SAARLAND UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR'STHESIS

STRONG NORMALIZATION OF THE λ-CALCULUS IN LEAN

Author
Sarah Mameche

Advisor
Kathrin Stark

Supervisor
Prof. Dr. Gert Smolka

Reviewers
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Submitted: 14th January 2019

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, 14th January, 2019

Abstract

We study syntactic theories with variable binders in the Lean Theorem Prover. In a general-purpose interactive prover such as Lean, formalizing binders and substitution is technical and highly repetitive.

Autosubst is an existing automation tool that reduces boilerplate around binders in the Coq proof assistant. It relies on the use of parallel de Bruijn substitutions and their equational theory, the σ-calculus. Autosubst derives the substitution operations of an extension of the σ-calculus for custom language specifications in second-order abstract syntax. It implements a decision procedure for equations with substitution applications.

Our goal is to adapt Autosubst to Lean to simplify normalization proofs in Lean. We implement the key features of Autosubst in Lean: the ability to derive generalized substitution lemmas as well as automation tactics for equational reasoning. In the process, we take a closer look at Lean's metaprogramming capabilities and we study how its extensions can be used to optimize the decision procedure in terms of proof term size and efficiency. As an application of the Autosubst adaptation, we formalize proofs of weak and strong normalization of the simply typed λ-calculus in Lean.

Acknowledgements

I dearly thank Kathrin Stark for investing so much of her time in advising me. Sessions with her were always motivating and I really appreciated her help, input and comprehensive feedback. Moreover, I would like to thank my supervisor Prof. Smolka for offering me a thesis at his chair even though I was very new to the field of Computational Logic at the time. I thank Prof. Smolka and Prof. Hermanns for rewiewing this thesis, and also Yannick Forster for providing the thesis template that is used. Special thanks to my family for all their love and support.

Contents

Chapter 1

Introduction

Formalizations of languages and logical systems are interesting for many purposes. During the design of a programming language, proving that certain correctness specifications are met is important to ensure reliable behavior and avoid flawed definitions. As one aspect, we might want to make statements about termination. For logical systems, correctness properties include decidability or expressiveness.

Both programming languages and logical calculi often have binding structures such as functions or let-expressions. Consequently, formalizing metatheory about them requires the treatment of *variable binders*.

Consider for example the λ-*calculus*, a term language with binders and application. Using binders, we can represent local functions such as $\lambda x.\lambda y.x$. Note that occurrences of bound variables are potentially substituted during function application, e.g. $(\lambda x.\lambda y.x)\ (\lambda z.z)$ reduces to $\lambda y.\lambda z.z$. Reduction behavior as in the example can be modeled with a reduction relation on terms. If we are interested in formalizing termination properties of a language, a reduction relation with the right semantics can be analyzed. The proofs of weak and strong normalization for the λ-calculus reviewed in this thesis work in this way.

Working out all the details of such proofs by hand is lengthy and error-prone. Thus, it is convenient to use a proof assistant for proof management. In addition, we can benefit from the system's type checker to ensure fully verified results. The framework we will be using is the *Lean Theorem Prover* [11]. Lean is a recently developed interactive prover with an axiomatic foundation based on constructive type theory and a meta language for tactic programming.

Aiming at proofs of weak and strong normalization in Lean, we first need adequate ways to handle binders, reduction and substitution. As it turns out, binders produce a lot of technical overhead in proof environments without built-in support for them like Lean. For instance, we need to treat terms that are equal under α-equivalence the same and make precise what happens if variables are substituted, taking care

that no free variables are captured. Such details can be often abstracted from in a paper proof, but we have to be precise about them in a machine-verified formalization.

There has been research towards automating the work associated with binders in general-purpose proof assistants, with the goal of mechanizing repetitive details. A benchmark for automation tools in this field is the POPLMark challenge [5]. It proposes a set of challenge problems that require reasoning with binders. Normalization proofs for the simply-typed λ-calculus also appear among the problems of a recent extension, POPLMarkReloaded [2].

Among the tools that provide solutions to the POPLMark challenge is Coq's Autosubst [25, 26]. Autosubst is designed to reduce the boilerplate in proofs with binders, shifting the focus of proofs to the actual properties that are proven.

We are interested in examining how the approach of Autosubst adapts to other proof assistants. Conveniently, Autosubst is implemented as an external tool that generates Coq code and can be extended to print code for other provers, mainly Lean. Thus, it is a good candidate for a tool for binder support in Lean.

1.1 Contribution

In this thesis, we will describe an implementation of Autosubst for Lean along with case studies of weak and strong normalization of the λ-calculus. As of yet, Lean has no support for variable binders and to our knowledge, there are no Lean-formalizations of weak or strong normalization of the λ-calculus in Lean.

Because Lean is designed to make tactic programming easier, the focus will be on how Lean's automation can be used in Autosubst. In particular, we target the shortcoming that Autosubst relies on inefficient rewriting and produces long proof terms, and look at how Lean's extensions can be used to approach these problems.

We will conclude with normalization proofs, considering both weak and strong normalization of the simply typed λ-calculus. The proofs have been implemented in Coq and Lean using Autosubst. All proofs can be found online.

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

1.2 Related Work

The Lean Theorem Prover Lean [11] is a new theorem prover under development at Microsoft Research and Carnegie Mellon University. It is an open source project started in 2013, and the current version Lean 3, as well as the reference manual and documentation, are available open-source at

`https://leanprover.github.io/`

Lean provides an interactive environment to write axiomatic proofs. It has a standard library containing definitions of common mathematical objects, and another library specific to Homotopy Type Theory. Its type theory is a version of the Calculus of Inductive Constructions [7, 8] with support for classical reasoning. The prover has a type-checker and a tactic interface. A new version Lean 4 is currently under development.

Metaprogramming in Lean A design goal of Lean is to also allow the user to write more flexible and hands-on automation. For this purpose, it offers another layer besides the pure object language, the so-called *meta language*. Meta definitions can access or construct expressions of the object language or inspect and modify the proof goal and context. They are mostly used to implement tactics, i.e., small pieces of automation. Details on the metaprogramming approach of Lean can be found in [13]. The paper also presents examples of larger pieces of automation and includes an evaluation of Lean's automation against other tactic languages.

The Coq Proof Assistant Besides Lean, we will use the Coq Proof Assistant [27]. Coq is based on the Calculus of (Co)Inductive Constructions [7, 8]. Tactic languages that can be used for Coq are `Ltac` or `Mtac`.

De Bruijn representation There are several common approaches to represent variables formally, such as a locally nameless [6], using nominal sets[22], or higher-order abstract syntax [20]. The design choice of Autosubst is to use a nameless representation of variables due to *de Bruijn* [10], where variables are just numerical indices that point to a binder.

The σ-calculus The theoretical basis of Autosubst is the σ-*calculus*, an equational theory proposed by Abadi et al [1]. The σ−calculus models variable substitution with explicit substitution functions from natural numbers to terms. This is in accordance with the de Bruijn view of variables as natural numbers. Along with the substitution functions, a set of substitution operations are added to the language. There is a set of rewriting rules for the calculus which have later shown to be confluent [9], and complete [24]. As a consequence, each expression containing substitution primitives can be associated with a unique normal form, thus equality in the calculus is decidable.

Autosubst The above theoretical results are used by Coq's Autosubst. It provides tactics to simplify substitution expressions according to the rewriting rules and to decide equality. The first version [25] is implemented in `Ltac` using type classes. Autosubst can be used for the synthesis of the substitution operations and rewriting lemmas, and as a decision procedure on equalities in later proofs. As input language, Autosubst 1 accepts inductive term types in Coq with annotations of the binders.

Autosubst 2 The most recent version, Autosubst 2 [26], extends the $\sigma-$calculus to handle syntax with different variable sorts more flexibly, for example languages with both term and type variables. It additionally supports *well-scoped* syntax [16]. In brief, in a well-scoped setting the maximal variable indices are bounded by the number of bound variables in the context. Substitutions become functions from finite, indexed types to terms, instead of functions from the natural numbers. The theory of the extended, well-scoped $\sigma-$calculus will be looked at in more detail in Chapter 2. In contrast to the first version, Autosubst 2 generates the relevant definitions and lemmas with an external tool written in Haskell. As input, it takes a second-order HOAS specification [20] of the target language.

Weak and Strong Normalization Proofs There are several proofs of the two reduction properties we study in the literature. One reduction property is weak normalization which states every term reduces to a normal form. In other words, at least one reduction sequence of any given term is finite. The result has been proven for example in [21]. We formalize a proof using logical relations following [12].

The second property is a stronger normalization statement. A system is strongly normalizing if every possible reduction leads to an irreducible expression. We will follow Girard's proof by *Kripke*-style logical relations [14, 2, 18].

1.3 Outline of the Thesis

Chapters 2 and 3 contain the preliminaries for this thesis. Firstly, we describe Autosubst and the σ-calculus. Secondly, we will give more details on Lean, also on metaprogramming. The two aspects come together in Chapter 4 which gives details on the implementation of Autosubst in Lean. We show which components were added to Coq's Autosubst and mention implementation issues. In Chapter 5, we look more closely at the tactics needed for Autosubst and automation approaches possible in Lean. In chapters 6 and 7 we first give a mathematical proof of weak and strong normalisation of the simply-typed $\lambda-$calculus and then show how they are realised in Lean and Coq. To conclude, Chapter 8 evaluates the work of this thesis and points out possible directions for future work.

Chapter 2

The σ-calculus and Autosubst

We start out by describing the theory behind Autosubst. One aspect is the choice of representation in the current version: de Bruijn and well-scoped syntax. The second aspect is the equational theory of Autosubst: the calculus of explicit substitutions, also known as the σ-calculus, and its extension to vector substitutions.

2.1 Parallel De Bruijn Substitutions

Binders introduce local definitions of variables, usually written informally as $\lambda x.s$, where x can occur as a bound variable in s. Because variable names are exchangeable, the named representation makes a syntactic distinction between terms which are α-equivalent, like $\lambda x.x$ and $\lambda y.y$.

The *de Bruijn* representation [10] abstracts from variable names to simplify formal implementations. In de Bruijn notation, a variable is a numerical index pointing to the binder that introduced it. Enclosing binders are counted from zero. For example, the term $\lambda f.\lambda x.fx$ is denoted by $\lambda\ \lambda\ 1\ 0$, where variables greater than 2 are out of the scope of the term.

In the following, the untyped λ-calculus, short UTLC, is considered, a simple term language with abstraction and application. With variables seen as de Bruijn references, terms have the form

$$s^m, t^m \in tm_m ::= x^m \mid \lambda\ s^{m+1} \mid s^m\ t^m \ (x \in \mathbb{I}_m, m \in \mathbb{N}).$$

Terms are well-scoped [23], which means their type carries as additional information how many bound variables the term contains. This is achieved by taking variables from an m-element finite type $\mathbb{I}_m$ instead of $\mathbb{N}$. The finite type is obtained by iterating the option type $\mathcal{O}$ on the empty type, i.e, $\mathbb{I}_0 ::= \emptyset$ and $\mathbb{I}_{n+1} ::= \mathcal{O}(\mathbb{I}_n)$.

$$
\begin{aligned}
x[\sigma] &= \sigma(x) \\
s\ t[\sigma] &= s[\sigma]\ t[\sigma] \\
\lambda s[\sigma] &= \lambda(s[\Uparrow \sigma])
\end{aligned}
\qquad\qquad
\begin{aligned}
(\sigma \circ \tau)\ x &::= \sigma(x)[\tau] \\
\Uparrow \sigma &::= 0 \cdot (\sigma \circ \uparrow)
\end{aligned}
$$

Figure 2.1: Operations of the σ-calculus.

An example for a *variable instantiation* is β-reduction in UTLC. A β-reduction takes place if a λ-abstraction is applied to another term, as in $(\lambda x.s)\ t$. This results in a redex s_t^x where the variable x is substituted by t in s.

A way to represent instantiation is the use of *explicit substitutions* [1]. Substitutions map the variables that occur in a term to a term they should be substituted with. In our de Bruijn model where variables are natural numbers, substitutions are represented as functions $\sigma, \tau : \mathbb{I}_m \to tm_n$.

If a substitution only exchanges indices, it is called a *renaming* $\xi, \rho : \mathbb{I}_m \to \mathbb{I}_n$. Examples for renamings are the identity $\mathrm{id}\ x ::= x$ and the shift renaming that increases all indices by one, $\uparrow x ::= x + 1$.

Because only the order of the de Bruijn indices matters, substitutions can be thought of as finite sequences of terms $[\sigma_0, \sigma_1, ...]$. They can be extended to the front using the *cons* operation:

$$t \cdot \sigma = [t, \sigma_0, \sigma_1, ...] ::= \lambda\, n.\ \text{if } n = 0 \text{ then } t \text{ else } \sigma_{n-1}.$$

More syntactic operations for substitutions are given in Figure 2.1. Their goal is to represent instantiation of variables in a term with a substitution. To this end, an instantiation operation $s[\sigma]$ describes how a substitution σ acts on term s. The substitution descends a term in parallel and replaces all de Bruijn indices at once.

When a binder is traversed, the interpretation of the indices in the substitution has to be adapted to a context with a new binder. The *lifting* operation $\Uparrow$ performs this index change by preserving index zero and incrementing the indices in the other terms in the sequence. We also have *forward composition* $\circ$ for substitutions.

Using instantiation and cons, β-reduction of $(\lambda s)\ t$ can be expressed as $s[t \cdot \mathrm{id}]$.

The λ-calculus with substitutions and the syntactic operations given above forms a model of the σ-calculus. Originally [1], substitutions were introduced as functions $\sigma : \mathbb{N} \to tm$. In the well-scoped setting, their domain is restricted to a finite type. This way, substitutions only instantiate variables in the scope of a term [26]. This

$$
\begin{aligned}
(st)\,[\sigma] &= s[\sigma]t[\sigma] & \mathrm{id} \circ \sigma &= \sigma \\
(\lambda s)\,[\sigma] &= \lambda\,(s[\Uparrow \sigma]) & \sigma \circ \mathrm{id} &= \sigma \\
0[s \cdot \sigma] &= s & (\sigma \circ \tau) \circ \theta &= \sigma \circ (\tau \circ \theta) \\
\uparrow \circ (s \cdot \sigma) &= \sigma & (s \cdot \sigma) \circ \tau &= s[\sigma] \cdot (\sigma \circ \tau) \\
s[\mathrm{id}] &= s & s[\sigma][\tau] &= s[\sigma \circ \tau] \\
0[\sigma] \cdot (\uparrow \cdot \sigma) &= \sigma & 0 \cdot \uparrow &= \mathrm{id}
\end{aligned}
$$

Figure 2.2: Rewriting system of the σ-calculus.

is reflected in the type of the instantiation operation:

$$[\cdot]\cdot : \forall n\ m, (\mathbb{I}_n \to tm_m) \to tm_n \to tm_m.$$

2.1.1 The σ-calculus as a Rewriting System

Besides the fact that the σ-calculus can express substitution operations needed for reductions, it induces a terminating rewriting system. The rules of the system are confluent[9] and complete [24]. Thus, every expression that only consists of substitution operations and the syntactic operations in UTLC has a unique normal form. The rewriting rules needed for deciding equality in the calculus are given above.

2.1.2 Vector Substitutions

More complex languages than the UTLC have multiple sorts of binders, for example term and type binders as present in System F. System F has the following syntax:

$$A^m, B^m \in ty_m ::= x_{ty}^m \mid A \to B \mid \forall\, A^{m+1} (x \in \mathbb{I}_m, m \in \mathbb{N})(..)$$

Here we have substitutions $\sigma_{tm} : \mathbb{I}_m \to tm_n$ and $\sigma_{ty} : \mathbb{I}_m \to ty_n$ for both term and type instantiation. We might be tempted to just apply them to a term one by one in some order, e.g. $s[\sigma_{tm}][\sigma_{ty}]$. Say that a variable k is instantiated with a term t in σ_{tm} and t contains type variables. Now, if we instantiate with σ_{ty}, the type variables in t change under σ_{ty}. However, if we permute the substitutions $s[\sigma_{ty}][\sigma_{tm}]$, the type variables in t are unchanged. To avoid such interference problems, the substitutions are combined into a vector $[\sigma_{tm}; \sigma_{ty}]$. This way, term and type variables can be instantiated simultaneously.

The operations of the σ-calculus can be adapted to vector substitutions. The rewriting rules from the previous section also scale to vector substitutions. Furthermore, the extension of the σ-calculus is generic enough to match not only System F but any term language that can be specified in second-order abstract syntax, which is used by Autosubst.

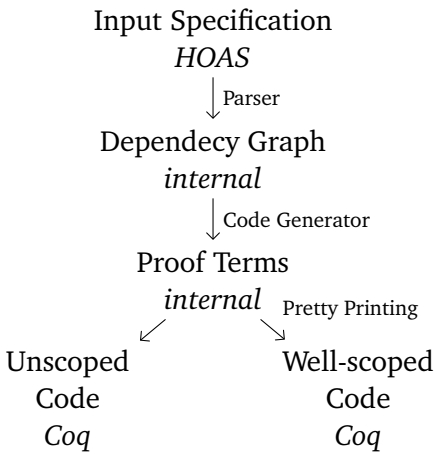

Figure 2.3: Components of Autosubst 2.

2.2 Autosubst: An Implementation for Coq

Autosubst implements a model of the σ-calculus in Coq. It defines a suitable instantiation operation for a given language specification. Instantiation definitions typically have to be set up for renamings first before they can be generalized to substitutions, otherwise, the definition is not structurally recursive in Coq.

Besides definitions of the substitution operations, Autosubst also derives and proves lemmas that correspond to the rules of the rewriting system. It provides a tactic `asimpl` that rewrites with them.

We focus on Autosubst 2 [26] which we want to extend to Lean. Autosubst 2 takes a second-order HOAS specification θ as input, with type and constructor declarations:

$$\theta ::= T_1 : \mathit{type}, \ldots C_1 : U_1$$
$$U ::= T \mid (T_1 \to \ldots \to T_n) \to U$$

The tool thus supports mutual inductive types with different variable sorts. It is based on the extended σ-calculus with vector substitutions. Instead of relying on `Ltac`, which does not allow mutual definitions, the library is written in Haskell.

Figure 2.3 shows how Autosubst is set up. The Haskell tool parses a given syntax description in HOAS into a graphical representation of syntax dependencies. This way, different binder sorts can be handled. From the dependency graph, Autosubst generates the definition for instantiation, substitution lemmas with corresponding proof terms, and the `Ltac` tactic `asimpl`. The output is a file of Coq definitions, either in unscoped or in well-scoped de Bruijn syntax.

Chapter 3

The Lean Theorem Prover

Most proof assistants are designed for the interactive construction of proofs and provide type checkers that guarantee the correctness of the results. Because providing all proof steps by hand is detailed and tedious work, it is desirable to let the prover perform generic steps automatically. This brings formal theorem proving closer to automated theorem proving, where the entire proof is found by the system.

Lean is a theorem prover designed for this purpose. It has the same framework for fully verified proofs as conventional proof assistants, and can also be used as a programming language for automation. Tutorials on both theorem proving and tactic programming in Lean are available online [4, 3].

In the following, we will look Lean more closely with the goal of supporting substitution in the prover. Lean's proof environment is described because it is used in the normalization proofs. Also, we look at Leans meta language as we will need it for implementing the rewriting tactics of Autosubst.

3.1 Axiomatic Foundation and Syntax

Lean is a functional language with dependent types. It is based on a version of dependent type theory known as the *Calculus of Constructions* [7] with inductive types [8], in short CIC. Dependent type theory is expressive enough to define common mathematical structures or functions and formulate assertions about them or to formalize parts of programs and state correctness claims. What sets it apart from other similarly expressive logical frameworks like set theory is the fact that every definable expression has a computable type. In particular, propositions have a type which is empty if a proposition is false and inhabited if it holds. Thus, to verify a claim it suffices to construct an expression in dependent type theory for it and to make sure that it type-checks. The expression itself can be seen as the proof. The propositions-as-types paradigm is used by most proof assistants, including Lean.

3.1.1 Object Language

Lean has a hierarchy of type universes, indexed with universe levels $u \in \mathbb{N}$. Every type is of type `Sort u` for a universe u, where `Sort u` itself has type `Sort u+1`. `Type u` is also used for `Sort u+1`. The bottom universe level `Sort 0` or `Type` can be used for the type of propositions, `Prop`, which is impredicative and can be marked as proof-irrelevant. Without `Prop`, Lean implements Martin-Löf type theory [17].

Lean provides syntax for inductive datatypes, structures, records, and type classes. It also has dependent function types, implicit arguments and type polymorphism. The following example shows how well-scoped terms of the λ-calculus can be defined in Lean. An indexed inductive type is used, where the index is a scope variable. Note that there is a binder Π for dependent function types, which can also be written using ∀.

```
inductive tm : ℕ → Type
  | var : Π {n : ℕ}, Fin n → tm n
  | app : Π {n : ℕ}, tm n → tm n → tm n
  | lam : Π {n : ℕ}, tm (nat.succ n) → tm n
```

Above, `Fin n` is the finite type $\mathbb{I}_n$. As in Coq, definitions can be grouped into namespaces or sections with local constants or variables. If they are not opened, objects in them have to be prefixed with their name. Note that this is also the case for inductive types like `nat` and the successor constructor `succ`.

Technically, Lean has a small kernel and a C++ code base. Definitions are compiled to bytecode and can be fully evaluated via a virtual machine, where missing or implicit type information is inferred. As opposed to Coq, the kernel only supports primitive recursion, more complicated forms are compiled to eliminators. Lean supports well-founded structural recursion, though not yet for mutually recursive definitions. This is of relevance for our development because it restricts the input syntax to non-mutual syntax types.

3.1.2 Classical Reasoning

Because substitutions are represented as functions, we often need to reason about equality of functions. Coq's Autosubst assumes the principle that two functions are equal if they agree on all arguments, known as *functional extensionality*. In Coq, this is a classical axiom, but can be safely assumed in dependent type theory [15].

Lean, in contrast, has a fews axiomatic extensions built-in, namely propositional extensionality, quotients and the axiom of choice. To an extent, these classical axioms are compatible with the computational interpretation of Lean [4]. The principle

of functional extensionality follows from the quotient construction and is thus also built-in. Here is its definition.

```
@funext : ∀ {α : Type u₁} {β : α → Type u₂} {f g : Π(x : α), βx},
    (∀x, fx = gx) → f = g
```

3.1.3 Proof Language

Because we use Lean as a proof environment in the case study, this section describes how to write proofs in Lean. Proofs can be stated declaratively or with use of tactics. In declarative proofs, the proof term is given directly similar to Coq. For better readability, the proof term can be structured using the keywords `assume`, `have`, `suffices`, and `show`.

There is also syntax for calculational proofs which allows a step-by-step proof of equalities or equivalences, where each step is labelled by a term that justifies it.

```
variables {α β : Type} (f : α → α → β)
variable symm : ∀ xy, f x y = f y x
variable fixpoint : ∀ x, f x x = x

example (a b c : α) (h₁ : f a b = f c c) : f b a = c :=
calc       f b a = f a b : symm b a
           ...   = f c c : h₁
           ...   = c     : fixpoint c
```

Alternatively, a proof can be constructed imperatively using tactics, i.e. commands that say how to construct a term. Statements in tactic mode are enclosed with the keywords `begin` and `end` or `by` for a single proof step.

Here is a simple example that mixes declarative use and tactics.

```
example (p q : Prop) : p ∧ q ↔ q ∧ p :=
begin
    apply iff.intro; intro h,
    {     have h₁ : p := h.left,
          have h₂ : q := h.right,
          show q ∧ p, exact ⟨h₂,h₁⟩ },
    {     exact ⟨h.right,h.left⟩ },
end
```

Note that a subgoal can be focused using curly brackets. The angle brackets ⟨.⟩ serve as an anonymous constructor, e.g., ⟨h₂,h₁⟩ is resolved as `and.intro h₁ h₂` above.

3.1.4 Lean's Simplifier

In the implementation of the rewriting system for Autosubst, we will need to reduce expressions with a set of directed rewriting lemmas. Lean provides a simplifier that is convenient for this purpose. The simplifier can be used via the tactic `simp` and computes the normal form of expressions, performing similar reductions as Coq's tactics `cbn` and `simpl`. However, the user can also specify simplification rules that tell the simplifier which definitions and theorems to use for reducing terms. Such hints can be given by tagging statements with `@[simp]` or writing
`attribute [simp] ` *——definition to unfold or statement to rewrite*

Similarly, if `@[refl]` is used, the reflexivity tactic uses hints. These mechanisms will be convenient for the component of the Autosubst implementation that normalizes expressions. Other approaches besides the simplifier are possible using Lean's meta language.

3.2 Lean as a Meta Language

Lean can be used for *meta programming*, that is, it provides methods to reason about the language itself. Meta programs are written in the same language as ordinary definitions and proofs, except that they are marked with the keyword `meta`.

Meta definitions can make use of the datatypes of the object language but also of methods or objects outside the axiomatic foundation that are untrusted. For example, expressions of the object language can be constructed or manipulated, and the current proof goal or definitions in the environment can be retrieved and modified.

3.2.1 Elaborator State and Tactics

The meta type `state` stores the current state of the elaborator, which contains the stack of proof goals and declarations in the local context such as datatype definitions or lemmas. The elaborator state can be modified by tactics. Because tactics are stateful, fallible operations, they are implemented with the `tactic` monad, described in more detail in the Appendix.

The monad can be thought of as an environment where untrusted meta objects live. Tactics executed in this environment only change the proof state if they are successful and leave it as is otherwise. The meta constant `state` is hidden in the monad, and thus in fact has type `tactic state`. The same holds for the meta types `environment` or `expr`. Tactics can operate on such objects and usually have return type `tactic` α for some type α.

```
meta def solve1 (tac : tactic unit) : tactic unit :=
do gs ← get_goals,
 match gs with
   | [] :=fail
   | g:: rs :=do
        set_goals [g],
        a ← tac,
        gs' ← get_goals,
        match gs' with
            | [] :=set_goals rs
            | _ :=fail
        end
   end
```

Figure 3.1: `solve1` Tactic in the Meta Language.

There tactic combinators such as $t_1 <|> t_2$, which backtracks and executes t_2 in case t_1 fails. Tactics can be sequenced with the operator >>. We also have t_1;t_2 or t_1 >> `all_goals` t_2 which execute t_2 on every subgoal produced by t_1, whereas `any_goals` only executes it on subgoals only if possible.

Useful primitives that read or modify the environment or proof goal are defined in the namespace `tactic`, such as `get_goals` and `set_goals` which read and update the list of proof goals, respectively. They are used by tactics like `solve1` (Figure 3.1).

By similar means, tactics such as `focus`, `assumption`, `apply` or `cases` can be implemented in Lean itself. They are defined under the namespace `tactic`.

3.2.2 Expressions and Quotation

The expressions of Lean's dependently typed object language can be described by dependent type theory itself. The meta type `expr` defines expressions inductively in the way that they are represented internally. There are syntax constructors for types, constants, applications, term and type abstractions, and let expressions. Furthermore, metavariables are used for the representation of placeholders and implicit arguments. Objects of type `expr` also live in the `tactic` monad and are always type-checked before being added to the context. `Meta` tactics that construct expressions that are not well-typed thus fail.

Lean provides a *quotation mechanism* to translate expressions as we write them into the corresponding objects of type `expr`.

For example, `'theorem1` is an object of type `expr` if `theorem1` is a defined name. The parser infers the full name of an object if double backticks are used, adding prefixes of open namespaces in the environment if necessary. Similarly, a compound expression `e` can be mapped to `expr` with `'(e)`. The expression can optionally contain placeholders that have to be resolved by the parser or elaborator. In that case, the expression is called pre-expression because it is only partially constructed. Pre-expressions have type `pexpr` and are created using double or triple backticks. The placeholders they contain are called *antiquotations* and are of the form `%%t`, where `t` is a variable for an expression.

```
example (p q : Prop) : p → q → q ∧ p :=
by do e₁ ← intro `h₁, e₂ ← intro `h₂,
      e₃ ← to_expr ``(and.intro %%e₂ %%e₁), exact e₃
```

If double backticks are used as in the above example, the names in the expression are resolved at parse time. For triple backticks, names are parsed at elaboration time, which is needed if local names occur:

```
example (p q : Prop) : p → q → q ∧ p :=
by do e₁ ← intro `h₁, e₂ ← intro `h₂,
      refine ```(and.intro h₂ h₁)
```

3.3 Comparison to Coq

As Autosubst was developed for Coq, we use Coq for comparison in the case study. This section points out some differences between the provers.

Both provers are based on versions of the CIC. There are a few differences like Lean's built-in classical constructions mentioned previously. Lean also has explicit universe levels for types with universe polymorphism which is only an experimental extension in Coq at present.

The prover provide mostly the same basic tactics, in some cases under different names. For example, `inversion` and `remember` are named `cases` and `generalize` in Lean. The `assumption` and `apply` tactics in Lean can infer implicit arguments, so they correspond to `assumption` and `eapply`. Similarly, Leans rewriting tactic `rewrite` or `rw` performs Coq's setoid-rewriting.

The tactics `auto`, `eauto` and similar ones are not provided in Lean. Using metaprogramming, though, we can easily inspect the context and look for hypotheses that can be applied, as done by `eauto`.

Worth noting for proofs is that Lean only supports tidy matches, whereas Coq tolerates them. If inductions would lead to an untidy match, they are not possible in Lean and generalizations have to be done first.

Finally, Lean's unfolding mechanism is a little different from Coq. Lean's definitions are not unfolded eagerly by default, which can be changed by marking it with one of the attributes `[reducible]`, `[semireducible]`, or `[irreducible]`. As pointed out before, we can also mark definitions in Lean with a simplification attribute to if it should be unfolded by simp.

Chapter 4

Autosubst in Lean

In the following, the adaptation of Autosubst 2 to Lean is explained in more detail. We have modified code generation and printing to produce well-scoped Lean code with the substitution definitions needed to support binders. Also, abstract syntax for Lean's meta language is used to generate tactics for rewriting and automation. An overview is shown in Figure 4.1, where dotted arrows show the extensions to Lean.

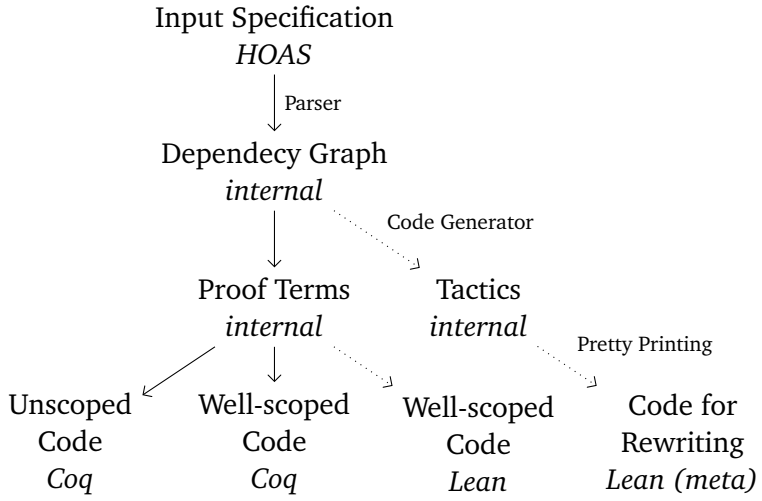

Figure 4.1: Components of Autosubst 2.

4.1 Components

As explained in Chapter 2, the existing Coq implementation translates a HOAS language specification to internal representations of the substitution primitives we need and prints Coq definitions from these.

The first step is to input parsing to identify syntax dependencies. This includes determining which syntax types can have variables and, in case there are multiple sorts, which substitution vectors are needed. The parsing step fits the Lean implementation as well.

4.1.1 Code Generator

In the next step, intermediate *abstract syntax objects* are generated. There are different Haskell data types for common Coq syntax objects. For instance, there are sentence types for recursive or inductive definitions and lemmas, and term types for terms and their components.

The Lean implementation needs similar abstract syntax entities. Some more syntax types are added to represent its meta language. Here is an example sentence type:

```
data MetaDefinition = MetaDefinition (String) (Binders) (Term) (Term)
```

Note that uses more syntax components that represent the name, arguments, return type and body of the meta definition. Meta objects like expressions and quotations are also added which can occur in terms.

The Coq implementation already generates the sentence types for substitution definitions and their proof terms. They can be mostly reused for Lean, with slight modifications (4.2).

Besides the usual substitution definitions, the Lean code generator generates tactics that normalize substitution expressions. As a simple rewriting approach, we generate a tactic that implements ordered rewriting with the substitution lemmas. For a second approach, we need tactics that perform expression matching on the goal and create a proof term for rewriting. How both approaches work in detail is described Chapter 5.

4.1.2 Pretty Printing

Printing code from abstract syntax objects is implemented using type classes in Haskell. A backend to Lean can be added via a new type class that translates the internal objects to Lean instead of Coq. For definitions and lemma declarations, we need straightforward adaptations of the syntax. The same goes for proof terms, using the simplification steps.

In addition to the substitution definitions and lemmas that are also present in the Coq version, meta definitions are printed from the syntax objects for meta types that were generated in the previous step.

```
inductive tm : ℕ -> Type
  | var_tm : Π {ntm : ℕ}, Fin ntm -> tm ntm
  | app : Π {ntm : ℕ}, tm ntm -> tm ntm -> tm ntm
  | lam : Π {ntm : ℕ}, tm (nat.succ ntm) -> tm ntm

def subst_tm : Π {mtm ntm : ℕ} (sigmatm : Fin mtm -> tm ntm)
           (s : tm mtm), tm ntm
  | mtm ntm sigmatm (var_tm s) :=sigmatm s
  | mtm ntm sigmatm (app s0 s1) :=app (subst_tm sigmatm s0)
                                      (subst_tm sigmatm s1)
  | mtm ntm sigmatm (lam s0) :=lam (subst_tm (up_tm_tm sigmatm) s0)

lemma instId_tm {mtm : ℕ} : subst_tm (@var_tm mtm) =id :=
  funext (λ x , idSubst_tm (@var_tm mtm) (λ n , by refl) (id x))
```

Figure 4.2: Lean code printed by Autosubst for the STLC.

Figure 4.2 shows part of the Lean code printed for the λ-calculus. The generated syntax for the λ-calculus will be also used in the case studies of weak and strong normalization.Lean outputs a term type definition `tm` and an instantiation operation `subst_tm`. Note that the instantiation definition corresponds to the one seen in Chapter 2.

One of the generated lemmas is `instId`. It implements the rewriting rule $s = s[\mathrm{id}]$. Its proof uses functional extensionality and a previous definition `idSubst` with the following type:

$$\texttt{idSubst} : \forall\, \{m\}\ (\sigma : \mathbb{I}_m \to \mathrm{tm}_m), (\forall x, \sigma x = x) \to \forall s, s[\sigma] = s$$

4.2 Implementation Details

Some of the proof terms derived for Coq have to be adjusted for Lean. Because Lean's definitions are semi-reducible, some additional unfolding steps are necessary to prove lemmas about definitionally equal objects. To do this, definitions can tagged with an attribute for Leans built-in simplifier such that the necessary unfolding steps can be done automatically by `simp with substLemmas`.

```
run_cmd mk_simp_attr ‘substLemmas
@[substLemmas]
——definition
```

```
def upId {m n : ℕ} {σ : Fin m → tm n} (Eq : ∀ x, σ x =var x) :
    ∀ x, (up σ) x =(@var (nat.succ m)) x :=
  λ n , match n with
  | (Fin.fs n) :=
        have h : _, from ap (ren shift) (Eq n), --proof term
      begin simp with substLemmas at *, assumption end
  | Fin.fz :=by refl
  end
```

Figure 4.3: Proof term modifications.

An example for the λ-calculus is given in Figure 4.3. Some of the usual substitution operations occur, that is, `up`, `shift` and instantiation with a renaming `ren`. Also, `fs`, `fz` are the constructors of our finite type. The details are not important here, this is just to show that the proof term `ap (ren shift) (Eq n)` has to be reduced with the simplifier before it can be used.

4.3 Limitations

In principle, the same second-order HOAS specifications are accepted as in the Coq version. Support for well-founded mutual recursion is limited in the current version of Lean, though. If a term sort is defined mutually, most definitions such as instantiation are mutually recursive with structural recursion on the first argument. However, Lean does not recognize that the structural recursion is well-founded.

For small definitions, a meta tactic can be used prove that the recursions are well-founded (see `wf_single_arg` in the development), but this timeouts as soon as definitions get larger. Timeout problems already occur for small languages such as a simply typed λ-calculus with a mutual type for terms and values. Future versions of Lean will hopefully provide native support for mutual recursion for a more stable solution.

Chapter 5

Automation for Rewriting

To implement the rewriting system of the extended σ-calculus, we need a procedure that converts a target expression to its normal form according to the rewriting rules. In this chapter, different approaches to do this are examined. In examples, expressions of the simple σ-calculus are used, which can be terms of the λ-calculus, substitutions or mixed expressions constructed with the substitution operations (Definition 5.1). We assume a set of rewriting rules $r : e_1 = e_2$ and denote the irreducible normal form of an expression with $\overline{e}$.

Definition 5.1 (Syntax of expressions, λ-calculus)

$$e, e' \in exp := n \mid \lambda.e \mid e\ e' \mid e \cdot e' \mid e[e'] \\ \mid \mathrm{id} \mid \uparrow \mid e \circ e' \mid \Uparrow e$$

5.1 Rewriting Tactics and the Simplifier

A straightforward way to simplify expressions is to greedily rewrite with applicable rules until the normal form is reached. In Lean, a meta definition that tries to rewrite with a given list of lemmas can be implemented as shown below.

```
meta def rw_pexpr (e : pexpr) : tactic unit :=
do e ← tactic.to_expr e,
   t ← target,
   (p,h,_) ← tactic.rewrite e t,
   replace_target p h

meta def rw_list (default : tactic unit): list pexpr → tactic unit
| [] :=default
| (e:: es) :=do (rw_pexpr e >> tactic.skip) <|> rw_list es
```

The tactic expects a list of pre-expressions which are the names of the rewriting lemmas. The list of pre-expressions can be created automatically during code generation.

Alternatively, the built-in simplifier of Lean can be used to reduce expressions. It is provided with hints which statements to use, so definitions that should be unfolded and lemmas for rewriting have to be tagged with a simplifying attribute.

5.2 Proof Term Construction

Rewriting with the rewriting tactic or the simplifier has the drawback of producing very large proof terms because rewriting can happen at an arbitrary subposition of the expression. Additionally, repeatedly invoking the rewrite-tactic can slow down the simplification process.

We adress these issues by choosing the rewriting lemmas and the order of rewriting manually before starting to rewrite. This can be done with Lean's reflection mechanism on expressions. With pattern matching on the target expression, the rewriting rules that match it syntactically can be selected and combined into a minimal proof term. Rewriting can be done in a single step using this proof term.

Proof Terms and Inferences

The goal is to find a proof term with type $e = \overline{e}$ for a goal expression e. The term is created bottom-up. To build it, the rules $r : e_1 = e_2$ of the rewriting system are replaced by lemmas with equality assumptions, called *transitivity lemmas*, of shape

$$\lambda a_1, \ldots, a_n.\ e = \overline{e}\ (e, \overline{e} \in \mathit{exp})$$

Their assumptions $a_i : e_i = \overline{e_i}$ contain subexpressions $e_i, \overline{e_i} \in \mathit{exp}$ of e and $\overline{e}$. The idea is that subexpressions e_i are already in normal form $\overline{e_i}$ and each assumption is a proof $e_i = \overline{e_i}$. The transitivity lemma then returns a proof $e = \overline{e}$ for the compound expression.

Proof terms of irreducible terms such as `id` or n are created with an reflexivity principle $\mathit{refl} : \forall(e : \mathit{exp}).\ e = e$.

Compound proof terms are created with the transitivity lemmas. Each proof term can be seen as the root of an inference tree because we can identify a transitivity lemma L: $\lambda a_1, \ldots a_n.\ e = \overline{e}$ with an inference

$$\frac{a_1 : e_1 = \overline{e_1} \qquad \ldots \qquad a_n : e_n = \overline{e_n}}{e = \overline{e}}\ L$$

In the inference tree, the right-hand sides are always irreducible subexpressions and the leaves are reflexity proofs.

$$\frac{}{\uparrow = \uparrow}\ \text{refl} \qquad \frac{e_1 = \overline{e_1} \qquad e_2 = \overline{e_2}}{e_1[e_2] = \overline{e_1}[\overline{e_2}]}\ \text{congrInst}$$

$$\frac{e = \overline{e}}{\lambda.e = \lambda.\overline{e}}\ \text{congrLam} \qquad \frac{e_1 = \overline{e_1} \qquad e_2 = \overline{e_2}}{e_1\ \overline{e_1} = e_2\ \overline{e_2}}\ \text{congrApp}$$

Figure 5.1: Examples of Reflexivity and Congruence Rules.

Deriving Proof Terms

When a compound term is normalized, subexpressions can be normalized in any order due to confluency of our rewriting system. As a consequence, we can start with atomic subexpressions of an expression e and normalize compound expressions bottom-up until we have an an inference tree for $e = \overline{e}$.

Leaves of an expression's syntax tree are irreducible expressions, thus the associated proof is reflexivity.

A compound expression decomposes into one or more subexpressions and either a type constructor or substitution operation. We can hereby assume proof terms for all subexpressions, because the inference tree is constructed bottom-up.

In the case of constructors, additional rules are necessary to prove that normalization agrees with the application of a constructor. We extend the set of transitivity lemmas with straightforward *congruence rules*, e.g. for a unary constructor C,

$$\forall\ e, e' \in \mathrm{exp},\ e = \overline{e} \to \mathrm{C}\ e = \mathrm{C}\ \overline{e}.$$

Some basic rules for the expressions in Definition 5.1 are shown in Figure 5.1.

More importantly, for a compound expression with a constructor or substitution operation at top level, we need a means to decide whether a rewriting rule applies. Before making this formal, we consider two examples.

The normalization rule $s[\mathrm{id}] = s$ for instance is only applicable to expressions of the shape $e_1[e_2]$. Due to our invariant that subexpressions are normalized, we assume proofs of $e_1 = \overline{e_1}$ and $e_2 = \overline{e_2}$. The rule applies in case $\overline{e_2}$ is id and we need to construct a proof term for $e_1[e_2] = \overline{e_1}$. This suggests that a suitable inference lemma is

$$\text{idInst: } \lambda(e_1 = \overline{e_1})(e_2 = \mathrm{id}).\ e_1[e_2] = \overline{e_1}$$

In the inference tree, this lemma is an inference as shown in Figure 5.2 from nodes $e_1 = \overline{e_1}$ and $e_2 = \mathrm{id}$ that were constructed before.

$$\frac{e_1 = \overline{e_1} \qquad e_2 = \mathrm{id}}{e_1[e_2] = \overline{e_1}}\ \text{idInst}$$

$$\frac{\begin{array}{cc} e_1 = \overline{e_1} & e_2 = \overline{e_2} \qquad e_3 = \overline{e_3} \\ \overline{e_2} \circ \overline{e_3} = \overline{e_4} & \overline{e_1}[e_4] = \overline{e_5} \end{array}}{e_1[e_2][e_3] = e_5}\ \text{instComp}$$

Figure 5.2: Example Inferences.

The rule in the previous example is simple in the sense that it only contains a single operator on the left-hand-side and normalizes to an atomic term. The rewriting system also contains more complex rules like $s[\sigma][\tau] = s[\sigma \circ \tau]$. Recall that equations in our tree have irreducible right-hand-sides, thus we aim at an inference lemma with return type $e_1[e_2][e_3] = e_3'$. In turn, all assumptions should also have an irreducible expression on the right. Evidently, we need assumptions covering the normalization of all subexpressions of $e_1[e_2][e_3]$, i.e. $e_1 = \overline{e_1}, e_2 = \overline{e_2}, e_3 = \overline{e_3}$. Additional assumptions capture how the normal expressions $\overline{e_1}, \overline{e_2}, \overline{e_3}$ reduce when plugged into $s[\sigma \circ \tau]$.

$$\begin{aligned} \text{instComp}\colon\ & \lambda(e_1 = \overline{e_1})(e_2 = \overline{e_2})(e_3 = \overline{e_3}) \\ & (\overline{e_2} \circ \overline{e_3} = \overline{e_4})(\overline{e_1}[\overline{e_4}] = \overline{e_5}).\ e_1[e_2][e_3] = \overline{e_5} \end{aligned}$$

Finding Lemmas

The inference lemmas can be derived generically for rewriting rules $r : e_1 = e_2$. Note that the left-hand-side e must be a compound expression because elementary expressions cannot be normalized further. Let $e_1, \ldots e_n$ be the subexpressions of e.

Firstly, we add assumptions $e_i = e_i'$ $(\forall i)$ to the transitivity lemma.

Secondly, we look for subexpressions $e_{i_1}, \ldots e_{i_n}$ which occur under a common operator O in $\downarrow e$ (in our case, $n \leqslant 2$). The subexpression $O(e_{i_1}, \ldots e_{i_n})$ should also not contain any other operators. An assumption

$$a_i : O(e_{i_1}', \ldots e_{i_n})' = e_i''$$

is added to the lemma. Also, we replace $O(e_{i_1}, \ldots e_{i_n})$ with e_i'' in $\downarrow e$ and continue the process with the resulting term e'. When all operations in $\downarrow e$ have been shifted/moved to the assumptions, we obtain the inference lemma

$$L_r : \lambda a_1, \ldots a_m, e = e'$$

Application of Inference Lemmas

At every inference node, we have the current expression e, an operator or constructor at its top level, and inference trees $e_i = \overline{e_i}$ for all subexpressions e_i of e. It remains to find an applicable lemma and grow the inference tree.

As observed earlier on, a lemma is applicable only if the left-hand-side of its head has the same shape as e. Also, the lemma's assumptions have to hold true. By construction, assumptions make statements about subexpressions of e or about the right-hand sides of earlier assumptions.

Assumptions about subexpressions of e determine whether the lemma is applicable. Whether they hold can be simply read off the inference tree constructed at this point. All assumptions of idInst are of this type.

For the remaining assumptions, we do not have constructed derivations yet. Thus all right-hand-sides are unknown, including the final normalized expression $\downarrow e$. All left-hand sides contain subexpressions $e_i, \overline{e_i}$ that have been normalized earlier on, therefore we can recurse on them to find an inference tree and the unknown normal forms. In terms of the example lemma instComp, there is a recursion on the expression $\overline{e_2} \circ \overline{e_3}$ to find a derivation for $\overline{e_2} \circ \overline{e_3} = e_4$. The result e_4 is used to find the normal form e_5, this time by recursing on $\overline{e_1}[e_4]$.

To sum up, we have developed a method that constructs an inference tree $e = \overline{e}$ for a given expression e. The derivation lemmas associated with the nodes of the tree can be chained into a proof term with type $e = \overline{e}$.

Implementation

The transitivity lemmas can be generated along with the usual substitution lemmas.

To apply the inference lemmas, we need meta tactics for each syntax operation. They take two proof terms as arguments which prove how the subexpressions normalize. For example, `normalizeInst:expr` $\to$ `expr` $\to$ `expr` expects arguments $\mathrm{Eqs} : \vec{=}\overline{s}$ and $\mathrm{Eq}\sigma : \sigma = \overline{\sigma}$ and returns a proof term $s[\sigma] = \overline{s[\sigma]}$. It applies rewriting lemmas if necessary, for instance if $s = \overline{s}$ and $\sigma = \mathrm{id}$, it returns `idInst Eqs` $\mathrm{Eq}\sigma$, and if no Lemma is applicable, `congrInst Eqs` $\mathrm{Eq}\sigma$

To find out which lemmas match, the function has to match on the proof terms for the subexpressions. For the idInst lemma from the previous section, it finds out whether $\mathrm{Eq}\sigma$ normalizes an expression to the identity. For more complicated lemmas such as instComp, the function has to know which constructor is at top level of the normalized subexpressions.

This is taken over by auxiliary functions which look for the relevant syntactic operators. A minimal example is shown below. The pair of functions take a proofterm $L : e = \overline{e}$ as input and check whether $\overline{e}$ is a λ-abstraction. This is the case if the transitivity lemma L that was used to construct the proofterm returns an abstraction, for example if it is a reflexivity of a λ-term or a λ-congruence. If L is some other lemma that returns an equality on terms, such as idInst, we have to recurse on its second argument which tells us whether $\overline{e}$ is an abstraction. Similarly, all other lemmas that return equalities on terms are matched on.

```
meta def isLam : expr → bool
  | `( refl (lam  %%Eqs)) :=tt
  | `( congrLam %%Eqs) :=tt
  | `( idInst %%Eqσ %%Eqs) :=isLam Eqs
  -- ...
  | _ :=ff

meta def destructLam : expr → tactic expr
  | `( refl (lam %%Eqs)) :=return Eqs
  | `( congrLam %%Eqs) :=return Eqs
  | `( idInst %%Eqσ %%Eqs) :=destructLam Eqs
  -- ...
  | e :=return e
```

Functions like the above ones are needed for the other term constructors and for operations that return terms, such as the instantiation operations. For operations that return substitutions, such as composition or cons, the lemmas that are recursed on are different ones, namely those that return equalities on substitutions. An example is the following lemma:

$$\text{idCompLeft}\colon \lambda(\sigma_1 = \overline{\sigma_1})(\sigma_2 = \mathrm{id}).\ \sigma_1 \circ \sigma_2 = \overline{\sigma_1}$$

Using auxiliary functions, the parts of `normalizeInst` that look for idInst and instComp look as follows.

```
meta def normalizeInst : expr → expr → expr
| s σ :=
   if (isId σ) then
      to_expr (idInst %%s %%σ)
   else if (isInst s) then
      do (t, τ) ← destructInst s,
      θ ← normalizeComp τ σ,
      s' ← normalizeInst t θ,
      to_expr (instComp %%s %%τ %%σ %%θ %%s')
   else -- ... more lemmas about instantiation
```

Note that the destruct function `destructInst` has type `expr` $\to$ (`expr` $\times$ `expr`) because it returns proof terms for both subexpressions.

To summarize, the functions needed are `normalizeInst`, `normalizeComp`, `normalizeCons` and `normalizeUp`. Because they are mutually recursive and Lean does not support mutual recursion for meta definitions yet, they are implemented in a single function using a natural number as indicator, i.e. the normalize function has type `nat` $\to$ `expr` $\to$ `expr` $\to$ `expr`. Because the operator $\Uparrow$ is unary, the part for `normalizeUp` ignores the second `expr` argument.

After the proof term has been synthesized, a simple meta definition can rewrite with it in the context. The tactic monad ensures that the constructed proof term is well-typed.

Chapter 6

Weak Normalization of the λ−Calculus

A reduction system for a language is weakly normalizing if every term has a reduction sequence to an irreducible term. The property holds for well-typed terms of the simply typed λ-calculus (STLC) and call-by-value reduction. It can be proved using a common way of reasoning by *logical relations* [12].

Logical relations are a versatile proof technique usually used to express a semantic model [2]. In our context, logical relations characterize weakly normalizing terms semantically. The main part of the weak normalization proof is a soundness property which shows that well-typed terms are in the logical relation.

The proof [12] is formalized as a first case study for Lean's Autosubst. We first outline the mathematical proof and then connect it to the implementation.

6.1 The Simply Typed λ-Calculus

The STLC with numerical constants and addition is shown in Figure 6.1. The representation is *de Bruijn*, thus variables are taken from finite types $\mathbb{I}_m,\ m \in \mathbb{N}$.

We define values as terms which are either a constant or a λ-abstraction.

Typing contexts can be elegantly represented in well-scoped syntax as functions

$$
\begin{aligned}
A, B \in \mathsf{ty} &:= \mathsf{int} \mid A \to B \\
\Gamma \in \mathsf{ctx} &:= \emptyset \mid A \cdot \Gamma \\
s, t \in \mathsf{tm} &:= x \mid n \mid \lambda s \mid s\ t \mid s + t \ (x \in \mathbb{I}_m, n, m \in \mathbb{N})
\end{aligned}
$$

Figure 6.1: Syntax of STLC.

from finite types $\Gamma_m : \mathbb{I}_m \to \mathrm{ty}$. Analogous to substitutions, they can be extended to the front using cons which adds a new type for a variable to the context.

$$\frac{\Gamma x = A}{\Gamma \vdash x : A} \qquad \frac{A \cdot \Gamma \vdash s : B}{\Gamma \vdash \lambda s : A \to B}$$

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B} \qquad \frac{\Gamma \vdash n_1 : \mathbb{N} \qquad \Gamma \vdash n_2 : \mathbb{N}}{\Gamma \vdash n_1 + n_2 : \mathbb{N}}$$

$$\frac{}{\Gamma \vdash \overline{n} : \mathbb{N}}$$

Figure 6.2: Typing judgment $\vdash$.

In order to describe typing in this setting, we use contexts $\Gamma : \mathbb{I}_m \to \mathrm{ty}$ to assign a type Γx to every bound variable x in the context.

The *typing judgment* in Figure 6.2 makes precise how typing contexts and terms interact. Well-typed terms in STLC are those that obey the typing rules under a given context.

The type for variables can be looked up in the context as we use context functions. An application types if the left-hand side has an adequate function type. The essential typing rule for abstraction needs an extension of the typing context to a new bound variable with the cons operation.

The reduction operation on terms that is analyzed in this section is the small step semantics $\succ$ shown in Figure 6.3. Applications and addition reduce left-to-right, and reduction of abstractions is `call-by-value`.

Since we want to analyse reduction sequences of more than one step, a big-step semantics will also be used. A big-step semantics is given by the reflexive-transive closure $\succ^*$ (Figure 6.4).

$$\frac{s_1 \succ s_1'}{s_1\ s_2 \succ s_1'\ s_2} \qquad \frac{\text{value } s_1 \qquad s_2 \succ s_2'}{s_1\ s_2 \succ s_1\ s_2'}$$

$$\frac{s_1 \succ s_1'}{s_1 + s_2 \succ s_1' + s_2} \qquad \frac{\text{value } s_1 \qquad s_2 \succ s_2'}{s_1 + s_2 \succ s_1 + s_2'}$$

$$\frac{\text{value } t}{\lambda s\ t \succ s[t \cdot \mathrm{id}]} \qquad \frac{n_1 + n_2 = n_3}{\overline{n_1} + \overline{n_2} \succ \overline{n}_3}$$

Figure 6.3: Small-step reduction $\succ$ for call-by-value $\lambda-$calculus.

$$\frac{}{s \succ^* s} \qquad \frac{s_1 \succ s_2 \qquad s_2 \succ^* s_3}{s_1 \succ^* s_3}$$

Figure 6.4: Reflexive-transitive closure of $\succ$.

6.2 Weak Normalization

In order to prove that reduction via $\succ^*$ is weakly normalizing, the purely syntactic characterization of typing introduced before is not sufficient. It is convenient to use a semantic model $\vDash$ in place of $\vdash$.

6.2.1 Logical Relations

A well-typed closed term is weakly normalizing if it reduces under $\succ^*$ to a well-typed value of the same type. This intuition can be made formal using a logical relation. For each type A, we give a set of terms that behave as intended when typed as A, so to say semantically well-typed terms. This yields the expression relation

$$\mathcal{E}[\![A]\!] := \{s \mid \exists t,\ s \succ^* t \wedge t \in \mathcal{V}[\![A]\!]\}.$$

To describe a semantically well-typed value, a logical relation on values is introduced. Its argument types are restricted to $\mathbb{N}$ and $A \to B$. For $\mathbb{N}$, the relation simply contains all constants. More interestingly, an abstraction of type $A \to B$ is well-typed if the application to any semantically well-typed value results in a semantically well-typed term.

$$\begin{aligned} \mathcal{V}[\![\mathbb{N}]\!] &:= \{\overline{n}\} \\ \mathcal{V}[\![A \to B]\!] &:= \{\lambda s \mid \forall v \in \mathcal{V}[\![A]\!].\ s[v \cdot \mathrm{id}] \in \mathcal{E}[\![B]\!]\} \end{aligned}$$

To handle free variables in a term, semantical well-typedness is extended to substitutions. The context relation $\mathcal{G}$ defines agreement of a type substitution $\Gamma : \mathbb{N} \to \text{type}$ and a substitution on the free variables, $\sigma : \mathbb{N} \to \text{tm}$.

$$\mathcal{G}[\![\Gamma]\!] := \{\sigma \mid \forall x.\ \sigma x \in \mathcal{V}[\![\Gamma x]\!]\}$$

Putting it all together, semantic typing can be defined in terms of $\mathcal{E}$ and $\mathcal{G}$.

Definition 6.1 (Semantic typing)

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

6.2.2 Compatibility and soundness

In order to use the semantic interpretation of typing for the normalization proof, it first has to be shown that it is sound with respect to syntactic typing, i.e., we have an inclusion $\vdash \subseteq \vDash$.

To this end, we establish the following properties about the logical relations $\mathcal{E}, \mathcal{V}, \mathcal{G}$ and the relationship of $\vdash$ and $\vDash$.

Lemma 6.2 (Value inclusion) *If $s \in \mathcal{V}[\![A]\!]$, then also $s \in \mathcal{E}[\![A]\!]$.*

Proof By definition using reflexivity of $\succ^*$.

For each typing rule of $\vdash$, a *compatibility lemma* will be proven which states that the respective rule also holds for $\vDash$.

Lemma 6.3 (Compatibility of $\vDash$ with variables) *If $\Gamma\ x = A$, then $\Gamma \vDash x : A$.*

Proof Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$.
Then $(\sigma\ x) \in \mathcal{V}[\![A]\!]$ by definition. By value inclusion also $\mathcal{E}[\![A]\!](\sigma\ x)$.

Lemma 6.4 (Compatibility of $\vDash$ with abstraction) *If $A \cdot \Gamma \vDash s : B$ then $\Gamma \vDash \lambda s : A \to B$.*

Proof Let $\sigma \in \mathcal{G}[\![\Gamma]\!]$, then we need $(\lambda s)[\sigma] = \lambda s[\sigma] \in \mathcal{E}[\![A \to B]\!]$.
By value inclusion, $\lambda s[\sigma] \in \mathcal{E}[\![A \to B]\!]$ suffices. Thus, let $v \in \mathcal{V}[\![A]\!]$.

To prove $s[v \cdot \sigma] \in \mathcal{E}[\![A]\!]$, the assumption $A \cdot \Gamma \vDash e : B$ can be used.
Its premise $v \cdot \sigma \in \mathcal{G}[\![A \cdot \Gamma]\!]$ follows from pointwise agreement of $v \cdot \sigma$ and $A \cdot \Gamma$ given $v \in \mathcal{V}[\![A]\!]$ and $\sigma \in \mathcal{G}[\![\Gamma]\!]$.

Lemma 6.5 (Compatibility of $\vDash$ with application) *If* $\Gamma \vDash s : A \to B$ *and* $\Gamma \vDash t : A$ *then* $\Gamma \vDash s\ t : B$.

Proof Again let $\sigma \in \mathcal{G}[\![\Gamma]\!]$. We need $st[\sigma] = s[\sigma]\ t[\sigma] \in \mathcal{E}[\![B]\!]$.

From $\Gamma \vDash s : A \to B$ follows that there exists a v such that $s[\sigma] \succ^* v, v \in \mathcal{V}[\![A \to B]\!]$.

Similarly, from $\Gamma \vDash t : A$, there exists a v' such that $t[\sigma] \succ^* v', v' \in \mathcal{V}[\![A]\!]$. Because of $v = \lambda s' \in \mathcal{V}[\![A \to B]\!], (\lambda s')\ v' = s'[v' \cdot \mathrm{id}] \in \mathcal{E}[\![B]\!]$. Unfolding the definition of $\mathcal{E}$ provides a $v'' \in \mathcal{V}[\![B]\!]$ such that $s[\sigma]\ t[\sigma] \succ^* v''$ which concludes the proof.

Lemma 6.6 (Compatibility of $\vDash$ with addition) *If* $\Gamma \vDash s : \mathbb{N}$ *and* $\Gamma \vDash t : \mathbb{N}$ *then* $\Gamma \vDash s\ t : B$.

Proof Given $\sigma \in \mathcal{G}[\![\Gamma]\!]$, the goal is $(s + t)[\sigma] = s[\sigma] + t[\sigma] \in \mathcal{E}[\![B]\!]$.

From our assumptions for $\vDash$, we get that there exist $v, v' \in \mathcal{V}[\![\mathbb{N}]\!]$ that $s[\sigma]$ and $s[\sigma]$ reduce to, respectively. By definition of $\mathcal{V}[\![]\!]\mathbb{N}$, they must be constants $v = \overline{n}, v' = \overline{n'}$.

We have $s[\sigma] + t[\sigma] \succ^* \overline{n + m}$ from the reduction rules and transitive closure. Additionally, $\overline{n + m} \in \mathcal{V}[\![\mathbb{N}]\!]$, thus $\overline{n + m} \in \mathcal{E}[\![\mathbb{N}]\!]$ as needed.

As the compatibility lemmas suggest, every inference rule of $\vdash$ can be simulated with $\vDash$. As a consequence, the inclusion $\vdash \subseteq \vDash$ holds, called semantic soundness.

Theorem 6.7 (Semantic soundness) *If* $\Gamma \vdash s : A$ *then* $\Gamma \vDash s : A$.

Proof By induction on $\Gamma \vdash s : A$. For each rule of $\vdash$, the resulting inductive hypotheses state that the assumptions hold for $\vDash$. Thus the application of the matching compatibility lemma proves the claim.

6.2.3 Weak Normalization

By passing from $\vdash$ to $\vDash$ with soundness, we can prove weak normalization.

Theorem 6.8 (Weak normalization of $\succ^*$) *If* $\emptyset \vdash s : A$ *then* $\exists v,\ s \succ^* v \wedge$ *value* v

Proof By semantic soundness, $\emptyset \vDash s : A$. Specialized to the identity substitution $\mathrm{id} \in \mathcal{G}[\![\emptyset]\!]$, this implies $\mathcal{E}[\![A]\!]s$ which proves the claim. □

```
Fixpoint fin (n : nat) : Type :=
  match n with
      | 0 => False
      | S m => option (fin m)
  end.
```

```
inductive Fin : ℕ → Type
    | fz {n} : Fin (succ n)
    | fs {n} : Fin n → Fin (succ n)
```

Figure 6.5: Definition of finite types in Coq (left) and Lean (right).

6.3 Realization Lean and Coq

In the following, more details on the formalization in Lean and Coq are given. As one aspect, we focus on syntax representations and proof steps that are different in the provers. Secondly, we emphasize issues related to binders and substitution to evaluate the performance of Autosubst.

The term type needed to represent the syntax of STLC is an indexed inductive family in Lean as shown in Chapter 3 with a straightforward extension to constants and addition. The Coq definition is similar.

We are in the single-sorted setting, that is, the only binders are term binders and there is an instantiation operation for a single substitution. Recall that substitutions are represented as functions from our finite type to terms.

Worth noting is that we use a slightly different definition of finite types in the provers. Coq defines them with a match on natural numbers whereas an inductive version is used in Lean (Figure 6.5). The inductive version has a similar induction principle as the fixpoint definition in Coq. It has been chosen over an equational definition because the definitions made with the equation compiler are less easy to work with. For example, matching on hypotheses of type `Fin` reduces the term into long complicated expression if an equational definition is used.

As another example, recall the cons operation

$$t \cdot \sigma = [t, \sigma_0, \sigma_1, ...] ::= \lambda\, n.\ \text{if } n = 0 \text{ then } t \text{ else } \sigma_{n-1}$$

Using finite types, the defintion of cons in Lean is given as follows:

```
def scons {X : Type} {n : ℕ} (x : X) (f : Fin n → X) (m : Fin (succ n)) : X :=
    match m with
    | fz :=x
    | (fs x) :=(f x)
    end
```

As a side note, we can reason about finite types, the cons operation and function composition independently from the binder sorts present in the input language. Their equational theory is therefore pre-defined separately and Autosubst only needs to generate lifting and instantiation specific to the input syntax.

Example: Unfolding

Next, we consider part of a proof where support for substitution is needed. Recall the compatibility lemma for binders (6.4):

$$A \cdot \Gamma \vDash s : B \to \Gamma \vDash \lambda s : A \to B$$

Value inclusion can be used to reduce the goal to $([\lambda]s)\sigma \in \mathcal{E}[\![A \to B]\!]$. In Lean, our proof goal at this state looks as follows.

$\vdash V\ (A \to B)\ (\texttt{lam}\ s)\ [\sigma]$

Simplifying with Autosubst tactics leads to

$\vdash V\ (A \to B)\ \texttt{lam}\ s\ [\texttt{var\_tm fz} .: \sigma \gg \texttt{ren\_tm} \uparrow]$

Above, $\gg$ is notation for composition and $\cdot[\cdot]$ for instantiation. Both operations are provided by Autosubst. Furthermore, `var_tm fz` is lowest de Bruijn index in our finite type, previously denoted as 0 for readability.

Observe that the modification of σ is the index change we have seen in the definition of instantiation (Chapter 2):

$$\lambda s[\sigma] = \lambda(s[\Uparrow \sigma]) = s[0 \cdot (\sigma \circ \uparrow)]$$

Thus, Autosubst has carried out an unfolding step of the instantiation definition for terms and the up operation.

Technically, Coq's Autosubst incorporates unfolding in the tactic `asimpl`. In Lean, there are two options. If the respective definitions have been tagged with an attribute for the simplifier, `simp` can be used.

We can alternatively use the rewriting tactic `arw` which rewrites with suitable equation lemmas. For example, the equation lemma for the abstraction case of instantiation has the type

```
subst_tm.equations._eqn_3 :
  ∀ (m n : ℕ) (σ : Fin m → tm n) (s : tm (nat.succ m)),
      lam s[σ] =lam s[up_tm_tm σ]
```

Example: Rewriting

If we continue the proof and assume $\nu \in \mathcal{V}[\![A]\!]$, the following proof goal remains.

$$\vdash \exists\ (\nu_2 : \mathtt{tm}\ m),\ s\ [\mathtt{var\_tm}\ \mathtt{fz}\ .:\ \sigma \gg \mathtt{ren\_tm}\ \uparrow][\nu .:\ \mathtt{var\_tm}] \succ^* \nu_2 \wedge V\ B\ \nu_2$$

More involved equational reasoning than in the previous step is needed here. Autosubst has to use the rewriting lemmas. Among others, the following rule of our rewriting system is used:

$$s[\sigma][\tau] = s[\sigma \circ \tau]$$

Lean's Autosubst tactic `arw` rewrites with the necessary lemmas, as does `asimpl` in Coq. The resulting proof goal looks as follows:

$$\vdash \exists\ (\nu_2 : \mathtt{tm}\ m),\ e[(\nu .:\sigma)] \succ^* \nu_2 \wedge V\ B\ \nu_2$$

The target expression is similar to the paper proof again at this point, where we have the goal $s[\nu \cdot \sigma] \in \mathcal{E}[\![A]\!]$.

Chapter 7

Strong Normalization of the λ−Calculus

So far we considered weak normalization which states that for any well-typed term, there is at least one reduction sequence that leads to a value and is thus finite. Subsequently, it can be asked whether all possible reduction sequences for a term are finite. This problem is called *strong normalization*. Strong normalization of STLC has also been chosen in the POPLMark challenge [5] to compare how binder handling can be dealt with in different proof assistants.

For call-by-value lambda calculus, the stronger formulation follows as a straightforward consequence from the weak normalization result because the reduction system is formulated deterministically. However, if the reduction of terms is nondeterministic as in full lambda calculus, proving strong normalization is more involved.

We follow a well-known proof by Girard [14] that has also been referred to in [2]. The proof again builds on the use of logical relations. In more detail, we will use a single Kripke-style logical relation [18] that includes a quantification over possible words, here contexts.

7.1 Reduction Relation and Substitutivity

In full lambda-calculus, the conditions on reduction behavior are relaxed in the sense that reductions below a binder are allowed, and applications and addition can reduce on either side, irrespective of whether the left sub-term is a value. Terms and syntactic typing are defined as in the previous section.

The reduction relation is closed under substitution and preserves typing. This property is called *substitutivity* and will be of relevace for later proofs.

Lemma 7.1 *If* $s \succ t$ *then* $s[\sigma] \succ t[\sigma]$.

Proof By induction on $\succ$. □

$$\frac{s_1 \succ s_1'}{s_1\ s_2 \succ s_1'\ s_2} \qquad \frac{s_2 \succ s_2'}{s_1\ s_2 \succ s_1\ s_2'} \qquad \frac{s \succ s'}{\lambda s \succ \lambda s'}$$

$$\frac{s_1 \succ s_1'}{s_1 + s_2 \succ s_1' + s_2} \qquad \frac{s_1 \succ s_1'}{s_1 + s_2 \succ s_1' + s_2}$$

$$\frac{}{\lambda s\ t \succ s[t \cdot id]} \qquad \frac{n_1 + n_2 = n_3}{\overline{n_1} + \overline{n_2} \succ \overline{n}_3}$$

Figure 7.1: Small-step reduction for full λ-calculus.

Lemma 7.2 (Substitutivity) *If* $s \succ^* t$ *then* $s[\sigma] \succ^* t[\sigma]$.

Proof By induction on $s \succ^* t$ using substitutivity of $\succ$. □

Substitutivity for renamings ξ can be stated as $s \succ^* t \to s\langle\xi\rangle \succ^* t\langle\xi\rangle$ and follows as a special case of the previous lemmas.

To prove preservation of typing, reordering and extension of the context have to be allowed. Such a weakening of the context which can be concisely formulated using renamings. We speak of *agreement under renaming*:

$$\Gamma \preccurlyeq_\xi \Delta := \forall x.\ \Delta(\xi\ x) = \Gamma\ x$$

The important statement about typing under agreeing contexts is the following.

Lemma 7.3 (context morphism for renamings) *If* $\Gamma \vdash s : A$ *and* $\Gamma \preccurlyeq_\xi \Delta$ *then* $\Delta \vdash s\langle\xi\rangle : A$.

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

Context morphism is needed in the preservation proof because our typing system has rules that extends contexts, i.e., the abstraction rule.

Lemma 7.4 (Preservation) *If* $s \succ^* t$ *and* $\Gamma \vdash s : A$ *then* $\Gamma \vdash t : A$.

Proof By induction on $s \succ^* t$ and inversion on $\Gamma \vdash s : A$, using the morphism lemma for renaming. □

7.2 Strong Normalization Predicate

The strong normalization property can be expressed inductively using an accessibility predicate *SN* which holds for a term whenever it holds for all of its $\succ$-successors. As *SN* holds trivially for terms with no successors, it can access a term if all its successors have a finite reduction path to an irreducible term.
Strong normalization is defined with the following inference rule:

$$\frac{\forall\, t.\ s \succ t \to SN\ t}{SN\ s}$$

With the goal of proving that *SN* holds for all well-typed terms in STLC, we first look at some properties of the predicate. The strong normalization property straightforwardly extends to all successors of a term.

Fact 7.5 *If* $SN\ s$ *and* $s \succ^* t$ *then* $SN\ t$.

Proof By induction on $s \succ^* t$. □

The following properties describe how *SN* behaves under the syntax constructs of the STLC.

Fact 7.6 $SN\ (s + t)$ *is equivalent to* $SN\ s \wedge SN\ t$.

Proof ($\to$) By induction on $SN\ s + t$.
($\leftarrow$) By induction on $SN\ s$ and $SN\ t$. □

A similar equivalence does not hold for application, because even for strongly normalizing terms s, t, an application can make new reductions possible and potentially lead to infinite branches in the reduction trees. An example is given by $\omega := \lambda x.xx$. ω itself is irreducible but $\omega\ \omega \succ^* \omega\ \omega$ in more than zero steps which results in an infinite reduction sequence.

However, it can be shown that *SN* is always backwards closed with respect to application.

Fact 7.7 *If* $SN\ (s\ t)$ *then* $SN\ s \wedge SN\ t$.

Proof Both $SN\ s$ and $SN\ t$ is shown by induction on $SN\ (s\ t)$. □

The other direction holds only in some cases. We only need to consider applications $(\lambda s)t$ with an abstraction on the left, because well-typed abstractions will reduce to this form. If s, t, and the redex $s[t \cdot id]$ are strongly normalizing, then so is $(s\ t)$.

Fact 7.8 *If* $SN\ s$, $SN\ t$ *and* $SN\ s[t \cdot id]$, *then* $SN\ (\lambda s)t$.

$$\begin{aligned}\mathcal{R}_\Gamma[\![\mathbb{N}]\!] &:= \{s \mid \Gamma \vdash s : \mathbb{N} \ \wedge \ SN\ s\}\\ \mathcal{R}_\Gamma[\![A \to B]\!] &:= \{s \mid \Gamma \vdash s : A \to B \ \wedge \\ &\qquad \forall\ \xi\ \Delta\ t,\ \Gamma \preccurlyeq_\xi \Delta \to t \in \mathcal{R}_\Gamma[\![A]\!] \to (s\langle\xi\rangle\ t) \in \mathcal{R}_\Delta[\![B]\!]\}\end{aligned}$$

Figure 7.2: Reducibility Relation $\mathcal{R}$.

Proof By nested induction on *SN* s and *SN* t using substitutivity of $\succ^*$. □

The situation for instantiation is similar to application because substituting terms such as $\omega\omega$ in for variables can violate strong normalization. Thus only backward closure can be shown.

Fact 7.9 *If SN* $s[\sigma]$ *then SN* s.

Proof By induction on *SN* $s[\sigma]$ using substitutivity of $\succ^*$. □

For renamings ξ, we can even show equivalence since changing variable names does not affect the reduction behavior of terms. A lemma is needed first.

Fact 7.10 *If* $s\langle\xi\rangle \succ t$ *then there exists a* t' *such that* $t = t'\langle\xi\rangle$ *and* $s \succ^* t'$.

Proof By induction on s and inversion on $s\langle\xi\rangle \succ t$. □

Fact 7.11 *SN* $s\langle\xi\rangle$ *is equivalent to SN* s.

Proof $(\to)$ By induction on *SN* $s\langle\xi\rangle$.
$(\leftarrow)$ By induction on *SN* s using fact 7.10. □

7.3 Typing Relation

Rather than proving the strong normalization property for all well-typed terms directly, a relational model is used. The logical relation $\mathcal{R}$ characterizes well-typed terms under a typing context by recursion on types. Intuitively, this resembles the design of the weak normalization proof, but the distinction between value and expression is no longer made. In particular, $\mathcal{R}$ takes the typing context Γ as an additional argument such that an explicit relation on contexts is no longer needed.

We call terms in the relation reducible. All reducible terms have to type syntactically under the given context. For the base type $\mathbb{N}$, we additionally require that *SN* holds. For a function type $A \to B$ we need that applications of a term s to terms t in $\mathcal{R}_\Delta[\![A]\!]$ will be in $\mathcal{R}_\Delta[\![B]\!]$. Note that Δ could possibly bind more type variables than

Γ, or index the type variables in s differently than Γ. To keep the formalization of $\mathcal{R}$ general, Both context extension and reordering of variables are allowed.

Using $\mathcal{R}$, the idea of context agreement can be extended to substitutions.

Definition 7.12 (context agreement) $\Gamma \preccurlyeq_\sigma \Delta := \forall x.\ x \in \mathcal{R}_\Delta[\![\Gamma x]\!](\sigma x)$.

In contrast to renamings, we relate a term σx and its type Γx to the context Δ using semantic well-typedness. Now it has to be proved that syntactic typing is preserved under instantiation for agreeing contexts. This can be first proved for contexts that argee on all variables, and then for contexts that agree semantically as in 7.12.

Lemma 7.13 $\Gamma \vdash s : A \to \forall x.\ \Gamma \vdash (\sigma\ x) : (\Delta\ x) \to \Delta \vdash s[\sigma] : A$.

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

In the binder case, we have to show $A \cdot \Delta \vdash e[0 \cdot \sigma \circ \uparrow] : B$. By induction hypothesis, this is implied by $\forall x.\ A \cdot \Delta \vdash (0 \cdot \sigma)\ x : (A \cdot \Gamma)\ x$. □

Lemma 7.14 (context morphism under substitution)

$$\Gamma \vdash s : A \to \Gamma \preccurlyeq_\sigma \Delta \to \Delta \vdash s[\sigma] : A.$$

Proof By Lemma 7.13 and the fact that syntactic typing follows from $\mathcal{R}$. □

7.4 Strong Normalization

Following Girards proof, we establish three essential properties of the reducibility relation. First, reducible terms are strongly normalizing. Second, the relation is forward closed. Lastly, the relation is backwards closed in case of a term which is not an abstraction. Such terms are called *neutral*, with *neutral* $s = \bot \Leftrightarrow s = (\lambda s')$.

Theorem 7.15 (CR_1) $s \in \mathcal{R}_\Gamma[\![A]\!] \to SN\ s$.

CR_1 will be shown simultaneously with CR_3.

Lemma 7.16 (CR_2 for $\succ$) $s \in \mathcal{R}_\Gamma[\![A]\!] \to s \succ t \to t \in \mathcal{R}_\Gamma[\![A]\!]$.

Proof By induction on A.

$t \in \mathcal{R}_\Gamma[\![A]\!]$ follows in each case with preservation of typing.

For $\mathbb{N}$, *SN* t follows from the induction hypothesis as *SN* is forward closed.
For function types, we can apply the induction hypothesis modulo context renaming. □

Theorem 7.17 (CR_2) $s \in \mathcal{R}_\Gamma[\![A]\!] \to s \succ^* t \to t \in \mathcal{R}_\Gamma[\![A]\!]$.

Proof Follows from 7.16 with induction on $s \succ^* t$. □

Theorem 7.18 (CR_3)

$$\Gamma \vdash s : A \to \mathit{neutral}\ s \to (\forall t.\ s \succ t \to t \in \mathcal{R}_\Gamma[\![A]\!]) \to s \in \mathcal{R}_\Gamma[\![A]\!].$$

Lemma 7.19 $CR_1 \wedge CR_3$.

Proof By induction on the argument type.

1. **Case** $(A = \mathrm{int})$:

   CR_1 holds because *SN* is true by definition.
   CR_3 assumes that any successor t is in $\mathcal{R}$, thus *SN* t holds which proves *SN* s.

2. **Case** $(A = A \to B)$:

   For CR_1, $\mathcal{R}_\Gamma[\![A \to B]\!]$ gives us an assumption for applications, namely that $s\langle\xi\rangle\ t \in \mathcal{R}_\Delta[\![A \to B]\!]$ for an agreeing context Δ and $t \in \mathcal{R}_\Delta[\![A]\!]$.
   To make use of this, we need to go from *SN* s to *SN* $(s\langle\xi\rangle\ 0)$. Note that *SN* $(s\langle\xi\rangle\ 0)$ is a stronger statement because of Facts 7.7 and 7.11. The claim now follows with $IH_{B,CR1}$ and $IH_{A,CR3}$.

   For CR_3, we need to show $s\langle\xi\rangle\ t \in \mathcal{R}_\Delta[\![B]\!]$, where $\Gamma \preccurlyeq_\xi \Delta$ and $t \in \mathcal{R}_\Delta[\![A]\!]$.

   As a first step, we first deduce from IH_{A,CR_1} that *SN* t holds such that by induction, for all successors t' of t that are in $\mathcal{R}_\Delta[\![A]\!]$, $s\langle\xi\rangle\ t' \in \mathcal{R}_\Delta[\![B]\!]$ (IH_*).

   IH_{B,CR_3} leaves us with the following subcases.

   (a) $\Delta \vdash s\langle\xi\rangle\ t : B$.

   $\Delta \vdash t : A$ follows straightforwardly from $t \in \mathcal{R}_\Delta[\![A]\!]$.
   Because $\Gamma \vdash s : A \to B$, s is either a variable or an application. In the former case $\Gamma \preccurlyeq_\xi \Delta$ can be used. In the application case preservation of typing under renaming is needed.

   (b) *neutral* $(s\langle\xi\rangle\ t)$ by definition.

   (c) $t' \in \mathcal{R}_\Delta[\![B]\!]$ for a successor t' of $s\langle\xi\rangle\ t$. From our reduction rules, we know that t' must be of shape $s'\ t$, $s\langle\xi\rangle\ t'$ or $s'[t \cdot \mathrm{id}]$.

   If $s'\ t$ where $s\langle\xi\rangle \succ s'$, lemma 7.10 can be used.

   If $t' = s\langle\xi\rangle\ t'$ where $t \succ^* t'$, we use IH_*. The resulting claim $t' \in \mathcal{R}_\Delta[\![A]\!]$ is a consequence of CR_2.

   If $s'[t \cdot \mathrm{id}]$ where $s\langle\xi\rangle = \lambda s'$,we know that $s\langle\xi\rangle$ cannot be neutral, thus s is not neutral either, and exfalso can be used.

CR_1 allows us to easily pass from membership in $\mathcal{R}$ to strong normalization. What is still missing is the connection between syntactic typing and reducibility, i.e., a soundness theorem for $\vdash \subseteq \mathcal{R}$. Some more properties of $\mathcal{R}$ are needed first.

Lemma 7.20 $x \in \mathcal{R}_{A\cdot\Gamma}[\![A]\!]$.

Proof Because variables are neutral and without successors, CR_3 proves the claim. □

Lemma 7.21 $s \in \mathcal{R}_\Gamma[\![A]\!] \to \Gamma \preccurlyeq_\xi \Delta \to s\langle\xi\rangle \in \mathcal{R}_\Delta[\![A]\!]$.

Proof By induction on A.

The fact $\Delta \vdash s[\xi] : A$ can be shown for both the base type and function types with preservation of typing under renaming and $\vdash \subseteq \mathcal{R}$.

For $A = \mathtt{int}$, *SN* $s\langle\xi\rangle$ by statement 7.11.

For $A = A \to B$, we have to show $s\langle\xi\rangle\langle\rho\rangle \in \mathcal{R}_{\Delta'}[\![B]\!]$, where $\Gamma \preccurlyeq_\xi \Delta$ and $\Delta \preccurlyeq_\rho \Delta'$.
This follows from $s \in \mathcal{R}_\Gamma[\![A \to B]\!]$ because $\Gamma \preccurlyeq_{\xi\circ\rho} \Delta'$ holds. □

The previous two facts can be used to prove that context agreement is preserved under context extension.

Lemma 7.22 $\Delta \preccurlyeq_\sigma \Gamma \to A \cdot \Delta \preccurlyeq_{\sigma\circ\uparrow} A \cdot \Gamma$

Proof With lemma 7.20 for $x \in \mathit{Fin}_0$, and lemma 7.21 otherwise. □

Lemma 7.8 can be lifted to $\mathcal{R}$.

Lemma 7.23

$$t \in \mathcal{R}_\Gamma[\![A]\!] \to \mathit{SN}\ s \to (A \cdot \Gamma) \vdash s : B \to s[t \cdot \mathrm{id}] \in \mathcal{R}_\Gamma[\![B]\!] \to (\lambda s)\ t \in \mathcal{R}_\Gamma[\![B]\!].$$

Proof With $\mathcal{R}_\Gamma[\![A]\!]$ and CR_1, *SN* t. The proof is by induction on *SN* s and *SN* t. With CR_3, we have left to show:

1. $\Gamma \vdash (\lambda s)\ t : B$ which is clear from $s[t \cdot \mathrm{id}] \in \mathcal{R}_\Gamma[\![B]\!]$.
2. *neutral* $(s\langle\xi\rangle\ t)$.
3. $t' \in \mathcal{R}_\Gamma[\![B]\!]$ for any t' such that $(\lambda s)\ t \succ t'$.

   If $t' = (\lambda s')\ t$, use IH_s. Here, $(A \cdot \Gamma) \vdash t' : A$ and $\lambda t'[t \cdot \mathrm{id}] \in \mathcal{R}_\Gamma[\![B]\!]$ follow from preservation and CR_2, respectively.

   If $t' = (\lambda s)\ t''$, use IH_t. We get $t' \in \mathcal{R}_\Gamma[\![A]\!]$ from $t \in \mathcal{R}_\Gamma[\![A]\!]$ by CR_2.
   As for $s[t'' \cdot \mathrm{id}] \in \mathcal{R}_\Gamma[\![B]\!]$, we can use CR_2 because $s[t' \cdot \mathrm{id}] \succ^* s[t'' \cdot \mathrm{id}]$.

   If $t' = s[t \cdot \mathrm{id}]$, we are done. □

We conclude by giving a proof of the soundness theorem and the strong normalization result.

Theorem 7.24 (Soundness of $\mathcal{R}$) $\Gamma \vdash s : A \to \Gamma \preccurlyeq_\sigma \Delta \to s[\sigma] \in \mathcal{R}_\Delta[\![A]\!]$.

Proof By induction over the typing judgment.

Case $s[0 \cdot \sigma \circ \uparrow] \in \mathcal{R}_\Delta[\![A \to B]\!]$:

To show $\Delta \vdash \lambda s[0 \cdot \sigma \circ \uparrow] : A \to B$, we need $A \cdot \Delta \vdash s[0 \cdot \sigma \circ \uparrow] : B$.
Context morphism reduces this to $A \cdot \Gamma \vdash s : B$, which is an assumption, and $A \cdot \Delta \preccurlyeq_\sigma \circ \uparrow A \cdot \Gamma$ which has been shown in lemma 7.22.

To show $\lambda s[0 \cdot \sigma \circ \uparrow]\langle\xi\rangle t \in \mathcal{R}_\Delta[\![B]\!]$, where $\Delta \preccurlyeq_\xi \Delta'$, lemma 7.23 is applied.

The remaining cases follow with the hypotheses. □

Corollary 7.25 (Strong normalization) $\emptyset \vdash s : A \to SNs$.

Proof From fact 7.9, CR_2 and soundness of $\mathcal{R}$.

7.5 Realization Lean and Coq

Regarding definitions, similar primitives as in the weak normalization proofs are needed: the term type and substitution primitives Autosubst generates, and inductive types for reduction and typing.

The predicate *SN* is defined in Lean as follows:

```
inductive SN {n} (R : tm n → tm n → Prop ) : tm n → Prop
  | sn_step (e₁ : tm n) : (forall e₂, R e₁ e₂ → SN e₂) → SN e₁.
```

In the proofs, we are faced with more substitution-related subgoals than in weak normalization, in particular for substitutivity and morphism. To illustrate, here is how the substitutivity can be proven using Autosubst's simplifications.

```
lemma substitutivity {n m} (s t σ) : s ≻ t → s[σ] ≻ @subst_tm n m σ t :=
begin
    intro h, revert m σ , induction h; intros m σ,
    any_goals { arw, constructor; aauto },
    { arw,
     apply substitutivity_h,
     now_arw, }
end}
```

	Lean		*Coq*	
Components	Def.	Proof	Def.	Proof
Autosubst	167	160	194	57
Reduction and Typing	60	135	51	119
Weak Normalization	10	97	17	73
Strong Normalization	19	350	28	392
Meta, `Ltac`	~160	–	~ 90	–
Sum *(without tactics)* 256	742	290	641	

Table 7.1: Comparison of the lines of code in the Lean and Coq Formalizations.

In the substitutivity proof, the `aauto` tactic is just a custom automation tactic that tries to apply hypotheses in the context. The tactics of Autosubst that occur are `arw` for rewriting and `now_arw` which proves equations. Optionally, tracing can be enabled, that is, `arw` can print a list of lemmas that were used for rewriting.

In summary, the Coq and Lean proofs can be defined similarly if suitable automation tactics are available that take over the substitution-related work.

Table 7.1 shows a comparison of the implementations in Lean and Coq in terms of code lines [1]. The first row shows the code provided by Autosubst. In the overview, the Code is split into defined statements (Def.) and Lemmas or Theorems (Proof).

[1] Lines of code were counted with `loc` (`https://github.com/cgag/loc`) which supports both Coq and Lean.

Chapter 8

Conclusion

In this thesis, we have provided support for variable binding in Lean. Binders are one of the most tedious parts in language formalizations because they add a lot of technical and distracting details. Therefore, we wish to rely on automation tools to reduce the overhead. Lean with its promise for good automation fits this goal well.

We adapted Autosubst 2 which is based on an elegant equational theory. Additionally, its infrastructure is designed to allow adding a backend to another prover. Adapting Autosubst included deciding how to implement automation, for which we analysed Lean's metaprogramming approach.

Finally, the Autosubst implementation was put to use for the goal of proving weak and strong normalization of the λ-calculus in Lean. The substitution related technicalities were taken over by the tool. The case study also allowed a test and comparison of the different rewriting approaches.

8.1 Evaluation

Extending Autosubst 2 to generate Lean code was straightforward due to the fact that Autosubst layers an internal syntax representation between parsing and printing. As another point, the proofs are represented declaratively and built with Coq tactics. Thus, the intermediate syntax objects are mostly independent of Coq syntax and can be easily used to print Lean definitions and proof terms.

We compared different approaches regarding the automation of the rewriting system. Automation as in the `Ltac` version can be implemented with Lean's tactic language and its quotation mechanism for names.

It was also of convenience that Lean has a simplifier built in, especially because using the simplifier can be refined by declaring custom simplifying attributes for a more controlled way of rewriting.

Both approaches have the downside of constructing huge proof terms. So we also looked at a more elaborate form of automation in Lean. Constructing proof terms with meta tactics can make them smaller and more readable.

At the moment, Autosubst 2 for Lean does not support mutually inductive types because it needs some mutual recursions that are not recognized as well-founded by Lean. However, this restriction only holds until the language supports the recursion because the plain syntax definitions are provided also for mutual types.

Weak and strong normalization are fundamental, well-studied properties. As such, we want to be able to prove them in Lean which was facilitated by Autosubst's support for binders.

Strong normalization of STLC has also recently been proposed as a challenge problem for POPLMarkReloaded [2] to compare menchanization of metatheory across proof assistants.

For a small system like the simply typed λ-calculus, the normalization proofs have been a good starting point compare the prover to Coq and to test the support for binding of Autosubst in Lean.

To conclude, the benchmark also showcases the use of logical relations which is important proof technique.

8.2 Future Work

There are several directions for future work. The weak and strong normalization proofs could be extended to larger syntactic systems and more complex typing systems. One candidate is System F with subtyping as considered in the POPLmark challenge. Besides weak and strong normalization, other results of interest can be formalized, for example confluence properties.

As soon as support for well-founded recursion of mutual inductive types in Lean is added, we can also use specification languages with mutual inductive sorts.

As we have seen, the set-up of Coq's Autosubst 2 allowed for an extension to Lean without a full re-implementation of the tool. Thus it might be interesting to add a backend for another proof assistant such as Agda [19].

Another direction for future work would be to extend the expressivity of the original Autosubst tool with respect to the input languages that can be handled. Targets could be languages with more complex binding structures. Because adapting the syntax generation is mostly straightforward, such changes to the original tool should also carry over to Lean.

Last, the focus of future work could also be on additional automation approaches. The syntactic simplification on meta expressions explained before could be done on the object level for more efficiency by using reflection. This method is a verified decision procedure, as opposed to the tactic approaches that potentially fail. Additionally, matching on expressions in Lean is inefficient at present, which is why switching to the object language and working with reified terms would be preferable. A final advantage to this approach is that it can be also realized Coq.

Appendix A

Appendix

A.1 Monadic Programming in Lean

A monad is a type constructor $m : \texttt{Type} \to \texttt{Type}$ that can be used to simulate stateful operations in a functional programming language. It always comes with two operations: `return` produces an object $m\ \alpha$ for every type α, and `bind` allows to carry out operations in the monad. These are their types in Lean:

```
return : Π {m : Type → Type} [monad m] {α}, α → m α
bind : Π {m : Type → Type} [monad m] {α}, m α → (α → m β) → m β
```

For example, if we have partial functions $f : \alpha \to \mathcal{O}\beta$, where $\mathcal{O}$ is the option type, they can be applied to objects in the option monad $\mathcal{O} : \texttt{Type} \to \texttt{Type}$ [3]. The `return` function is the constructor `some`, and `bind` applies partial functions as follows:

```
bind {α β : Type} (a : Oα) (f : α → Oβ) :=
  match a with
      | some a :=f a
      | none :=none end
```

Lean provides the following notation.

$m\ \alpha >>= f$	`bind` $m\ \alpha\ f$
$m\ \alpha >> m\ \beta$	`bind` $m\ \alpha\ (\lambda a, m\ \beta)$
`do` $\alpha \leftarrow m\ \alpha, s$	`bind` $m\ \alpha\ (\lambda a, s)$

```
meta def rw_expr (p : pexpr) : tactic unit :=
do
  e ← tactic.to_expr e,
  t ← target,
  (p,h,_) ← rewrite e t,
  replace_target p h

meta def rw_exprs (default : tactic unit) (trc :=tt): list pexpr ? tactic unit
| [] :=default
| (e:: es) :=do
   rw_pexpr e  <|> rw_exprs es
```

Figure A.1: A Lean Tactic for Rewriting.

A.2 Autosubst Tactic Examples

In this section, it is shown how Autosubst's rewriting system can be implemented using tactic programming. First, we define a tactic that tries to rewrite the goal with a given expression if possible, see Figure A.1. We also have a tactic that tries a list of expressions, `rw_exprs`.

Next, rewriting lemmas have to be specified. We use a list of pre-expressions that contains quoted lemma names (`Lemmas`). Those are used by `arw`.

The unfolding of definitions can be done similarly. Definitions such as instantiation (`subst_tm`) are generated with Lean's equation compiler and the names of the equations can be used for rewriting. Definitions that contain a match, like `scons`, have similar equations for rewriting.

```
meta def Lemmas :=[“(instId_tm), ——...
meta def Eqns :=[“(subst_tm.equations._eqn_1), “(scons._match_1) ——...
```

Lists like in this example can be generated by Autosubst. The tactic `arw` can then be defined as in Figure A.2. Some possible customizations are also shown.

```
-- unfolding and rewritings
meta def arw : tactic unit :=tactic.repeat arw'
  do (rw_exprs tactic.failed Lemmas) <|>
    (rw_exprs tactic.failed Eqns)

-- rewrite until  failure
meta def arw : tactic unit :=tactic.repeat arw'

-- solve equations
meta def now_arw : tactic unit :=do arw, tactic.reflexivity

-- arw in hypothesis
meta def arw_at (h) : tactic unit :=
do
  hyp ← tactic.get_local h,
  tactic.revert hyp,
  arw,
  tactic.intro h,
  tactic.skip
```

Figure A.2: Possible Autosubst Tactics for Rewriting.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *Under consideration for publication in J. Functional Programming*, 2018. To appear.

[3] Jeremy Avigad, Leonardo de Moura, and Jared Roesch. Programming in Lean. `https://leanprover.github.io/programming_in_lean/programming_in_lean.pdf`, 2016.

[4] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem proving in Lean. `https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf`, 2017.

[5] 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 *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, pages 50–65, 2005. doi: 10.1007/11541868_4. URL `https://doi.org/10.1007/11541868_4`.

[6] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49:363–408, 2011.

[7] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[8] Thierry Coquand and Christine Paulin. Inductively defined types. In *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[9] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence prop-

erties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, March 1996. ISSN 0004-5411. doi: 10.1145/226643.226675. URL http://doi.acm.org/10.1145/226643.226675.

[10] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings) 75(5)*, pages 381–392, 1972.

[11] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. ISBN 978-3-319-21400-9. URL http://dblp.uni-trier.de/db/conf/cade/cade2015.html#MouraKADR15.

[12] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems, 2018. Lecture Notes. (2018).

[13] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110278. URL http://doi.acm.org/10.1145/3110278.

[14] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

[15] Martin Hofmann. Extensional concepts in intensional type theory. (1995). 1995.

[16] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de Bruijn syntax. *Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, January 8-9, 2018*, Jan 2018.

[17] P. Martin-Löf. Intuitionistic type theory. In *Bibliopolis*, 1984.

[18] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Ann. pure appl. logic*, 51(1-2):99–124, 1991.

[19] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481862. URL http://doi.acm.org/10.1145/1481861.1481862.

[20] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *Proceedings of*

the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988, pages 199–208, 1988.

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

[22] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013. doi: 10.1017/CBO9781139084673.

[23] Adams R. Formalized 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. Springer, Berlin, Heidelberg, 2006.

[24] 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. ACM, 2015.

[25] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, Aug 2015.

[26] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de Bruijn terms and vector substitutions. *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, 2019. To appear.

[27] The Coq Proof Assistant. `http://coq.inria.fr`.