SAARLAND UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR'S THESIS

GENERATING INFRASTRUCTURAL CODE FOR TERMS WITH BINDERS USING METACOQ AND OCAML

Author
Adrian Dapprich

Advisor
Dr. Andrej Dudenhefner

Reviewers
Prof. Dr. Gert Smolka
Dr. Andrej Dudenhefner

Submitted: 26th October 2021

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig 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, 26th October, 2021

Abstract

The metatheory of languages with binders (e.g. programming languages) often involves intuitive reasoning about the behavior of substitutions. But when formalizing metatheory in proof assistants based on type theory, like Coq, these intuitive notions can turn out to be boilerplate heavy and result in tedious proofs.

The Autosubst family of programs is concerned with generating substitution boilerplate code for languages with binders in Coq.
We present two programs that reimplement and extend Autosubst 2, one written in OCaml and one written in MetaCoq.
The program written in OCaml uses the Coq implementation as a library for code generation. The program written in MetaCoq uses the metaprogramming facilities of MetaCoq for code generation.

We discuss the implementation details and implementation challenges of both programs and how they extend the original Autosubst 2.

Acknowledgements

I would like to thank my friends for keeping me company during these last two crazy years.
I also want to thank Professor Smolka, for holding up the beacon of type theory at the University of Saarland, Kathrin, for her counsel and work on Autosubst 2, and Andrej, for helping and advising me first during ACP and now during my Bachelor's thesis.

And most importantly, I want to thank my family for always supporting me.

Contents

Abstract **v**

1 Introduction **1**
1.1 Coq . . . 1
1.2 MetaCoq . . . 2
1.3 Overview . . . 3
1.4 Related Work . . . 4
1.5 Contributions . . . 4

2 Autosubst **6**
2.1 Binders . . . 6
2.2 Substitutions . . . 8
2.3 De Bruijn Algebras . . . 9
2.3.1 Substitution Equations . . . 11
2.4 Traversals . . . 12
2.5 Autosubst Compiler . . . 13
2.5.1 Input Syntax . . . 13
2.5.2 Dependency Analysis . . . 16
2.5.3 Output Syntax . . . 19

3 Autosubst OCaml **21**
3.1 Motivation . . . 21
3.2 How it Works . . . 22
3.2.1 Usage of Coq as a Library . . . 22
3.2.2 Parsing User Input . . . 28
3.2.3 Code Generation . . . 30
3.3 Using Monads in OCaml . . . 34

4 Autosubst MetaCoq **36**
4.1 How it Works . . . 36

4.1.1 Parsing User Input 37
4.1.2 Programming in MetaCoq 39
4.1.3 Code Generation 42
4.2 Missing Features 43
4.3 Extension to MetaCoq 43

5 Case Studies **45**
5.1 Stark's Existing Case Studies 45
5.2 TAPL Exercise 23.6.3 47

6 Extensions **49**
6.1 Asimpl 49
6.1.1 Functional Extensionality 49
6.1.2 Asimpl with Functional Extensionality 49
6.1.3 Asimpl with Setoid Rewriting 51
6.2 Traced Syntax 53
6.2.1 Formalization 55
6.3 Autosubst 2 57

7 Conclusion **59**
7.1 Autosubst OCaml 59
7.2 Autosubst MetaCoq 59
7.3 Future Work 60

A Appendix **61**
A.1 System F_{cbv} 61
A.2 System F 62
A.3 σ_{SP}-calculus 62
A.4 Rewrite Laws of the de Bruijn Algebra of the λ-calculus 63
A.5 Reuse from the Coq Implementation 64

Bibliography **65**

Chapter 1

Introduction

1.1 Coq

Coq [32] is a proof-assistant based on type theory.
It can be used to formalize a mathematical proof by constructing a corresponding term in the functional programming language Gallina. This due to the Curry-Howard correspondence, which says that a mathematical proposition can be encoded as a type in type theory. Constructing an inhabitant of that type then corresponds to a proof of that proposition.

Coq's formal language is the Calculus of Inductive Constructions. As such it supports dependent types and inductive type definitions. Everything has a type, even a type itself, which leads to a hierarchy of types. For example starting at the trivial proof of the true proposition it holds that

$$\mathrm{I} \;:\; \top \;:\; \mathbb{P} \;:\; \mathbb{T} \;:\; \mathbb{T} \;:\; \ldots$$

with $\mathbb{P}$ being an impredicative universe of propositions.

Inductive type definitions in particular are useful to model various predicates, for example logical conjunction with a single constructor.

$$\begin{aligned} \wedge &: \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\ \mathtt{conj} &: \forall (\mathrm{P}\ \mathrm{Q} : \mathbb{P}), \mathrm{P} \to \mathrm{Q} \to \mathrm{P} \wedge \mathrm{Q} \end{aligned}$$

We can then prove a simple fact about conjunction by constructing the following Gallina term.

```
fun P Q c => match c with conj p q => conj q p end
```
$$: \forall (\mathrm{P}\ \mathrm{Q} : \mathbb{P}), \mathrm{P} \wedge \mathrm{Q} \to \mathrm{Q} \wedge \mathrm{P}$$

A user interacts with the Coq environment by way of vernacular commands. For example the `Inductive` vernacular command is used to define a new inductive

type.
Coq also implements a tactic language that can be used to construct proofs interactively.

Inductive type definitions can also be used to model programming languages (or other languages like first-order logic), as well as reduction semantics and type systems for these programming languages.
With this, metatheorems of programming languages can be expressed and proved in Coq. Two important metatheorems of programming languages are **progress** and **preservation**. For terms $\mathtt{tm}$, types $\mathtt{ty}$, an evaluation relation on terms $\succ : \mathtt{tm} \to \mathtt{tm} \to \mathbb{P}$ and a typing relation[1] on terms and types $\mathtt{has_type} : \mathtt{tm} \to \mathtt{ty} \to \mathbb{P}$

- the progress theorem states that for any well-typed term $\mathtt{t}$ there is a term $\mathtt{t}'$ so that $\mathtt{t} \succ \mathtt{t}'$,
- the preservation theorem states that for any well-typed term $\mathtt{t}$ that reduces to a term $\mathtt{t}'$, $\mathtt{t}'$ is still well-typed.

During proofs of theorems like the above, one often needs auxiliary lemmas about substitutions that are intuitively clear but tedious to prove.
There are several ways to address this problem, the Autosubst family of programs being one of them.

1.2 MetaCoq

The MetaCoq project [29] is metaprogramming library for the Coq proof assistant. It aims to formally verify metatheorems about Coq, in Coq, and to allow the development of metaprogramming plugins that can manipulate Gallina terms.
In order to do this, MetaCoq defines a reification of Gallina terms in Coq, i.e. a representation of the Gallina AST[2] as an inductive type definition, called `term`[3]. It also implements a way of converting from Gallina terms to the reified terms (**quoting**) and back (**unquoting**).

Quoting and unquoting is done by evaluating terms of the **`TemplateMonad`** type. The `TemplateMonad` is a monad that encapsulates side-effects that can interact with the Coq environment. A term of type `TemplateMonad A` represents a computation that will return a value of type `A` when evaluated.
For example, the

- `tmQuote :∀ A, A -> TemplateMonad term`
- and `tmUnquoteTyped :∀ A, term -> TemplateMonad A`

[1] where for a term $\mathtt{t}$ and type $\mathtt{T}$, $\mathtt{has_type}\ \mathtt{t}\ \mathtt{T}$ means that $\mathtt{t}$ is well-typed
[2] abstract syntax tree
[3] which clashes with other names we want to use, so we call it the MetaCoq AST

constructors can be used for quoting and unquoting.
The `TemplateMonad` also supports other interactions with the Coq environment like adding new definitions and inductive types to the environment using the constructors

- `tmDefinition : ident -> ∀ A, A -> TemplateMonad A`
- and `tmMkInductive : mutual_inductive_entry -> TemplateMonad unit`.

So by using the `TemplateMonad`, we can write metaprograms in Gallina that construct, unquote, and define other Gallina terms in the Coq environment.
Evaluation of `TemplateMonad` terms is implemented as a Coq plugin and is triggered by the `MetaCoq Run` vernacular command.

1.3 Overview

When formalizing metatheory of languages with binders one often needs boilerplate lemmas about substitutions. These lemmas are often intuitive, but a formal proof requires technicalities and is tedious.

Schäfer et al. [24] created the Autosubst tool for Coq. It derives proofs of certain rewrite lemmas about substitutions for a given language, and uses a completeness result about the σ_{SP}-calculus [23] to implement a decision procedure for equalities between terms.
However, it is implemented in `Ltac`, which is fragile and hard to debug because `Ltac` semantics are not well defined, and also imposes some limitations on the languages that Autosubst supports. For example, it does not support languages with variadic binders[4] or mutually inductive sorts[5].

Stark et al. [31] developed Autosubst 2 which improves upon Autosubst by having a simplified treatment of multi-sorted languages, and supports variadic binders, mutually inductive sorts, wellscoped syntax, modularly defined syntax, and other extensions.
It is implemented as a Haskell program so it does not suffer from the limitations of `Ltac`. The program takes a language specification and generates Coq source code that contains inductive types for the language, rewrite lemmas for substitutions and the `asimpl` tactic to automatically solve equalities between terms.
Autosubst 2 is described in detail in the dissertation of Stark [30], where she also mechanizes a convergence result of the σ_{SP}-calculus.

In this thesis we develop two reimplementations of Autosubst 2.
Autosubst OCaml is a reimplementation in OCaml, the language that Coq is implemented in. We explore how feasible it is to use the Coq implementation as a library

[4] An abstraction that can bind multiple variables
[5] Sorts that are defined in terms of each other like in the call-by-value System F (Appendix A.1)

for code generation.
Autosubst MetaCoq is a reimplementation in the MetaCoq framework for Coq. We explore how to use MetaCoq's metaprogramming capabilities to implement code generation.

We implement some improvements over Autosubst 2 in the OCaml reimplementation.
Neither program supports the modular syntax feature of Autosubst 2, as it is a complex extension that is out of scope for this work.

We discuss the general theory behind Autosubst (Chapter 2), our implementation in OCaml (Chapter 3), our implementation in MetaCoq (Chapter 4), case studies using the OCaml implementation (Chapter 5), and extensions to Autosubst 2 that we (attempt to) implement (Chapter 6).

1.4 Related Work

There are several other compilers that generate some amount of boilerplate code for languages with binders.
We give a brief overview here.

- Ott [25] is a compiler for language specifications with multiple backends. For example, it can generate code for Coq, Isabelle/HOL and Latex. This code can be used in formalizations but Ott does not generate any lemmas about the languages.
- LNGen [5] is related to Ott and uses its specification language. It generates code in the locally nameless style [9] and focuses on generating infrastructure lemmas.
- Needle&Knot [14] is a compiler for language specifications that also generates common infrastructure lemmas and tactics that use these lemmas.

1.5 Contributions

In this section we give a short summary of our contributions.

Note that the author wrote an initial reimplementation of Autosubst 2 in OCaml as part of his ACP project[6].
That version[7] only supports basic code generation and no generation of automation tactics.

Our contributions during the bachelor project are:

- Continuation of the reimplementation of Autosubst 2 in OCaml (Chapter 3).

[6] `https://courses.ps.uni-saarland.de/acp_20/`
[7] `https://github.com/uds-psl/autosubst-ocaml/tree/acp_submission`

- A partial reimplementation of Autosubst 2 as a MetaCoq plugin (Chapter 4).
- Extensions to the input syntax of the programs (Section 2.5.1).
- Extension to asimpl tactic (Section 6.1)
- Bugfixes in several projects:
  - One printing bug in Coq fixed (Technical Remark 3.5)
  - One bug in Autosubst 2 reported (Section 6.3)
  - One bug in the `monadic` library fixed (Section 3.3)

The MetaCoq plugin turned out to require a lot more work than we anticipated so there are some implementation details missing compared to Autosubst 2 and Autosubst OCaml, which is discussed in Section 4.2.

Chapter 2

Autosubst

In this chapter we discuss the general ideas underlying Autosubst 2, and by extension, our implementations. In the following we will just use the name Autosubst if the statement is general enough that it holds for all three implementations. Otherwise we use their names.

The goal of Autosubst is

- to lift the burden of generating tedious boilerplate code commonly used when proving metatheorems about languages with binders
- and to allow a user to automatically prove certain lemmas that appear in the context of substitutions by generating Coq tactics.

Autosubst achieves these goals by generating code for a de Bruijn algebra, which are introduced in Section 2.3.
We summarize the design decisions and properties of the theory behind Autosubst, which are a result of the work of Schäfer et al. [24] and Stark [30].
We also discuss the general behavior of an Autosubst compiler, which is a program implementing the code generation.

Since Autosubst is concerned with automatically generating boilerplate code for languages with binders some decisions have to be made about how to formalize binders and substitutions in the generated code. This is a long researched problem space with a lot of approaches [11, 12, 9, 19] that have different trade-offs.
We discuss the basic named syntax and single substitution approach below and contrast them with the approach that Autosubst takes.

2.1 Binders

The general definitions concerning binders are the following, based on [20, Section 5.1].
A **binder** is a construct which declares a variable to be local to a term. A **bound**

variable is any variable of a term that is associated with a binder. A **free variable** is any variable of a term that is not bound.
A term is **closed** if it contains no free variables.

The most prominent example of a binder in programming languages is the λ-binder of the eponymous λ-calculus which consists of a single sort $\mathtt{tm}$.

Definition 2.1 (λ-calculus)

$$s, t \in \mathtt{tm} := v \mid s\ t \mid \lambda v.s \qquad\qquad v \in \mathtt{string}$$

With a standard notion of substitution: $s[t/v]$ is a term where all free occurrences of v have been replaced by t.
Substitution naturally appears during β-reduction, which replaces a bound variable with an argument in the body of a λ-abstraction.

$$(\lambda v.s)t \to_\beta s[t\ /\ v]$$

Named Syntax

The textbook approach to binders is to use names for variables. In paper proofs this works well. However, when formalizing a language with this approach, one encounters some problems.

One problem is that of **α-equivalence**: two programs that only differ in the names of their bound variables have equivalent behavior. So by convention, equality is defined up to renaming of bound variables.

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

But with an implementation of the λ-calculus in Coq, these terms are not definitionally equivalent. So extra work needs to be done to reconstruct this α-equivalence.

Another problem is that of **capture-avoiding substitution**: when substituting a term s containing a free variable v into some other term where a variable of the same name is bound, the occurrence of v in s should not become bound.

$$(\lambda x.f\ x)[(\lambda y.y\ x)\ /\ f] \neq (\lambda x.(\lambda y.y\ x)\ x)$$

The free x in $(\lambda y.y\ x)$ would be captured because after the substitution it is associated with λx.

One solution is to say substitution can rename variables in a way that free variables are not captured. This might result in the following substitution.

$$(\lambda x.f\ x)[(\lambda y.y\ x)\ /\ f] = (\lambda z.(\lambda y.y\ x)\ z)$$

This, again, introduces more work in the formalization because substitution then needs the ability to pick fresh variable names. Some introductory materials resort to "sidestep this extra complexity" [21] by using substitutions only with closed terms.

De Bruijn Syntax

Instead of named syntax, Autosubst opts to use de Bruijn syntax [11] for the code it generates, which solves both of the above-mentioned problems. In de Bruijn syntax, a bound variable is a natural number n that specifies the variable is bound by the nth enclosing binder. And a free variable is a natural number greater than the number of enclosing binders.
The named term from above can be translated to the following term in de Bruijn syntax.

$$(\lambda f.\lambda x.f\ x)(\lambda y.y\ x) \simeq (\lambda.\lambda.1\ 0)(\lambda.0\ 1)$$

In de Bruijn syntax, terms have a canonical form, so there is no problem like α-equivalence.

Capture-avoiding substitution still has to be dealt with but it is a straightforward transformation. The indices of free variables are increased each time the substitution affects the body of a binder (the free variable 1 is shifted to 2 because substitution traverses into the body of λ).

$$\begin{aligned}(\lambda.1\ 0)[(\lambda.0\ 1)\ /\ 0] &= \lambda.(1\ 0)[(\lambda.0\ 2)\ /\ 1] \\ &= \lambda.1[(\lambda.0\ 2)\ /\ 1]\ 0[(\lambda.0\ 2)\ /\ 1] \\ &= \lambda.(\lambda.0\ 2)\ 0\end{aligned}$$

This regularity makes de Bruijn syntax a prime target for code generation.

2.2 Substitutions

Another design decision is about formalizing substitutions.

The textbook approach is using single substitutions, which replace a single variable with a term. This is what we have used above.
But for some proofs we still "need to build some (rather tedious) machinery to deal with the fact that we are performing multiple substitutions" [21] like the following lemma.

```
Lemma swap_subst : ∀ t x x1 v v1,
    x ≠ x1 → closed v → closed v1 →
    <{ t[v/x][v1/x1] }> = <{ t[v1/x1][v/x] }>.
```

Listing 2.1: `swap_subst` lemma from [21, Norm.v]

Instead of single substitutions, Autosubst directly uses multiple substitutions in the form of **parallel substitutions** which replace all free variables in a term at the same time.
When using de Bruijn syntax, a parallel substitution is a function from $\mathbb{N}$ to terms.
Autosubst even uses a generalization called **vector substitutions** which is a vector of parallel substitutions for independently substituting variables of different sorts.

2.3 De Bruijn Algebras

The code generation of Autosubst targets de Bruijn algebras.
Completeness results about de Bruijn algebras by Schäfer et al. [23] are what initially inspired work on Autosubst.

Definition 2.2 (De Bruijn Algebra) *A de Bruijn algebra for a given language is an algebra consisting of*

- *the syntax of terms,*
- *instantiations with renamings and substitutions,*
- *and associated substitution primitives*[1]*.*

Autosubst supports two different categories of de Bruijn algebras: **unscoped** and **wellscoped**. They are discussed in Section 2.5.3.
Below, we give the names of the substitution primitives and how they are represented in different contexts. Their exact definition depends on the category of the de Bruijn algebra at hand.

Name	Mathematical Symbol	Coq Identifier	Coq Notation
zero	Z	`var_zero`	
shift	$\uparrow$	`shift`	$\uparrow$
extension	$s \cdot \sigma$	`scons s sigma`	`s .: sigma`

Table 2.1: Comparison of Substitution Primitives

Zero is the lowest de Bruijn index, **shift** increases a de Bruijn index and **extension** defines a substitution by specifying its behavior on zero and a shifted index.
Generally, the following equation holds for the primitives.

$$(s \cdot \sigma)\ Z = s$$
$$(s \cdot \sigma)\ (\uparrow m) = \sigma\ m$$

As an example, we list the de Bruijn algebra of the λ-calculus in Figure 2.1.
So the generated code of de Bruijn algebras contains inductive type definitions for the terms, recursive function definitions for instantiations, and definitions of the liftings.

Additionally, Autosubst generates code for **rewriting laws** (or rewriting lemmas) about the behavior of renamings and substitutions.

[1]Note that renamings are special substitutions that only replace de Bruijn indices with de Bruijn indices. They are first class in Autosubst 2 but we still use the naming scheme "substitution primitives" instead of "renaming and substitution primitives" in this and other cases.

There are several groups of rewriting laws that are generated. The groups are listed in Appendix A.4 containing as an example the rewriting laws for the de Bruijn algebra of the λ-calculus.
In general, there are instances of the laws for all sorts of the language. For example, for the polymorphic lambda calculus, System F (Appendix A.2), Autosubst generates instances of the rewriting lemmas for `ty` and `tm`.

The liftings ($\Uparrow^*, \Uparrow$) are used to ensure instantiations are capture-avoiding.
In de Bruijn syntax, free variables in the body of a λ-binder are increased by one compared to outside the body, and the `0` index inside the body is bound by the λ-binder.
In the definition of instantiation, σ must only affect free variables of a term.
Therefore, in the case of a λ-binder, σ is extended by var `0`, so that any `0` inside the body stays unaffected and all increased indices are passed to σ. It is also composed with a shift to increase all free variables in the result of σ, to account for the λ-binder.
Note that in this thesis the composition operator ∘ always denotes **forward composition** and binds stronger than extension.

The choice to generate code for a de Bruijn algebra with its associated rewriting lemmas enables Autosubst to automatically prove certain lemmas, which are discussed in section 2.3.1.

This is due to a metatheoretical property that Schäfer et al. [23] have shown for the de Bruijn algebra of the λ-calculus: With respect to equality, it is a sound and complete model of the **σ_{SP}-calculus**.

The σ_{SP}-calculus by Curien et al. [10] (Appendix A.3) is a calculus of explicit substitutions [1]. This is a form of λ-calculus where substitutions are part of the syntax and a reduction relation is defined to reduce substitutions.
The σ_{SP}-calculus is interesting because its reduction has the property that it is **confluent** and **terminating** (for abstract rewriting systems in general and the definitions of confluent and terminating in particular, see [7]).

Schäfer et al. [23] use the confluence and termination properties to prove equality is decidable for terms of the σ_{SP}-calculus. And since the de Bruijn algebra of the λ-calculus is a sound and complete model of the σ_{SP}-calculus, decidability of equality is transported there.
Finally, Autosubst can use the decidability of equality in the de Bruijn algebra of the λ-calculus to automatically prove equations between terms.

Note that the proofs of soundness, completeness, confluence, and termination have only been mechanized by Schäfer et al. [23] and Stark [30] for the de Bruijn algebra of the λ-calculus and the associated σ_{SP}-calculus.

$$s, t \in \mathit{tm} := \mathit{var}\ n \mid \mathit{app}\ s\ t \mid \lambda.s \qquad\qquad n \in \mathbb{N}$$

(a) The λ-calculus using de Bruijn syntax.

$$\begin{aligned}
_\langle_\rangle &: \mathit{tm} \to (\mathbb{N} \to \mathbb{N}) \to \mathit{tm} \\
(\mathit{var}\ n)\langle\xi\rangle &= \mathit{var}\ (\xi\ n) \\
(\mathit{app}\ s\ t)\langle\xi\rangle &= \mathit{app}\ s\langle\xi\rangle\ t\langle\xi\rangle \\
(\lambda.s)\langle\xi\rangle &= \lambda.s\langle\Uparrow^*\ \xi\rangle
\end{aligned}
\qquad
\begin{aligned}
\Uparrow^* &: (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\
\Uparrow^*\ \xi &= 0 \cdot \xi \circ \uparrow
\end{aligned}$$

$$\begin{aligned}
_[_] &: \mathit{tm} \to (\mathbb{N} \to \mathit{tm}) \to \mathit{tm} \\
(\mathit{var}\ n)[\sigma] &= \sigma\ n \\
(\mathit{app}\ s\ t)[\sigma] &= \mathit{app}\ s[\sigma]\ t[\sigma] \\
(\lambda.s)[\sigma] &= \lambda.s[\Uparrow\ \sigma]
\end{aligned}
\qquad
\begin{aligned}
\Uparrow &: (\mathbb{N} \to \mathit{tm}) \to (\mathbb{N} \to \mathit{tm}) \\
\Uparrow\ \sigma &= \mathit{var}\ 0 \cdot \sigma\ \circ \langle\uparrow\rangle
\end{aligned}$$

(b) Instantiations with renamings and substitutions, and associated liftings.

Figure 2.1: The de Bruijn algebra of the λ-calculus.

The conjecture is that the same holds for the de Bruijn algebras and associated σ-calculi of other languages, like System F or first-order logic.
To our knowledge, the conjecture has not been disproven so far and evidence in favor of the conjecture is given in the form of case studies (Chapter 5) that use different languages.

2.3.1 Substitution Equations

The theory of de Bruijn algebras allows Autosubst to generate Coq tactics that automatically solve **substitution equations**. These are assumption free (i.e. there is no additional information about them) equations between instantiations and they appear ubiquitously as goals when proving metatheorems about languages with binders.
The example in Figure 2.2 is from the proof that instantiation preserves reduction in the λ-calculus.

One can intuitively check that the goal holds by tracing the behavior of free variables in s.

$$0 \xrightarrow{[\mathtt{t\ .:\ id}]} \mathtt{t} \xrightarrow{[\sigma]} \mathtt{t}[\sigma] \xleftarrow{[\mathtt{t}[\sigma]\ \mathtt{.:\ id}]} 0 \xleftarrow{[\Uparrow\sigma]} 0$$

$$\mathtt{S\ n} \xrightarrow{[\mathtt{t\ .:\ id}]} \mathtt{n} \xrightarrow{[\sigma]} \sigma\ \mathtt{n} \xleftarrow{[\mathtt{t}[\sigma]\ \mathtt{.:\ id}]} (\sigma\ \mathtt{n})\langle\uparrow\rangle \xleftarrow{[\Uparrow\sigma]} \mathtt{S\ n}$$

```
Theorem ≻_inst_tm (σ : ℕ → tm) (s t : tm) :
  (s ≻ t) → (s[σ] ≻ t[σ]).
```

(a) A metatheorem that instantiation preserves reduction.

$$s[t \cdot id][\sigma] \;=\; s[\Uparrow \sigma][t[\sigma] \cdot id]$$

(b) A substitution equation goal that appears in the proof of the above theorem.

Figure 2.2

But Autosubst generates a tactic, called `asimpl`, that can automatically prove it.

The tactic works off the conjecture that for all of our generated de Bruijn algebras, the rewriting lemmas constitute a confluent and terminating rewriting system, as it does for the λ-calculus.
The algorithm implemented by the tactic does the following:

- On both sides of the equation, use the rewrite lemmas in any order (confluent) to arrive at a normal form (terminating).
- If the normal forms of both terms are equal then the goal is solved.
- If they are not equal we can be sure the goal is not provable. Because of confluence the two terms have exactly one normal form.

2.4 Traversals

We summarize traversals [13, 2], which are a specialization of recursive function definitions for syntax with binders. Autosubst uses the theory of traversals to implement generation of recursive function definitions.

A traversal is way of defining recursive functions for a sort, e.g. `tm` of the λ-calculus.

Definition 2.3 (`tm`-Traversal [13, Definition 3.1]) *A tm-traversal* $T = (\mathbb{V}, \mathbb{A}, \mathbb{L}) : \mathbb{T}^{V}_{D}$ *for terms of the λ-calculus consists of semantic counterparts to the syntactic constructors.*

$$\begin{aligned}
&\mathbb{V} : V \to D\\
&\mathbb{A} : D \to D \to D\\
&\mathbb{L} : ((\mathbb{N} \to \mathbb{N}) \to V \to D) \to D
\end{aligned}$$

where the types V, D *are referred to as* ***denotation domains*** *for de Bruijn indices and terms.*

Intuitively, a `tm`-traversal describes a recursive function on `tm` by specifying (1) what happens to the variable in the `var` case, (2) how to use the results of recur-

sive calls in the app case, (3) how to use the result of a recursive call that has some additional information in the lam case.

Traversals are useful in the context of Autosubst because they encapsulate the process of lifting arguments to recursive calls, which is done with the additional information.
In Figure 2.3 we show instantiation with substitutions and a rewriting lemma of the de Bruijn algebra of the λ-calculus. Both are defined recursively and certain arguments are lifted (denoted by $\Uparrow, \Uparrow'$) in recursive calls on the subterms of binders.

When Autosubst generates these proof terms, it uses a function that generates an inlined traversal.
Stark notes in her dissertation [30, Section 11.2.2] that it would be interesting future work to generate an explicit traversal. Then one can reuse this for other recursive function definitions.

2.5 Autosubst Compiler

We give a general description of the Autosubst compiler. Specifics can be found the respective chapters for the OCaml and MetaCoq implementations.

The Autosubst compiler consists of three stages.
First, it parses the input syntax that specifies the target language.
Then, it analyzes the language to precompute some information.
And finally, it generate the actual code.

The code generation is again divided into two parts: the first generates the de Bruijn algebra and rewriting lemmas (we say **algebra generation**) and the second generates code that is related to the `asimpl` tactic (we say **automation generation**).

2.5.1 Input Syntax

We define an input syntax for specifying languages with binders based on Autosubst 2's EHOAS[2].

We also implement some extensions to the syntax based on the extensions of EHOAS. The extensions are: variadic binders (binders that bind multiple variables), functors (to specify that arguments of a constructor are of a composed type), and constructor parameters. The exception is we do not support the modular syntax feature of EHOAS, which is out of scope for this work.
Also, even though we support above-mentioned extensions in our two programs, we generally do not discuss them in the thesis itself because they are not pertinent to

[2]Extended Higher Order Abstract Syntax based on HOAS [19]

```
Fixpoint subst_tm (σ_tm : ℕ → ty) (s : ty) {struct s} :
    ty :=
  match s with
  | var_tm s0 ⇒ σ_tm s0
  | app s0 s1 ⇒ app (subst_tm σ_tm s0) (subst_tm σ_tm s1)
  | lam s0 ⇒ lam (subst_tm (⇑ σ_tm) s0)
  end.
```

(a) Instantiation with substitution for ty of System F_{cbv}

```
Fixpoint idSubst_tm (σ_tm : ℕ → ty)
                    (Eq_tm : ∀ x, σ_tm x = var_tm x)
                    (s : ty) {struct s} :
    subst_tm σ_tm s = s :=
  match s with
  | var_tm s0 ⇒ Eq_tm s0
  | app s0 s1 ⇒
      congr_app (idSubst_tm σ_tm Eq_tm s0) (idSubst_tm σ
         _tm Eq_tm s1)
  | lam s0 ⇒
      congr_lam (idSubst_tm (⇑ σ_tm) (⇑' _ Eq_tm) s0)
  end.
```

(b) Substitution with identity lemma for ty of System F_{cbv}

Figure 2.3

```
ident          ::= <Coq identifier>
ct             ::= <Coq term>
lang           ::= decl+
decl           ::= sortDecl
                 | functorDecl
                 | constructorDecl
sortDecl       ::= ident_s ':' 'Type'
                 | ident_s '(' ident ')' ':' Type
functorDecl    ::= ident_F ':' 'Functor'
constrDecl     ::= ident ':' (arg '->')* ident_s
                 | ident params ':' (arg '->')* ident_s
params         ::= '(' param (',' param)* ')'
param          ::= ident ':' ct
arg            ::= arghead
                 | 'bind' binder (',' binder)* 'in' arghead
binder         ::= ident_s
                 | '<' ident ',' ident_s '>'
arghead        ::= ident_s
                 | ident_F ct arghead+
```

Figure 2.4: Our Custom Input Syntax. The `ident`$_s$ (`ident`$_F$) indicates that this particular `ident` must have been declared by a `sortDecl` (`functorDecl`).

Autosubst's code generation. More information about the extensions can be found in [30, Chapter 5].

The input syntax is described in Figure 2.4 and concrete examples can be found in Section 3.2.2, Section 4.1.1 and (slight modifications necessary) in Stark's dissertation [30, Section 5.2].
Note that "bind x in y" declares a binder that binds variables of sort "x" in an argument of sort "y".

Our input syntax has some improvements compared to EHOAS.

- Usability Improvement
  - The Autosubst OCaml implementation allows the user to customize the name of the variable constructor of a sort using the second `sortDecl` variant.
    Autosubst MetaCoq and Autosubst 2 auto-generate the name.
  - We allow any Coq identifier, i.e. unicode strings, for sorts, constructors,

```
Inductive ty : Type :=
  | var_ty : ℕ → ty
  | arr : ty → ty → ty
  | all : ty → ty.

Inductive tm : Type :=
  | app : tm → tm → tm
  | tapp : tm → ty → tm
  | vt : vl → tm
with vl : Type :=
  | var_vl : ℕ → vl
  | lam : ty → tm → vl
  | tlam : tm → vl.
```

Figure 2.5: Inductive types for System F_{cbv} generated in order of dependency.

etc.
Autosubst 2 only allows ASCII identifiers.

- Usability Improvement / Forced Change
  - We use an explicit `bind` construct to specify bound sorts in an `arg`. Autosubst 2's EHOAS uses arrow syntax because it is based on HOAS. We think it is an improvement because this prevents a false impression that EHOAS is a higher-order syntax, since only first-order binders are allowed anyways.
    But is also a necessary change because we were not able to implement a notation for nested arrows for the MetaCoq implementation of Autosubst (Section 4.1.1).

2.5.2 Dependency Analysis

After parsing an input language, it is analyzed to extract information for code generation. Most important is the order in which to generate the sorts, as there are dependencies between them when one sort is defined in terms of another.

For example in System F_{cbv} (Figure 2.6) the sort `tm` depends on the sort `ty`. Therefore, we must define the inductive type for `ty` before the one for `tm` in Coq. Similarly, the sorts `tm` and `vl` are defined in terms of each other, which necessitates a mutually inductive type definition in Coq.

The dependency analysis builds a dependency graph structure out of the input language for which we need the following definitions.

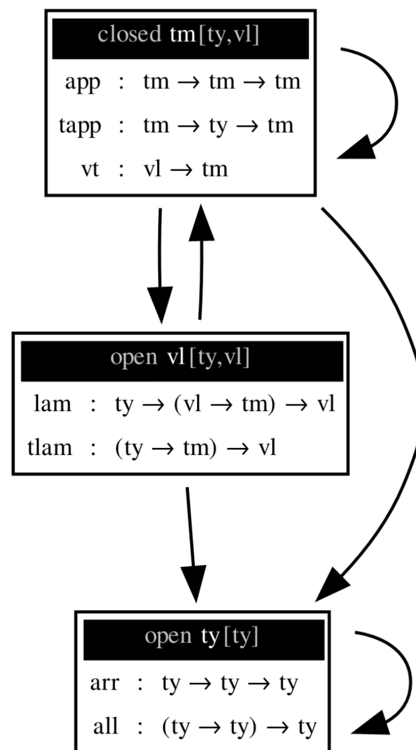
```
ty : Type
tm : Type
vl : Type

arr : ty -> ty -> ty
all : (bind ty in ty) -> ty

app  : tm -> tm -> tm
tapp : tm -> ty -> tm
vt   : vl -> tm

lam  : ty -> (bind vl in tm) -> vl
tlam : (bind ty in tm) -> vl
```

(a) Specification of System F_{cbv} in our input syntax.



(b) The dependency graph of System F_{cbv} [30, Figure 8.2].

Figure 2.6

Definition 2.4 (Occurrence [30, Page 111]) *A sort y has a **direct occurrence** in a sort x if y is an argument head – either immediately or in a nested argument head of a functor – in one of x's constructors.*
***Occurrence** is the transitive closure of direct occurrence.*

The nodes of the dependency graph are the language's sorts and there is an edge (x, y) if y occurs in x.

In this dependency graph Autosubst computes the strongly connected components. Each strongly connected component consists of sorts that have to be defined as mutually inductive types because they are defined in terms of each other.
Autosubst starts code generation at the terminal strongly connected component, i.e. the one with no outgoing edges and thus no dependencies, and then moves backwards in the graph.

Additionally, the following information is also computed from the dependency graph.

Definition 2.5 (Open Sort [30, Page 111]) *A sort x is **open** if and only if x is bound in a sort y and also occurs in y.*

Autosubst generates a variable constructor for all open sorts.

Definition 2.6 (Substitution Vector [30, Page 111]) *The **substitution vector** for a sort x consists of those sorts y that are open and occur in x.*

The substitution vector of a sort x contains those sorts whose variables can appear in terms of x. Autosubst must keep track of which substitutions it needs for each sort in the instantiation operation.

For each sort, Autosubst either generates just instantiation with renaming, just instantiation with substitution, or both. For the rules of this we refer to [30, Section 8.1].

We change dependency analysis only in one way for the MetaCoq reimplementation of Autosubst 2:

- Forced Change
  - We already compute which sorts have a renaming during this stage while Autosubst 2 computes it during code generation.
    The function that computes this in Autosubst 2 uses non-structural recursion. We want to avoid this because it would be harder to port the code to MetaCoq.

2.5.3 Output Syntax

Autosubst 2 supports code generation for two categories of de Bruijn algebras: unscoped and wellscoped (and we discuss a third one in Section 6.2).

Unscoped Syntax

Our examples of de Bruijn algebras so far fall into the category of **unscoped syntax**. In unscoped syntax, de Bruijn indices are represented by natural numbers.

```
Inductive tm : Type :=
| var_tm : ℕ → tm
| app : tm → tm → tm
| lam : tm → tm.
```

Listing 2.2: λ-calculus terms using unscoped syntax.

The substitution primitives are readily defined.

Definition 2.7 (Unscoped Syntax Primitives)

$$\begin{aligned} Z &: \mathsf{nat} \\ Z &= 0 \end{aligned}$$

$$\begin{aligned} \uparrow &: \mathsf{nat} \to \mathsf{nat} \\ \uparrow\ m &= S\ m \end{aligned}$$

$$\begin{aligned} (_ \cdot _) &: X \to (\mathsf{nat} \to X) \to \mathsf{nat} \to X \\ (x \cdot f)\ 0 &= x \\ (x \cdot f)\ (S\ m) &= f\ m \end{aligned}$$

Wellscoped Syntax

In **wellscoped syntax**, de Bruijn indices are elements of the finite type `fin n` where `n` is a natural number. The finite type `fin n` denotes the numbers from zero to `n` and is recusively defined on `n`.

```
Fixpoint fin (n : ℕ) : Type :=
  match n with
  | 0 ⇒ ⊥
  | S m ⇒ option (fin m)
  end.
```

There are other definitions of finitye types [17] but we use this one for its reduction behavior.

Terms are also parameterized by a natural number `n` so that the variable constructor can take an element of type `fin n`.
Therefore, a term of type `tm n` is constrained to contain free variables up to `n`.

```
Inductive tm (n: ℕ) : Type :=
| var_tm : fin n → tm n
| app : tm n → tm n → tm n
| lam : tm (S n) → tm n.
```

Listing 2.3: λ-calculus terms using wellscoped syntax.

The crucial part is that we increase the bound in the body of the `lam` constructor because it can contain one more free variable.
Wellscoped syntax can be useful in a statement on a term being closed, e.g. a term of type `tm 0` can contain no free variables. Also, even though they increase the amount of bookkeeping one has to do to manage the bounds, this can be used as a safety net during programming, because it prevents incompatible terms from being combined.

The substitution primitives are defined as follows, as the `fin n` type is an n-fold nesting of option types.

Definition 2.8 (Wellscoped Syntax Primitives)

$$Z_{\mathbb{I}} : \mathsf{fin}\ (S\ n)$$
$$Z_{\mathbb{I}} = \mathsf{None}$$

$$\uparrow_{\mathbb{I}} : \mathsf{fin}\ n \to \mathsf{fin}\ (S\ n)$$
$$\uparrow_{\mathbb{I}}\ m = \mathsf{Some}\ m$$

$$\cdot_{\mathbb{I}} : X \to (\mathsf{fin}\ n \to X) \to \mathsf{fin}\ (S\ n) \to X$$
$$(x \cdot_{\mathbb{I}} f)\ \mathsf{None} = x$$
$$(x \cdot_{\mathbb{I}} f)\ (\mathsf{Some}\ m) = f\ m$$

Chapter 3

Autosubst OCaml

One part of the thesis is the continuation of the authors port of Autosubst 2 to the OCaml language. We call this reimplementation **Autosubst OCaml**[1]. Analogous to Autosubst 2, Autosubst OCaml parses a user-supplied language specification and generates Coq code for proving substitution equations of that language, in the form of Coq source files.

3.1 Motivation

Autosubst 2 is written in Haskell and is distributed in source-form. This makes it necessary to maintain a Haskell environment if one wants to use the program. In order to represent the generated code, custom ASTs for Gallina terms, tactics and Coq vernacular commands are defined. Additionally, for each of these ASTs, pretty-printers to generate the concrete syntax had to be written from scratch.
With a reimplementation in OCaml, the language Coq is implemented in, users can reuse their existing OCaml environment to run the program. Also, we can reuse the ASTs and pretty-printers from the Coq implementation in the definition of our program.

The ad-hoc nature of the abstract syntax and pretty-printers in Autosubst 2 is a disadvantage because for one it replicates functionality but also there are differences to the existing implementations from Coq.
For example, the pretty-printers produce concrete syntax that uses more parentheses than necessary, has irregular spacing and is indented according to custom rules.

```
Definition upId_ty_ty  (σ : ( fin ) → ty ) (Eq : ∀ x, σ x = (
    var_ty ) x) : ∀ x, (up_ty_ty σ) x = (var_ty ) x :=
  fun n ⇒ match n with
  | S fin_n ⇒ (ap) (ren_ty (shift)) (Eq fin_n)
```

[1]The implementation can be found at https://github.com/uds-psl/autosubst-ocaml.
In order run the program, consult the README.org

```
  | 0  ⇒ eq_refl
end.
```

Listing 3.1: Excerpt of code generated by Autosubst 2.

This makes it harder to read than code written by a human or printed by the pretty-printers form the Coq implementation.
Also, we found one instance where non-parseable code is generated. This is possible due to a combination of an abstract syntax term which should not be possible to construct and the pretty-printer naively printing the malformed term. An explanation can be found in Section 6.3.

As for the abstract syntax, Autosubst 2's definition of the abstract syntax of Coq has a node[2] that contains function arguments which are lifted in recursive calls. This bookkeeping behaviour is specific to syntax traversals [13] and does not belong in an abstract syntax tree. While porting the code to use the AST from the Coq implementation, this became apparent as there was no analog to map this node to.

Therefore, our plan for a reimplementation in OCaml is to reuse the abstract syntax and pretty-printers from the Coq implementation. This means we can only construct sensible abstract syntax terms, the generated code has a canonical look, and it is more likely[3] that it can be parsed again (but with the current design of the program we have no guarantees that the code also typechecks).

3.2 How it Works

In the following we discuss the implementation of the Autosubst compiler in OCaml. First, we analyze how to interact with the OCaml implementation of Coq. Then, we implement a parser for the language specification and the dependency analysis. Finally, we discuss how the abstract syntax terms themselves are generated.

We do not discuss how exactly the types and proof terms for most of the generated lemmas look, as it is no different (except for formatting due to the pretty-printers) from the way Stark describes it in her dissertation [30, Section 8.2].

3.2.1 Usage of Coq as a Library

To use the exposed facilities of the Coq implementation we need to link the implementation's modules to our own OCaml code. This allows us to use, among others, the Coq parser, several abstract syntax trees, and pretty printers for terms.

[2] called `TermSubst`

[3] We found and fixed one instance where the pretty-printer generated wrong concrete syntax discussed in Technical Remark 3.5.

Technical Remark 3.1

One roadblock was that when linking certain modules of the Coq implementation one needs to add the `-linkall` flag[a] during compilation, which causes all modules of the Coq library to be linked in the resulting executable.
This is because these modules depend on other modules to set up state. If `-linkall` is omitted, the state is not initialized correctly and the program will crash at runtime with an unhelpful error message.

[a] `https://github.com/coq/coq/issues/9547`

Gallina Term AST

The Coq implementation exposes multiple ASTs for Gallina terms, which we use to construct types and proof terms, and one AST for vernacular commands, which we use to construct commands like `Inductive`s and `Definition`s that use the aforementioned types and proof terms.

The three term ASTs are

- `constr_expr`,
- `glob_constr`,
- and `econstr`.

When Coq parses a term, it is first parsed into the `constr_expr` AST and then translated down into the other ASTs. So `constr_expr` is closely related to concrete syntax, `glob_constr` is an intermediate form, and `econstr` is used by the kernel for typechecking,

We opt to use the `constr_expr` AST to construct Gallina terms because it is simpler to construct and easier to use with other parts of the code than the other two ASTs due to the following reasons.

- Other functions that we want to use, like the pretty-printers and constructors for vernacular commands, use `constr_expr`.
- It has variadic lambdas and applications whereas the other two have curried forms, meaning the application node applies the function only to a single argument.
- It uses named variables instead of de Bruijn indices for locally bound variables.
- It uses named variables instead of references into the environment for globally bound variables.

We define smart constructors to take care of boilerplate in the `constr_expr` AST. These constitute the DSL[4] in which we construct all Gallina terms[5].

Technical Remark 3.2

To be more precise about the last point, a globally defined constant (e.g. defined by `Definition` or `Parameter` etc.) consists of a kernel name and a universe instance. The kernel name is the fully qualified name, e.g. `"Coq.Arith.PeanoNat.Nat.add"` for the addition function, and the universe instance specifies the universe levels of the constant's arguments.
So to construct a term for `add x y` in the `econstr` AST, we would have to know the fully qualified name of `add`, set fitting universe levels and calculate the de Bruijn indices for `x` and `y`, assuming they are locally bound.

Especially the last two points show that it is simpler to construct terms of `constr_expr` than terms of the other ASTs.
For one, while it is good that proofs with de Bruijn indices handle variables in a straightforward manner, it is arguably harder for humans to read. In the same manner, programming with de Bruijn indices is harder and more error-prone so we prefer an AST with named variables.
And second, the other ASTs are **globalized**, meaning all globally bound variables (e.g. `nat` or one of our generated `Lemma`s) are represented by references into the environment. To fill that environment we would have to start a Coq instance, parse the standard library, and also put in any of our defined constants that we want to reference in other terms.

We can think of only one advantage of using one of the globalized ASTs. Assuming Autosubst might generate a term that does not typecheck, a user notices this when they try to compile the generated source code.
If we used a globalized AST, then Autosubst can use the kernel's typechecker during code generation to make sure all constructed terms typecheck. Then the error would already surface during execution of Autosubst.
Note that it is only a matter of convenience for when the error occurs.
Since our generated code does not rely on Axioms (except **Functional Extensionality** if enabled) it is safe to use, i.e. does not make the development inconsistent.

In a discussion in the Coq chatroom Zulip[6] we noted that it is generally possible to start a Coq instance – either directly or through a tool like SerAPI[7] – and handle

[4]Domain Specific Language
[5]defined in `lib/gallinaGen.mli`
[6]`https://coq.zulipchat.com/#narrow/stream/237656-Coq-devs.20.26.20plugin.20devs/topic/Constructing.20Notations.20in.20OCaml/near/227594459`
[7]`https://github.com/ejgallego/coq-serapi/`

the environment in our program.
But weighing user-convenience versus developer-convenience we decided to use with the `constr_expr` AST to keep our program simple. From experience with Autosubst 2, we know that the program does not construct untypable terms.

One disadvantage of using `constr_expr` is that we need to explicitly construct notation terms because a notation is a separate node in the `constr_expr` AST.
We must use notations because otherwise an equality is printed as "eq a b", which was the state of code generation after the ACP project. But this goes against our target of improving the look of the generated code.
We use two notations: `"x = y"` for equality and `"A -> B"` for non-dependent function types. We register both in the Coq environment in the beginning of the program and define smart constructors to generate the notation nodes.

Technical Remark 3.3

Notations were perhaps the most challenging part of using the `constr_expr` AST.
We did not find any documentation on the `CNotation` constructor and had to reverse engineer the arguments by dropping[a] from a running Coq instance and reading out the notation storage.
It turns out the `notation_key` argument must contain the notation with underscores in place of variable names. So for the equality notation `"x = y"`, the key is `"_ = _"`.

[a] `https://github.com/coq/coq/blob/master/dev/doc/debugging.md`

Vernacular Command AST

Vernacular commands are represented by `vernac_expr` terms which often contain `constr_expr` terms, e.g. a `Definition` has a type and a body which are `constr_exprs`.
For the code generation, our program must construct and print the following commands:

```
Inductive, Definition, Lemma, Fixpoint, Proof, Qed,
Notation, Class, Instance, Ltac, Tactic Notation
```

Most of these are straightforward to construct and use (except `Proof`, `Ltac` and `Tactic Notation`) and we again define smart constructors to take care of boilerplate[8].

We use pretty-printers from the Coq implementation to convert the abstract `vernac_expr`

[8] defined in `lib/vernacGen.mli`

terms to concrete syntax. We introduce one abstraction, `vernac_unit`, to group related `vernac_expr`s and implement the `Ltac` and `Tactic Notation` commands[9].

```
(* TacGen.t is a type alias for Coq's tactic AST *)
type vernac_unit =
  | Vernac of vernac_control list
  | TacticLtac of string * TacGen.t
  | TacticNotation of string list * TacGen.t
```

The grouping is necessary for better pretty-printing, to insert whitespace between different `vernac_unit`s without introducing whitespace within a `vernac_unit` of related commands. This is handy in an interactive proof of a lemma (which is composed of `Lemma`, `Proof` and `Qed` commands).
The necessity of the `TacticLtac` and `TacticNotaton` constructors is discussed in the next section.

Technical Remark 3.4

One irregularity when constructing a `Proof` command is that there is the `VernacExactProof` node to give a proof term directly with the command. It is pretty-printed as

```
Proof (foo).
```

But proof-general[a] does not work[b] with this syntax and there is discussion to deprecate it entirely so we decided to use a custom pretty-printer for this case which instead prints

```
Proof.
  exact (foo).
```

[a] A popular emacs mode for poof assistants `https://proofgeneral.github.io/`
[b] `https://github.com/ProofGeneral/PG/issues/498#issuecomment-638479230`

Technical Remark 3.5

While implementing code generation we found that the pretty printer generated wrong code for the `Existing Instance` command.
The command is used to register an existing constant in the environment as a typeclass instance [28]. It can take several arguments separated by spaces, but the pretty printer separated them with commas.

[9] Note that `vernac_control` is a wrapper around `vernac_expr` to annotate vernacular commands with attributes like `#[ local ]`.

It was a simple fix and a pull request has been merged[a].

[a] https://github.com/coq/coq/pull/15040

Tactic AST

Tactics also have multiple ASTs and we opt again to use the one closest to the concrete syntax, `raw_tactic_expr`. Combinators like `try`, `repeat`, `progress` and basic tactics like `rewrite`, `cbn`, `unfold` are part of this AST. We again define smart constructors to take care of boilerplate and to act as a DSL to construct tactics[10].

```
let rewrites = List.map (fun t →
                            try_ (rewrite_ t))
                         ["lemma0"; "lemma1"] in
let unfold = calltac_ "auto_unfold" in
let tac = then_ (unfold :: rewrites) in
TacticLtac ("mytactic", tac)

(* Ltac mytactic := auto_unfold; try rewrite lemma0; try
   rewrite lemma1 *)
```

Listing 3.2: Using the tactic DSL to construct an `Ltac` definition.

Other tactics like `setoid_rewrite` are defined using Coq's syntax extension mechanism so they are not a native part of the AST and we cannot construct them directly. We can work around this fact because `vernac_expr` allows us to reference a tactic by a string.
Semantically, this is meant for tactics defined by an `Ltac` command, but it works for `setoid_rewrite` because the concrete syntax is the same.

Technical Remark 3.6

The canonical way to construct a term representing a `setoid_rewrite` is to define the syntax extension, put it in the environment and reference it with the `TacML` AST node.
But to avoid manipulation of the environment, we instead use the `TacCall` node, for calling user-defined tactics, which results in the same concrete syntax `"setoid_rewrite foo"`.
The only problem is that we cannot give a rewrite orientation this way because ← is not a valid identifier in an argument position of `TacCall`. We define a left-rewrite version of `setoid_rewrite` and reference this instead.

[10] defined in `lib/automationGen.mli`

```
Ltac setoid_rewrite_left t := setoid_rewrite <- t.
```

Due to the combinatory explosion if there are many flags this is not suitable in general, but in our case this was the only special case.

This AST only allows us to construct the tactic terms.
Commands like `Ltac` to bind a tactic term to a name are also defined using Coq's syntax extension mechanism so there is no easy way to construct and print it.
Therefore, we define our own trivial printer for this which prints `"Ltac␣some_name ␣:="` and delegates the right-hand side to Coq's pretty-printer for tactic terms.
The same holds for the `Tactic Notation` command.

3.2.2 Parsing User Input

In Autosubst OCaml the user writes their language specification in a text file which is passed to the program.
The language specification is written in our custom input syntax (Section 2.5.1) for which we implemented a parser using the `angstrom`[11] parser combinator library.
We also evaluated the parser combinator libraries `mparser` and `planck` but decided for `angstrom` because it is the most modern and fully featured.

`planck` seems abandoned as the website[12] is not reachable and it was last published in 2016 on opam[13].

`mparser` has the nice feature of reporting line numbers in error messages. But since out language specifications are line based, with `angstrom` we just report the next unparsed line if there is any parsing error.

Technical Remark 3.7

The backtracking behavior of `angstrom` complicates this but as long as we commit[a] the parser after each successfully parsed line, error reporting works well this way.

[a] In parser combinator vernacular: To prevent backtracking beyond a certain point.

Also, while `mparser` has a monadic interface, it does not support the modern let operator syntax that `angstrom` and other libraries use to unify the handling of monads in OCaml (see Section 3.3 for further explanation of let operator syntax).

There are two deviations in the variant of our input syntax for Autosubst OCaml.

[11] `https://github.com/inhabitedtype/angstrom/`
[12] `https://bitbucket.org/camlspotter/planck`
[13] An OCaml package manager

- For functor arguments in constructors we follow the same pattern as in Autosubst 2 where the functor and its constant arguments are enclosed in quotation marks (Listing 3.2.2).
  In Autosubst 2 this is done so that the pretty printer can simply copy the string to the generated code.
  In Autosubst OCaml it is done because initially it was a time-constrained project of ACP and the aim was to keep the input syntax compatible. But this can be removed in the future.
- We allow line comments beginning at `--`, like Autosubst 2 does.

Next we show two examples of the input syntax.

The first is for System F_{cbv} where all the common features of the syntax are used. First several types are defined and then their constructors (although in general the definitions can be in any order).

```
-- the types
ty : Type
tm : Type
vl : Type

-- the constructors for ty
arr : ty -> ty -> ty
all : (bind ty in ty) -> ty

-- the constructors for tm
app  : tm -> tm -> tm
tapp : tm -> ty -> tm
vt   : vl -> tm

-- the constructors for vl
lam  : ty -> (bind vl in tm) -> vl
tlam : (bind ty in tm) -> vl
```

Listing 3.3: Specification of System F_{cbv} in our input syntax.

The second is for a variadic version of the simply typed λ-calculus.
An application can take a variable amount of operands by using the `list` functor.
And a lambda abstraction is parameterized over the number of bound variables and uses the variadic syntax for the binders.

```
tm: Type

list : Functor

app : tm -> "list" (tm) -> tm
```

```
lam (p: nat) : (bind <p, tm> in tm) -> tm
```

Listing 3.4: Specification of variadic simply typed λ-calculus in our input syntax.

Improvements over the Autosubst 2 parser are the ones already mentioned in Sectio 2.5.1, and also:

- Usability Improvement
  - We support the same identifiers that Coq allows for sorts and constructors, i.e. identifiers containing unicode, by reusing functions from the Coq parser.
    Autosubst 2 only allows ASCII identifiers.
  - We do a sanity check that identifiers are not declared twice.

The dependency analysis of the parsed specification is implemented with the `ocamlgraph` library[14,15] and works analogously to Autosubst 2.

3.2.3 Code Generation

The next step after the dependency analysis is the algebra generation and automation generation.
In this section we discuss how Autosubst OCaml generates the abstract syntax of types, proof terms, vernacular commands, and tactics, as well as differences in behavior to Autosubst 2.

Most of the generated types and proof terms are the same as in Autosubst 2. For these we refer to [30, Section 8.2].

For each non-variable constructor, Autosubst generates a congruence lemma stating that if we have equalities for all subterms, then the constructors applied to the respective subterms are also equal.
These lemmas are used in the generation of recursive functions to combine the results of recursive calls on subterms. Below is the congruence lemma for System F_{cbv}'s function type constructor.

```
Lemma congr_arr {s0 : ty} {s1 : ty} {t0 : ty} {t1 : ty}
  (H0 : s0 = t0) (H1 : s1 = t1) :
    arr s0 s1 = arr t0 t1.
```

Listing 3.5: Type of `congr_arr` from System F_{cbv}.

[14] `https://github.com/backtracking/ocamlgraph/`

[15] Incidentally, there was a bug in the computation of strongly connected components, which would have given us unexpected problems during implementation of the dependency analysis (`https://github.com/backtracking/ocamlgraph/issues/85`). A fix was just released a couple of weeks before we started the project. But the bug had been open for 3 years and the fix had been implemented but unpublished for 1.5 years, which reflects somewhat poorly on the OCaml ecosystem.

Autosubst 2 proves these lemmas by using the `congruence` tactic.
In Autosubst OCaml we explicitly generate the proof terms to become familiar with the process of code generation.
Also, the tactic language of Coq could change so using proof terms is more stable.
The proof terms use chains of `eq_trans` to swap out one s_i for t_i at a time and `eq_refl` as the base case.

```
Proof.
exact (eq_trans
         (eq_trans eq_refl
                   (f_equal (fun x ⇒ arr x s1) H0))
         (f_equal (fun x ⇒ arr t0 x) H1)).
Qed.
```

Listing 3.6: Proof of `congr_arr`

For some lemmas, including the compositionality lemmas, Autosubst 2 generates one version with extensional equality and one version with equality using the functional extensionality axiom. We generate a third version that abstracts the extensional equality[16] which our version of the `asimpl` tactic uses. This is further discussed in Section 6.1.

For all `Fixpoint`s we annotate the structurally decreasing argument. Without the annotation, Coq's inference causes slowdowns with some pathologically large languages (ca. 20 sorts). For a `Fixpoint` with n bodies each with m arguments, the inference tries all $n * m$ combinations.

Another way the output differs from Autosubst 2 is the way Autosubst OCaml organizes the generated code.
To avoid naming conflicts and problems with keeping multiple files in sync, Autosubst OCaml generates one output file. In the file, code is separated into several modules:

- `Core`, for the basic code of the de Bruijn algebra,
- `Fext`, for code involving the functional extensionality axiom,
- `Allfv`, for code belonging to the allfv extension discussed in section **??**,
- `Extra`, for any extra code,
- and `Interface`, to define the interface of the generated file.

The `Core` module contains the biggest part of the generated code: the de Bruijn algebra, the rewriting lemmas and tactics like `asimpl`.

[16] based on an idea by Yannick Forster

The `Fext` module contains all lemmas that involve the functional extensionality axiom and the `asimpl_fext` tactic that uses them. By default the module and thereby any use of axioms is excluded from code generation. It will only be generated if (1) the user passes the `-fext` command line flag or (2) when the given language uses the codomain functor because it requires functional extensionality[17]

The `Allfv` module contains all the lemmas about evaluating predicates on all free variables of a term. By default it is excluded from code generation but can be turned on if the user passes the `-allfv` command line flag.

The `Extra` module contains all code that should be run after everything else, like opaqueness hints and implicit argument declarations.

The `Interface` module exports all other generated modules in order to make them available to the user in one batch.

The module organization is implemented in the `AutosubstModules` module that uses an association list of tags to `vernac_units`. A code generation function can simply tag a piece of generated code to put it into a module. Also, `AutosubstModules` implements the monoid signature so that we can aggregate generated code from multiple sources.

```
module AutosubstModules = sig
  type module_tag = Core | Fext | Allfv | Extra
  type t = (module_tag * vernac_unit list) list

  val add_units : module_tag → vernac_unit list → t
  val empty : t
  val append : t → t → t
  ...
end
```

Listing 3.7: Excerpt of `AutosubstModules`.

This makes it straightforward to add further extensions into their own modules.
The tags also make it easy to filter out a module to a disable a feature.
Additionally, it simplifies the existing code generation because it is more encapsulated. Each code generation function can emit definitions that are put into some modules and functions higher in the call-chain just aggregate the results from below because it's a monoid.

[17]The codomain functor abstracts a function type: `cod A B = A -> B`.
Autosubst needs certain composition lemmas for functors and the lemmas for the codomain functor require functional extensionality.

Even though OCaml has support for unrestricted side-effects we choose to implement Autosubst OCaml using monads and with as little mutable state as possible, in order make reasoning about the program easier. For general information about monads and the monad variants that we use, we refer to [35]. We build a monad transformer stack [18] out of the `State` monad, to control side-effects, `Error` monad, for error handling, and `Reader` monad for dependency injection.
With the `Reader` monad, functions deep in the call chain still have access to the dependency graph.
This monad transformer stack is called `RSEM`.

One major difference to Autosubst 2 is our use of the `State` monad to record information when generating the rewriting system for the de Bruijn algebra.
To generate the automation code we need certain information that is already available during algebra generation. For example to generate tactics like "`setoid_rewrite idSubst_tm`", information about the lemma name is needed.
We use the `State` monad to store the relevant information in the `state` and then use it later when generating the automation. The design works, however it proved to be problematic due to a couple of reasons.

- The purpose of the the `State` monad is to emulate mutable state. But this system is not inherently mutable. This creates a discrepancy because we just use the `state` as an extra function output.
- The `State` monad makes it harder to test our automation generation functions because they are coupled to the code generation functions.
- A related problem inherent when using the `State` monad is that order of execution matters.
  When calling two functions that both interact with the `state`, the order of calling them matters. This is because the second one will be implicitly passed the result `state` of the first.
  Therefore, if we change the code generation we can also affect the automation generation which lead to at least one bug[18].
- Finally, although the algebra generation and the automation generation share some information, there are some differences.
  As a result, in some places we have to change the algebra generation functions to put the proper information into the `state` so that the automation generation functions can use it. This shows that our solution does not properly fit the problem.

However, we decided to keep this design for now because it works and the thesis has a fixed timeframe.

[18] Which was caught in our CI and is fixed.

In Autosubst MetaCoq we derive the information for the automation generation directly from the dependency graph and it is left as future maintenance work to also implement this in Autosubst OCaml.

3.3 Using Monads in OCaml

This section focuses on OCaml's support for monads in general.
Unfortunately, the OCaml standard library lacks an implementation of monads. We use the `monadic`[19] library because it is easier to write monad transformer stacks than with the other monad library we evaluated, called `monads`[20].

At the time we started the Autosubst OCaml implementation, the `monadic` library was very new and we found one typo which resulted in a bug in the monadic filter function. We fixed it and opened a pull request[21] but it seems the author has since stopped maintaining the library.

Technical Remark 3.8

The problem with the `monads` package is that their transformers do not compose well.
Monad transformers like ReaderT are implemented using a functor `Reader.Make(T: T)(M: Monad)`.
Module `T` contains the type that Reader is parameterized over and module `M` is the inner monad of the transformer.
But the resulting module of the `Make` functor does not conform to the `Monad` signature again because it is missing a type definition. Therefore, the transformers do not compose well and intuitive code like the following does not work because `M1` is not a `Monad`.

```
module M1 = Reader.Make(String)(Identity)
module M2 = State.Make(Int)(M1) (* ↯ *)
```

The workaround is to include the `Make` functors output in another module that defines the type, for example by using the type-deriving functor `T1`.

```
module M1 : Monad = struct
    include Reader.T1(String)(Identity)
    (* or defining the type manually *)
    (* type 'a t = string → 'a *)
    include Reader.Make(String)(Identity)
end
```

[19] https://github.com/Denommus/monadic/
[20] https://opam.ocaml.org/packages/monads/
[21] https://github.com/Denommus/monadic/pull/2

```
module M2 = State.Make(Int)(M1)
```

This is rather uncomfortable so we raised an issue[a] and discussed a way to implement better `Make` functors that build a proper `Monads`. We plan to implement this after submission of this thesis since it is not important for our current implementation, although we might switch to this library in the future if the `monadic` library proves to be abandoned.

[a] https://github.com/BinaryAnalysisPlatform/bap/issues/1354

Both `monadic` and the `angstrom` parser combinator library implement a recent OCaml syntax feature to make working with monads easier.
OCaml has support for defining custom let operator syntax[22] since version 4.08. This can be used to define natural syntax sugar for monadic bind in OCaml (see Figure 3.1), analogous to Haskell's do-notation, which is very comfortable when programming with monads.

```
let* a = monadic_value in
let* b = monadic_func a in
return b
```

```
do
  a <- monadic_value
  b <- monadic_func a
  return b
```

Figure 3.1: Monadic bind syntax sugar using let operator syntax and do-notation.

One problem that is inherent when programming with monads is that they are very pervasive. For any monadic function, all calling functions must either be monadic as well, or have some method of evaluating a monadic value. This "infecting" behavior can lead to some tedious refactoring. If we change some function to be monadic, this change propagates recursively through its call-sites until the monadic value is evaluated. In Autosubst OCaml, evaluation of the monadic value is done at the entrypoint of the code generation.

After implementing two large programs using monads, we now believe it would be worthwhile to make almost all functions monadic by default. This approach would preempt this kind of refactoring.

[22] https://ocaml.org/releases/4.08/htmlman/manual046.html

Chapter 4

Autosubst MetaCoq

As a continuation of our effort to reimplement Autosubst 2 in OCaml, we also explore how to reimplement it in the MetaCoq framework [29]. This implementation is then called **Autosubst MetaCoq**[1].

But unfortunately we were not able to bring this implementation to a level with Autosubst 2 or Autosubst OCaml. We discuss missing features in Section 4.2

4.1 How it Works

A user interacts with Autosubst OCaml on the command line and receives source code files they can use in their development, similar to Autosubst 2.
But with an Autosubst implementation in MetaCoq, a user can interact with it from within a running Coq instance. So this implementation aims to make the interaction more seamless and easy.
Apart from the usability standpoint, Autosubst MetaCoq is part of a line of work exploring the metaprogramming capabilities of the MetaCoq framework [33, 8, 3].

MetaCoq provides an AST for constructing reified Gallina terms. The abstract syntax terms can be unquoted into Gallina terms using the `TemplateMonad` of MetaCoq. The `TemplateMonad` can also be used to create new inductive types and lemmas in the Coq environment.
Both is enabled by side-effects produced by evaluating a `TemplateMonad` expression.

Therefore, we can implement the code generation of Autosubst 2 by

- computing abstract syntax terms of types, proof terms and inductive types, analogously to Autosubst 2 and Autosubst OCaml, using the provided AST by MetaCoq,

[1] The implementation can be found at `https://github.com/uds-psl/autosubst-metacoq`. In order run the program, consult the README.org

- and constructing a `TemplateMonad` expression that unquotes the terms and puts the inductive types and lemmas into the environment.

The main entrypoint of Autosubst MetaCoq is the function `runAutosubst` which receives the language specification and the output syntax category, and returns a `TemplateMonad` expression as described above.
We define a notation for it, so that a user can execute the command `MetaCoq Run Autosubst Unscoped for lang.` for a language specification `lang`.

The architecture of the program is then mostly the same as Autosubst OCaml but we replace the pretty-printers that give us the string representation of an abstract syntax term with `TemplateMonad` constructors that unquote a term or create a new inductive type or definition.

Since the program is implemented in Coq, it can use dependent types and certified programming.
Generally, we think certified programming is a great idea, but you need enough time for it. We experienced productivity slowdowns while developing Autosubst MetaCoq using certified functions so we decided to go for a more traditional functional programming approach.
However, there are some parts of the program that could benefit from a certified programming style, like the dependency analysis and graph algorithms. We leave this as future work.
Being implemented in MetaCoq, it might also be possible to verify the whole program. But with about 3500 lines of code we expect it to be a lot of effort that is out of scope for this work.

4.1.1 Parsing User Input

Because the user interacts with Autosubst MetaCoq in a running Coq instance we cannot receive the language specification in a text file like in previous implementations. There are several possible ways how the user could transmit the language specification to our program.

Inductive Types

One possibility is that a user implements the inductive types for their language directly and we analyze them using MetaCoq.

```
Definition bind {X Y:Type} (a: X) (b: Y) := b.

Inductive tm : Type :=
| app : tm -> tm -> tm
| lam : (bind tm tm) -> tm.
```

But there are multiple problems with this approach.

- Do we reuse the inductive type the user writes?
  If yes, the user would have to additionally specify a variable constructor and scope parameters depending on the output syntax.
  If no, there would be two similar copies of the inductive types.
- Coq supports syntax for inductive types like `Cumulative`, `Polymorphic`, and indices which we have to take care to not accept.

Custom Entry Notations

Another possibility is embedding a DSL for our input syntax in Coq using custom entry notations [32]. This approach is based on Pit-Claudel et al. [22].
With custom entry notations we replace the Coq parser inside of designated delimiters (for example "`{{`" and "`}}`") so that we can define our own interpretation for tokens.
In our case, the notations desugar to some inductive types modelling the input syntax, analogous to those after the parsing step in Autosubst OCaml.

To enable a nice input syntax we redefine parsing for '`:`' and '`->`' to not be a typecast or function arrow, respectively. And we also allow any Coq identifier as constructor and sort names[2,3]

```
Definition utlc : autosubstLanguage :=
  {| al_sorts := <{ tm : Type }>;
     al_ctors := {{ app : tm → tm → tm;
                    lam : (bind tm in tm) → tm }} |}.
```

Listing 4.1: Specification of sorts and constructors of the λ-calculus.

This allows a natural input syntax that looks almost the same as in the other implementations.

Dependency analysis is based on the same graph analysis as with Autosubst 2. Except we precompute the sorts that have renamings because the function that computes this in Autosubst 2 does not use structural recursion. To define a non-structurally recursive function in Coq you need to prove termination which makes it harder to port this function from OCaml to Coq.

For the graph analysis we write our own graph implementation based on finite maps (that we also implement ourselves) with a couple of related algorithms.
The existing implementations in the Coq ecosystem for either one were not suitable

[2] The fact that we can write the identifier app instead of the string `"app"` is possible due to a workaround described in [22].

[3] Note that "`{|`" is Coq's syntax for a record and not one of our custom entry notations.

for us[4].
The one that comes closest is the `FinMap` implementation from the Coq standard library. But it also has some problems with computation because it carries around a proof that the map is ordered.

4.1.2 Programming in MetaCoq

Because Gallina is a functional language whose syntax is inspired by ML we can use almost the same architecture as with Autosubst OCaml. Some concessions have to be made because the MetaCoq AST works differently than the `constr_expr` AST we use in Autosubst OCaml. Instead, it is more like the `econstr` AST.
This means:

- References to globally bound variables are not strings but pointers into the environment. A reference always uses a fully qualified name. We can use MetaCoq functionality to compute the fully qualified names.

> **Technical Remark 4.1**
>
> Anything that is already in the environment before we run the program can be quoted to retrieve a reference.
>
> For any named term that our program constructs and that we want to reference in some other term, we can compute the fully qualified name.
> The `TemplateMonad` constructor `tmCurrentModPath` returns the fully qualified name of the current module. Any definition or inductive type that we create using the `TemplateMonad` will be put into the current module.
> Therefore, we compute fully qualified names by prepending the module name.

- Locally bound variables are not strings but de Bruijn indices. This has the downside that it is harder to program in. How we deal with this is explained below.

- With the MetaCoq AST it's not possible to write a function with implicit ar-

[4]There are several implementations of finite maps and graphs but they are for proving instead of computing. We looked at the below implementations and even those that were able to compute did not have satisfactory performance.
https://github.com/math-comp/finmap
https://github.com/coq-community/coq-ext-lib/blob/master/theories/Structures/Maps.v
https://gitlab.mpi-sws.org/iris/stdpp/-/blob/master/theories/fin_maps.v
https://github.com/coq-contribs/graph-basics
https://github.com/coq-community/graph-theory

guments. We also explain below how to work around this.

Implicit Arguments

Because the MetaCoq AST does not contain information about which arguments of a function are implicit, we cannot generate functions with implicit arguments. If one writes a function application in Coq and wants to treat certain arguments as implicit, one can use underscores in their place like `(f _ a _ b)`.
In the MetaCoq AST, we can analogously construct function applications that explicitly pass `tHole` nodes in place of arguments that should be implicit like `(tApp f_q [tHole; a_q; tHole; b_q])`[5]

TECHNICAL REMARK 4.2

Underscores are concrete syntax for existential variables (**evars**[a]), which are typed placeholders for some other term. The MetaCoq AST has a node for evars, called **tHole**, so when we generate a function application we can put `tHole`s in place of implicit arguments which turn into evars after unquoting and are then inferred by Coq.
This is the same way that implicit arguments work in Coq except you don't have to explicitly pass a placeholder in Coq.
`tHole`s in an abstract term necessitate the use of `tmUnquoteTyped` to unquote MetaCoq AST terms. There is also the `tmUnquote` command which infers the return type but it fails if the given term contains holes.
As far as we know this is undocumented in MetaCoq and we only found out in private communication with Joomy Korkut and Kathrin Stark[b].

[a] `https://coq.github.io/doc/v8.14/refman/language/extensions/evars.html`
[b] We opened an issue for future reference.
`https://github.com/MetaCoq/metacoq/issues/545`

So to keep track of and correctly pass implicit arguments, for each function that we generate we remember a list of booleans that indicate implicit argument. Whenever we encounter an application we look up if any implicit arguments are needed, and if so, intersperse the `tHole`s.

[5] "_q" denotes a quoted term.

De Bruijn Indices

The biggest difference to `constr_expr` is that the MetaCoq AST uses de Bruijn indices for local variables. While de Bruijn indices work well in proofs, programming with them is complicated and error prone. When generating expressions with de Bruijn indices, one way of dealing with them is to use counters to keep track of under how many binders a given term is. This results in code like the following, which is both hard to write and understand, as Ullrich also notes [33, Section 7.4.2].

```
let fpos := 1+allIndCount in
let Hstart := fpos in
let Hpos := Hstart+(ctorCount -i) in
let ppos := 1+Hstart+ctorCount in
let paramOffset := 1+ppos in
let recPos := paramOffset + trueParamCount in
...
```

Listing 4.2: Code excerpt from Ullrich's Bachelor Thesis

Our solution is to implement a custom AST, **nterm**, that is based on the subset of the the MetaCoq AST that we need. Except that it has named variables. This allows us to build a term with named variables and then translate it to de Bruijn indices at once which makes the code generation functions much easier and in line with our Autosubst OCaml implementation.

```
Inductive nterm : Type :=
| nRef : string → nterm (* turns into tRel, tConst, tInd
    , tConstruct from the term type *)
| nHole : nterm
| nTerm : term → nterm (* embed MetaCoq AST terms *)
| nProd : string → nterm → nterm → nterm
| nArr : nterm → nterm → nterm
| nLambda : string → nterm → nterm → nterm
| nApp : nterm → list nterm → nterm
| nFix : mfixpoint nterm → ℕ → nterm
| nCase : string → ℕ → nterm → nterm → list (ℕ *
    nterm) → nterm.

Inductive term : Type :=
| tRel : ℕ → term
| tEvar : ℕ → list term → term
| tProd : aname → term → term → term
| tLambda : aname → term → term → term
| tApp : term → list term → term
```

```
| tConst : kername → Instance.t → term
| tInd : inductive → Instance.t → term
| tConstruct : inductive → ℕ → Instance.t → term
| tCase : (inductive * ℕ) * relevance →
          term → term → list (ℕ * term) → term
| tFix : mfixpoint term → ℕ → term
| ...
```

Listing 4.3: Our `nterm` AST in comparison to the MetaCoq AST (abridged)

4.1.3 Code Generation

The main architectural difference between Autosubst OCaml and Autosubst MetaCoq is that the latter uses two monads instead of one: `TemplateMonad` and a custom `RWSEM` monad.
We give a graphical representation of their relationship in Figure 4.1.

We do not use a monad library to define a monad transformer stack. Instead, we implement `RWSEM` from scratch as a combination of `Reader`, `Writer`, `State`, and `Error`.

Code generation still advances topologically through the strongly connected components of the dependency graph. For every component, the `genCode` function is called, which returns an expression of the `RWSEM` monad. This expression is evaluated and gives the generated abstract syntax terms and the state of the code generation.
The abstract syntax terms are used with the `TemplateMonad` to put the generated code into the environment.
The state of the code generation contains information like our implicit argument tracking and is passed to the next call to `genCode`.

The `genCode` function implements the code generation for a component and is defined analogously to Autosubst 2 and Autosubst OCaml.

Generating the automation proved to be uncomfortable because we cannot define notations and tactics using the `TemplateMonad`. As a workaround, we define simple printers that generate the tactic code as a string.
This string is printed in the Coq message buffer and the user can then paste it back into the Coq buffer to define tactics. The user can automatically format the output this way, so we can omit formatting logic in our printer.
This detour is certainly not nice to use and a drawback of Autosubst MetaCoq.
We had problems with generating notations so we do not generate them at all.

To add the this functionality to the `TemplateMonad` would require a reification of notations and tactics. But we do not believe this is in the scope of MetaCoq.

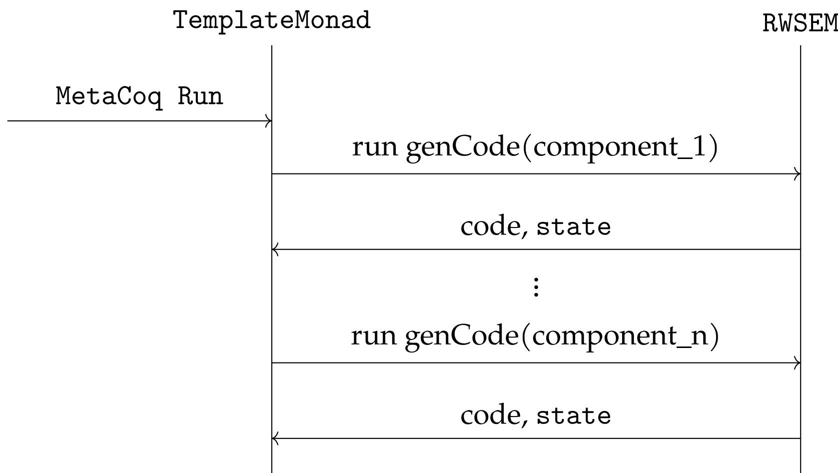


Figure 4.1: Relationship of `TemplateMonad` and `RWSEM`.

4.2 Missing Features

In comparison to Autosubst OCaml there are some missing features that make this implementation incomplete.

- We do not generate lemmas with functional extensionality.
- We do not organize generated code in modules. We suspect that is not possible with MetaCoq.
- We do not generate custom variable names. The feature is in the syntax but it is ignored.
- We do not generate lemmas with `pointwise_relation` for the new `asimpl`.

4.3 Extension to MetaCoq

Tracking and generating implicit arguments as discussed in section 4.1.2 adds complexity to the program. Also, it hurts performance because we do a string-keyed association list lookup for every application node with a named variable, to check if that variable has any implicit arguments.

As a possible fix we implement a `TemplateMonad` command corresponding to the Coq `Arguments` vernacular command[6]. Using this command one can declare arguments as implicit like so:

```
MetaCoq Run (tmArguments "f" ["x"; "y"]).
Arguments f {x} {y}. (* Equivalent to this *)
```

[6] `https://github.com/addap/metacoq/blob/tmArguments/test.v`

At the time of writing this thesis, it is just a proof-of-concept that can only declare arguments as maximally-inserted implicit but in general it is possible to support more functionality of the `Arguments` command.

Using this would reduce complexity and increase performance because we do not have to instantiate implicit arguments with evars in MetaCoq. Instead, the Coq runtime takes care of it.
We also think this would be a sensible addition the `TemplateMonad` so that other MetaCoq users can use implicit arguments in their generated terms more comfortably.

However, a proper implementation is left as future work due to time and scope constraints.

Chapter 5

Case Studies

In this chapter we discuss the case studies that exist for Autosubst OCaml.

Stark has already implemented several case studies for Autosubst 2 [30, Part III]. We port a subset to Autosubst OCaml to ensure compatibility of our implementation. Also we add one case study that arose naturally when the author tried to mechanize a metatheorem found in the book "Types and Programming Languages" [20].

Stark's case studies along with our own are part of our CI[1] process on GitHub[2] to ensure that there are no regressions in Autosubst OCaml.

Unfortunately, we did not implement noteworthy case studies for Autosubst MetaCoq because of a lack of time and Autosubst MetaCoq is not as easy to use as Autosubst OCaml at the time of writing this thesis.
We only test the code generation of Autosubst MetaCoq on some input languages ensure that it is working[3].

5.1 Stark's Existing Case Studies

To show the usefulness of Autosubst, Stark has already implemented numerous case studies for Autosubst 2 in her dissertation. We port[4] only the subset that does not deal with modular syntax, because Autosubst OCaml does not support this feature. The case studies contain proofs of metatheorems mainly for the, simply typed λ-calculus and System F with subtyping. And there are some short proofs for other systems like variants of System F and the untyped λ-calculus.

- Instantiation preserves reduction in the untyped λ-calculus with pairs.
- Instantiation preserves reduction in System F_{cbv}.

[1]Continuous Integration
[2]https://github.com/uds-psl/autosubst-ocaml/actions
[3]https://github.com/uds-psl/autosubst-metacoq/tree/master/test/examples.v
[4]https://github.com/uds-psl/autosubst-ocaml/tree/master/case-studies/kathrin/coq

- Type preservation in the multivariate λ-calculus.
- Weak head normalization in the simply typed λ-calculus.
- Strong normalization via Raamsdonk's Characterization [34] in the simply typed λ-calculus with sums.

The most extensive case study is solving parts 1 and 2 of the POPLMark challenge [6].
Both parts of the challenge are about proving metatheorems for a variant of System F.
Part one uses System F extended with subtyping and part two uses System F extended with subtyping, record types and pattern matching.

We use our version of `asimpl` where possible in order to reduce reliance on the functional extensionality axiom. Porting the code is straightforward because Autosubst OCaml provides a very similar interface to Autosubst 2.
The major differences that had to be accounted for are due to differences in behavior of our new version of `asimpl`, further discussed in Section 6.1.

- Autosubst OCaml does not provide the `asimpl in *` version of `asimpl` because there is no `setoid_rewrite in *`.
  The solution is to instead use `asimpl in H` for any hypothesis H that needs to be simplified.
- We have to instantiate existential variables in a few places because `asimpl` cannot deal with them.
- `asimpl` needs morphisms for certain user-defined types.
  We discuss this below.

Stark defines the subtype relation sub for System F with record types as the following.

```
(* chapter10/sysf_pat.v::10 *)
Inductive ty (n_ty : ℕ) : Type :=
  | var_ty : fin n_ty → ty n_ty
  | top : ty n_ty
  | arr : ty n_ty → ty n_ty → ty n_ty
  | all : ty n_ty → ty (S n_ty) → ty n_ty
  | recty : list (ℕ * ty n_ty) → ty n_ty.

(* chapter10/POPLmark21.v::292 *)
Inductive sub {n} (Delta : fin n → ty n) :
    ty n → ty n → ℙ :=
...
```

```
| SA_rec (xs ys : list (ℕ * ty n)) :
    (∀ l T', In (l, T') ys → ∃ T, In (l, T) xs ∧ sub
       Delta T T') →
    unique xs → unique ys →
    sub Delta (recty xs) (recty ys).
```

Here sub is nested in an existential type.
To rewrite inside the subtype relation with `asimpl` we have to prove a morphism for it. To do this, we first have to prove a stronger induction principle for `sub`. Since `sub` has a nested recursive occurrence below an existential quantifier, Coq's automatically derived induction principle is not strong enough. We first tried to use Marcel Ullrich's MetaCoq plugin from his bachelor thesis [33] to generate a strong induction principle. But unfortunately, it does not support nesting in an existential quantifier, so we implemented the stronger induction lemma ourselves[5].

In conclusion, our new version of `asimpl` is mostly a drop-in replacement. Sometime it does not work however, which we discuss in Section 6.1.3.

5.2 TAPL Exercise 23.6.3

To experience how a user works with Autosubst OCaml we also add a case study of our own[6].
For this we choose to mechanize the first three steps of exercise 23.6.3 from TAPL [20, Page 378].

The exercise is about proving the non-typability of the term $\mathtt{omega}$ in System F. The term is defined as:

$$\mathtt{omega} = (\lambda x.\, x\, x)(\lambda y.\, y\, y)$$

It is not typable because $\mathtt{omega}$ reduces to itself and System F has the normalization property, i.e. reduction of all well-typed terms terminates.
The exercise consists of seven steps and we mechanize the first three, but only the second actually uses substitution lemmas.

We first use Autosubst OCaml to generate code for our specification of System F (Appendix A.2). Then we define a typing relation, the exposed predicate, and erasure into untyped λ-calculus using the generated inductive types. Finally, we mechanize the three steps.

We use the automation of Autosubst mainly in the proofs of progress and preservation for System F.

[5] From private communication we gather that his tool is planned to support existential quantifiers in the future so users of the new `asimpl` tactic could eventually use it also for this use-case.
[6] The development can be found in `https://github.com/uds-psl/autosubst-ocaml/tree/master/case-studies/tapl-exercise`

In turn, we need the preservation property in the proof of step two.
Steps one and three can be proved without the automation of Autosubst because no substitutions occur their proofs. Still, they of course use the language implementation provided by Autosubst.

We made the experience that Autosubst lowers the burden of starting to work on a mechanization. If we were not aware of Autosubst at the time of solving the exercise on paper, we would not have attempted to mechanize it.
This is due to the ratio of necessary boilerplate code to code of the exercise itself. The boilerplate is much longer. Autosubst made it easy to just concentrate on the exercise itself, so it has achieved its purpose for us.

Note that this development was done before Autosubst OCaml supported generation of the `asimpl` tactic. So initially, we proved everything using the rewriting lemmas directly.
Therefore, even without the automation, Autosubst OCaml is helpful.

Chapter 6

Extensions

In this chapter we discuss extensions over the original Autosubst 2 that we implemented in Autosubst OCaml and also what we fixed

6.1 Asimpl

6.1.1 Functional Extensionality

Functional extensionality (funext) is an axiom that states that two functions are equal if they are **extensionally equal**, which means they are equal applied to all arguments.

```
Axiom funext : ∀ (X Y: 𝕋) (f g : X → Y),
    (∀ x : X, f x = g x) → f = g.
```

It is consistent as an axiom in Coq, however, introducing an axion will hurt adoption of a library. We know of at least two development that do not use Autosubst because of its reliance on the funext axiom [15, 16].
This is why one of the targets of this work is to remove the dependency.

Note however, that funext is only needed for the automation of Autosubst. One can still solve the same substitution equations by manually rewriting with lemmas that do not use funext.

6.1.2 Asimpl with Functional Extensionality

Autosubst 2 uses funext for easier rewriting in the `asimpl` tactic[1]. There are a number of lemmas in the equational theory of a given de Bruijn algebra that state the extensional equality of certain functions. In case of the λ-calculus (which we use as a running example in this section), the following is one of them.

```
Lemma scons_comp (t: tm) (f: ℕ → tm) (g: ty → tm) :
  ∀ n, ((t .: f) >> g) n = (t[g] .: f >> g) n.
```

[1] Which we call `asimpl_fext` in the following.

Using funext we can promote the extensional equality of the functions to an equality of the functions.

```
Lemma scons_comp_fext (t: tm) (f: ℕ → tm)
                              (g: ty → tm) :
  (t .: f) >> g = t[g] .: f >> g.
```

Now, we can easily rewrite the function on the left-hand side to the function on the right-hand side in arbitrary contexts, like in the following instantiation.

```
s[(t .: var_tm) >> g] = s[t[g] .: var_tm >> g]
```

However, funext does not give us strictly more power in the context of Autosubst because we can also prove the equation above using an extensionality lemma combined with `scons_comp`. The extensionality lemma states that instantiations with extensionally equal functions are equal.

```
Lemma ext_tm (s: tm) (f g : ℕ → tm) :
  (∀ n, f n = g n) → s[f] = s[g]
```

Intuitively `ext_tm` makes sense because a substitution replaces all free variables in a term. If the two substitutions are equal for all free variables, the instantiations must also result in the same term.
These extensionality lemmas are already generated by Autosubst 2 and our implementations.

But using funext is certainly more convenient, because we can use the normal rewriting facilities to replace one function with another. Whereas with the extensionality lemma we need to apply it and then argue why the substitutions are extensionally equal.
In the example above this is directly implied by `scons_comp`. But in general the function we want to replace might be contained in a deeper context like in the following.

```
s[h >> ((t .: var_tm) >> g)] = s[h >> (t[g] .: var_tm >> g)]
```

To prove this substitution equation we also need to know that composing extensionality equal functions results in extensionality equal functions.

Generally, substitutions (and renamings) in Autosubst can be composed of opaque functions `f`, extension `scons` (or `.:`) and function composition `funcomp` (or `>>`). Generally, we need some automated way of proving the extensional equality of functions composed of those primitives.

6.1.3 Asimpl with Setoid Rewriting

One possible way is to use Coq's generalized rewriting – or setoid rewriting – library [27]. Setoid rewriting enables rewriting with relations other than equality, for example with extensional equality.
In order to use setoid rewriting to replace the argument of a function `f` we have to define a **morphism** $\texttt{morph}_\texttt{f}$ for it.
The morphism specifies that if the arguments are in some relation `R_1 x y`, then the function values are in some other relation `R_2 (f x)(f y)`.
Setoid rewriting can then automatically derive a **strategy** for composing morphisms so that for example starting from an assumption `H : R_1 x y` we can prove `R_3 (g (f x))(g (f y))` using intermediate morphisms from `R_1` to `R_2` and from `R_2` to `R_3`.

So we can implement `asimpl` with setoid rewriting if we define morphisms that can compose to a strategy starting at one of our rewrite lemmas to a substitution equation.

Morphisms

We need to define morphisms that can be composed to reduce substitution equations to one of the rewriting lemmas. In total we need morphisms for

- instantiation with substitutions,
- instantiation with renamings,
- extension,
- and function composition.

The proofs of the morphisms for the instantiations are applications of the extensionality lemmas that we already generate and the other two morphisms can be defined statically. The morphisms are instances of the `Proper` typeclass. We generate `vernac_expr` terms to construct the instances.

Lemmas

In general, we need extensional variants of all rewriting lemmas `H` that are used with a `rewrite H` in the `asimpl_fext` tactic. These are the four compositionality lemmas, the left-identity lemma for renamings and substitutions, and the right-identity lemma for renamings and substitutions.

Technical Remark 6.1

Setoid rewriting has some problems rewriting with lemmas that use an extensional equality explicitly, like the following:

```
Lemma renRen_tm (xi_tm : ℕ → ℕ) (zeta_tm : ℕ → ℕ) :
    ∀ (s : tm),
    ren_tm zeta_tm (ren_tm xi_tm s) = ren_tm (funcomp zeta_tm
       xi_tm) s.
```

It works better if we instead use the `pointwise_relation` predicate from the `Setoid` library, which abstracts extensional equality.

```
Lemma renRen_tm_pointwise (xi_tm : ℕ → ℕ) (zeta_tm : ℕ → ℕ) :
  pointwise_relation _ eq (funcomp (ren_tm zeta_tm) (ren_tm
     xi_tm))
    (ren_tm (funcomp zeta_tm xi_tm)).
```

We suspect it is a weakness of the strategy search used by setoid rewriting. The `pointwise_relation` might be easier than an explicit quantifier because it does not mention a bound variable.
So when we speak of extensional variants of lemmas in this Section, we mean those using the `pointwise_relation` predicate.

For the λ-calculus this means we need the following 8 extensional rewrite lemmas.

$$
\begin{aligned}
[\sigma][\tau] &\equiv [\sigma \circ [\tau]] & \mathrm{var} \circ [\sigma] &\equiv \sigma \\
[\sigma]\langle\zeta\rangle &\equiv [\sigma \circ \langle\zeta\rangle] & \mathrm{var} \circ \langle\xi\rangle &\equiv \xi \circ \mathrm{var} \\
\langle\xi\rangle[\tau] &\equiv [\xi \circ \tau] & [\mathrm{var}] &\equiv \mathrm{id} \\
\langle\xi\rangle\langle\zeta\rangle &\equiv \langle\xi \circ \zeta\rangle & \langle\mathrm{id}\rangle &\equiv \mathrm{id}
\end{aligned}
$$

The extensional versions of the compositionality lemmas are already generated by Autosubst 2, with an explicit quantifier, so we wrap it in the `pointwise_relation`. The extensional versions of the other lemmas are newly generated by our implementations.

Tactic

We generate an ltac script that is similar to the original `asimpl_fext`. This is because whenever `asimpl_fext` uses `rewrite H`, the new `asimpl` can use `setoid_rewrite H'` with `H'` being an extensional version of `H`.
Also, `asimpl_fext` unfolds function composition, which the new `asimpl` avoids since it is one of the relations that setoid rewriting uses for the strategy search.

Evaluation

In our experience, working with setoid rewriting can be hard because sometimes it is not obvious why it fails to infer a strategy the way that one has in mind[2]. Small

[2]Failure to infer a strategy is typeclass inference error, which are notoriously hard to decipher in Coq.

changes in the definition of a morphism or rewrite lemmas can easily introduce regressions in asimpl's power.
During the process of developing the new asimpl, our CI process turned out to be very useful for catching these regressions by testing it on all case studies.

To conclude, our new version of asimpl works and successfully avoids the use of functional extensionality for rewriting in Stark's original case studies (Section 5.1). But at the moment we can only give a weaker conjecture of completeness for this decision procedure because for some suitably complex cases, setoid rewriting fails to infer a strategy.

In the end we adopted a workflow that used both tactics. The new asimpl as long as it works, and if it cannot solve the goal we check with asimpl_fext and help manually.

This is sometimes necessary because goals can appear that confuse setoid rewriting. It happens when the relations that setoid rewriting needs for strategy search – function composition and pointwise_relation – are unfolded. For example the following goal is not immediately solved with our new asimpl.

```
Goal ∀ m n k l (s: tm l n) (σ: fin m → tm l n) (f: fin n → tm l k)
    (x: fin (S m)),
    subst_tm var_ty f ((s .: σ) x)
    = (subst_tm var_ty f s .: σ >> subst_tm var_ty f) x.
```

It only solves the goal if the left side is formulated with function composition and the explicit extensionality is replaced with pointwise_relation.

```
Goal ∀ m n k l (s: tm l n) (σ: fin m → tm l n) (f: fin n → tm l k),
    pointwise_relation _ eq ((s .: σ) >> (subst_tm var_ty f))
                (subst_tm var_ty f s .: σ >> subst_tm var_ty f).
```

6.2 Traced Syntax

Recall that Autosubst 2 supports generation of unscoped and wellscoped syntax as discussed in Section 2.5.3.
To explore the extensibility of our implementation we explore how to implement an additional syntax variant: **traced syntax** [12].

Traced syntax can be regarded as a generalization of wellscoped syntax.
Recall that in wellscoped syntax an inductive type is parameterized by a natural number that is the upper bound of free variables that can appear in terms of this type.
In contrast, in traced syntax an inductive type is parameterized by a list of natural numbers which specify exactly which free variables can appear inside terms of this type.

Our idea of traced syntax is inspired by [12] but we have to adapt several aspects so that it works with our use-case.

- Herbelin et al. explicitly renounce de Bruijn indices while we use them extensively.
- They carry a proof in the variable constructor, that a given name is in the list of free variables, while are just interested in carrying the number itself.
- They define substitutions to only insert closed terms, which Autosubst does not do.

This formalization is based on the existing formalization of wellscoped syntax.

Terms are readily defined as follows

```
Inductive tm (vs: list ℕ) : Type :=
| var_tm : finL vs → tm vs
| app : tm vs → tm vs → tm vs
| lam : tm (0 :: vs) → tm vs.
```

The three differences to an analogous definition in wellscoped syntax (Section 2.5.3) are:

- The type `tm` is indexed by a list of natural numbers.
- The variable constructor takes a value of type `finL vs`.
  This type represents numbers in the list `vs`.
- The body argument of the binder `lam` is lifted by prepending a zero to the list `vs`.
  We discuss our reasoning for this lifting in Section 6.2.1.

Note that, analogous to the `n` parameter in wellscoped syntax, the list `vs` is only a superset of the allowed free variables in terms.
That means it can contain numbers that do not appear in a given term, e.g. the term `(lambda . 0 1)` can be typed with `(tm [0; 4])`[3].

This means that wellscoped syntax can be regarded as a special case of traced syntax where the list represents all de Bruijn indices up to some number `n`.
This observation leads us to our formalization of traced syntax.

[3]The 0 in the list corresponds to the variable 1 in the term, because the 1 appears under a binder

6.2.1 Formalization

In traced syntax, de Bruijn indices have the type `finL vs` that is parameterized by a list of natural numbers.
There are exactly as many inhabitants of `finL vs` as members in the list `vs`.

Definition 6.1 (finL)

$$\mathit{wrapNat}(v : \mathbb{N}) : \mathbb{T} ::= wN$$
$$\mathrm{finL}\ [] = \bot$$
$$\mathrm{finL}\ (v :: vs') = (\mathit{wrapNat}\ v) + (\mathrm{finL}\ vs')$$

`finL` is implemented as a recursive function definition on the given list. And it uses `wrapNat` to lift a given natural number to the type level, so that for example the type `finL [1;2]` can intuitively be read as "1 or 2 or $\bot$".

But a key simplification that we do from an intuitive interpretation of `finL vs` is the natural numbers in `vs` do not immediately specify the de Bruijn indices that `finL vs` represents.
Instead, for a list $vs = [v_0; \ldots; v_i; v_{i+1}; \ldots; v_n]$, the number v_{i+1} represents a de Bruijn index $w_{i+1} = w_i + v_{i+1} + 1$. So v_{i+1} specifies the gap of missing de Bruijn indices (plus one) since v_i.
For example the type `finT [0;0;0]` represents the de Bruijn indices 0, 1, and 2.
And the type `finT [0;2;1]` represents the de Bruijn indices 0, 3, and 5 (and not literally 0, 2, and 1).
This simplification is done so that the represented set of de Bruijn indices is implicitly strictly ordered. And so that lifting by prepending a zero implicitly shifts the de Bruijn indices so that we do not have to change `vs`.

We want it to be strictly ordered to ensure that there is only a single type for a set of de Bruijn indices.
With a literal treatment of `vs`, ordering would have to be ensured some other way and lifting the list, like in the body of the `lam` constructor, would have to increase all numbers.

```
| lam : tm (0 :: map S vs) → tm vs
```

This transformation on `vs` makes code generation harder because in some places we then have to "unshift" it to remove the mapping.
Therefore, to make code generation simpler, we opt for the more complex interpretation.

Technical Remark 6.2

The definition of the finite type `finL` has a big impact on the ease of code generation. There are other possible definitions with different trade-offs [17]. We also tried one formalization with an indexed inductive type. But this results in conflicts with Coq's index condition [26, Section 21.2] which makes code generation harder.
Therefore, we opt for a recursive definition that is derived from `fin` of wellscoped syntax using the algebraic properties of Coq's inductive types. Recall that `fin` is defined as follows.

```
Fixpoint fin (n: ℕ) : Type :=
  match n with
  | 0 ⇒ ⊥
  | S n' ⇒ option (fin n')
  end.
```

So `fin 0` is not inhabited and for each iteration of the `S` constructor, the type gets one new member due to the `None` constructor.
For `finL` we want a similar behavior that gives the type one new member for each iteration of the `cons` constructor. To arrive at this we start by inlining the `option` type, and then replacing the `unit` type with a type `wrapNat n` that represents single natural number.

```
Fixpoint fin' (n: ℕ) : Type :=
  match n with
  | 0 ⇒ ⊥
  | S n' ⇒ unit + fin' n'
  end.

Inductive wrapNat (n: ℕ) := wN : wrapNat n.
Fixpoint finL (vs: list ℕ) : Type :=
  match vs with
  | [] ⇒ ⊥
  | v :: vs' ⇒ wrapNat v + finL vs'
  end.
```

Using this formalization the code generation has to be amended in only a few ways.

- Generating the type of scope variables as `list ℕ`.
- Generating the type `finT` instead of `fin`.
- Inserting a match on the `wrapNat` constructor in certain places.

This exemplifies that our formalization of traced syntax is very similar to wellscoped syntax, since it is a natural generalization.

The primitives for the de Bruijn index $0_{\mathbb{L}}$, shifting $(\uparrow_{\mathbb{L}})$, and stream extension $(\cdot_{\mathbb{L}})$ are defined very similarly to wellscoped syntax (Figure 2.8).

Definition 6.2 (Traced Syntax Primitives)

$$
\begin{aligned}
Z_{\mathbb{L}} &: \mathsf{finL}\ (0 :: vs) \\
Z_{\mathbb{L}} &= \mathsf{inl}\ (wN\ 0)
\end{aligned}
$$

$$
\begin{aligned}
\uparrow_{\mathbb{L}} &: \mathsf{finL}\ vs \to \mathsf{finL}\ (0 :: vs) \\
\uparrow_{\mathbb{L}}\ m &= \mathsf{inl}\ m
\end{aligned}
$$

$$
\begin{aligned}
\cdot_{\mathbb{L}} &: X \to (\mathsf{finL}\ vs \to X) \to \mathsf{finL}\ (0 :: vs) \to X \\
(x \cdot_{\mathbb{L}} f)(\mathsf{inl}\ (wN\ 0)) &= x \\
(x \cdot_{\mathbb{L}} f)(\mathsf{inr}\ m) &= f\ m
\end{aligned}
$$

We have not evaluated the merits of our formalization on a case study and we conjecture that our special interpretation of the list `vs` makes traced syntax harder to use than with a literal interpretation.
However, in the future we want to explore different formalization, with a literal treatment of `vs`, and with different primitives.
We implement formalization in this section manually for one language[4]. It is not part of code generation in Autosubst OCaml.

6.3 Autosubst 2

We found one bug in Autosubst 2 during our reimplementations.

If the first sort in a component of size $\geqslant 2$ is not recursive[5], the substitution operation is falsely generated as a `Definition` instead of a `Fixpoint` while still containing multiple bodies. This concrete syntax is not parseable by Coq.

```
Definition subst_vl ... := ...
 with subst_tm ... := ...
```

Listing 6.1: Instantiation with substitution if the `sysf_cbv.sig` signature is changed to declare `vl` before `tm`.

[4] https://github.com/addap/autosubst-ocaml/blob/master/traced/utlc_traced.v
[5] the sort occurs in itself

The bug was reported[6] and the solution was to drop the check for recursivity. It was not needed anyways and the bug just never surfaced because it depends on the exact signature definition.

In comparison to Autosubst 2 we also cleanup up the static files to remove unused code, put notations into a separate module to only include them optionally, and factor out the code that uses the functional extensionality axiom into a different file to also only include it optionally[7].

[6]in private email communication with Kathrin Stark

[7]The updated static files can be found in `https://github.com/addap/autosubst-ocaml/tree/master/share/autosubst`

Chapter 7

Conclusion

To conclude the thesis we summarize our main results.

7.1 Autosubst OCaml

We reimplement Autosubst 2 in OCaml to reuse code from the Coq library for code generation.
Autosubst OCaml supports all syntax features of Autosubst 2 except modular syntax.
By default, it does not generate lemmas and tactics with functional extensionality, but they can be enabled so that Autosubst OCaml can be a drop-in replacement for Autosubst 2.

Autosubst OCaml generates a new kind of `asimpl` tactic, which is based on setoid rewriting and can normally replace the old `asimpl_fext`. We adjust the case studies by Stark to use the new `asimpl` almost everywhere[1]. This shows it is an adequate replacement of `asimpl_fext` in most cases.

But there are certain goals that confuse setoid rewriting. If that happens, a user can either solve the goal manually, fall back on `asimpl_fext`, or try to change the goal so that setoid rewriting works again. So the new `asimpl` is not a perfect solution.

7.2 Autosubst MetaCoq

We partially reimplement Autosubst 2 in MetaCoq to use MetaCoq's metaprogramming facilities for code generation.
Autosubst MetaCoq supports most syntax features of Autosubst 2. It does not support modular syntax and has limited support for functors because for languages with functors it cannot define the inductive type due to universe constraint errors.

Autosubst MetaCoq generates the basic rewrite lemmas for a given input language. But it generates neither rewrite lemmas with functional extensionality, nor rewrite

[1] `variadic_fin.v` uses the `cod` functor, which implies functional extensionality

lemmas using the `pointwise_relation` predicate. Furthermore, we have only an experimental generation of tactics in Autosubst MetaCoq because there is no predefined tactic AST provided.

This precludes users of Autosubst MetaCoq from using any automation, until these lemmas are generated along with the correct tactics.

7.3 Future Work

Our main focus for future work lies in Autosubst OCaml. After working on both projects we can decidedly say that it is much easier to understand, extend, and use Autosubst OCaml.

The main reasons are because we think OCaml is a better language than Gallina for implementing programs:

- The OCaml language is easier to understand than Gallina. Gallina sometimes has counter-intuitive reduction behavior which can make terms explode in size, or not reduce at all, if some definition uses `Qed` instead of `Defined`.
- OCaml is more efficient than Gallina. Even on relatively small languages like System F_{cbv}, Autosubst MetaCoq is a lot slower than Autosubst OCaml.
- Development tools for OCaml are better and allow easier programming.
- OCaml has a fairly large standard library. For Coq, most libraries help with proofs. So to implement computation in Gallina, one often has to reinvent wheels which places an additional burden on the programmer.

In Autosubst OCaml we want to further explore:

- An implementation of traced syntax.
- An implementation of `asimpl` that matches on the syntax of goals. This should give us more control than relying on setoid rewriting to find a solution.

In Autosubst MetaCoq we want to at least generate the lemmas with functional extensionality. We expect this is not much work because we already generate the corresponding lemmas with extensional equality.

Finally, both programs need a lot more documentation in order to be understandable to other people. This is a failure of our implementations. We postponed writing documentation for too long. Some parts of Autosubst OCaml are well-documented, mostly parts that were newly implemented by us.
But the code generation itself still needs more documentation.

Appendix A

Appendix

A.1 System F_{cbv}

We define System F_{cbv} that we use throughout the thesis.
Also, we give its specification in our input syntax and its associated de Bruijn algebra.

Definition A.1 (**System F_{cbv}**)

$$\begin{aligned} T, U \in ty &:= n \mid T \to U \mid \forall.T & n \in \mathbb{N} \\ s, t \in tm &:= s\ t \mid s\ T \mid v \\ v \in vl &:= n \mid \lambda.s \mid \Lambda.s & n \in \mathbb{N} \end{aligned}$$

$\lambda.s$ is a term-abstraction and its type is an arrow type $T \to U$.
$\Lambda.s$ is a type-abstraction and its type is a quantifier $\forall.T$.

```
ty : Type
tm : Type
vl : Type

arr : ty -> ty -> ty
all : (bind ty in ty) -> ty

app  : tm -> tm -> tm
tapp : tm -> ty -> tm
vt   : vl -> tm

lam  : ty -> (bind vl in tm) -> vl
tlam : (bind ty in tm) -> vl
```

Listing A.1: Specification of System F_{cbv} in our input syntax.

Definition A.2 (**Rewrite Lemmas of the de Bruijn algebra of System F_{cbv}**)

A.2 System F

We define System F. Also, we give its specification in our input syntax.

Definition A.3 (System F)

$$T, U \in ty := n \mid T \to U \mid \forall.T \qquad n \in \mathbb{N}$$
$$s, t \in tm := s\ t \mid s\ T \mid \lambda.s \mid \Lambda.s \qquad n \in \mathbb{N}$$

$\lambda.s$ is a term-abstraction and its type is an arrow type $T \to U$.
$\Lambda.s$ is a type-abstraction and its type is a quantifier $\forall.T$.

```
ty : Type
tm : Type

arr : ty -> ty -> ty
all : (bind ty in ty) -> ty

app : tm -> tm -> tm
tapp : tm -> ty -> tm
lam : ty -> (bind tm in tm) -> tm
tlam : (bind ty in tm) -> tm
```

Listing A.2: Specification of System F in our input syntax.

A.3 σ_{SP}-calculus

We define the σ_{SP}-calculus and its reduction[1] (taken from [30, Figure 4.1]).

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

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

[1] This relation excludes β-reduction. When adding β-reduction, we explicitly write $\lambda\sigma_{SP}$-calculus.

There have been a range of calculi of explicit substitutions proposed (summarized in [4]), starting with the original σ-calculus by Abadi et al. [1].

Note that the reduction rules of the σ_{SP}-calculus correspond with the lemmas about substitution of the de Bruijn algebra of the λ-calculus from Section 2.3, so it is intuitive that the de Bruijn algebra models the σ_{SP}-calculus.

A.4 Rewrite Laws of the de Bruijn Algebra of the λ-calculus

Lemma A.4 (Interaction Laws [30, Fact 3.1])

$$\mathrm{id} \circ f \equiv f \equiv f \circ \mathrm{id} \tag{A.1}$$
$$(f \circ g) \circ h \equiv f \circ (g \circ h) \tag{A.2}$$
$$(s \cdot \sigma) \circ f \equiv (f\, s) \cdot (\sigma \circ f) \tag{A.3}$$
$$\uparrow \circ (s \cdot \sigma) \equiv \sigma \tag{A.4}$$
$$0 \cdot \uparrow \equiv \mathrm{id} \tag{A.5}$$
$$(\sigma\, 0) \cdot (\uparrow \circ \sigma) \equiv \sigma \tag{A.6}$$

Lemma A.5 (Monad Laws [30, Page 26])

$$s[var] = s \tag{A.7}$$
$$s[\sigma][\tau] = s[\sigma \circ [\tau]] \tag{A.8}$$

Lemma A.6 (Compositionality Laws [30, Lemma 3.4])

$$s\langle\xi\rangle\langle\zeta\rangle = s\langle\xi \circ \zeta\rangle \tag{A.9}$$
$$s\langle\xi\rangle[\tau] = s[\xi \circ \tau] \tag{A.10}$$
$$s[\sigma]\langle\zeta\rangle = s[\sigma \circ \langle\zeta\rangle] \tag{A.11}$$
$$s[\sigma][\tau] = s[\sigma \circ [\tau]] \tag{A.12}$$

Lemma A.7 (Supplementary Laws [30, Fact 3.5])

$$var \circ [\sigma] \equiv \sigma \tag{A.13}$$
$$(\sigma \circ [\tau]) \circ [\theta] \equiv \sigma \circ [\tau \circ [\theta]] \tag{A.14}$$
$$\sigma \circ [var] \equiv \sigma \tag{A.15}$$

Lemma A.8 (Extensionality Laws [30, Lemma 3.7])

$$\text{If } \xi \equiv \zeta, \text{ then } s\langle\xi\rangle = s\langle\zeta\rangle \tag{A.16}$$
$$\text{If } \sigma \equiv \tau, \text{ then } s[\sigma] = s[\tau] \tag{A.17}$$

Lemma A.9 (Coincidence Laws [30, Lemma 3.8])

$$\Uparrow^* \xi \circ var \equiv \Uparrow (\xi \circ var) \tag{A.18}$$
$$s[\xi \circ var] = s\langle\xi\rangle \tag{A.19}$$

A.5 Reuse from the Coq Implementation

One goal of Autosubst OCaml is to reuse code from the Coq implementation. Here we list a table containing the most important types and definitions from the Coq implementation that we reuse.

`Constrexpr.constr_expr`	Gallina terms
`Constrexpr.local_binder_expr`	Binders of dependent products or lambdas
`Constrexpr.branch_expr`	Branch of a match expression
`Vernacexpr.vernac_expr`	Vernacular command terms
`Vernacexpr.vernac_control`	Vernacular command terms with attributes
`Vernacexpr.opacity_flag`	Whether a definition is opaque or transparent
`Vernacexpr.proof_end`	End of a proof
`Vernacexpr.syntax_modifier`	Notation arguments like "at level n"
`Attributes.vernac_flag`	Attributes of vernacular commands
`Tacexpr.raw_tactic_expr`	Ltac tactic terms
`Locus.clause_expr`	Location annotations like "in *"
`Glob_term.binding_kind`	If a binder is explicit or implicit
`Name.t, Id.t, lident, qualid`	Different kinds of identifiers
`ident_decl, name_decl`	Different kinds of identifiers with attached universe declaration
`Ppconstr.pr_lconstr_expr`	Printing a Gallina term
`Ppvernac.pr_vernac`	Printing a vernacular command with attributes
`Ppvernac.pr_vernac_expr`	Printing a vernacular command
`Tacexpr.pr_raw_tactic`	Printing a tactic term
`Global.env`	Global environment
`Evd.from_env`	Evar management
`Notations.declare_scope`	Declaring notation scopes
`Metasyntax.add_notation`	Adding a new notation
`Pp.seq, Pp.str`	Primitive pretty-printing functions
`CAst.make`	Wrapper for AST terms that adds source file location

Bibliography

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. *Journal of functional programming*, 1(4):375–416, 1991.

[2] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 195–207, 2017.

[3] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting functional programs from coq, in coq. *arXiv preprint arXiv:2108.02995*, 2021.

[4] Mauricio Ayala-Rincón and Cesar Munoz. Explicit substitutions and all that. 2000.

[5] Brian Aydemir and Stephanie Weirich. Lngen: Tool support for locally nameless representations. 2010.

[6] 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 p opl m ark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

[7] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[8] Matteo Cavada, Andrea Colò, and Alberto Momigliano. Mutantchick: Type-preserving mutation analysis for coq. In *CILC*, pages 105–112, 2020.

[9] Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

[10] Pierre-Louis Curien, Thérese Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)*, 43(2):362–397, 1996.

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

[12] Hugo Herbelin and Gyesik Lee. Formalising logical meta-theory semantical normalisation using kripke models for predicate.

[13] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 293–306, 2018.

[14] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & knot: Binder boilerplate tied up. In *European Symposium on Programming*, pages 419–445. Springer, 2016.

[15] Dominik Kirst and Marc Hermes. Synthetic undecidability and incompleteness of first-order axiom systems in coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[16] Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's theorem in coq: A constructive approach to finite model theory. In *International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France*, Paris, France, 2020. Springer.

[17] Jan Christian Menz. A coq library for finite types. *Bachelor's thesis, Universität des Saarlandes*, 2016.

[18] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008.

[19] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23(7):199–208, 1988.

[20] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[21] Benjamin C Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. Programming language foundations, 2020. URL `https://softwarefoundations.cis.upenn.edu/current/plf-current/index.html`.

[22] Clément Pit-Claudel and Thomas Bourgeat. An experience report on writing usable dsls in coq.

[23] 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*, pages 67–73, 2015.

[24] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In *International Conference on Interactive Theorem Proving*, pages 359–374. Springer, 2015.

[25] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of functional programming*, 20(1):71–122, 2010.

[26] Gert Smolka. Modeling and proving in computational type theory using the coq proof assistant. 2021.

[27] Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of formalized reasoning*, 2(1):41–62, 2009.

[28] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[29] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. doi: 10.1007/s10817-019-09540-0. URL `https://hal.inria.fr/hal-02167423`.

[30] Kathrin Stark. Mechanising syntax with binders in coq. 2019.

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

[32] The Coq Development Team. The coq proof assistant, January 2019. URL `https://doi.org/10.5281/zenodo.2554024`.

[33] Marcel Ullrich. Generating induction principles for nested inductive types in metacoq. 2020.

[34] Femke van Raamsdonk, Paula Severi, Morten Heine B. Sørensen, and Hongwei Xi. Perpetual reductions in-calculus. *Information and Computation*, 149(2):173–225, 1999. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1998.2750. URL `https://www.sciencedirect.com/science/article/pii/S089054019892750X`.

[35] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.