclusion	-	x	-
Congruence	x	-	-
Fundamental lemma	x	-	-
WN	-	x	-
LR for SN	x	-	-
Monotonicity LR	x	-	-
Closure properties	-	x	-
Substitutivity reduction	x	-	-
Anti-renaming reduction	-	-	x
Fundamental lemma SN	x	-	-
SN	-	x	-

Figure 9.8: Overview of modular proofs for preservation, weak head normalisation, and strong normalisation.

Assuming the variable component. As the lambda feature contains binders, we further have to assume certain properties of the variable feature. First, we assume how instantiation handles variables. Next, we assume how typing handles variables, i.e. have a parameter:

```
Variable hasty_var : ∀ Γ s A, has_ty_var Γs A → has_ty Γs A.
```

This assumption is required in the context morphism lemma and during preservation. These assumptions can be dropped in the boolean and arithmetic feature.

Differences in the proofs. There are moreover small changes when we want to apply a constructor, rely on the evaluation of functions, or want to do inversion on a full predicate. We hence rely on the following three tactics: The tactic `msimpl` simplifies goals using the injections for functions. These equations have to be registered in a hint database for auto-rewriting after their definition, together with defining equations for retracts. Such equations exist for all functions, i.e. instantiation, values, and the logical relation. The tactic `minversion` is an extension of Coq's `inversion` tactic to modular syntax. It applies registered inversion lemmas, then uses Coq's `inversion` tactic and resolves contradictory cases using the injectivity of inj. These inversion lemmas exist for all inductive predicates, i.e. reduction and typing. Last, `mconstructor` is a combination of `msimpl` and the `constructor` tactic.

Let us show the example of abstraction of the context substitution lemma (Lemma 9.8). Here we have to show

$$\Gamma \vdash \lambda_{iA}.s[\sigma] : A \to_i B$$

To use the constructor tactic, we hence have to know that every feature predicate indeed is part of the whole predicate, i.e. is a tight retract. The `mconstuctor` tactic has this information registered, so using it we first specialise to the feature predicate and then apply Coq's constructor tactic. We can then solve the goal using the inductive hypothesis. To show that the resulting contexts still form a context morphism, we have to use the assumed variable rule for typing in the case of the variable to be 0.

In preservation, `minversion` comes in useful: Recall that we first do an induction on the reduction, and then a case analysis on the typing predicate. For example, in the case of abstraction, we have the assumption that

$$\Gamma \vdash \mathsf{app}_i\,(\lambda_{iA}.s)\,tB :$$

For a proper inversion, we have to know that this rule stems from lambda typing (what we do not up-front). The `minversion` automatically applies the corresponding lemma, which can be proven once we know that feature typing is a tight retract.

MetaCoq support. In the code, we use the MetaCoq support to state lemmas in a modular manner and provided automated definitions and proofs for variants.

Declarative Equivalence $(\Gamma \vdash s \equiv t : A)$

$$\frac{A :: \Gamma \vdash s \equiv s' : B \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{app}\ (\lambda.s)\ t \equiv s'[t'..] : B} \qquad \frac{A :: \Gamma \vdash \mathsf{app}\ (s\langle\uparrow\rangle)\ (\mathsf{var}\ 0) \equiv \mathsf{app}\ (s'\langle\uparrow\rangle)\ (\mathsf{var}\ 0) : B}{\Gamma \vdash s \equiv s' : A \to B}$$

$$\frac{}{\Gamma \vdash \mathsf{var}\ x \equiv \mathsf{var}\ x : \Gamma\, x} \qquad \frac{A :: \Gamma \vdash s \equiv s' : B}{\Gamma \vdash \lambda.s \equiv \lambda.s' : A \to B}$$

$$\frac{\Gamma \vdash s \equiv s' : A \to B \quad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{app}\ s\ t \equiv \mathsf{app}\ s'\ t' : B} \qquad \frac{\Gamma \vdash s \equiv t : A}{\Gamma \vdash t \equiv s : A} \qquad \frac{\Gamma \vdash s \equiv t : A \quad \Gamma \vdash t \equiv u : A}{\Gamma \vdash s \equiv u : A}$$

Figure 9.9: Declarative equivalence.

9.8 Decidability of Beta Eta Equivalence

So far, we have seen typing contexts only. In this final proof for the λ-calculus, we consider binary logical relations. The substitution properties are very similar to before.

In this section we hence show that declarative equivalence for the simply-typed λ-calculus (Figure 9.9) is decidable following a proof by Crary [28] and later mechanised in Beluga by Cave and Pientka [22]. We largely follow the later mechanisation. Asides from being a standard proof for the equational theory of the λ-calculus, the proof requires several properties on substitutions, in particular, concerning the correct handling of renamings.

Declarative equivalence contains β-reduction, η-equivalence, and congruence rules. As a consequence, it is hard to give a decider directly. Instead, we show that this form of equivalence is related to algorithmic equivalence, for which decidability is immediate. We omit this proof in this thesis. The proof requires binary logical relations.

See Figure 9.10 for algorithmic equivalence. The definition is a mutually inductive between algorithmic equivalence for all terms and algorithmic equivalence for neutral terms, i.e. all terms except λ-terms in the case of the λ-calculus. On neutral terms, we follow the structure of the terms. On non-neutral terms, we follow the type, using **weak head reduction** $s \succ_h s'$ in the case of the base type.

Most interesting, we see how to express **η-equivalence** in the simply-typed λ-calculus. In a named representation this rule would state that the following equivalence has to hold for a fresh variable x:

$$\frac{x : A, \Gamma \vdash \mathsf{app}\ s\ (\mathsf{var}\ x) \equiv \mathsf{app}\ s'\ (\mathsf{var}\ x) : B}{\Gamma \vdash s \equiv s' : A \to B}$$

In de Bruijn syntax, we create a specific free variable $\mathsf{var}\ 0$. We ensure freshness by lifting all variables in s and s' via the shifting operation.

Weak head reduction $(s \succ_h t)$

$$\frac{}{\mathsf{app}\,(\lambda.s)\,t \succ_h s[t..]} \qquad \frac{s \succ_h s'}{\mathsf{app}\,s\,t \succ_h \mathsf{app}\,s'\,t}$$

Algorithmic Equivalence $(\Gamma \vdash s \equiv_{\mathsf{alg}} t : A)$

$$\frac{s \succ_h^* s' \quad t \succ_h^* t' \quad \Gamma \vdash s' \equiv_{\mathsf{alg}\downarrow} t' : \mathsf{Base}}{\Gamma \vdash s \equiv_{\mathsf{alg}} t : \mathsf{Base}} \qquad \frac{A \cdot \Gamma \vdash \mathsf{app}\,(s\langle\uparrow\rangle)\,(\mathsf{var}\,0) \equiv_{\mathsf{alg}} \mathsf{app}\,(t\langle\uparrow\rangle)\,(\mathsf{var}\,0) : B}{\Gamma \vdash s \equiv_{\mathsf{alg}} t : A \to B}$$

Neutral Algorithmic Equivalence $(\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A)$

$$\frac{x : A \in \Gamma}{\Gamma \vdash \mathsf{var}\,x \equiv_{\mathsf{alg}\downarrow} \mathsf{var}\,x : A} \qquad \frac{\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} s' : A \to B \quad \Gamma \vdash t \equiv_{\mathsf{alg}} t' : A}{\Gamma \vdash \mathsf{app}\,s\,t \equiv_{\mathsf{alg}\downarrow} \mathsf{app}\,s'\,t' : B}$$

Figure 9.10: Algorithmic equivalence.

See Figure 9.11 for an overview of the three central relations that appear in this proof. To achieve our goal, we use the intermediate notion of a logical relation recursive on the types, $(s,t) \in R_\Gamma(A)$. In general, we can imitate most rules except for an extended context. This is where we need a logical relation.

Weak head reduction. As we use pure syntax, a list of types represents a context. Weak head reduction satisfies similar results as full reduction:

Lemma 9.30 (Properties of Weak Head Reduction).

1. *If* $s \succ_h^* s'$, *then* $\mathsf{app}\,s\,t \succ_h^* \mathsf{app}\,s'\,t$.
2. *If* $s \succ_h s'$, *then* $s[\sigma] \succ_h s'[\sigma]$.
3. *If* $s \succ_h^* s'$, *then* $s[\sigma] \succ_h^* s'[\sigma]$.

Properties of algorithmic equivalence. We prove that algorithmic equivalence satisfies the rules of declarative equivalence, i.e. is backwards-closed under reduction, symmetric and transitive. Moreover, we require that algorithmic equivalence is monotone. These are precisely the properties of declarative equivalence. Although the proofs of symmetry and transitivity are technically interesting, they do not showcase features of the Autosubst tool. We hence mention them only in the appendix (Appendix C).

Similarly to the case of strong normalisation, we have **context renamings** and **context morphisms**, this time for the contexts of equivalence. They are defined analogously to the previous case.

In particular, the following lemma holds:

Fact 9.31.

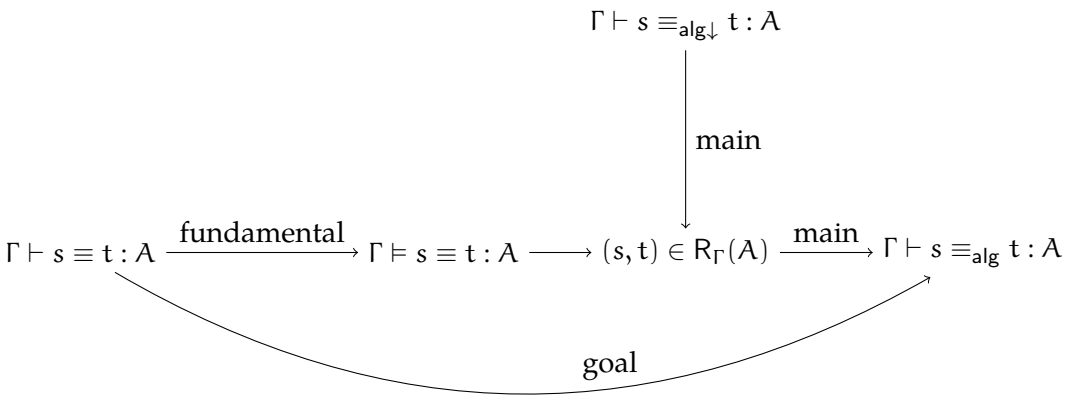

Figure 9.11: Connection between the three equivalences.

1. $\Gamma \leqslant_{\uparrow} A :: \Gamma$
2. *If* $\Gamma \leqslant_{\xi} \Delta$, *then* $A :: \Gamma \leqslant_{\Uparrow^{*tm}_{tm}\xi} A :: \Delta$

We can then show monotonicty.

Lemma 9.32 (Monotonicity).

1. *If* $\Gamma \vdash s \equiv_{alg} t : A$ *and* $\Gamma \leqslant_{\xi} \Delta$, *then* $\Delta \vdash s\langle\xi\rangle \equiv_{alg} t\langle\xi\rangle : A$.
2. *If* $A \vdash s \equiv_{alg\downarrow} t : \Gamma$ *and* $\Gamma \leqslant_{\xi} \Delta$, *then* $\Delta \vdash s\langle\xi\rangle \equiv_{alg\downarrow} t\langle\xi\rangle : A$.

Proof. By a mutual induction on $\Gamma \vdash s \equiv_{alg} t : A$ and $\Gamma \vdash s \equiv_{alg\downarrow} t : A$, using monotonicity of reduction (Lemma 9.30).

In the case of a function, we have to show that $\Delta \vdash s\langle\xi\rangle \equiv_{alg} t\langle\xi\rangle : A \to B$, i.e. $A :: \Delta \vdash \mathsf{app}\uparrow\langle s\langle\xi\rangle\rangle\,\mathsf{var}\,0 \equiv_{alg} \mathsf{app}\uparrow\langle t\langle\xi\rangle\rangle : B$. This follows using the inductive hypothesis with the lifted context (Lemma 9.31). In the assumptions, we have that $A :: \Delta \vdash \mathsf{app}\, s\langle\xi\rangle\langle\Uparrow^{*tm}_{tm} \xi\rangle\,\mathsf{var}\ \Uparrow^{*tm}_{tm} \xi\,0 \equiv_{alg} \mathsf{app}\, t\langle\xi\rangle\langle\Uparrow^{*tm}_{tm} \xi\rangle\,\mathsf{var}\ \Uparrow^{*tm}_{tm} \xi\,0 : B$ and can solve the equation using `asimpl`. □

Lemma 9.33 (Backward Closure). *If* $\Gamma \vdash s \equiv_{alg} t : A$ *and* $s' \succ_h^* s$ *and* $t' \succ_h^* t$, *then* $\Gamma \vdash s' \equiv_{alg} t' : A$.

Proof. By induction on $\Gamma \vdash s \equiv_{alg} t : A$. In the base case, the claim follows directly; in the function case, using the inductive hypothesis, monotonicity of weak head reduction, and congruence of the reduction under application (Lemma 9.30). □

Logical equivalence. We use logical equivalence as an intermediate relation. We define logical equivalence by a recursive function over the type:

Definition 9.34 (Logical Equivalence).

$$\begin{aligned} R_\Gamma(\mathsf{Base}) &:= \{(s,s') \mid \Gamma \vdash s \equiv_{\mathsf{alg}} s' : \mathsf{Base}\} \\ R_\Gamma(A \to B) &:= \{(s,s') \mid \forall t\, t'.(t,t') \in R_\Gamma(A) \\ &\quad \to \forall \xi \Delta.\Gamma \leqslant_\xi \Delta \to (\mathsf{app}\,(s\langle\xi\rangle)\,t, \mathsf{app}\,(s'\langle\xi\rangle)\,t') \in R_\Delta(B)\} \end{aligned}$$

Note that similar to the previous logical relations, we use a **Kripke-style logical relation**, i.e. in the definition of abstraction we allow to extend the context by a renaming ξ.

Similar to algorithmic equivalence, we have to show that logical equivalence is symmetric, transitive, monotone, and closed under backward closure.

Lemma 9.35 (Monotonicity). *If* $(s,t) \in R_\Gamma(A)$ *and* $\Gamma \leqslant_\xi \Delta$, *then* $(s\langle\xi\rangle, t\langle\xi\rangle) \in R_\Delta(A)$.

Proof. By induction on A, using monotonicity of algorithmic equivalence if $A = \mathsf{Base}$ (Lemma 9.32), and `asimpl` in the function case. □

Lemma 9.36 (Backward Closure). *If* $(s,t) \in R_\Gamma(A)$ *and* $s' \succ_h^* s$ *and* $t' \succ_h^* t$, *then* $(s',t') \in R_\Gamma(A)$.

Proof. By induction on A, using backward closure of algorithmic equivalence if $A = \mathsf{Base}$ (Lemma 9.33), using the inductive hypothesis and the congruence and monotonicity properties of weak head reduction in the function case (Lemma 9.30). □

Soundness. We want to show that logically related terms are algorithmically related. This requires to show at the same time that neutral, algorithmically related terms are logically related because of the function case. Note the similarity to strong normalisation on the simply-typed λ-calculus, where we had a similar structure.

Lemma 9.37 (Main Lemma). *We have:*

1. *If* $(s,t) \in R_\Gamma(A)$, *then* $\Gamma \vdash s \equiv_{\mathsf{alg}} t : A$.
2. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$, *then* $(s,t) \in R_\Gamma(A)$.

Proof. We show the two statements simultaneously by induction on A, the cases for $A = \mathsf{Base}$ follow directly.

To show that logical equivalence implies algorithmic equivalence in the function case, and to apply the statement of logical equivalence, we require that $\uparrow$ connects two contexts (Lemma 9.31).

For the other case, using the inductive hypothesis, we require monotonicity of algorithmic equivalence (Lemma 9.32). □

We say that two substitutions σ and τ are logically related in contexts Γ and Δ, written $(\sigma,\tau) \in R_\Gamma(\Delta)$, if for all x in the range of Γ, we have $(\sigma x, \tau x) \in R_\Delta(\Gamma x)$.

Fact 9.38. $(\mathsf{var}, \mathsf{var}) \in \mathsf{R}_\Gamma(\Gamma\, x)$.

Proof. Using the main lemma (Lemma 9.37). □

We say that two terms are **semantically logically equal**, $\Gamma \vDash s \equiv t : A$, if for $(\sigma, \tau) \in \mathsf{R}_\Gamma(\Delta)$, then $(s[\sigma], t[\tau]) \in \mathsf{R}_\Gamma(A)$.

We turn to the fundamental theorem.

Theorem 9.39 (Fundamental Lemma). *If* $\Gamma \vdash s \equiv t : A$, *then* $\Gamma \vDash s \equiv t : A$.

Proof. By induction on declarative equivalence.

In the variable case, the claim directly follows with our environment assumption. The case for symmetry and transitivity follow using symmetry and transitivity of logical equivalence (Lemma C.6, Lemma C.7). The congruence application case follows using the inductive hypothesis and instantiating the assumption for the function type with the identity renaming. We require `asimpl` to resolve instantiation with the identity renaming.

Most interesting are β-reduction, η-expansion, and abstraction. For η-expansion, we use backwards-closure of logical equivalence to also reduce the application on the left side of the equation. The remainder follows using the inductive hypothesis.

For abstraction, we have to show that $\lambda.s$ and $\lambda.s'$ are logically connected under the type $A \to B$, i.e. for $\Gamma \leqslant_\xi \Delta$, and $(t, t') \in \mathsf{R}_\Delta(A)$, then $(\mathsf{app}\,(\lambda.s[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma]\langle \Uparrow^{*\mathsf{tm}}_{\mathsf{tm}} \xi\rangle)\, t, \mathsf{app}\,(\lambda.s'[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau]\langle \Uparrow^{*\mathsf{tm}}_{\mathsf{tm}} \xi\rangle)\, t') \in \mathsf{R}_\Delta(A)$. Using backward closure (Lemma 9.36), we show instead that $(s[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \sigma]\langle \Uparrow^{*\mathsf{tm}}_{\mathsf{tm}} \xi\rangle[t..], s'[\Uparrow^{\mathsf{tm}}_{\mathsf{tm}} \tau]\langle \Uparrow^{*\mathsf{tm}}_{\mathsf{tm}} \xi\rangle[t'..]) \in \mathsf{R}_\Delta(A)$. We use the `asimpl` command to simplify both sides of the logical equivalence such that we can apply the inductive hypothesis. Note that this requires the composition of renamings and substitutions. Showing that the resulting substitutions, $t \cdot (\sigma \circ [\xi \circ \mathsf{var}])$ and $t' \cdot \tau \circ [\xi \circ \mathsf{var}]$, are connected requires monotonicity of the logical relation.

For η-equivalence, we have to show that $(s[\sigma], t[\tau]) \in \mathsf{R}_{\Gamma'}(A \to B)$. We use the inductive hypothesis and again require monotonicity of the logical relation. □

Theorem 9.40 (Completeness). *Declarative equivalence implies algorithmic equivalence.*

Proof. Using the fundamental lemma (Lemma 9.39) and Lemma 9.38, we get that $(s[\mathsf{var}], t[\mathsf{var}]) \in \mathsf{R}_\Gamma(A)$ and, using the main lemma (Lemma 9.37) and identity under identity substitutions that also $\Gamma \vdash s \equiv_{\mathsf{alg}} t : A$. □

9.9 Evaluation

We have given a variety of lean proofs of the λ-calculus and variations, all with parallel substitutions. In all cases, Autosubst generated the respective substitution boilerplate.

	Specification	Proofs	Pure/Scoped
Reduction	40	45	Scoped
Preservation STLC	20	25	Scoped
Preservation Multivariate	40	45	Scoped
Weak Head Normalisation	20	40	Scoped
Strong Normalisation Schäfer	35	80	Scoped
Strong Normalisation Raamsdonk	100	140	Scoped
Modular code	540	665	Pure
Decidability $\beta\eta$-equivalence	90	135	Pure

Figure 9.12: Line numbers.

Our case studies were chosen such that a wealth of substitution properties are required. We moreover used standard proofs, to be able to compare it with other approaches. We omitted results such as confluence for the λ-calculus, as substitutions are mainly orthogonal. Our proofs contained both β-reduction and η-expansion; none caused a problem. Except for the de Bruijn substitutions, all proofs closely follow the paper-based descriptions.

See Figure 9.12 for the lines of code. All developments satisfy the condition of being concise. Most developments use a scoped representation of syntax, an exception is the development for modular syntax, as so far we do not support modular scoped syntax. Decidability of $\beta\eta$-equivalence uses a pure development because of historical reasons.

Let us start with the proofs for the simply-typed λ-calculus. We had two very different proofs of strong normalisation. In the appendix in Appendix B, we provide for reference a third proof following Girard's proof more closely. The proof requires very similar substiution properties as the one using Schäfer's expression relation.

Both a proof of strong normalisation via Raamsdonk's characterisation and the equivalence proof were mechanised in Beluga [85], a proof assistant based on contextual modal type theory [80]. Although Beluga is a special-purpose proof assistant, for the proofs verified, our development was competitive. An approach in Agda based on a universe of syntax is very similar in that it uses an approach based on de Bruijn syntax, but uses a universe of codes and is hence an example of the algebraic approach. Substitution lemmas have still to be applied manually. Our development is comparable in size to other solutions to the POPLMark Reloaded challenge.[2]

Although we scale the development to sums,[3] there are no new substitution properties worth mentioning. The extension is available with the thesis. We found that tactics are a great advantage for mechanised meta-theory. In the case of sums, many lemmas could

[2]This is hard to evaluate because all three solutions were developed in unison and use all different proof assistants.

[3]https://github.com/andreasabel/strong-normalisation/tree/master/coq

be automatically extended and in fewer lines of code than the Agda or Beluga solution, proof assistants which support proof terms only.

The original challenge uses typed syntax, and a typed version of the challenge is available.[4] The scoped variant is almost identical and has no disadvantage in the number of lines. It only differs in the requirement of an additional, extensional typing predicate (Figure 9.2) during the fundamental lemma, to do induction on.

While we found scoped syntax to be useful to avoid mistakes in shifting, we found no similar improvement in using typed syntax. Proofs did not shorten. As typed syntax requires induction-recursion already in a typed representation of System F [42], we see scoped syntax as a particular sweet spot.

Interesting enough, anti-renaming was a problem in all approaches and for example, led to extensions in Beluga. It would be interesting to know whether there is any realisation which requires no anti-renaming lemma at all. There seems to be no way to get around anti-renaming.

Kripke-style logical relations have particularly many renamings. As renamings are never transformed to substitutions, in our proofs we basically only use the `substify` tactic.

The scaling to *variadic syntax* was trivial in the case of type safety. The obvious next step is to explore whether this elegant treatment also transfers to further case studies such as weak head or strong normalisation.

An extension to *modular syntax* was no problem once we developed placeholders for previous tactics. In fact, then the proofs are basically identical if we take care to follow the previously defined design principles. This is the largest proof for modular syntax we know of, and the only one using instantiation. Interesting, the anti-renaming lemma was again a problem. We expect this problem to appear in a possible modular proof via Raamsdonk's characterisation as well.

Autosubst takes care of substitution equations, and substitution equations only. It hence often helps to state definitions and lemmas in an equational way. For example, for β-reduction it is helpful to state the following obvious consequence:

$$s' = s[t..] \to \mathsf{app}\,(\lambda_A.s)\,t \succ s'$$

There is still room of improvement, besides the apparent lacking support for modular scoped syntax and typed syntax: Applying a substitution equation is hard, and sometimes we need conversions. For example, the final result of weak and strong normalisation always require us to explicitly convert a term s to $s[\mathsf{var}]$. If we want to apply the lemma directly, we require Autosubst support for matching.

[4] https://github.com/andreasabel/strong-normalisation/tree/master/coq

We will see that many concepts such as context renaming and context morphism lemmas re-appear in System F.

Chapter 10

System F with Subtyping

In Chapter 9, we have seen how Autosubst handles several variants of the λ-calculus. We covered polyadic binders, variadic binders, functors, and modular syntax; no system required vector substitutions. In this chapter, we consider type safety for System F with subtyping [21, 29], a system with both type and term variables. For the full proof with pattern matching, we assume suitable primitives for pattern typing and pattern matching.

This particular set of problems is also known as the POPLMark challenge [12], a benchmark for evaluating the state of art of mechanised meta-theory, and in particular, binders, complex induction principles, and component reuse. In this chapter, we focus on binders only.

The POPLMark challenge suggests to prove type safety for two systems building up on each other: First System F with subtyping (commonly known as Part 1), then that system enriched with records and pattern matching (Part 2).

For each system, we first show reflexivity, transitivity, and substitutivity of subtyping (Part A). Second, we prove type safety via progress and preservation à la Wright and Felleisen [120] (Part B). The proof outline for preservation is the same as in the λ-calculus and hence again requires context renaming and context morphism lemmas [52, 63]. Preservation requires additional inversion lemmas due to the subtyping rule.

The proof contains many interesting substitution properties not occurring in the previous chapter. One such property is substitutivity of subtyping. In the way subtyping is posed, transitivity and substitutivity have to be shown in a mutual induction. In [100], the authors offer an elegant solution to disentangle this proof for de Bruijn substitutions which we follow and extend to record types. This is an orthogonal problem to the proper handling of substitution equations (although this is a precondition).

Next recall that we have both type variables and term variables. We hence have to handle both a type context and a term type context that depends on the type context. Vector substitutions come in useful for the adaption.

A variety of approaches have handled Part 1 of the POPLMark challenge [1], also in de Bruijn syntax [11, 67, 70, 100, 116]. Almost all solutions in de Bruijn syntax use single-point substitutions, i.e. substitutions as pairs of natural numbers and replaced terms. In contrast, Schäfer et al. [100] use de Bruijn substitutions and context morphism lemmas [52, 63]. With the shortest solution at its publication, a simple equational theory, and only two simplification tactics, it meets the requirements of a concise, transparent, and accessible solution to Part 1 of the POPLMark challenge.

In spite of its prominence, only a few solutions to Part 2 exists. A Twelf solution [9] comes with around 4500 lines of code, and — in contrast to the expectation of the POPLMark authors that de Bruijn syntax does not scale for this proof — all remaining solutions [17, 67, 116] use (single-point) de Bruijn syntax and require to handle substitution lemmas manually. The only solution with tool support is the one by Keuchel et al. [67] using Needle&Knot.

In this chapter, we present concise, transparent, and accessible solutions of the POPLMark challenge: a full solution of Part 1 and a solution of Part 2 that only assumes suitable definitions of pattern typing and pattern matching, but from there on covers all substitution-relevant parts.

In our description, we split the proof into three parts, corresponding to the increasingly complex systems: First, System F without records, $\mathsf{F}_{<:}$ (known as Part 1); second, $\mathsf{F}_{<:}$ enriched with record types, record terms, and projection, $\mathsf{F}_{<:,\mathsf{rec}}$; and finally, $\mathsf{F}_{<:,\mathsf{rec}}$ extended with pattern matching, $\mathsf{F}_{<:,\mathsf{pat}}$ (known as Part 2).

We base our representation on de Bruijn substitutions, as already Schäfer et al. [100] did for Part 1. Different to this previous proof, we use scoped syntax [18] and vector substitutions. We could shorten the proofs by a third.

For the second part, we use Coq's predefined list type to define record types and record terms. Using the predefined list type instead of a custom definition is essential to avoid redundancy in larger developments and in proving further results. Only the solution of Berghofer [17] uses predefined lists as well. Uniqueness properties and new induction principle were hardest to handle, but substitution-wise all mechanised proofs were similar to the paper-based ones. With suitable automation, the parts concerning substitutions did not change at all.

For the third and final part, a let-constructor with pattern matching as stated in the POPLMark challenge needs more elaborate handling of variadic binders. The previous solutions of Vouillon [116] and Keuchel et al. [67] adapt the definition and type of pattern typing such that they circumvent the problem of variadic substitutions. As our goal is the handling of such substitutions, we leave its type unchanged. The definition of pattern typing and pattern matching, as well as its properties, are an orthogonal problem to binders. In this thesis, we assume that suitable predicates exist, are substitutive, and satisfy progress and a type property (see Assumption 10.16); our solution

```
ty, tm : Type

⊤ : ty
arr : ty → ty → ty
all : ty → (ty → ty) → ty

app  : tm → tm → tm
tapp : tm → ty → tm
abs  : ty → (tm → tm) → tm
tabs : ty → (ty → tm) → tm
```

Figure 10.1: EHOAS specification for $\mathsf{F}_<$.

is hence parametric in these assumptions and leaves these definitions for future work. Due to Autosubst's support, the remaining proof of type safety for variadic binders was a non-issue and we can use the context morphism lemma as before.

The first part of the POPLMark challenge was already mechanised in and hence partially reuses the description in [108].

10.1 Type Safety for $\mathsf{F}_<$

We start with System F with subtyping, $\mathsf{F}_<$, specified in Figure 10.1. Autosubst outputs the following many-sorted representation:

$$\begin{aligned} A, B \in \mathsf{ty}^k &:= \mathsf{var}_{\mathsf{ty}}\, x \mid \top \mid A^k \to B^k \mid \forall_{\mathcal{A}^k}.B^{k+1} && x \in \mathbb{I}^k \\ s, t \in \mathsf{tm}^{k;l} &:= \mathsf{var}_{\mathsf{tm}}\, x \mid s^{k;l}\, t^{k;l} \mid s^{k;l}\, A^k \\ &\quad \lambda_{\mathcal{A}^k}.s^{k;l+1} \mid \Lambda_{\mathcal{A}^k}.s^{k+1,l} && x \in \mathbb{I}^l \end{aligned}$$

We say that a term s is a **value**, $\mathsf{value}\, s$, if it is either an abstraction or a type abstraction.

See Figure 10.2 for subtyping, typing, and the weak semantics of $\mathsf{F}_<$.

Most apparent difference to previous systems is the definition of **subtyping**, $\Delta \vdash A <: B$, for types $A, B : \mathsf{ty}^m$ and a **type context** $\Delta : \mathbb{I}^m \to \mathsf{ty}^m$, which contains subtyping information for each occurring type variable. In the typing predicate, we have one rule which allows us to replace a type with a more general one.

Each type A is a subtype of the most general type $\top$ and subtyping propagates through the type constructors as expected. Most interesting are the rules for variables and universal quantification. A variable is a subtype of itself and any more general type of its look-up. Implicitly these rules ensure that subtyping is reflexive and transitive, see the next section. For universal quantification, we extend the context by the respective extended type B_1, and – as all types still exist in the smaller contexts – have to lift the context. As we use scoped syntax, a forgotten shifting would yield a type error.

Subtyping $\Delta \vdash A <: B$

$$\frac{}{\Delta \vdash A <: \top} \qquad \frac{}{\Delta \vdash \mathsf{var}_{\mathsf{ty}}\, x <: \mathsf{var}_{\mathsf{ty}}\, x} \qquad \frac{\Delta \vdash \Delta\, x <: B}{\Delta \vdash \mathsf{var}_{\mathsf{ty}}\, x <: B}$$

$$\frac{\Delta \vdash B_1 <: A_1 \qquad \Delta \vdash A_2 <: B_2}{\Delta \vdash A_1 \to A_2 <: B_1 \to B_2} \qquad \frac{\Delta \vdash B_1 <: A_1 \qquad (B_1, \Delta) \circ \langle\uparrow\rangle \vdash A_2 <: B_2}{\Delta \vdash \forall_{A_1}.A_2 <: \forall_{B_1}.B_2}$$

Typing $\Delta; \Gamma \vdash s : A$

$$\frac{}{\Delta; \Gamma \vdash \mathsf{var}\, x : \Gamma\, x} \qquad \frac{\Delta; \Gamma \vdash s : A \to B \qquad \Delta; \Gamma \vdash t : A}{\Delta; \Gamma \vdash \mathsf{app}\, s\, t : B} \qquad \frac{\Delta; A, \Gamma \vdash s : B}{\Delta; \Gamma \vdash \lambda_A.s : A \to B}$$

$$\frac{\Delta; \Gamma \vdash s : \forall_A.C \qquad \Delta \vdash B <: C}{\Delta; \Gamma \vdash s\, B : A[B \cdot \mathsf{var}_{\mathsf{ty}}]} \qquad \frac{(A \cdot \Delta) \circ \langle\uparrow\rangle; \Gamma \circ \langle\uparrow\rangle \vdash s : B}{\Delta; \Gamma \vdash \Lambda_A.s : \forall_A.B}$$

$$\frac{\Delta \vdash A <: B \qquad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash s : B}$$

Weak Semantics $s \succ t$

$$\frac{\mathsf{value}\, v}{(\lambda_A.s)\, v \succ s[\mathsf{var}_{\mathsf{ty}}; v \cdot \mathsf{var}_{\mathsf{vl}}]} \qquad \frac{}{(\Lambda_A.s)\, B \succ s[B \cdot \mathsf{var}_{\mathsf{ty}}; \mathsf{var}_{\mathsf{vl}}]}$$

$$\frac{s \succ s'}{\mathsf{app}\, s\, t \succ \mathsf{app}\, s'\, t} \qquad \frac{t \succ t' \quad \mathsf{value}\, v}{\mathsf{app}\, v\, t \succ \mathsf{app}\, v\, t'} \qquad \frac{s \succ s'}{s\, A \succ s'\, A}$$

Figure 10.2: Subtyping, typing, and weak semantics of $\mathsf{F}_{<:}$.

We now turn to *typing*, written $\Delta; \Gamma \vdash s : A$. As in the λ-calculus, we have a **term context** $\Gamma : \mathbb{I}^n \to \mathsf{ty}^m$ with typing information for each term variable $\mathsf{var}\, x : \mathsf{tm}^{m,n}$. Newly, we need a type context Δ with again subtyping information for type variables to use the subtyping rule correctly. Note that the types in Γ might contain subtyping information in Δ; so we have to hold the two contexts in accordance with each other.

The typing rules for variables, term abstraction and term application remain unchanged, except for the additional context. Only the already mentioned subtyping rule, type application, and type abstraction have to be added. Type application requires us to substitute B in A – the first time in a typing rule. As a consequence, this rule will require additional reasoning on substitutions. For type abstraction, we bind a new type variable and remember the attached subtyping information in Δ. Additionally, all types in Γ and Δ have to be lifted to be in accordance with the free variables in s.

For *reduction*, we consider a weak semantics, and hence β-reduction presumes that the substituted term is a value and is restricted to left-most reduction. For the reduction of type abstraction, we simply use the type component of substitution.

In the following, we use these definitions first to show several properties of subtyping, then to continue with progress and preservation.

10.1.1 Properties of Subtyping

We start with the properties of subtyping; mainly reflexivity, transitivity, and substitutivity. Proving substitutivity is non-trivial because it has to be proved mutually inductive with transitivity. The proof structure is due to Schäfer et al. [100] and completely analogous for scoped syntax. We still present the proof as (1) we require the same structure for the extended record types, (2) we encounter type context morphism lemmas needed in the second part, and (3) we encounter several proofs with interesting substitution equations and where first-class renamings come in handy.

Reflexivity is straightforward and follows by an induction on the type:

Lemma 10.1 (Reflexivity). $\Delta \vdash A <: A$.

For monotonicity of subtyping, we need the notion of a reordering: Δ' is a **type context reordering** of Δ with the renaming ξ, written $\Delta' \leqslant_\xi \Delta$, if $(\Delta' x)\langle \xi \rangle = \Delta(\xi x)$ for all x.

The proof is straightforward:

Lemma 10.2 (Monotonicity of Subtyping). *If* $\Delta \vdash A <: B$ *and* $\Delta' \leqslant_\xi \Delta$, *then* $\Delta' \vdash A\langle \xi \rangle <: B\langle \xi \rangle$.

Proof. By induction on $\Delta \vdash A <: B$. Most interesting, in the case of universal quantification we have to show that:

$$\forall x : \mathbb{I}^{1+n}.(((B \cdot \Delta) \circ \langle \uparrow \rangle)\, x)\langle \mathsf{var}\, 0 \cdot \xi \circ \uparrow \rangle = (B\langle \xi \rangle \cdot \Delta' \circ \langle \uparrow \rangle)((0 \cdot \xi \circ \uparrow)\, x)$$

which can be solved using a case analysis on x, $\Delta' \leqslant_\xi \Delta$ and `asimpl`. □

For substitutivity, we require context morphisms. Δ' is a **type context morphism** for Δ, $\Delta' \leqslant_\sigma \Delta$, if $\Delta' \vdash \sigma\, x <: (\Delta\, x)[\sigma]$ for all x. As mentioned before, the proof is intertwined with transitivity:

Lemma 10.3 (Transitivity and Substitutivity).

1. *If* $\Delta \vdash A <: B$ *and* $\Delta \vdash B <: C$, *then* $\Delta \vdash A <: C$.
2. *If* $\Delta \vdash A <: B$ *and* $\Delta' \leqslant_\sigma \Delta$, *then* $\Delta' \vdash A[\sigma] <: B[\sigma]$.

Proof. The proof requires several sub-lemmas to disentangle the dependency. See the description of Schäfer et al. [100] and the Coq code for more details.

While transitivity does not contain statements about instantiation, let us have a look at substitutivity. We use reflexivity of subtyping (Lemma 10.1) in the variable case. The look-up rule requires transitivity of subtyping. Most interesting, for universal quantification, we need to show a statement similar to monotonicity:

$$\forall x : \mathbb{I}^{1+n}.(B[\sigma] \cdot \Delta') \circ \langle\uparrow\rangle \vdash (\mathsf{var}\, 0 \cdot \sigma \circ \langle\uparrow\rangle)\, x <: ((B1 \cdot \Delta) \circ \langle\uparrow\rangle\, x)[\mathsf{var}\, 0 \cdot \sigma \circ \langle\uparrow\rangle]$$

This requires among others reflexivity of subtyping (Lemma 10.1) and weakening of subtyping (Lemma 10.2). □

Note that this is the first proof of substitutivity where we needed a more involved structure than before. This is due to the unconventional rules of subtyping for the variable constructor.

10.1.2 Progress

We show type safety for $F_<$. The proof is separated into progress and preservation [120]. Progress requires no substitution properties, but the following inversion lemmas:

Lemma 10.4 (Inversion Lemmas).

1. *If* $\emptyset;\emptyset \vdash s : A \to B$ *and* $\mathsf{value}\, s$, *then there are types* C *and a term* t *such that* $s = \lambda_C.t$.
2. *If* $\emptyset;\emptyset \vdash s : \forall_A.B$ *and* $\mathsf{value}\, s$, *then there are types* C *and a term* t *such that* $s = \Lambda_C.t$.

Proof. By dependent induction on $\emptyset;\emptyset \vdash s : A \to B$ and $\emptyset;\emptyset \vdash s : \forall_A.B$, respectively. □

The proof is then straightforward.

Lemma 10.5 (Progress). *If* $\emptyset;\emptyset \vdash s : A$, *then either* s *is a value or there is a* t *such that* $s \succ t$.

Proof. By a dependent induction on $\emptyset;\emptyset \vdash s : A$. In the case of an application or type application, we use the inductive hypothesis and the inversion lemma (Lemma 10.4). □

10.1.3 Preservation

Like preservation for the simply-typed λ-calculus (Section 9.2), preservation for $F_<$ requires a context morphism lemma. This lemma depends on reorderings and context morphisms for both contexts. Substitutivity requires that all subcomponents of the definition are compatible with substitutions; for example, subtyping, as established in the last section.

Term reorderings depend on a whole vector of substitutions. Γ' is a **term context reordering** of Γ with the renaming vector $\langle \xi; \zeta \rangle$, $\Gamma' \leqslant_{\xi;\zeta} \Gamma$, if $(\Gamma' x)\langle \xi \rangle = \Gamma(\zeta x)$ for all x.

Lemma 10.6 (Term Context Renaming Lemma). *Assume that* $\Delta' \leqslant_\xi \Delta$ *and* $\Gamma' \leqslant_{\xi;\zeta} \Gamma$.

If $\Delta'; \Gamma' \vdash s : A$, *then* $\Delta; \Gamma \vdash s\langle \xi; \zeta \rangle : A\langle \xi \rangle$.

Proof. By induction on $\Delta'; \Gamma' \vdash s : A$, requiring compatibility of subtyping with renaming. In the case of subtyping, we need monotonicity of subtypings (Lemma 10.2).

The proof requires substitution reasoning in three cases: type application, term abstraction, and type abstraction. Let us have a closer look at type abstraction.

We have to show that $\Delta; \Gamma \vdash \Lambda_s.\langle \xi; \zeta \rangle : \forall_A.B$, i.e. $\Delta; \Gamma \vdash \Lambda_{A\langle \xi \rangle}.s\langle \Uparrow^{ty}_{ty} \xi; \Uparrow^{ty}_{vl} \zeta \rangle : \forall_A.B$. With the corresponding typing rule and the inductive hypothesis, it remains to show that both $(A\langle \xi \rangle \cdot \Delta) \circ \langle \uparrow \rangle$ and $\Gamma \circ \langle \uparrow \rangle$ still fulfil the corresponding preconditions, i.e.

$$\forall x.(\Uparrow^{ty}_{ty} \xi)(((A \cdot \Delta') \circ \langle \uparrow \rangle)\, x) = ((A\langle \xi \rangle \cdot \Delta) \circ \langle \uparrow \rangle)((\Uparrow^{ty}_{ty} \xi)\, x)$$

and

$$\forall x.((\Gamma' \circ \langle \uparrow \rangle)\, x)\langle \Uparrow^{ty}_{ty} \xi \rangle = (\Gamma \circ \langle \uparrow \rangle)((\Uparrow^{ty}_{vl} \zeta)\, x).$$

This requires reasoning on the composition of renamings and substitutions, the interaction between extension and composition, and a case analysis on x. For both equations, we use `asimpl` to simplify the goals and then solve the goal with $\Delta' \leqslant_\xi \Delta$ and $\Gamma' \leqslant_{\xi;\zeta} \Gamma$. □

We turn to context morphisms. Γ' is a **term context morphism** for Γ in context Δ', $\Delta'; \Gamma' \leqslant_{\sigma;\tau} \Gamma$ if $\Delta'; \Gamma' \vdash \tau x : (\Gamma x)[\sigma]$ for all x.

Lemma 10.7 (Term Context Morphism Lemma). *Assume that* $\Delta' \leqslant_\sigma \Delta$ *and* $\Delta'; \Gamma' \leqslant_{\sigma;\tau} \Gamma$.

If $\Delta'; \Gamma' \vdash s : A$, *then* $\Delta; \Gamma \vdash s[\sigma; \tau] : A[\sigma]$.

Proof. By induction on $\Delta'; \Gamma' \vdash s : A$. We need substitutivity of subtyping for the subtyping rule (Lemma 10.3). We use equational reasoning similar to the reasoning in the context renaming lemma, using the context renaming lemma in the case of binders.

For abstraction, we have to show that:

$$\forall x : \mathbb{I}^{1+n}.\Delta'; A[\sigma] :: \Gamma' \vdash (\mathsf{var}\, 0 \cdot \tau \circ \langle \mathsf{id}; \uparrow \rangle)\, x : ((A \cdot \Gamma)\, x)[\sigma]$$

which requires a case analysis and the context renaming lemma. Note the similarity to the proof in the simply-typed λ-calculus.

For type abstraction, we have to show that

$$\forall x : \mathbb{I}^{1+n}.A[\sigma] \cdot \Delta' \circ \langle\uparrow\rangle \vdash (\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle)\,x <: ((A \cdot \Delta \circ \langle\uparrow\rangle)\,x)[\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle]$$

This requires weakening of subtyping (Lemma 10.2). Moreover, we prove:

$$\forall x.A[\sigma] \cdot \Delta' \circ \langle\uparrow\rangle; \Gamma' \circ \langle\uparrow\rangle \vdash (\tau \circ \langle\uparrow; \mathsf{id}\rangle)\,x : ((\Gamma \circ \langle\uparrow\rangle)\,x)[\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle]$$

For type application, we need the following equation:

$$B[A'[\sigma] \cdot \sigma] = B[\mathsf{var}\,0 \cdot \sigma \circ \langle\uparrow\rangle][A'[\sigma]..]$$

again proved via `asimpl`. □

Preservation further requires a subtyping morphism lemma. We say that Δ' is a **subcontext** of Δ, $\Delta' <: \Delta$, if and only if $\Delta' \vdash \Delta'\,x <: \Delta\,x$ for all variables x (in scope). In Coq, we define this as:

```
∀ x. sub Δ' (Δ' x) (Δ x).
```

The scoping information is implicit in x.

Lemma 10.8 (Subtyping Morphism). *If* $\Delta' <: \Delta$ *and* $\Delta; \Gamma \vdash s : A$, *then* $\Delta'; \Gamma \vdash s : A$.

Proof. By induction on $\Delta; \Gamma \vdash s : A$ with compatibility of subtyping with renaming and substitution (Lemma 10.3). □

As we work with subtyping, we need the following inversion lemmas for preservation:

Lemma 10.9 (Typing Inversion).

1. *If* $\Delta; \Gamma \vdash \lambda_A.s : C$ *and* $\Delta \vdash C <: A' \to B$, *then* $\Delta \vdash A' <: A$ *and there is* B' *such that* $\Delta; A \cdot \Gamma \vdash s : B'$ *and* $\Delta \vdash B' <: B$.
2. *If* $\Delta; \Gamma \vdash \Lambda_A.s : C$ *and* $\Delta \vdash C <: \forall_{A'}.B$, *then* $\Delta \vdash A' <: A$ *and there is* B' *such that* $A' \cdot \Delta \circ \langle\uparrow\rangle; \Gamma \circ \langle\uparrow\rangle \vdash s : B'$ *and* $A' \cdot \Delta \circ \langle\uparrow\rangle \vdash B' <: B$.

Preservation then follows by induction on $\Delta; \Gamma \vdash s : A$, using the context morphism lemma for type applications and type abstraction.

Theorem 10.10 (Preservation). *If* $\Delta; \Gamma \vdash s : A$, *and* $s \succ t$, *then* $\Delta; \Gamma \vdash t : A$.

```
...
label : Type
ℒ, × : Functor
...
recty : ℒ (label × ty) → ty
...
rectm : ℒ (label × tm) → tm
proj : tm → label → tm
```

Figure 10.3: Adapted EHOAS specification for $\mathsf{F}_{<,\mathsf{rec}}$.

Proof. Analogous to Lemma 9.9, but by induction on $s \succ t$ and then a dependent induction on $\Delta; \Gamma \vdash s : A$ due to subtyping. Otherwise, the proofs resemble previous ones, and we omit the cases for abstraction and application.

Similar to term abstraction, type abstraction requires us to use the context morphism lemma, and hence we have to show that:

$$\forall x : \mathbb{I}^{1+m}.\Delta \vdash B..x <: ((A\langle\uparrow\rangle \cdot \Delta \circ \langle\uparrow\rangle)\, x)[B..]$$

which follows with our automation. □

We hence reached our goal of type safety. Note that we fulfilled the well-formedness condition of the POPLMark challenge intrinsically.

10.2 Type Safety for $\mathsf{F}_{<,\mathsf{rec}}$

In the second step, we extend $\mathsf{F}_{<}$ with record types, record terms, and projections to $\mathsf{F}_{<,\mathsf{rec}}$.

Record types, respective record terms extend both types and terms, using a sort of labels, label:

$$\begin{aligned} A^k, B^k \in \mathsf{ty}_k &:= \cdots \mid \mathsf{recty}\,\{l_i : A_i\} \\ s^{k;l}, t^{k;l} \in \mathsf{tm}_{k;l} &:= \cdots \mid \pi_l\, s \mid \mathsf{rectm}\,\{l_i = s_i\} \qquad l \in \mathsf{label} \end{aligned}$$

See Figure 10.3 for the extension of the EHOAS specification with record types, record labels, and projections. We will frequently write $\mathsf{recty}\,\{l_i : A_i\}$ as simply $\mathsf{recty}\,As$. Together with the new syntactic objects, we have a new category of values: $\mathsf{rectm}\,xs$ is a value if all terms in $\mathsf{rectm}\,xs$ are values.

We implement records with the predefined lists of Coq and hence use the functor mechanism of Autosubst. This contrasts other solutions, which introduce a mutual definition with a new inductive type of custom lists to avoid custom induction principles [67, 116].

$$\frac{}{\text{unique nil}} \qquad \frac{\forall y.(l, y) \notin xs \quad \text{unique } xs}{\text{unique } ((l, x) \cdot xs)}$$

Figure 10.4: Uniqueness.

Subtyping $\Delta \vdash A <: B$

$$\frac{\text{unique Bs} \quad \forall l\, T.(l, T') \in Bs \to \exists T.(l, T') \in As \wedge \Delta \vdash T <: T'}{\Delta \vdash \text{recty As} <: \text{recty Bs}}$$

Typing $\Delta; \Gamma \vdash s : A$

$$\frac{\Delta; \Gamma \vdash s : \text{recty As} \qquad (l, A) \in As}{\Delta; \Gamma \vdash \pi_l\, s : A}$$

$$\frac{xs \equiv_l As \qquad \text{unique } xs \quad \text{unique As} \quad \forall l\, s\, A.(l, s) \in xs \to (l, A) \in As \to \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash \text{rectm } xs : \text{recty As}}$$

Weak Semantics $s \succ t$

$$\frac{t \succ t' \quad (l, t) \in xs}{\text{rectm } xs \succ \text{rectm } xs[l \mapsto t']} \qquad \frac{s \succ s'}{\pi_l\, s \succ \pi_l\, s'} \qquad \frac{(s, l) \in xs}{\pi_l\, (\text{rectm } xs) \succ s}$$

Figure 10.5: Subtyping, typing, and evaluation for $F_{<:,\text{rec}}$.

In the informal syntax, there is the implicit assumption that each label appears only once. Lists alone do not reflect this uniqueness. We hence have a predicate that labels are unique (see Figure 10.4) and require that all terms are well-formed.

We start with extending subtyping, typing, and reduction to $F_{<:,\text{rec}}$, see Figure 10.5 for an overview. *Subtyping* of record types requires two conditions: First, all labels in Bs have to appear in As. Second, for each element (l, T') in Bs, there has to be a pair (l, T) in As with $\Delta \vdash T <: T'$. Labels in Bs must still be unique, a precondition for preservation.

Typing requires two new rules, one for projection and one for record terms. Typing of projections is straightforward and only requires that the respective term has a record type and the label appears in the record type. For typing a record term, both record terms and record types require the same labels.[1] We say that two record objects are **label equivalent**, $xs \equiv_l ys$, it they contain the same labels:

$$xs \equiv_l ys := \forall l.(\exists x.(l, x) \in xs) \leftrightarrow (\exists y.(l, y) \in ys)$$

For each defined label, the typing statement has to hold. Note that we further need

[1] We could weaken this precondition, but this is the way it is posed in the POPLMark challenge.

uniqueness properties for xs and As.

For reduction, we require both a new reduction rule and several congruence rules. For *projection reduction*, the label has to appear in the record term. We have further one new congruence rule for projection (which is trivial) and one for record terms. Reduction in a record term reduces one component and leaves all other components unchanged. We hence require a function which updates a record component, defined with list mapping (and requires decidability of record labels). We write $\mathsf{xs}[\mathsf{l} \mapsto \mathsf{s}]$ to denote the record list xs with the component at label l replaced by s. Updating a component fulfils the following preservation properties:

Lemma 10.11.

1. *If unique* xs, *then unique* $\mathsf{xs}[\mathsf{l} \mapsto \mathsf{s}]$.
2. *If* $\mathsf{xs} \equiv_{\mathsf{l}} \mathsf{ys}$, *then* $\mathsf{xs}[\mathsf{l} \mapsto \mathsf{s}] \equiv_{\mathsf{l}} \mathsf{ys}$.
3. *If* $(\mathsf{l}, \mathsf{s}) \in \mathsf{xs}[\mathsf{l}' \mapsto \mathsf{t}]$, *then either* $(\mathsf{l}, \mathsf{s}) \in \mathsf{xs}$ *or* $\mathsf{l} = \mathsf{l}'$ *and* $\mathsf{s} = \mathsf{t}$.

10.2.1 Transitivity

We show transitivity of the respective subtyping system. There are the following changes:

First, as subtyping of record types requires that the labels are unique, we need a well-formed condition such that reflexivity of subtyping goes through. Second, we require substitutivity of uniqueness and membership. We use the following lemmas:

Lemma 10.12. *If unique* xs, *then unique* $(\mathsf{map}\,(\mathsf{id} \times \mathsf{f})\,\mathsf{xs})$.

Lemma 10.13. *If* $(\mathsf{l}, \mathsf{s}) \in \mathsf{xs}$, *then* $(\mathsf{l}, \mathsf{f}\,\mathsf{s}) \in \mathsf{map}\,(\mathsf{id} \times \mathsf{f})\,\mathsf{xs}$.

Third and last, record types and subsumption need manual, *strengthened induction principles*. For example, for record types, Coq's induction principle requires us to prove that:

$$\forall \mathsf{xs}.\mathsf{P}\,(\mathsf{recty}\,\mathsf{xs})$$

but we actually need that:

$$\forall \mathsf{xs}.(\forall \mathsf{l}\,\mathsf{s}.(\mathsf{l}, \mathsf{s}) \in \mathsf{xs} \to \mathsf{P}\,\mathsf{s}) \to \mathsf{P}\,(\mathsf{recty}\,\mathsf{xs})$$

We hence generate these induction principles manually. While trivial to prove, they require 100 (!) lines of code, most for the statement alone.

Last, we need the characteristic equations of mapping. During weakening of subtyping, we show for example that:

$$\forall \mathsf{l}\,\mathsf{T}'.(\mathsf{l}, \mathsf{T}') \in \mathsf{map}\,(\mathsf{id} \times \langle \xi \rangle)\,\mathsf{ys} \to \exists \mathsf{T}.(\mathsf{l}, \mathsf{T}) \in \mathsf{map}\,(\mathsf{id} \times \langle \xi \rangle)\,\mathsf{xs} \wedge \Delta' \vdash \mathsf{T} <: \mathsf{T}'$$

which can only be resolved using the characteristic lemma of mapping.

10.2.2 Progress

We require a new inversion statement for progress:

Lemma 10.14. *If* $\Delta; \Gamma \vdash s : \mathit{recty}\,\mathsf{As}$ *and* $\mathsf{value}\,s$*, then there is a list* xs *such that* $s = \mathit{rectm}\,\mathsf{xs}$ *and whenever* $(\mathsf{l}, \mathsf{A}) \in \mathsf{As}$*, then there exists* s' *with* $(\mathsf{l}, s') \in \mathsf{xs}$*.*

Proof. Similar to before. The proof requires that $\mathsf{xs} \equiv_{\mathsf{l}} \mathsf{As}$. □

Otherwise, progress gets considerably harder. First, we require a strengthened induction principle for the typing predicate.

For projection, we have to show that $\pi_{\mathsf{l}}\, s$ can indeed do a step. We know that $\emptyset; \emptyset \vdash s : \mathrm{recty}\,\mathsf{As}$, $(\mathsf{l}, \mathsf{A}) \in \mathsf{As}$, and with Lemma 10.14 that $s = \mathrm{rectm}\,\mathsf{xs}$ for some xs. For reduction, we further need that l appears in xs. We hence require that all labels in As also appear in xs (as in the definition for record typing).

For record terms, $\mathrm{rectm}\,\mathsf{xs}$, we have to lift the decision to the whole list of elements: Either all elements in xs are values or there exists a component (l, s) in xs such that $s \succ s'$. In the first case, the whole record term is a value, while in the second case we reduce.

10.2.3 Preservation

Let us turn to the context renaming and context morphism lemma. In both cases, we require monotonicity of uniqueness (Lemma 10.12). For the typing of record terms, we additionally use the characteristic equation of mapping. For projection, we require substitutivity of membership (Lemma 10.13). With Autosubst almost nothing changes, as instantiation is simply propagated.

The following inversion lemma for records comes in useful:

Lemma 10.15. *Assume that* $\Delta; \Gamma \vdash \mathit{rectm}\,\mathsf{xs} : \mathsf{A}$ *and* $\Delta \vdash \mathsf{A} <: \mathit{recty}\,\mathsf{As}'$*. Then, if* $(\mathsf{l}, s) \in \mathsf{xs}$ *and* $(\mathsf{l}, \mathsf{B}') \in \mathsf{As}'$*, there exists* B *such that* $\Delta; \Gamma \vdash s : \mathsf{B}$ *and* $\Delta \vdash \mathsf{B} <: \mathsf{B}'$*.*

Proof. As before, by a dependent induction. We require the uniqueness property. □

For preservation, two new cases appear. Projection simply requires the new inversion lemma (Lemma 10.15).

For records, we have to show that reduction, i.e. updating a record, preserves uniqueness and label equivalence (Lemma 10.11). We then do a case analysis on the respective value: In case it is unchanged by the reduction, typing holds; otherwise, we use the strengthened inductive hypothesis.

```
pat : Sort
...
patvar : ty → pat
patlist : ℒ (label × pat) → pat
...
letpat p: pat → tm → (𝕍 p tm → tm) → tm
```

Figure 10.6: EHOAS specification for pattern matching.

10.3 Type Safety for $\mathsf{F}_{<,\mathsf{pat}}$

Let us reconsider patterns as defined in the POPLMark challenge:

$$p \in \mathsf{pat} \ := \ x : A \mid \{l_i : p_i\}$$

A pattern is either a variable with a type, or a list of patterns with one label each.

Patterns $\mathsf{pat}\ m$, are used during the reduction of a let-expression:

$$\mathsf{let}_n\ p = s\ \mathsf{in}\ t$$

The term t contains a variadic number n of free variables; during reduction of a let-abstraction, s is matched against the pattern p with n free variables and then creates n terms following the labels in a record term, and then replaces the free variables in t with this terms. We call the process of retrieving the terms corresponding to a pattern **pattern matching**.

In Autosubst we define patterns by a new sort, see Figure 10.6 for the extensions to the EHOAS specification. In scoped syntax, `let`-abstraction has the following type:

$$\mathsf{let} : \forall n.\mathsf{pat}\ m_{\mathsf{ty}} \to \mathsf{tm}^{m_{\mathsf{ty}};m_{\mathsf{vl}}} \to \mathsf{tm}^{m_{\mathsf{ty}};p+m_{\mathsf{tm}}} \to \mathsf{tm}^{m_{\mathsf{ty}};m_{\mathsf{tm}}}$$

Note that as-is the number of new binders, n, and the pattern p are disconnected – this connection has to be incorporated into the definition of pattern typing.

Transitivity of subtyping remains unchanged. For type safety, the extension consists of two parts:

1. We require suitable definitions of pattern typing and pattern matching, describing the descend into a record type respectively record term.
2. We use pattern typing and pattern matching in the actual definition of typing and evaluation and prove type safety.

Substitution-wise, only the second part is interesting; the first point is orthogonal to the problem of substitution boilerplate and is more concerned with technical properties on records.

For the remainder of the chapter and this thesis, we assume that suitable predicates exist and satisfy the following properties:

Assumption 10.16. *There exist predicates for pattern typing and pattern matching*

$$\_ : \_ \Rightarrow_n \_ : \mathsf{pat}\ m_{ty} \to \mathsf{ty}^{m_{ty}} \to (\mathbb{I}^n \to \mathsf{ty}^{m_{ty}})$$
$$\_, \_ \succ_n \_ : \mathsf{pat}\ m_{ty} \to \mathsf{tm}^{m_{ty}; m_{tm}} \to (\mathbb{I}^p \to \mathsf{tm}^{m_{ty}; m_{tm}})$$

that fulfil the following three properties:

1. *Pattern typing is substitutive.*
2. *If s is a value, $\emptyset; \emptyset \vdash s : A$, and $p : A \Rightarrow_n \Gamma$, then there exists σ such that $p, s \succ_n \sigma$. We call this the* progress property *of pattern typing.*
3. *If $\Delta; \Gamma \vdash s : A$, $p : A \Rightarrow_n \Gamma'$, and $p, s \succ_n \sigma$, then $\Delta; \Gamma \vdash \sigma\, x : \Gamma'\, x$. We call this the* typedness property *of pattern matching.*

All these conditions are fulfiled by a reasonable implementation, which we leave for future work. See e.g. Berghofer [17] for predicates in pure de Bruijn syntax which satisfy these properties. A solution in scoped syntax might require additional axioms to generate the respective functions; vectors might be a solution.

We can then continue with the formal definition of typing and evaluation for let-abstractions. We define typing as:

$$\frac{\Delta; \Gamma \vdash s : A \qquad p : A \Rightarrow_n \Gamma' \qquad \Delta; \Gamma' \cdot_n \Gamma \vdash t : B}{\Delta; \Gamma \vdash \mathsf{let}_n\ p = s\ \mathsf{in}\ t : B}$$

Note that to show that t has type B, we assume that its free variables have the types as declared in the partial context Γ' generated by pattern typing. We use a variadic extension to connect these two contexts.

Similarly, evaluation of a let-abstraction uses pattern matching to find values for the free variables in t:

$$\frac{p, s \succ_n \sigma}{(\mathsf{let}_n\ p = s\ \mathsf{in}\ t) \succ t[\sigma \cdot_n \mathsf{var}]}$$

We can then continue with the proof of type safety.

Progress

In the case of a let abstraction, we do a case analysis whether s is a value. If s is a value, we show that we can reduce using the progress property of pattern typing.

Preservation

We return to the context renaming lemma. In the case of typing of a let abstraction, we have that

$$\Delta; \Gamma \vdash \mathsf{let}_n\ p\langle\xi\rangle = s\langle\xi;\zeta\rangle\ \mathsf{in}\ t\langle\xi; \mathsf{hd}_n \cdot_n \zeta \circ \uparrow^n\rangle : A\langle\xi\rangle.$$

Using the corresponding typing rule, we then have to show that both $p\langle\xi\rangle : A\langle\xi\rangle \Rightarrow_n \Gamma' \circ \langle\xi\rangle$ and that $\Delta; \Gamma' \circ \langle\xi\rangle \cdot_n \Gamma \vdash t\langle\xi; \mathsf{hd}_n \cdot_n \zeta \circ \uparrow^n\rangle : B\langle\xi\rangle$. With the inductive hypothesis, we hence have to show that:

$$\forall x : \mathbb{I}^{n+m_{ty}}.(\Gamma' \cdot_n \Gamma\, x)\langle\xi\rangle = ((\Gamma' \circ \langle\xi\rangle) \cdot_n \Gamma)(\mathsf{hd}_n \zeta \circ \uparrow^n \cdot_n x)$$

We use `asimpl` to do so. We further require monotonicity of pattern typing.

For the context morphism lemma, we analogously have to show that

$$\Delta; \Gamma' \circ [\sigma] \cdot_n \Gamma \vdash t[\sigma; \mathsf{hd}_n \cdot_n \tau \circ \langle \mathsf{id}; \uparrow^n\rangle] : A[\sigma]$$

and hence that

$$\forall x.\Delta; \Gamma' \circ [\sigma] \cdot_n \Gamma \vdash (\mathsf{hd}_n \cdot_n \tau \circ \langle \mathsf{id}, \uparrow^n\rangle)\, x : (\Gamma' \cdot_n \Gamma\, x)[\sigma]$$

We hence do a case analysis on x using Lemma 6.18, then prove the lemma with the context renaming lemma and `asimpl`.

Note that while these equations might seem daunting, they pose no hindrance in the proof. Autosubst automatically solves all equations, and all the user has to do is using its automation mechanism.

Finally, we turn to preservation.

Theorem 10.17 (Preservation). *If* $\Delta; \Gamma \vdash s : A$, *and* $s \succ t$, *then* $\Delta; \Gamma \vdash t : A$.

Proof. There is only one new rule. For pattern reduction, we show that

$$\Delta; \Gamma \vdash t[\mathsf{var}; \sigma \cdot_n \mathsf{var}] : A$$

knowing that $\Delta; \Gamma' \cdot_n \Gamma \vdash t : B$, that $p : A \Rightarrow_n \Gamma'$, and that $p, s \succ_n \sigma$. We hence use the context morphism lemma. For the preconditions we hence have to show that

$$\forall x.\Delta \vdash \mathsf{var}\, x <: (\Delta\, x)[\mathsf{var}]$$

which follows directly with our rewriting automation and reflexivity of subtyping, and

$$\forall x.\Delta; \Gamma \vdash (\sigma \cdot_n \mathsf{var})x : (\Gamma' \cdot_n \Gamma)[\mathsf{var}_{ty}; \mathsf{var}_{vl}].$$

The second equation requires a case analysis on $x : \mathbb{I}^{n+m_{ty}}$. If x is in the second segment, then the claim directly follows with our simplification and the variable typing rule. Otherwise, after simplification, we have to show that $\Delta; \Gamma \vdash \sigma\, x : \Gamma'\, x$ and hence require the typedness property of pattern typing and pattern matching. □

10.4 Discussion

We compare our mechanisation to other solutions of the POPLMark challenge.

10.4.1 POPLMark Part A

We start with Part A of the POPLMark challenge to which many solutions have been submitted.

Let us start and compare the solution to the one of Autosubst 1. Already Autosubst 1 offers a concise, transparent, and accessible solution to Part A for the POPLMark challenge. De Bruijn substitutions relieve us from many intermediate lemmas for single-point substitutions.

We could simplify the solution of Schäfer et al. [100] even further, from 435 lines to 315 lines. The proofs are shorter for two reasons:

First, a scoped syntax allowed us to remove the separate scopedness judgment required by the POPLMark specification. Scoped syntax enables us to work with contexts in a more concise, functional fashion.

Second, vector substitutions allow us to unify context lemmas for type and term renamings, respectively substitutions. Previous proof efforts of Schäfer et al. [100] could hence be simplified by using merely the context renaming and context morphism lemmas. We see the simple structure of lemmas as a big advantage.

See Figure 10.7 for an overview of the lines of code of different solutions using general-purpose proof assistants, adapted from Schäfer et al. [100] where the authors give a detailed discussion on the different solutions. New is the solution of Keuchel et al. [67] which also offers tool support called Needle&Knot and hence offer a very concise solution as well.

Different to Autosubst, Needle&Knot uses single-point substitutions. In the example of type safety, these seem to offer a similarly simple proof structure as de Bruijn substitutions — if one is firm with the lemmas needed. These lemmas have to be applied manually, which means that users require knowledge on the library used. While the generated substitution lemmas suffice for type safety, this is not so clear if we want to prove more extended results such as normalisation. Unfortunately, the authors do provide proofs of type safety only.

10.4.2 POPLMark Part B

We compare our solution to the ones using de Bruijn syntax of Berghofer [17], Vouillon [116], and Needle&Knot. All three solutions use single-point de Bruijn syntax, i.e. substitution are of the form $\mathbb{N} * \mathsf{tm}$. See Figure 10.8 for a summary of the design decision. Of course, the line numbers given are only of very limited significance, as our solution

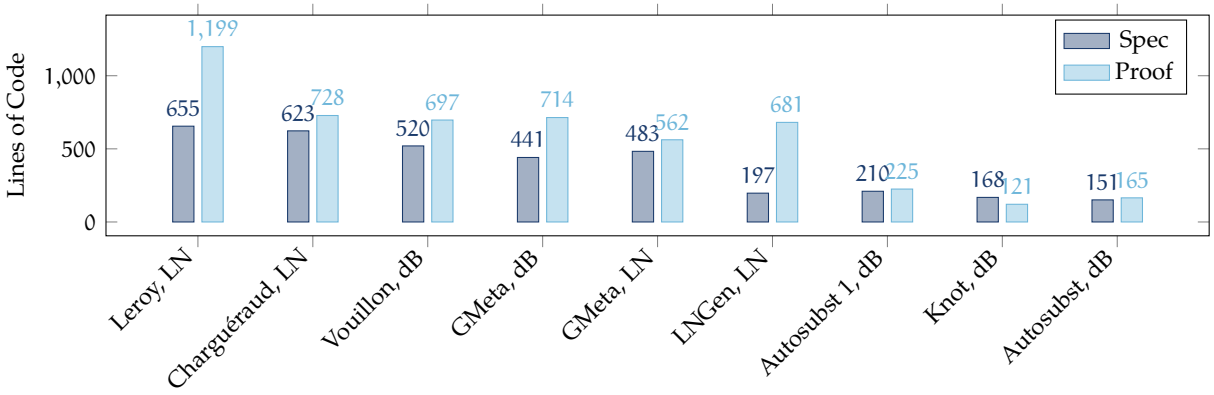

Figure 10.7: Comparison of Coq solutions to part A of the POPLMark challenge, measured using the coqwc utility. Adapted from Schäfer et al. [100]. LN = locally nameless syntax, dB = de Bruijn syntax.

axiomatises pattern typing and pattern matching. Needle&Knot required around 120 lines of code for these proofs.

Let us start with the handling of many-sorted syntax. While Berghofer accumulates type variables and term variables into one sort, Vouillon and Needle&Knot have different sorts for type variables and term variables, just as we have. We found different variables useful to separate our concerns, and they were indeed crucial to state context renaming and context morphism lemma.

We appreciated the easy proof structure of preservation. While single-point de Bruijn syntax requires several lemmas, our proof via context renaming and context morphism lemmas is straightforward. This easy structure remains unchanged for Part 2 of the challenge.

Despite the separation into two kinds of variables, all Berghofer, Vouillon and Needle&Knot define **contexts** as custom lists which contain both type variables and value variables. The main difference is whether we adapt the context when we insert or when we look up variables. During look-up, the whole remaining context is shifted. We found the separation of contexts convenient, as the access of elements was straightforward, and we could reuse previous definitions. The immediate shifting did not cause a problem.

The solutions of Needle&Knot and Vouillon handle record types and record terms by a mutual inductive type of custom lists. An exception is the solution of Berghofer. Although these lead to the correct induction principles in Coq, the approach is otherwise cumbersome – all definitions on lists, such as looking up or mapping, have to be defined separately for each of these types. This would cause redundancy in larger developments. All solutions, including the one of Berghofer, use separate inductive predicates for typing and evaluation on records and lists of records. We are the only solution which

uses membership predicate and predefined list functions. The solution scaled well. Our only annoyance are Coq's induction principles for nested inductive type.

Scopedness is an external property in all solutions. Needle&Knot automatically generate custom scopedness properties. We found working with scoped syntax convenient. Well-formedness adds a considerable amount of lines. As in our case, uniqueness conditions are in the typing judgment. Needle&Knot ignore the uniqueness conditions altogether.

We turn to *reasoning on syntax*. Recall, that all solutions use single-point de Bruijn syntax. However, Berghofer, Vouillon, and Needle&Knot use all different variations of *when* the variable is shifted. This shifting is applied not *during* the lambda, but just later on when talking about the variables. The solution of Needle&Knot seems to be particularly suited for proofs of type safety. We do the (parallel) shifting at once with a renaming. In Vouillon, shifting appears during the abstraction; he claims that the equational system gets more natural if we use this in the case of abstraction. The equational theory still requires to sort the variables, and we know of no completeness result. All solutions except Needle&Knot have to prove manual substitution properties; all approaches, including the one of Needle&Knot, apply substitution lemmas manually.

For patterns, all solutions calculate the size of a pattern to lift. In Vouillon, patterns are defined as an inductive type (with a mutual inductive type for lists of patterns). A generic mapping calculates the size of patterns. Instantiating this one with $\mathbb{N}$, 0, and S would lead to the size of the pattern. All solutions mentioned define pattern typing and pattern matching, although Vouillon and Needle&Knot substitute in-place, and hence bypass the original intention of Part B to compare solutions w.r.t. the handling of non-standard binders. Note that this is no longer possible if we use recursive lets such as in [94] or indeed want to use the result of pattern matching in a direct manner. Only Berghofer generates a list of terms and replaces them at once.

As a consequence, it is unfortunately impossible to compare the solutions w.r.t. their reasoning on variadic syntax. Although Keuchel et al. [67] support data type with variadic binders, we found no development actually reasoning with variadic binders.

Only Needle&Knot offer tool support for substitution boilerplate. In a later paper [68], the authors mention the application to generate substitutivity of typing judgments. However, the tool cannot handle the type system of the POPLMark challenge with the separate transitivity statement.

The obvious next step in our solution is to implement pattern typing and pattern matching in an elegant manner.

	Berghofer	Vouillon	Needle	Autosubst 2
Presentation	single-point	single-point	single-point	de Bruijn
Scoped	No	No	No	Yes
Separated variables	No	Yes	Yes	Yes
Separated context	No	No	No	Yes
Lists with predefined type	Yes	No	No	Yes
Multivariate reduction	Yes	No	No	Yes
LoC without records	1300	1300	290	310
LoC with records	2500	2200	830	520 + patterns
Proof Assistant	Isabelle	Coq	Coq	Coq

Figure 10.8: Comparison of the solutions to POPLMark Part B of Berghofer [17], Vouillon [116], Needle&Knot.

Chapter 11

Other Developments

There are further developments which use and benefit from the current implementation of Autosubst.

Mameche [73] extends the Autosubst output to the Lean proof assistant [35] and mechanises a proof of strong normalisation using Kripke-style logical relations (see Appendix B). Not surprising, the mechanisation is very similar to its counterpart in Coq.

Forster et al. [46] use Autosubst for a de Bruijn representation of parameterised first-order logic, recall its description in Figure 5.8. They use this representation to study variants of completeness for first-order logic. Previous work in [44] uses a locally named representation. A de Bruijn representation simplifies the previously difficult proof of weakening significantly and does not affect the remaining development. Forster et al.'s work inspired the simplified Autosubst output for first-order binders.

Next, Spies and Forster [107] use Autosubst to formalise undecidability of higher-order-unification in a λ-calculus with β-conversion.

Finally, call-by-push-value, short CBPV, is a subsuming paradigm for call-by-value and call-by-name [71]. The main characteristic is its mutual inductive sort of computations and values. Apart from its interest for compilers (see e.g. Rizkallah et al. [94]), CBPV is of interest for mechanisations as many meta-theoretical results (weak and strong normalisation, confluence, a denotational semantics) can be adapted to call-by-value and call-by-name variants. Progress and preservation for call-by-push-value were formalised in Abella by Forster et al. [47]. The authors of Rizkallah et al. [94] provide a sound equational theory for call-by-push-value with recursive let using locally nameless syntax.

In [45], we provide a case study using weak normalisation, strong normalisation, and observational equivalence. In particular, we extended the transport of results to a strong semantics. See Figure 11.1 for the EHOAS specification of our mechanised variant of CBPV. From the syntactic side, we have mutual inductive syntax. This formal system

```
valtype : Sort
comptype : Sort
value : Sort
comp : Sort
𝔹 : Sort

zero: valtype
one: valtype
U: comptype → valtype
Sigma: valtype → valtype → valtype
cross: valtype → valtype → valtype

cone: comptype
F: valtype → comptype
Pi: comptype → comptype → comptype
arrow: valtype → comptype → comptype

u: value
pair: value → value → value
inj: 𝔹 → value → value
thunk: comp → value

cu: comp
force: value → comp
lambda: (value → comp) → comp
app: comp → value → comp
tuple: comp → comp → comp
ret: value → comp
letin: comp → (value → comp) → comp
proj: 𝔹 → comp → comp
caseZ: value → comp
caseS: value → (value → comp) → (value →
comp) → comp
caseP: value → (value → value → comp) →
comp.
```

Figure 11.1: EHOAS specification of CBPV.

Contents	Spec	Proofs
Setup	350	250
Translating CBV and CBN to CBPV	1250	500
Weak Reduction	200	450
Weak Normalisation	100	150
Strong Reduction	800	950
Strong Normalisation	200	300
Observational Equivalence	250	350
Equational Theory	100	200
Denotational Semantics	700	600
Total	3,950	3,750

Figure 11.2: Overview of the mechanised results [45].

would hence have been impossible in Autosubst 1. See Figure 11.2 for an overview of the results and lines of code; the paper [45] contains more technical details.

Substitution-wise, we have to handle both mutual inductive syntax and different syntactic systems. In the first version, renamings were second-class, which lead to several problems and manual statements. First-class renamings hence simplified the development tremendously.

In this development, we mechanised results fundamentally different from the ones in the previous two chapters, e.g. confluence, a denotational semantics, and translations between different formal systems. As Autosubst was able to provide the expected support, we are confident that similar results can be proven for other formal systems as well. Our mechanisation further shows that usage of Autosubst scales to large developments. We think this development showcases how working on a problem in the meta-theory of formal systems should be: The development would have been more work *without* a proof assistant.

In this paper, we worked with several co-authors who used de Bruijn syntax for the first time. As a consequence, scoped syntax proved incredibly valuable to state the correct lemmas.

There are two possible improvements: First, traversals as mentioned in [64] would have been useful to define the above-mentioned translations from call-by-name and call-by-value to CBPV. We had to provide manual proofs that these translations are substitutive — proofs which we could omit with traversals. Further, we required manual proofs of the preservation of injectivity which Autosubst does not provide so far.

Conclusion

Conclusion

In this final section, we first summarise the results of this thesis and then discuss possible extensions and future work.

11.1 Summary of Results

In this thesis, we have examined mechanisations of syntax with both pure and scoped de Bruijn syntax, de Bruijn substitutions, and basic syntactic lemmas inspired by the σ_{SP}-calculus forming a convergent rewriting system. As outlined in the introduction, we approach this topic from three angles: from a theoretic one, justifying our strategy for substitution equations via the convergence of reduction in the σ_{SP}-calculus; from a practical one, generalising existing reasoning principles to broader classes of syntax realised in the Autosubst compiler; and finally, from an empirical one, testing the Autosubst output on several case studies.

11.1.1 Calculi of Explicit Substitutions

We start our exposition with the untyped λ-calculus, and extend the previous result of soundness and completeness of the σ_{SP}-calculus for the de Bruijn algebra [99] with a formalisation and mechanisation of a proof that reduction of the σ_{SP}-calculus is convergent. We hence mechanically verify that the syntactic set of equations Autosubst generates for the λ-calculus is convergent and that rewriting is a suitable proof method for assumption-free substitution equations.

As is standard, convergence is split into local confluence and termination, then connected using Newman's lemma. Both proofs are of fundamentally different difficulty.

Establishing local confluence is surprisingly straightforward. Unlike previous developments, which use a critical-pair analysis, the proof is by a row of nested inversions and uses Coq's depth-first search with backtracking. We consider this one of the advantages of a proof assistant with tactics.

The second and more intricate part is the termination of the σ_{SP}-calculus, a proof that Schäfer et al. [99] have called "far from straightforward". Although three paper-based proofs exist [30, 55, 121], all proofs omit many details. In contrast to Kamareddine and Qiao [65], who formalise termination of the similar σ-calculus in ALF following Curien

et al. [30] very closely, we first simplify the approach of Curien et al. [30] and then mechanise a variant for the σ_{SP}-calculus. Our simplification integrates an additional calculus with fewer reduction rules. As far as we know, our proof is the first mechanised proof of termination for the actual σ_{SP}-calculus, though the differences concerning termination to the σ-calculus are small. We plan to use this proof as a base for future work, as unfortunately the proof in ALF could not be retrieved.

11.1.2 Compiling Syntactic Specifications

We generalise the approach of Autosubst 1 to more extensive syntactic systems. To implement first-class renamings, scoped syntax, many-sorted syntax, modular syntax, and variadic syntax, we require a fundamentally new approach. We have transitioned from Coq's internal tactic language LTac to an external compiler and a new HOAS-like specification language.

We opt for a compiler-based approach, since it is easily extensible, and most importantly, allows us to generate the desired lemmas, proofs, notations, and tactics without compromises. While the first prototype of Autosubst 2 generates textual output with a one-phase compiler, an intermediate phase in the compiler has proven highly beneficial to extend Autosubst to other syntactic output. In hindsight, we are happy how little the compiler changed when we added new features.

In our realisations of formal systems, we follow three design principles: First, we restrict ourselves to a finite set of substitution primitives; second, we use only a single operation of instantiation; and third, we take special care of the choice of syntactic substitution lemmas. We are more strict in following these design principles than Autosubst 1, and in exchange are rewarded with a more elegant equational theory for many-sorted syntax. In all extensions, the similarity to the primitives and equations of the σ_{SP}-calculus is not a coincidence, but the result of careful design.

The fact that these primitives extend in such a regular manner to a variety of formal systems, leads us to decisively contradict the claim that de Bruijn syntax fails to scale to larger systems: in fact, it scales to all examined systems in such a regular manner that we can automatically generate the necessary substitution boilerplate.

A major extension is the one for modular syntax. Although this problem is orthogonal to binders, it is amenable to the same techniques. Our first approach to modular syntax used algebras, but it turned out that a much simpler approach via injections together with supplementary tool support already scales to proofs of strong normalisation.

Although we have shown formal completeness of the σ-calculus for the λ-calculus only, in practice this seems to be no restriction: in our case studies, we were able to solve all appearing equations.

11.1.3 Case Studies

Throughout the thesis, our focus is on a practical development and hence Autosubst has been strongly driven by the practical needs of various case studies. All extensions have been evaluated on these case studies. Where possible, we have also compared against other developments.

First-class renamings appear during proofs with Kripke-style logical relations as in the POPLMark Reloaded challenge [4]. The challenge further requires anti-renaming lemmas in the case of strong normalisation and highlights the need for scoped syntax, which we now think of as a particular sweet spot. Last, functors and variadic syntax first appeared in proofs with records during the POPLMark challenge.

We are able to mechanise a variety of results from the literature. Most importantly, those proofs require little knowledge about substitutions: Although we expect familiarity with de Bruijn syntax and de Bruijn substitutions, there is no need to know how to prove substitution equations. The current version of Autosubst was successfully used by various students. We have observed that in particular scoped syntax simplifies the start.

In contrast to the assumption of the POPLMark authors, de Bruijn syntax, in fact, allows a concise, transparent, and accessible mechanisation of many proofs about the meta-theory of formal systems. In the challenges posed, Autosubst could well hold up against other solutions in general-purpose proof assistants and even special-purpose proof assistants.

We think that the advantages of general-purpose proof assistants – more users and hence more libraries, advanced features such as tactics and type classes, conveniences like custom notations – are invaluable when mechanising the meta-theory of formal systems.

11.2 Open Questions and Challenges

While Autosubst already supports a variety of formal systems, we still see many possibilities for future work.

11.2.1 Calculi of Explicit Substitution

Autosubst generates instantiation and substitution lemmas in a principled manner, but there is no formal justification that these reasoning principles are well-formed for custom syntax. In this point, we fall behind Needle&Knot, who together with their compiler Needle have a Coq formalisation which shows the existence of instantiation and a set of syntactic lemmas.

Even if this has not been a practical restriction in previous developments, we are still interested in examining the resulting equational theory of custom and more extensive

syntax in detail. Even if we as developers are careful, it is easy to forget syntactic lemmas necessary for a complete equational theory.

The first step is a scoped calculus of explicit substitutions. As in the scoped λ-calculus, its primitives would be annotated with a scope; soundness w.r.t. the scoped λ-calculus is then trivial. Most interesting, we require a counterpart of expansion to be complete: this type-driven rule unifies two substitutions with the empty domain. First experiments have shown that local confluence can be recovered with the right closure and that termination can be recovered using the previous results on the pure σ_{SP}-calculus. Most interesting will be completeness, for which we hope to get a simplified proof: with termination, it could be possible to omit Schäfer et al.'s [99] intermediate notion of normal forms; scopedness helps during completeness of instantiation.

Generalising our results to custom calculi requires the definition of custom calculi of explicit substitutions which could be realised via a signature similar to Knot. There are then four desirable properties: the soundness for the corresponding de Bruijn algebra, completeness for the corresponding de Bruijn algebra, local confluence, and termination.

We expect soundness, local confluence, and termination to hold in all cases. The proof of local confluence should still be straightforward, possibly in a similarly simple manner as before. For termination, the hardest part was the termination of the distribution calculus, and probably most calculi can be reduced to a similar intermediate calculus. In the existing proof of termination, it could be that we can simplify the last proof step – termination of the distribution calculus – using the techniques of Zantema [121].

We expect completeness to be the main challenge, and unprovable in many cases. The main problem is that finding a normal form might be type-driven, for example, when examining a syntactic system with one constant only.

Finally, we are interested in going beyond the rewriting system of the σ_{SP}-calculus. While Autosubst helps with assumption-free equations, it also unfolds the lifting operation and hence clutters proof statements if they are not solved immediately. It might be of interest to investigate whether a calculus with an explicit lifting operation such as the one by Hardin and Lévy [56] is suitable and can be adapted to a complete and convergent rewrite system for de Bruijn algebras. The current variant is not complete, as discussed in Chapter 4.

Next, in the case studies, we repeatedly encountered the case that we wanted to apply a lemma but had to manipulate a term explicitly, e.g. changing a term to itself instantiated with the identity substitution. It might be interesting to see whether results about matching in calculi of explicit substitutions could help us in this case.

It would be further interesting to examine whether other approaches to binders (e.g. single-point de Bruijn syntax such as [67] or locally nameless syntax) offer similar completeness result as de Bruijn substitutions, i.e. if we canonically order the indices of

substitutions.

11.2.2 Compiling Syntactic Specifications

Aside from extending meta-theoretic results of calculi of explicit substitutions, the current implementation of Autosubst allows many more extensions.

One could extend Autosubst either to other proof assistants; the generation of the de Bruijn algebra and the syntactic rewriting lemmas has already been implemented in Lean [73]. The main challenge is to transport rewriting to a language without tactics, e.g. Agda.

On the automation side, a variant without functional extensionality is of interest. While Autosubst already generates lemma without functional extensionality, automation has to be adapted. Further investigations are necessary for a faster version of automation, e.g. via reflection.

Another solution to functional extensionality would be to implement scoped syntax via vectors and hence omit the problem of functional extensionality altogether; see e.g. the intrinsically typed realisations in [4]. In fact, in the compiler this would probably only require to change the printing of the substitution primitives and should hence be relatively straightforward. Note that probably certain equations no longer hold definitionally; we encountered (and solved) a similar problem in the extension to variadic syntax.

So far, the transformation of substitutions to renamings is rather rudimentary. While our separation into first-class renamings never needed this direction, we are still interested in making these steps more elegant. Rewriting up to associativity [20] could simplify the definition.

Another promising area is to extend the expressiveness of Autosubst. Obvious extensions are intrinsically typed syntax, dependent predicates, (substitutive) functions, and dependent types. Intrinsically typed syntax would require a different definition of substitutions via an inductive definition, very similar to the one need for a scoped representation by vectors.

An extension of Needle&Knot already generates substitution boilerplate for a subset of well-formed inductive predicates [68]. Inductive predicates require merely the definition of instantiation (which corresponds to the context renaming and context morphism lemma), as we are in the universe of propositions and might be proof-irrelevant anyways. Stating these via relations might be helpful. The definition of dependent inductive types is probably much harder as it requires definitions to be invertible to prove the substitution lemmas.

We are moreover interested in implementing traversals, as first sketched in [64]. Further investigation is necessary to decide the best trade-offs in an implementation.

Finally, when proving properties *within* a formal system with binders (as opposed to establishing global properties), it is more convenient to work with actual names. Inside a system, it seems reasonable to provide automatic convertibility functions between a named and unnamed representation. As our base would be unnamed syntax, we could hopefully omit statements about α-equivalence. We do not think that names in a mechanisation of formal systems are worth sacrificing the simplicity of reasoning on syntax.

11.2.3 Modular Syntax

In our exposition, we allow modular syntax for pure syntax and simple second-order syntax only. The general question is how far our approach to modular syntax scales.

So far, we have neither implemented the POPLMark challenge nor the POPLMark Reloaded challenge with modular syntax. The main reason is that, so far, we support only pure de Bruijn syntax, but the extension to scoped syntax should need no conceptual changes. We are highly interested in providing the first truly modular proofs to both challenges.

Next, an obvious extension is the automatic generation of modular inductive predicates. As we already present manual variants, automation should be straightforward (and far easier than supporting the respective substitution boilerplate). Feature interaction, i.e. how to include features in other features, is the most pressing question. Also, the definition of custom automatic induction principles as in Section 7.2 probably requires more care. We expect previous case studies to be easily adaptable.

Further research is needed to allow the modular extension of types with different substitution vectors; for example from a variant with the arithmetic feature alone to a variant with both abstraction and a feature for abstraction, or from the lambda calculus to System F. This step immediately introduces the challenge of different types, previously tackled by [37].

Finally, if our approach scales as expected, it would be useful to provide a modular library with standard results (e.g. context renaming and context morphism lemmas, preservation, type safety, weak and strong normalisation, semantics) for common formal systems.

Appendix A

Abstract Reduction Systems

The semantics of programming languages, as well as the rewriting rules of the σ_{SP}-calculus, are defined via a small-step semantics and thus reuse many of the notions of abstract reduction systems as by Baader and Nipkow [13], Huet [59]. The Coq mechanisation is based on lecture notes and reuses parts of a development by Smolka [103].

We call a binary predicate of type $R : X \to X \to \mathsf{Prop}$ a **relation** over type X. We say that x and y are in the union of a relation, short $(R \cup R')\,x\,y$, if either $R\,x\,y$ or $R'\,x\,y$.

A relation R' over Y **preserves** a relation R over X, if there is a **morphism** $f : X \to Y$, i.e. $\forall x\,y.R\,x\,y \to R'\,(f\,x)\,(f\,y)$.

We define the **reflexive-transitive** and **transitive** closure of a relation as an inductive predicate (Figure A.1). Both closures are transitive, and the transitive closure is contained in the reflexive-transitive closure:

Fact A.1. *Let R be a relation over a type X.*

1. *R^+ and R^* are transitive.*
2. *If $R^+\,x\,y$, then $R^*\,x\,y$.*

Consider a relation R over X. A term $x : X$ is **normal** w.r.t. R, if there is no y such that $R\,x\,y$. It is **weakly normalising** w.r.t. R, if there is some normal y with $R^*\,x\,y$.

Reflexive-Transitive Closure $(R^*\,x\,y)$

$$\frac{}{R^*\,x\,x} \qquad \frac{R\,x\,y \qquad R^*\,y\,z}{R^*\,x\,z}$$

Transitive Closure $(R^+\,x\,y)$

$$\frac{R\,x\,y \qquad R^*\,y\,z}{R^+\,x\,z}$$

Figure A.1: Reflexive-transitive and transitive closure.

A term $x : X$ is said to be **strongly normalising** w.r.t. a relation R over X, $\mathsf{sn}_R(x)$, if it satisfies the following inductive predicate:

$$\frac{\forall y.R\,x\,y \to \mathsf{sn}_R(y)}{\mathsf{sn}_R(x)}$$

We simplify the notation to $\mathsf{sn}(x)$ if the relation R is clear from the context. A relation R is **terminating**, if $\mathsf{sn}_R(x)$ for all x.

The following facts about strong normalisation appear throughout the thesis:

Fact A.2. *Let X and Y be types, R be a relation over X and R' a relation over Y.*

1. [*Forward Propagation.*] *If $\mathsf{sn}_R(x)$ and $R^*\,x\,y$, then also $\mathsf{sn}_R(y)$.*
2. [*Morphism Lemma.*] *Assume that R' simulates R via a morphism f. Then $\mathsf{sn}_{R'}(f\,x)$ implies $\mathsf{sn}_R(x)$.*

The morphism lemma is useful to prove propagation of strong normalisation in many cases, e.g.

Corollary A.3. *If $\mathsf{sn}_{R\cup R'}(s)$, then $\mathsf{sn}_R(s)$ and $\mathsf{sn}_{R'}(s)$.*

Confluence is another important notion. We say that two objects x and y are **joinable** by a relation R whenever the following holds:

$$\exists z.R\,x\,z \wedge R\,y\,z$$

A relation R over X is **confluent**, if for $R^*\,x\,y$ and $R^*\,x\,y'$, y and y' are joinable by R^*, i.e. there is a unifying z such that $R^*\,y\,z$ and $R^*\,y'\,z$. A relation R over X is **locally confluent**, if $R\,x\,y$ and $R\,x\,y'$ implies that y and y' are joinable, i.e. there is some z such that $R^*\,y\,z$ and $R^*\,y'\,z$.

The following lemma [81] is a standard lemma and allows to simplify the proof obligation to local confluence if the rewriting system is terminating:

Lemma A.4 (Newman's Lemma). *A terminating and locally confluent relation is also confluent.*

A relation R is **convergent** if it is terminating and confluent.

Appendix B

Strong Normalisation à la Girard

This chapter contains the third proof of strong normalisation for full reduction in the simply-typed λ-calculus, see Chapter 9. The proof is very similar to the original proof by Girard et al. [51] but is extended with Kripke-style logical relations.

We require the following properties of strong normalisation:

Lemma B.1 (Properties of sn).

1. *If* $\mathsf{sn}\,(\mathsf{app}\ s\ t)$, *then* $\mathsf{sn}\,(s)$ *and* $\mathsf{sn}\,(t)$.
2. *If* $\mathsf{sn}\,(\lambda_A.s)$, *then* $\mathsf{sn}\,(s)$.
3. *If* $\mathsf{sn}\,(s[\sigma])$, *then* $\mathsf{sn}\,(s)$.

Proof. (1) and (2) follow directly with the homomorphism lemma. (3) additionally requires that reduction is substitutive (Lemma 9.1). □

Note that in the definition of strong normalisation w.r.t. full reduction, open terms only appear at negative position. Hence, the last statement, which is an anti-substitution lemma, is actually straightforward to prove. On the other hand, substitutivity no longer holds and showing the statement for renamings is hard:

Lemma B.2. *If* $\mathsf{sn}\,(s)$, *then* $\mathsf{sn}\,(s\langle\xi\rangle)$.

Proof. By induction on $\mathsf{sn}\,(s)$, using anti-renaming of reduction (Lemma 9.5). □

We define the logical relation w.r.t. a context by a recursive function over the type:

$$\begin{aligned} R_\Gamma\,(\mathsf{Base}) &:= \{s \mid \Gamma \vdash s : \mathsf{Base} \wedge \mathsf{sn}\,(s)\} \\ R_\Gamma\,(A \to B) &:= \{s \mid \Gamma \vdash s : A \to B \wedge \forall \xi\, v.v \in R_\Gamma\,(A) \to (\mathsf{app}\,(s\langle\xi\rangle)\,v) \in R_\Gamma\,(B)\} \end{aligned}$$

Clearly, if $s \in R_\Gamma\,(A)$, then $\Gamma \vdash s : A$.

Again, the logical relation has to be substitutive.

Lemma B.3 (Monotonicity). *If* $s \in \mathsf{R}\,(A)$, *then* $s\langle\xi\rangle \in \mathsf{R}\,(A)$.

Proof. By induction on A. To show that the typing statement is preserved in both cases, we require the context renaming lemma for typing (Lemma 9.7). Otherwise, for the base case, we require anti-monotonicity of strong normalisation (Lemma B.2). For the function case, we have to show that renamings compose to mirror the definition. □

Lemma B.4 (Preservation of the Logical Relation). *If* $s \succ t$ *and* $s \in \mathsf{R}\,(A)$, *then* $t \in \mathsf{R}\,(A)$. *Moreover, if* $s \succ^* t$ *and* $s \in \mathsf{R}\,(A)$, *then* $t \in \mathsf{R}\,(A)$.

Proof. By induction on A. We require preservation (Section 9.2) and that reduction is substitutive for the function case. □

Lemma B.5 (Closure of the Logical Relation).

1. *If* $s \in \mathsf{R}_\Gamma\,(A)$, *then* $\mathsf{sn}\,(s)$.
2. *If* $\Gamma \vdash s : A$ *and* s *is neutral, and* s *is forward closed, i.e. if* $s \succ t$, *then* $t \in \mathsf{R}_\Gamma\,(A)$, *then* $s \in \mathsf{R}_\Gamma\,(A)$.

Proof. The two statements are shown simultaneously by induction on A. In itself we have no interesting substitution equations to show, but require a number of the previously proven statements: Anti-monotonicity of reduction (Lemma 9.5), preservation (Lemma 9.9), the context morphism lemma (Lemma 9.8), and substitutivity of strong normalisation (Lemma B.1). □

Corollary B.6. $\mathsf{var}\,x \in \mathsf{R}_\Gamma\,(\Gamma\,x)$.

Proof. Directly, using Lemma B.5. □

Lemma B.7 (Closure under Beta Expansion). *If* $\mathsf{sn}\,(t)$ *and* $A \cdot \Gamma \vdash s : B$, *and* $s[t..] \in \mathsf{R}_\Gamma\,(B)$, *then* $\mathsf{app}\,(\lambda_A.s)\,t \in \mathsf{R}_\Gamma\,(B)$.

Proof. By a nested induction on $\mathsf{sn}\,(M)$ (using Lemma B.1) and $\mathsf{sn}\,(N)$. We then apply Lemma B.5 and have to show the three properties. Typing and neutrality directly follow. Forward closure is harder and follows by a case analysis on $\mathsf{app}\,(\lambda A s)\,t \succ s'$. The reduction case follows directly with our assumption; otherwise, we have to show that we can apply the respective inductive hypothesis and all assumptions are preserved, using the context morphism lemmas, preservation of the logical relation, naturality of reduction, and substitutivity. □

We lift syntactic typing to semantic typing analogous to weak head normalisation. The only difference is that we do not split the definition into a value and an expression relation:

$$\mathcal{G}\,(\Gamma) := \{\sigma \mid \forall x.(\sigma\,x) \in \mathsf{R}_\Delta\,(\Gamma\,x)\}$$
$$\Gamma \vDash s : A := \forall \Delta\,\sigma.\sigma \in \mathcal{G}_\Delta\,(\Gamma) \to s[\sigma] \in \mathsf{R}_\Delta\,(A)$$

Theorem B.8 (Fundamental Lemma). *If* $\Gamma \vdash s : A$, *then* $\Gamma \vDash s : A$.

Proof. By induction on $\Gamma \vdash s : A$. From the substitution view, everything is standard, but we require context morphism lemmas and the closure under β-expansion. □

Corollary B.9. *If* $\Gamma \vdash s : A$, *then* $\mathsf{sn}\,(s)$.

Proof. Using the fundamental lemma with Γ and the identity substitution, simplifying the goal using `asimpl`. □

Appendix C

Symmetry and Transitivity of Algorithmic and Logical Equivalence

This section outlines mechanised proofs of symmetry and transitivity of algorithmic and logical equivalence, defined and required in Section 9.8.

Weak head reduction is easily shown to be confluent.

Lemma C.1 (Confluence). *If* $s \succ_h^* t$ *and* $s \succ_h^* t'$, *then there exists* u *such that* $t \succ_h^* u$ *and* $t' \succ_h^* u$.

Proof. By a nested induction on $s \succ_h^* t$ and case analysis on $s \succ_h^* t'$. □

Lemma C.2 (Symmetry).

1. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}} t : A$, *then* $\Gamma \vdash t \equiv_{\mathsf{alg}} s : A$.
2. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$, *then* $\Gamma \vdash t \equiv_{\mathsf{alg}\downarrow} s : A$.

Proof. Directly by mutual induction. □

Showing transitivity is harder, as not all definitions transfer for algorithmic equivalence. For example, in the case of the base type, we have no guarantee that we go back to the same type, or for application, the type A is unknown in the conclusion. We thus require the following two intermediate lemmas.

Lemma C.3. *If* $s \succ_h t$ *and* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} u : A$, *then* $s = t$.

Proof. By induction on $\Gamma \vdash u \equiv_{\mathsf{alg}\downarrow} : s$. □

Lemma C.4. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$ *and* $\Gamma \vdash t \equiv_{\mathsf{alg}\downarrow} u : B$, *then* $A = B$.

Proof. By induction on $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$, case analysis on $\Gamma \vdash t \equiv_{\mathsf{alg}\downarrow} u : B$. □

Lemma C.5 (Transitivity).

1. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}} t : A$ *and* $\Gamma \vdash t \equiv_{\mathsf{alg}} u : A$, *then* $\Gamma \vdash s \equiv_{\mathsf{alg}} u : A$.
2. *If* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$ *and* $\Gamma \vdash t \equiv_{\mathsf{alg}\downarrow} u : A$, *then* $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} u : A$.

Proof. By a mutual induction on $\Gamma \vdash s \equiv_{\mathsf{alg}} t : A$ and $\Gamma \vdash s \equiv_{\mathsf{alg}\downarrow} t : A$. The claim follows directly in the function and the variable case by case analysis on the second assumption.

If $\Gamma \vdash s \equiv_{\mathsf{alg}} t : \mathsf{Base}$, then there are s' and t' such that $s \succ_h^* s'$, $t \succ_h^* t'$ and $\Gamma \vdash s' \equiv_{\mathsf{alg}\downarrow} t' : \mathsf{Base}$. At the same time, for $\Gamma \vdash t \equiv_{\mathsf{alg}} u : \mathsf{Base}$, there are t'' and u' with $t \succ_h^* t''$ and $u \succ_h^* u'$ and $\Gamma \vdash t'' \equiv_{\mathsf{alg}\downarrow} u' : \mathsf{Base}$. To find fitting neutral terms to combine s and u, we first use confluence (Lemma C.1) to note that there is a term t''' such that $t' \succ_h^* t'''$ and $t'' \succ_h^* t'''$. Using neutral confluence (Lemma C.3), moreover $t''' = t''$. With symmetry (Lemma C.2), also $\Gamma \vdash t \equiv_{\mathsf{alg}} s : \mathsf{Base}$, and so also $t''' = t'$. With algorithmic backward closure, it moreover suffices to show that two successors of s and u are algorithmically equivalent.

In the application case, the type of function arguments is not necessarily equal. We use Lemma C.4, to prove that they are indeed equal, and the claim follows. □

Lemma C.6 (Symmetry). *If* $(s, t) \in \mathsf{R}_\Gamma(A)$, *then* $(t, s) \in \mathsf{R}_\Gamma(A)$.

Proof. By induction on A, using symmetry of algebraic equivalence for the case that $A = \mathsf{Base}$ (Lemma C.2). □

Lemma C.7 (Transitivity). *If* $(s, t) \in \mathsf{R}_\Gamma(A)$ *and* $(t, u) \in \mathsf{R}_\Gamma(A)$, *then* $(s, u) \in \mathsf{R}_\Gamma(A)$.

Proof. By induction A, using transitivity of algebraic equivalence for the case that $A = \mathsf{Base}$ (Lemma C.5). We require symmetry of logical equivalence (Lemma C.6) to solve the function case. □

Bibliography

[1] Website of the POPLMark challenge. https://www.seas.upenn.edu/~plclub/poplmark/, Accessed: 2019-10-30.

[2] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. *Journal of functional programming*, 1(4):375–416, 1991.

[3] Andreas Abel, Alberto Momigliano, and Brigitte Pientka. POPLMark Reloaded. In *Proceedings of the Logical Frameworks and Meta-Languages: Theory and Practice Workshop*, 2017.

[4] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark Reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29:e19, 2019.

[5] Robin Adams. Formalised metatheory with terms represented by an indexed family of types. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs*, TYPES'04, pages 1–16, 2004. ISBN 3-540-31428-8, 978-3-540-31428-8. URL http://dx.doi.org/10.1007/11617990_1.

[6] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 195–207. ACM, 2017. ISBN 978-1-4503-4705-1.

[7] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages*, 2(ICFP): 90, 2018.

[8] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalised inductive types. In *International Workshop on Computer Science Logic*, pages 453–468. Springer, 1999.

[9] Michael Ashley-Rollman, Karl Crary, and Robert Harper. Solution to the POPLMark challenge in Twelf. https://www.seas.upenn.edu/~plclub/poplmark/cmu.html, Accessed: 2019-09-10.

[10] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, Stephanie Weirich, and Stephanie Weirich. Engineering formal metatheory. In *Acm sigplan notices*, volume 43, pages 3–15. ACM, 2008.

[11] Brian E. Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical report, University of Pennsylvania, 2010.

[12] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. In *TPHOLs*, volume 3603, pages 50–65. Springer, 2005.

[13] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[14] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

[15] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.

[16] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in Coq. *Journal of automated reasoning*, 49(2):141–159, 2012.

[17] Stefan Berghofer. A solution to the POPLMark challenge using de Bruijn indices in Isabelle/HOL. *Journal of Automated Reasoning*, 49(3):303–326, 2012.

[18] Richard S Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of functional programming*, 9(1):77–91, 1999.

[19] Olivier Boite. Proof reuse with extended inductive types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2004.

[20] Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *International Conference on Certified Programs and Proofs*, pages 167–182. Springer, 2011.

[21] Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1-2):4–56, 1994.

[22] Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice, LFMTP 2015, Berlin, Germany, 1 August 2015*, pages 33–45, 2015. URL https://doi.org/10.4204/EPTCS.185.3.

[23] James Cheney. Scrap your nameplate:(functional pearl). *ACM SIGPLAN Notices*, 40(9):180–191, 2005.

[24] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of computer Programming*, 9(2):137–159, 1987.

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

[26] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[27] Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

[28] Karl Crary. Logical relations and a case study in equivalence checking. *Advanced Topics in Types and Programming Languages*, pages 223–244, 2005.

[29] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in fsub. *Mathematical structures in computer science*, 2 (1):55–91, 1992.

[30] Pierre-Louis Curien, Thérèse Hardin, and Alejandro Ríos. Strong normalization of substitutions. In *International Symposium on Mathematical Foundations of Computer Science*, pages 209–217. Springer, 1992.

[31] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)*, 43(2):362–397, 1996.

[32] Haskell B Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

[33] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972. ISSN 1385-7258.

[34] Washington de Carvalho Segundo, Flávio L. C. de Moura, and Daniel Ventura. Formalizing a named explicit substitutions calculus in coq. In Matthew England, James H. Davenport, Andrea Kohlhase, Michael Kohlhase, Paul Libbrecht, Walther Neuper, Pedro Quaresma, Alan P. Sexton, Petr Sojka, Josef Urban, and Stephen M. Watt, editors, *Joint Proceedings of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM co-located with Conferences on Intelligent Computer Mathematics (CICM 2014), Coimbra, Portugal, July 7-11, 2014*, volume 1186 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014. URL http://ceur-ws.org/Vol-1186/paper-19.pdf.

[35] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[36] David Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000.

[37] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *ACM SIGPLAN Notices*, volume 48, pages 207–218. ACM, 2013.

[38] 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.

[39] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems – lecture notes. 2018. URL https://plv.mpi-sws.org/semantics/2017/lecturenotes.pdf.

[40] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical frameworks*, 2:6, 1991.

[41] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 1–15. Springer, 1994.

[42] Fredrik Nordvall Forsberg and Anton Setzer. A finite axiomatisation of inductive-inductive definitions. *Logic, Construction, Computation*, 3:259–287, 2012.

[43] Yannick Forster and Kathrin Stark. Coq à la carte - a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, USA, January 20–21, 2020*, January 2020. To appear.

[44] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, New York, NY, USA, January 2019. ACM.

[45] 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, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 118–131, 2019.

[46] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory. In *Symposium on Logical Foundations Of Computer Science (LFCS 2020), January 4-7, 2020, Deerfield Beach, Florida, U.S.A.*, January 2020. To appear.

[47] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Journal of Functional Programming Special Issue: Post-Proceedings of ICFP 2017*, to appear.

[48] Andrew Gacek. The Abella interactive theorem prover (system description). In *International Joint Conference on Automated Reasoning*, pages 154–161. Springer, 2008.

[49] Lorenzo Gheri and Andrei Popescu. A formalised general theory of syntax with bindings. In *International Conference on Interactive Theorem Proving*, pages 241–261. Springer, 2017.

[50] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[51] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.

[52] Healfdene Goguen and James McKinna. Candidates for substitution. *LFCS report series-Laboratory for Foundations of Computer Science ECS LFCS*, 1997.

[53] Andrew D Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HOL Users' Group Workshop*, pages 413–425. Springer, 1993.

[54] Thérèse Hardin. Eta-conversion for the languages of explicit substitutions. In *International Conference on Algebraic and Logic Programming*, pages 306–321. Springer, 1992.

[55] Thérese Hardin and Alain Laville. Proof of termination of the rewriting system SUBST on CCL. *Theoretical Computer Science*, 46:305–312, 1986.

[56] Thérese Hardin and Jean-Jacques Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, volume 106, 1989.

[57] Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997. ISBN 978-3-540-76121-1.

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

[59] Gérard Huet. Conflunt reductions: Abstract properties and applications to term rewriting systems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 30–45. IEEE, 1977.

[60] Gérard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 23(1):11–21, 1981.

[61] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher Order Logics*, pages 152–167. Springer, 2004.

[62] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '17, pages 10–14. ACM, 2017.

[63] Jonas Kaiser, Tobias Tebbi, and Gert Smolka. Equivalence of System F and λ2 in Coq based on context morphism lemmas. In *Proceedings of CPP 2017*. ACM, 2017.

[64] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de Bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 293–306, 2018.

[65] Fairouz Kamareddine and Haiyan Qiao. Formalizing strong normalization proofs of explicit substitution calculi in ALF. *Journal of Automated Reasoning*, 30(1):59–98, 2003.

[66] Steven Keuchel and Tom Schrijvers. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2013.

[67] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & Knot: Binder boilerplate tied up. In *European Symposium on Programming Languages and Systems*, pages 419–445. Springer, 2016.

[68] Steven Keuchel, Tom Schrijvers, and Stephanie Weirich. Needle & Knot: Boilerplate bound tighter. Technical report, 2017.

[69] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[70] Gyesik Lee, Bruno C.D.S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. GMeta: A generic formal metatheory framework for first-order representations. In *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 436–455. Springer Berlin Heidelberg, 2012.

[71] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *International Conference on Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.

[72] Lena Magnusson. The new implementation of ALF. In *WORKSHOP ON*, page 249, 1992.

[73] Sarah Mameche. Strong normalisation of the lambda calculus in Lean. *Bachelor thesis*, 2019.

[74] Conor McBride and James McKinna. Functional pearl: I am not a number– I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004.

[75] Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–334. Springer, 1995.

[76] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[77] John C Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991.

[78] Masayuki Mizuno and Eijiro Sumii. Formal verification of the correspondence between call-by-need and call-by-name. In *International Symposium on Functional and Logic Programming*, pages 1–16. Springer, 2018.

[79] Anne Mulhern. Proof weaving. In *Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanising Metatheory*, 2006.

[80] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.

[81] Maxwell Herman Alexander Newman. On theories with a combinatorial definition of equivalence. *Annals of mathematics*, pages 223–243, 1942.

[82] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[83] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22–24, 1988*, pages 199–208. ACM, 1988.

[84] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, pages 202–206. Springer, 1999.

[85] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *International Joint Conference on Automated Reasoning*, pages 15–21. Springer, 2010.

[86] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[87] Andrew M Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, 186(2):165–193, 2003.

[88] Olivier Pons. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs*, pages 217–232. Springer, 2000.

[89] François Pottier. Revisiting the CPS transformation and its implementation. Unpublished, July 2017. URL http://gallium.inria.fr/~fpottier/publis/fpottier-cps.pdf.

[90] Garrel Pottinger. A tour of the multivariate lambda calculus. In *Truth or Consequences*, pages 209–229. Springer, 1990.

[91] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[92] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019.

[93] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[94] Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. A formal equational theory for call-by-push-value. In *International Conference on Interactive Theorem Proving*, pages 523–541. Springer, 2018.

[95] Andreas Rossberg, Claudio V Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 89–102. ACM, 2010.

[96] Amokrane Saïbi. Formalisation of a λ-calculus with explicit substitutions in Coq. In *International Workshop on Types for Proofs and Programs*, pages 183–202. Springer, 1994.

[97] Steven Schäfer. *Engineering Formal Systems in Constructive Type Theory*. PhD thesis, Saarland University, 2019.

[98] Steven Schäfer and Kathrin Stark. Embedding higher-order abstract syntax in type theory. In *Abstract for Types Workshop*, June 18 – 21 2018.

[99] Steven Schäfer, Gert Smolka, and Tobias Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15–17, 2015*, pages 67–73, Berlin, Heidelberg, 2015. Springer-Verlag. URL http://doi.acm.org/10.1145/2676724.2693163.

[100] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving — 6th International Conference, ITP 2015,*

Nanjing, China, August 24–27, 2015, Proceedings, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015. ISBN 978-3-319-22101-4. URL https://doi.org/10.1007/978-3-319-22102-1_24.

[101] Christopher Schwaab and Jeremy G Siek. Modular type-safety proofs in Agda. In *Proceedings of the 7th workshop on Programming languages meets program verification*, pages 3–12. ACM, 2013.

[102] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *ACM SIGPLAN Notices*, 42(9):1–12, 2007.

[103] Gert Smolka. Confluence and normalization in reduction systems. Lecture Notes https://courses.ps.uni-saarland.de/icl_17/dl/94/Retracts.pdf, Accessed: 2019-11-26.

[104] Gert Smolka. Retracts. Lecture Notes https://courses.ps.uni-saarland.de/icl_17/dl/94/Retracts.pdf, Accessed: 2019-11-26.

[105] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[106] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. 2019.

[107] Simon Spies and Yannick Forster. Undecidability of higher-order unification formalised in Coq. Technical report, January 2020. to appear.

[108] 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, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180, 2019.

[109] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4): 423–436, 2008.

[110] William W Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(2):198–212, 1967.

[111] The Coq Development Team. The Coq proof assistant, version 8.10.0, October 2019. URL https://doi.org/10.5281/zenodo.3476303.

[112] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *PACMPL*, 3(ICFP):105:1–105:28, 2019. URL https://doi.org/10.1145/3341709.

[113] Amin Timany and Matthieu Sozeau. Cumulative inductive types in Coq. *LIPIcs: Leibniz International Proceedings in Informatics*, 2018.

[114] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runst. *Proceedings of the ACM on Programming Languages*, 2(POPL):64, 2017.

[115] Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999. URL https://doi.org/10.1006/inco.1998.2750.

[116] Jérôme Vouillon. A solution to the POPLMark challenge based on de Bruijn indices. *Journal of Automated Reasoning*, 49(3):327–362, 2012. ISSN 0168-7433.

[117] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[118] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *PACMPL*, 2(ICFP):87:1–87:30, 2018. URL https://doi.org/10.1145/3236782.

[119] Stephanie Weirich, Brent A Yorgey, and Tim Sheard. Binders unbound. In *ACM SIGPLAN Notices*, volume 46, pages 333–345. ACM, 2011.

[120] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[121] Hans Zantema. Termination of term rewriting by interpretation. In *International Workshop on Conditional Term Rewriting Systems*, pages 155–167. Springer, 1992.

Index