[**TODO:** Make sure to exclude no files (not rewriting,rewriting-appendix)]

Performance Engineering of Proof-Based Software Systems at Scale

by

Jason S. Gross

B.S., Massachusetts Institute of Technology (2013)
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author .....
Department of Electrical Engineering and Computer Science
January 27, 2021

Certified by .....
Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Performance Engineering of Proof-Based Software Systems at Scale

by

Jason S. Gross

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Formal verification is increasingly valuable as our world comes to rely more on software for critical infrastructure. A significant and understudied cost of developing mechanized proofs, especially at scale, is the computer performance of proof generation. This dissertation aims to be a partial guide to identifying and resolving performance bottlenecks in dependently typed tactic-driven proof assistants like Coq.

We present a survey of the landscape of performance issues in Coq, with micro- and macro-benchmarks. We describe various metrics that allow prediction of performance, such as term size, goal size, and number of binders, and note the occasional surprising lack of a bottleneck for some factors, such as total proof term size. To our knowledge such a roadmap to performance bottlenecks is a new contribution of this dissertation.

The central new technical contribution presented by this dissertation is a reflective framework for partial evaluation and rewriting, already used to compile a code generator for field-arithmetic cryptographic primitives which generates code currently used in Google Chrome. We believe this prototype is the first scalably performant realization of an approach for code specialization which does not require adding to the trusted code base. Our extensible engine, which combines the traditional concepts of tailored term reduction and automatic rewriting from hint databases with on-the-fly generation of inductive codes for constants, is also of interest to replace these ingredients in proof assistants' proof checkers and tactic engines. Additionally, we use the development of this framework itself as a case study for the various performance issues that can arise when designing large proof libraries. We also present a novel method of simple and fast reification, developed and published during this PhD.

Finally, we present additional lessons drawn from the case studies of a category-theory library, a proof-producing parser generator, and cryptographic code generation.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Electrical Engineering and Computer Science

Dedicated to future users and developers of proof assistants.

Dedicated also to my mom, for her perpetual support and nurturing throughout my life.

Acknowledgments

I'm extremely grateful to my advisor Adam Chlipala for his patience, guidance, encouragement, advice, and wisdom, during the writing of this dissertation, and through my research career. I don't know what it's like to have any other PhD advisor, but I can't imagine having a PhD advisor who would have been better for my mental health than Adam. I want to thank my coworkers, with special thanks to Andres Erbsen for many engaging conversations and rich and productive collaborations. Special thanks to my mom, for taking every opportunity to enrich my life and setting me on this path, for encouraging me from my youth and always supporting me in all that I do. I want to thank my sister Rachel, my dad, and the rest of my family for always being kind and supportive. My friendship with Allison Strandberg through the years has been invaluable and fulfilling, and early discussions with her helped me shape the initial story I wanted to present with my dissertation. I will be eternally grateful to Rajee Agrawal for her faith in me, for everything she's done to help me excel, for her help in finding a much better story for my work than I had ever had, and for helping me find how to present this story in both my defense and this dissertation. While the technical work in proof assistants has always been a delight, writing papers has remained a struggle, and the process of completing my PhD with a thesis and a defense would have been a great deal more stressful without Rajee.

I want to thank the Coq development team—without whom I would not have a proof assistant to use—for their patience and responsiveness to my many, many bug reports, feature requests, and questions. Special thanks to Pierre-Marie Pédrot for doing the heavy lifting of tracking down performance issues inside Coq and explaining them to me, and fixing many of them. I'd also like to thank Matthieu Sozeau for adding support for universe polymorphism and primitive projections to Coq, and responding to all of my bug reports on the functionality and performance of these features from my work on the HoTT Category Theory library during the development of these features.

I'd like to thank the rest of my thesis committee—Professor Saman Amarasinghe and Professor Nickolai Zeldovich—for their support and direction during the editing of this dissertation and my defense.

Moving on to more specific acknowledgments, I want to thank Andres Erbsen for pointing out to me some of the particular performance bottlenecks in Coq that I made use of in this thesis, including those of subsubsection Sharing in Section 2.6.1 and those of subsections Name Resolution, Capture-Avoiding Substitution, Quadratic Creation of Substitutions for Existential Variables, and Quadratic Substitution in Function Application in Subsection 2.6.3. I'd like to thank András Kovács for a brief exchange with me and Andres in which it became clear that we could frame many of the performance issues we were encountering as needing to "get the basics right."

I'd like to thank Hugo Herbelin for sharing the trick with `type of` to propagate

universe constraints[1] as well as useful conversations on Coq's bug tracker that allowed us to track down performance issues.[2] I would like to thank Jonathan Leivent for sharing the trick of annotating identifiers with `: Type` to avoid needing to adjust universes[3]. I would like to thank Pierre-Marie Pédrot for conversations on Coq's Gitter and his help in tracking down performance bottlenecks in earlier versions of our reification scripts and in Coq's tactics. I would like to thank Beta Ziliani for his help in using Mtac2, as well as his invaluable guidance in figuring out how to use canonical structures to reify to PHOAS. I also thank John Wiegley for feedback on "Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac" [GEC18], which is included in slightly-modified form distributed between Chapter 5 and various sections of Chapter 3.

I'd like to thank Karl Palmskog for pointing me at Lamport and Paulson [LP99] and Paulson [Pau18].[4]

I'd like to thank Rajee Agrawal for her help in structuring, wording, and copy-editing of this document.

I'd like to thank Karl Palmskog, Talia Ringer, Ilya Sergey, and Zachary Tatlock for making the high-quality LaTeX bibliography file for "QED at Large: A Survey of Engineering of Formally Verified Software" [Rin+20] available on GitHub[5] and pointing me at it; it's been quite useful in polishing the bibliography of this document.

A significant fraction of the text of this dissertation is taken from papers I've co-authored during my PhD, sometimes with major edits, other times with only minor edits to conform to the flow of the dissertation.

In particular:

?? is largely taken from a draft paper co-authored with Andres Erbsen and Adam Chlipala.

Sections 5.1 and 3.2 are based on the introduction to "Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac" [GEC18]. Chapter 5 is largely taken from [GEC18], with some new text for this dissertation.

Chapter 6 is based largely on "Experience Implementing a Performant Category-Theory Library in Coq" [GCS14], and I'd like to thank Benedikt Ahrens, Daniel R. Grayson, Robert Harper, Bas Spitters, and Edward Z. Yang for feedback on this

[1] `https://github.com/coq/coq/issues/5996#issuecomment-338405694`

[2] `https://github.com/coq/coq/issues/6252`

[3] `https://github.com/coq/coq/issues/5996#issuecomment-670955273`

[4] @palmskog on gitter `https://gitter.im/coq/coq?at=5e5ec0ae4eefc06dcf31943f`

[5] `https://github.com/proofengineering/proofengineering-bib/blob/master/proof-engineering.bib`

paper. Sections 6.3, 6.5, 6.4.1 and 7.2.3 are largely taken from [GCS14]. Some of the text in Subsections 7.2.1 and 7.3.1 also comes from [GCS14].

For those interested in history, our method of reification by parametricity presented in Chapter 5 was inspired by the `evm_compute` tactic [MCB14]. We first made use of `pattern` to allow `vm_compute` to replace `cbv`-with-an-explicit-blacklist when we discovered `cbv` was too slow and the blacklist too hard to maintain. We then noticed that in the sequence of doing abstraction; `vm_compute`; application; β-reduction; reification, we could move β-reduction to the end of the sequence if we fused reification with application, and thus reification by parametricity was born.

This work was supported in part by a Google Research Award and National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584. Work on the category-theory library, one of the case studies presented in Chapter 6 and presented more fully in [GCS14], was supported in part by the MIT bigdata@CSAIL initiative, NSF grant CCF-1253229, ONR grant N000141310260, and AFOSR grant FA9550-14-1-0031. This material is also based upon work supported by the National Science Foundation Graduate Research Fellowship. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Part I

Introduction

Testing shows the presence, not the absence of bugs.

— Edsger Wybe Dijkstra, 1969 [BR70]

If you blindly optimize without profiling, you will likely waste your time on the 99% of code that isn't actually a performance bottleneck and miss the 1% that is.

— Charles E. Leiserson [Lei20]

Chapter 1

Background

1.1 Opportunity

In critical software systems, like the implementations of cryptography supporting the internet, there are opposing pressures to innovate and to let things be as they are. Innovation can help create more performant systems with higher security. But, if the new code has any bugs at all, it could leave the system vulnerable to attacks costing billions of dollars.

Attackers have financial incentive to find any bugs and exploit them, so guaranteeing a complete lack of bugs is essential. Testing, which is the de facto standard for finding bugs, is both expensive and does not guarantee a lack of bugs. For example, some bugs in cryptographic code only occur in as few as 19 out of 2^{255} cases [Lan14]. If we aim to catch such a bug using continuous random testing in a "modest" twenty years, then we would need over a thousand times as many computers as there are atoms in the solar system! This is not an accident. If computers become fast enough to complete this testing in reasonable time, then attackers can use the faster computers to get past the current level of cryptographic protection even if there are no bugs in the code. As a result, however fast computers get, ensuring security will require scaling up the size of the mathematical problem proportionally, and testing will continue to be inadequate at finding all bugs. So, in critical software systems, implementing new, innovative algorithms is a slow and risky process.

An appealing solution to this problem is to *prove* critical software correct. We do this by specifying in formal mathematics the intended behavior of the software and showing a correspondence between our math and the code. Ideally, the specification is relatively simple and easier to trust than the thousands of lines of code. Once we show a correspondence between our math and any new piece of code, we can confidently deploy the software in the world. This is known as verification.

While proofs of algorithms tend to be done with pen and paper (consider the ubiquitous proofs that various sorting algorithms are correct found in introductory algorithms classes), writing proofs of actual code is much harder. Proofs of code correctness tend to be filled with tedious case analysis and only sparse mathematical insights, and attempts to create and check these proofs by hand are subject to the same issues of human fallibility as writing the code in the first place. To avoid the problems from human fallibility, we use proof-checking programs; to cope with tediousness, we use proof assistants which are proof-checking programs that can also help us write the proof itself. Such programs are called *proof assistants*; they assist users in writing code which, when run, generates a proof that can be checked by the proof checker.

But now we are back where we started, with proof-checking programs as our critical software, and the question of how they could possibly give us the confidence to deploy new software in the world.[1] In order to trust proof-checking programs, we want them to be general enough that possible bugs in the proof-checking program are unlikely to line up with mistakes that we make in any individual proof (c.f. Subsection 1.3.1). Such programs which are general enough to check the statements and proofs of arbitrary mathematical theorems are called foundational tools. We also want proof-checking programs to be small so that we have a hope of verifying them by hand (c.f. Subsection 1.3.2).

Proof-checking programs have had many successes, in both software verification and traditional mathematics proof formalization. Examples abound, from compilers [Ler09] to microkernels [Kle+09] to web servers [Chl15] to cryptography code [Erb+19], from the Four-Color Theorem [Gon08] to the Odd-Order Theorem [Gon+13a] to Homotopy Type Theory [Uni13].

However, in almost all examples of software verification successes, there is an enormous overhead in the lines of proof written over the lines of code being verified. The proofs are so long and arduous to write that it typically requires multiple PhDs worth of work to verify one piece of software—see Figure 1-1 for some examples.

In order to utilize verification in the innovation pipeline, we need verification to provide fast feedback about the correctness of code. If we have to write each proof, this is not feasible. One proposal is to automate proof generation so we no longer need to replicate proof-writing effort for iterations of code with the same (or similar) mathematical specification. In this manner, we can decrease the marginal overhead of manual proof writing, by reusing proofs on several variations and optimizations of an algorithm, and deploy new code in the world with confidence.

[1] We cannot solve this problem with programs to check our proof-checking programs, under pain of infinite regress, Gödelian paradoxes [Raa20], and invisible untrustworthiness [Tho84]. However, the mathematically inclined reader might be interested to note that adding one more layer of meta does in fact help, and there are projects underway to verify proof checkers [AR14; Soz+19; Ana+17].

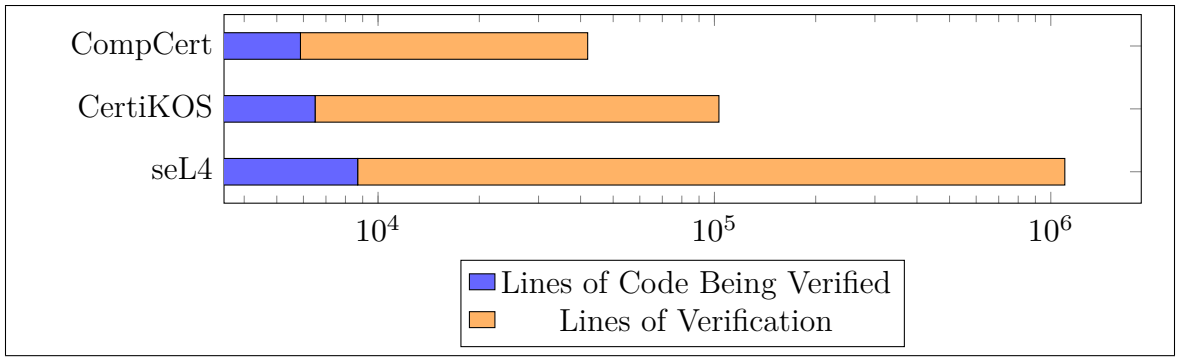


Figure 1-1: Overhead of lines of code of verification over lines of code being verified for some successful projects in microkernels and operating systems (seL4, CertiKOS) and compilers (CompCert). Note the log-scaling.

1.2 Our Work

In the quest to enable automated proof generation, the author encountered several performance bottlenecks in proof checking across projects. We draw particularly from the project in generation of verified low-level cryptographic code in Fiat Cryptography [Erb+19], with auxiliary case studies in category theory [GCS14] and parsers [Gro15a]. Unlike many other performance domains, the time it takes to check proofs as we scale the size of the input is almost always superlinear—quadratic at best, commonly cubic or exponential, occasionally even worse. Empirically, this might look like a proof script that checks in tens of seconds on the smallest of toy examples; takes about a minute on the smallest real-world example, which might be twice the size of the toy example; takes twenty hours to generate and check the proof of an example only twice that size; and, on a (perhaps not-quite-realistic) example twice the size of the twenty-hour example, the script might not finish within a year, or even a thousand years—see Section 2.2 for more details, which we preview here in Figure 1-2. In just three doublings of input size, we might go from tens of seconds to thousands of years. Moreover, in proof assistants, this is not an isolated experience. This sort of performance behavior is common across projects.

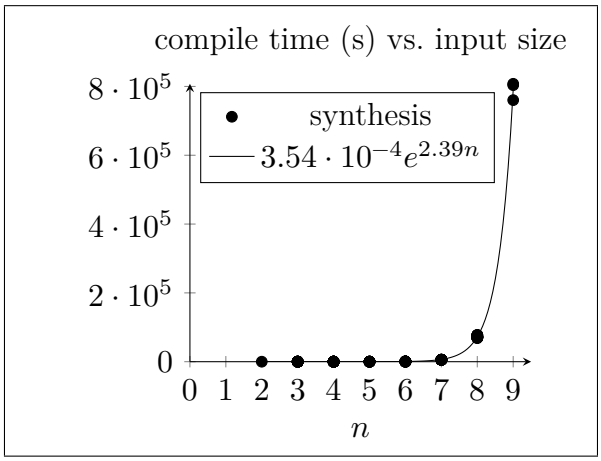


Figure 1-2: Example of superlinear performance scaling

While compiler performance—both the time it takes to compile code and the time it takes to run the generated code—has long been an active field of study [KV17; GBE07; Myt+09], to our knowledge there is no existing body of work systematically investigating the performance of proof assistants, nor even any work primarily fo-

cused on the problem of proof-assistant performance. We distinguish the question of interactive-proof-assistant performance from that of performance of fully automated reasoning tools such as SAT and SMT solvers, on which there has been a great deal of research [Bou94]. As we discuss in Subsection 1.2.1, interactive proof assistants utilize human creativity to handle a greater breadth and depth of problems than fully automated tools, which succeed only on more restricted domains.

This dissertation argues that the problem of *compile-time performance* or *proof-checking performance* is both nontrivial and significantly different from the problem of performance of typical programs. While many papers mention performance, obliquely or otherwise, and some are even driven by performance concerns of a particular algorithm or part of the system, [Gon08, p. 1382; Bou94; GM05; Bra20; Ben89; Pie90; CPG17; PCG18; GL02; Nog02; Bra20], we have not found any that investigate the performance problems that arise *asymptotically* when proof assistants are used to verify programs at large scale.

We present in Part II a research prototype of a tool and methodology for achieving acceptable performance at scale in the domain of term transformation and rewriting. We present in Part III design principles to avoid the performance bottlenecks we encountered and insights about the proof assistants that these performance bottlenecks reveal.

We argue that for proof assistants to scale to industrial uses, *we must get the basics of asymptotic performance of proof checking right*. Through work in this domain, we hope to utilize verification in the software innovation pipeline.

1.2.1 What Are Proof Assistants?

Before diving into the details of performance bottlenecks and solutions, we review the history of formal verification and proof assistants to bring the reader up to speed on the context of our work and investigation.

While we intend to cover a wide swath of the history and development in this subsection, more detailed descriptions can be found in the literature [Rin+20; Geu09; HUW14; HP03; Dar19; Dav01; MR05; Kam02; Moo19; MW13; Gor00; PNW19; Pfe02; Con+86, Related Work]. Ringer et al. [Rin+20, ch. 4] has a particularly clear presentation which was invaluable in assembling this section.

Formal verification can be traced back to the early 1950s [Dar19]. The first formally verified proof, in some sense, achieved in 1954, was of the theorem that the sum of two even numbers is even [Dav01]. The "proof" was an implementation in a vacuum-tube computer of the algorithm of Presburger [Pre29], which could decide, for any first-order formula of natural number arithmetic, whether the formula represented a true theorem or a false one; by implementing this algorithm and verifying that it returns

"true" on a formula such as $\forall a\ b\ x\ y,\ \exists z,\ a = x + x \to b = y + y \to a + b = z + z$, the machine can be said to prove that this formula is true.

While complete decision procedures exist for arithmetic, propositional logic (the fragment of logic without quantifiers, i.e., consisting only of $\to$, $\wedge$, $\vee$, $\neg$, and $\leftrightarrow$), and elementary geometry, there is no complete decision procedure for first-order logic, which allows predicates, as well as universal and existential quantification over objects [Dav01]. In fact, first-order logic is sufficiently expressive to encode the halting problem [Chu36; Tur37]. The problem gets worse in higher-order logic, where we can encode systems that reason about themselves, such as Peano arithmetic, and Gödel's incompleteness theorem proves that there must be some statements which are neither provably true nor provably false. In fact, we cannot even decide which statements are undecidable [Mak11]!

This incompleteness, however, does not sink the project of automated proof search. Consider, for example, the very simple program that merely lists out all possible proofs in a given logical system, halting only when it has found either a proof or a disproof of a given statement. While this procedure will run forever on statements which are neither provably true nor provably false, it will in fact be able to output proofs for all provable statements. This procedure, however, is uselessly slow.

More efficient procedures for proof search exist. Early systems such as the Stanford Pascal Verifier [Luc+79] and Stanford Resolution Prover were based on what is now known as Robinson's resolution rule [Rob65], which, when coupled with syntactic unification, resulted in tolerable performance on sufficiently simple problems [Dav01; Dar19]. A particularly clear description of the resolution method can be found in Shankar [Sha94, pp. 17–18]. In the 1960s, all 400-or-so theorems of Whitehead and Russell's *Principia Mathematica* were automatically proven by the same program [Dav01, p. 9]. However, as the author himself notes, this was only feasible because all of the theorems could be expressed in a way where all universal quantifiers came first, followed by all existential quantifiers, followed by a formula without any quantifiers.

In the early 1970s, Boyer and Moore began work on theorem provers which could work with higher-order principles such as mathematical induction [MW13, p. 6]. This work resulted in a family of theorem provers, collectively known as the Boyer–Moore theorem provers, which includes the first seriously successful automated theorem provers [MW13, p. 8; Dar19]. They developed the Edinburgh Pure LISP Theorem Prover, Thm, and later Nqthm [MW13, p. 8; Boy07; Wik20c], the last of which came to be known as *the* Boyer–Moore theorem prover. Nqthm has been used to formalize and verify Gödel's first incompleteness theorem in 1986 [Sha94; Moo19, p. 29], to verify the implementation of an assembler and linker [Moo07] as well as a number of FORTRAN programs, and to formally prove the invertibility of RSA encryption, the undecidability of the halting problem, Gauss' law of quadratic reciprocity [Moo19, pp. 28–29]. Nqthm later evolved into ACL2 [Moo19; KM20a; KM20b], which has been

used, among other things, to verify a Motorola digital signal processor, the floating-point arithmetic unit in AMD chips, and some x86 machine code programs [Moo19, p. 2].

In 1967, at around the same time that Robinson published his resolution principle, N. G. de Bruijn developed the Automath system [Kam02; Bru94; Bru70; Wik20b]. Unlike the Boyer–Moore theorem provers, Automath checked the validity of sequences of human-generated proof steps and hence was more of a proof checker or proof assistant than an automated theorem prover [Rin+20]. Automath is notable for being the first system to represent both theorems and proofs in the same formal system, reducing the problem of proof checking to that of type checking [Rin+20] by exploiting what came to be known as the Curry–Howard correspondence [Kam02]; we will discuss this more in Subsection 1.3.1. The legacy of Automath also includes de Bruijn indices, a method for encoding function arguments which we describe in Section 3.1.3; dependent types, which we explain in Subsection 1.3.1; and the *de Bruijn principle*—stating that proof checkers should be as small and as simple as possible—which we discuss in Subsection 1.3.2 [Rin+20; Kam02]. We are deferring the explanation of these important concepts for the time being because, unlike the methods of theorem proving described above, these methods are at the heart of Coq, the primary theorem prover used in this thesis, as well as proof assistants like it. One notable accomplishment in the Automath system was the translation and checking of the entirety of Edmund Landau's *Foundations of Analysis* in the early 1970s [Kam02].

Almost at the same time as Boyer and Moore were working on their theorem provers in Edinburgh, Scotland, Milner developed the LCF theorem prover at Stanford in 1972 [Gor00, p. 1]. Written as an interactive proof checker based on Dana Scott's 1969 logic for computable functions (which LCF abbreviates), LCF was designed to allow users to interactively reason about functional programs [Gor00, p. 1]. In 1973, Milner moved to Edinburgh and designed Edinburgh LCF, the successor to Stanford LCF. This new version of LCF was designed to work around two deficiencies of its predecessor: theorem proving was limited by available memory for storing proof objects, and the fixed set of functions for building proofs could not be easily extended. The first of these was solved by what is now called "the LCF approach": by representing proofs with an abstract `thm` type, whose API only permitted valid rules of inference, proofs did not have to be carried around in memory [Gor00, pp. 1–2; Har01]. In order to support abstract data types, Milner et al. invented the language ML (short for "Meta Language") [Gor00, p. 2], the precursor to Caml and later OCaml. The second issue—ease of extensibility—was also addressed by the design of ML [Gor00, p. 2]. By combining an abstract, opaque, trusted API for building terms with a functional programming language, users were granted the ability to combine the basic proof steps into "tactics". Tactics were functions that took in a goal, that is, a formula to be proven, and returned a list of remaining subgoals, together with a function that would take in proofs of those subgoals and turn them into a proof of the overall theorem. An example: a tactic for proving conjunctions might, upon being asked to prove $A \wedge B$, return the two-element list of subgoals $[A, B]$ together

with a function that, when given a proof of A and a proof of B (i.e., when given two `thm` objects, the first of which proves A and the second of which proves B), combines them with a primitive conjunction rule to produce a proof object justifying $A \wedge B$.

In the mid 1980s, Coq [Coq20], the proof assistant which we focus most on in this dissertation, was born from an integration of features and ideas from a number of the proof assistants we've discussed in this subsection. Notably, it was based on the Calculus of Constructions (CoC), a synthesis of Martin-Löf's type theory [Mar75; Mar82] with dependent types and polymorphism, which grew out of Dana Scott's logic of computable functions [Sco93] together with de Bruijn's work on Automath [HP03]. In the late 1980s, some problems were found with the way datatypes were encoded using functions, which lead to the introduction of inductive types and an extension of CoC called the Calculus of Inductive Constructions (CIC) [HP03]. Huet and Paulin-Mohring [HP03] contains an illuminating and comprehensive discussion of how the threads of Coq's development arose in tandem with the history discussed in the preceding paragraphs, and we highly recommend this read for those interested in Coq's history.

Major accomplishments of verification in Coq include the fully verified optimizing C compiler CompCert [Ler09], the proof of the Four Color Theorem [Gon08], and the complete formalization of the Odd Order Theorem, also known as the Feit–Thompson Theorem [Gon+13a]. This last development was the result of about six years of work formalizing a proof that every finite group of odd order is solvable; the original proof, published in the early 1960s, is about 225 pages long.

We now briefly mention a number of other proof assistants, calling out some particularly significant accomplishments of verification. Undoubtedly we will miss some proof assistants and accomplishments, for which we refer the reader to the rich existing literature, some of which is cited in the first paragraph of this subsection, as well as scattered among other papers which describe a variety of proof assistants [Wie09].

Inspired by Automath, the Mizar [Har96a; Rud92; MR05] proof checker was designed to assist mathematicians in preparing mathematical papers [Rud92]. The Mizar Mathematical Library already had 55 thousand formally verified lemmas in 2009 and was at the time (and might still be) the largest library of formal mathematics [Wie09]. LCF [Gor00; GMW79; Gor+78] spawned a number of other closely related proof assistants, such as HOL [Bar00; Gor00], Isabelle/HOL [PNW19; Wen02; NPW02; Pau94], HOL4 [SN08], and HOL Light [Har96c]. Among other accomplishments, a complete OS microkernel, seL4, was fully verified in Isabelle/HOL by 2009 [Kle+09]. In 2014, a complete proof of the Kepler conjecture on optimal packing of spheres was formalized in a combination of Isabelle and HOL Light [Hal06; Hal+14]. The functional programming language CakeML includes a self-bootstrapping optimizing compiler which is fully verified in HOL [Kum+14]. The Nqthm Boyer–Moore theorem prover eventually evolved into ACL2 [KM20b; KM20a]. Other proof assistants include LF [Pfe91; HHP93; Pfe02], Twelf [PS99], Matita [Asp+07; Asp+11],

PVS [Sha96; ORS92], LEGO [Pol94], and NuPRL [Con+86].

1.3 Basic Design Choices

Although the design space of proof assistants is quite large, as we've touched on in Subsection 1.2.1, there are only two main design decisions which we want to assume for the investigations of this thesis. The first is the use of dependent type theory as a basis for formal proofs, as is done in Automath [Wik20b; Bru70; Bru94], Coq [Coq20], Agda [Nor09], Idris [Bra13], Lean [Mou+15], NuPRL [Con+86], Matita [Asp+11], and others, rather than on some other logic, as is done in LCF [Gor00; GMW79; Gor+78], Isabelle/HOL [PNW19; Wen02; NPW02; Pau94], HOL4 [SN08], HOL Light [Har96c], LF [Pfe91; HHP93], and Twelf [PS99], among others. The second is the *de Bruijn criterion*, mandating independent checking of proofs by a small trusted kernel [BW05]. We have found that many of the performance bottlenecks are fundamentally a result of one or the other of these two design decisions. Readers are advised to consult Ringer et al. [Rin+20, ch. 4] for a more thorough mapping of the design axes.

In this section, we will explain these two design choices in detail; by the end of this section, the reader should understand what each design choice entails and, we hope, why these are reasonable choices to make.

1.3.1 Dependent Types: What? Why? How?

There are, broadly, three schools of thought on what is a *proof*. Geuvers [Geu09] describe two roles that a proof plays:

(i) A proof *convinces* the reader that the statement is correct.

(ii) A proof *explains* why the statement is correct.

A third conception of proof is that a proof is itself a mathematical object or construction which corresponds to the content of a particular theorem [Bau13]. This third conception dates back to the school of intuitionism of Brouwer in the early 1900s and of constructive mathematics of Bishop in the 1960s; see Constable et al. [Con+86, Related Works] for a tracing of the history from Brouwer to Martin-Löf, whose type theory is at the heart of Coq and similar proof assistants.

This third conception of proof admits formal frameworks where proof and computation are unified as the same activity. As we'll see shortly, this allows for drastically smaller proofs.

The foundation of unifying computation and proving is, in some sense, the *Curry–Howard–de Bruijn correspondence*, more commonly known as the Curry–Howard correspondence or the Curry–Howard isomorphism. This correspondence establishes the relationship between types and propositions, between proofs and computational objects.

The reader may be familiar with types from programming languages such as C/C++, Java, and Python, all of which have types for strings, integers, and lists, among others. A *type* denotes a particular collection of objects, called its *members*, *inhabitants*, or *terms*. For example, `0` is a term of type `int`, `"abc"` is a term of type `string`, and `true` and `false` are terms of type `bool`. Types define how terms can be built and how they can be used. New natural numbers, for example, can be built only as zero or as the successor of another natural number; these two ways of building natural numbers are called the type's *constructors*. Similarly, the only ways to get a new Boolean are by giving either `true` or `false`; these are the two constructors of the type `bool`. Note that there are other ways to get a Boolean, such as by calling a function that returns Booleans, or by having been given a Boolean as a function argument. The constructors define the only Booleans that exist at the end of the day, after all computation has been run. This uniqueness is formally encoded by the *eliminator* of a type, which describes how to use it. The eliminator on `bool` is the `if` statement; to use a Boolean, one must say what to do if it is `true` and what to do if it is `false`. Some eliminators encode recursion: to use a natural number, one must say what to do if it is zero and also one must say what to do if it is a successor. In the case where the given number is the successor of n, however, one is allowed to call the function recursively on n. For example, we might define the factorial function as

```
fact m =
  case m of
    zero   -> succ zero
    succ n -> m * fact n
```

Eliminators in programming correspond to *induction* and case analysis in mathematics. To prove a property of all natural numbers, one must prove it of zero, and also prove that if it holds for any number n, then it holds for the successor of n. Here we see the first glimmer of the the Curry–Howard isomorphism, which identifies each type with the set of terms of that type, identifies each proposition with the set of proofs of that proposition, and thereby identifies terms with proofs.

Table 1.1 shows the correspondence between programs and proofs. We have already seen how recursion lines up with induction in the case of natural numbers; let us look now at how some of the other proof rules correspond.

To prove a conjunction $A \wedge B$, one must prove A and also prove B; if one has a proof of the conjunction $A \wedge B$, one may assume both A and B have been proven. This

computation	set theory	logic
type	set	proposition
term / program	element of a set	proof
eliminator / recursion		case analysis / induction
type of pairs	Cartesian product ($\times$)	conjunction ($\wedge$)
sum type ($+$)	disjoint union ($\sqcup$)	disjunction ($\vee$)
function type	set of functions	implication ($\to$)
unit type	singleton set	trivial truth
empty type	empty set ($\emptyset$)	falsehood
dependent function type (Π)		universal quantification ($\forall$)
dependent pair type (Σ)		existential quantification ($\exists$)

Table 1.1: The Curry–Howard Correspondence

lines up exactly with the type of pairs: to inhabit the type of pairs $A \times B$, one must give an object of type A paired with an object of type B; given an object of the pair type $A \times B$, one can *project* out the components of types A and B.

To prove the implication $A \to B$, one must prove B under the assumption that A holds, i.e., that a proof of A has been given. The rule of *modus ponens* describes how to use a proof of $A \to B$: if also a proof of A is given, then B may be concluded. These correspond exactly to the construction and application of functions in programming languages: to define a function of type $A \to B$, the programmer gets an argument of type A and must return a value of type B. To use a function of type $A \to B$, the programmer must *apply* the function to an argument of type A; this is also known as *calling* the function.

Here we begin to see how type checking and proof checking can be seen as the same task. The process of *type checking* a program consists of ensuring that every variable is given a type, that every expression assigned to a variable has the type of that variable, that every argument to a function has the correct type, etc. If we write the Boolean negation function which sends `true` to `false` and `false` to `true` by case analysis (i.e., by an `if` statement), the type checker will reject our program if we try to apply it to, say, an argument of type `string` such as `"foo"`. Similarly, if we try to use *modus ponens* to combine a proof that $x = 1 \to 2x = 2$ with a proof that $x = 2$ to obtain a proof that $2x = 2$, the proof checker should complain that $x = 1$ and $x = 2$ are not the same type.

While the correspondence of the unit type to tautologies is relatively trivial, the correspondence of the empty type to falsehood encodes nontrivial principles. By encoding falsehood as the empty type, the principle of explosion—that from a contradiction, everything follows—can be encoded as case analysis on the empty type.

The last two rows of Table 1.1 are especially interesting cases which we will now cover.

Some programming languages allow functions to return values whose types depend on the *value*s of the functions' arguments. In these languages, the types of arguments are generally also allowed to depend on the values of previous arguments. Such languages are said to support dependent types. For example, we might have a function that takes in a Boolean and returns a string if the Boolean is `true` but an integer if the Boolean is `false`. More interestingly, we might have a function that takes in two Booleans and additionally takes in a third argument which is of type `unit` whenever the two Booleans are either both `true` or both `false` but is of type `empty` when they are not equal. This third argument serves as a kind of proof that the first two arguments are equal. By checking that the third argument is well-typed, that is, that the single inhabitant of the `unit` type is passed only when in fact the first two arguments are equal, the type checker is in fact doing proof checking. While compilers of languages like C++, which supports dependent types via templates, can be made to do rudimentary proof checking in this way, proof assistants such as Coq are built around such dependently typed proof checking.

The last two lines of Table 1.1 can now be understood.

A dependent function type is just one whose return value depends on its arguments. For example, we may write the nondependently typed function type

$$\texttt{bool} \to \texttt{bool} \to \texttt{unit} \to \texttt{unit}$$

which takes in three arguments of types `bool`, `bool`, and `unit` and returns a value of type `unit`. Note that we write this function in curried style, with $\to$ associating to the right (i.e., $A \to B \to C$ is $A \to (B \to C)$), where a function takes in one argument at a time and returns a function awaiting the next argument. This function is not very interesting, since it can only return the single element of type `unit`.

However, if we define $E(b_1, b_2)$ to be the type `if` $\texttt{b}_1$ `then (if` $\texttt{b}_2$ `then unit else empty) else (if` $\texttt{b}_2$ `then empty else unit)`, i.e., the type which is `unit` when both are `true` or both are `false` and is `empty` otherwise, then we may write the dependent type

$$(b_1 : \texttt{bool}) \to (b_2 : \texttt{bool}) \to E(b_1, b_2) \to E(b_2, b_1)$$

Alternate notations include

$$\Pi_{b_1:\texttt{bool}} \Pi_{b_2:\texttt{bool}} E(b_1, b_2) \to E(b_2, b_1)$$

and

$$\forall (b_1 : \texttt{bool})(b_2 : \texttt{bool}), E(b_1, b_2) \to E(b_2, b_1).$$

A function of this type witnesses a proof that equality of Booleans is symmetric.

Similarly, dependent pair types witness existentially quantified proofs. Suppose we

have a type $T(n)$ which encodes the statement "n is prime and even". To prove $\exists n, T(n)$, we must provide an explicit n together with a proof that it satisfies T. This is exactly what a dependent pair is: $\Sigma_n T(n)$ is the type of consisting of a pair of a number n paired with a proof that that particular n satisfies T.

As we mentioned above, one feature of basing a proof assistant on dependent type theory is that computation can be done at the type level, without leaving a trace in the proof term. Many proofs require intermediate arguments based solely on the computation of functions. For example, a proof in number theory or cryptography might depend on the fact that a particularly large number, raised to some large power, is congruent to 1 modulo some prime. As argued by Stampoulis [Sta13], if we are required to record all intermediate computation steps in the proof term, they can become prohibitively large. The *Poincaré principle* asserts that such arguments should not need to be recorded in formal proofs but should instead be automatically verified by appeal to computation [BG01, p. 1167]. The ability to appeal to computation without blowing up the size of the proof term is quite important for so-called reflective (or reflexive) methods of proof, described in great detail in Chapter 3.

Readers interested in a more comprehensive explanation of dependent type theory are advised to consult Chapter 1 (Type theory) and Appendix A (Formal type theory) of Univalent Foundations Program [Uni13]. Readers interested in perspectives on how dependent types may be disadvantageous are invited to consult literature such as Lamport and Paulson [LP99] and Paulson [Pau18].

1.3.2 The de Bruijn Criterion

> A Mathematical Assistant satisfying the possibility of independent checking by a small program is said to satisfy the *de Bruijn* criterion.
>
> — Henk Barendregt [BW05]

As described in the beginning of this chapter, the purpose of proving our software correct is that we want to be able to trust that it has no bugs. Having a proof checker reduces the problem of software correctness to the problem of the correctness of the specification, together with the correctness of the proof checker. If the proof checker is complicated and impenetrable, it might be quite reasonable not to trust it.

Proof assistants satisfying the de Bruijn criterion are, in general, more easily trustable than those which violate it. The ability to check proofs with a small program, divorced from any heuristic programs and search procedures which generate the proof, allows trust in the proof to be reduced to trust in that small program. Sufficiently small and well-written programs can more easily be inspected and verified.

The proof assistant Coq, which is the primary proof assistant we consider in this

dissertation, is a decent example of satisfying the de Bruijn criterion. There is a large untrusted codebase which includes the proof scripting language $\mathcal{L}_{tac}$, used for generating proofs and doing type inference. There's a much smaller kernel which checks the proofs, and Coq is even shipped with a separate checker program, `coqchk`, for checking proof objects saved to disk. Moreover, in the past year, a checker for Coq's proof objects has been implemented in Coq itself and formally verified with respect to the type theory underlying Coq [Soz+19].

Note that the LCF approach to theorem proving, where proofs have an abstract type and type safety of the tactics guarantees validity of the proof object, forms a sort-of complementary approach to trust.

1.4 Look Ahead: Layout and Contributions of the Thesis

In the remainder of Part I, we will finish laying out the landscape of performance bottlenecks we encountered in dependently typed proof assistants; Chapter 2 (The Performance Landscape in Type-Theoretic Proof Assistants) gives a more in-depth investigation into what makes performance optimization in dependent type theory hard, different, and unique, followed by describing major axes of superlinear performance bottlenecks in Section 2.6 (The Four Axes of the Landscape).

Part II (Program Transformation and Rewriting) is devoted, in some sense, to performance bottlenecks that arise from the de Bruijn criterion of Subsection 1.3.2. We investigate one particular method for avoiding these performance bottlenecks. We introduce this method, variously called *proof by reflection* or *reflective automation*, in Chapter 3 (Reflective Program Transformation), with a special emphasis on a particularly common use case—transformation of syntax trees. **??** (**??**) describes our original contribution of a framework for leveraging reflection to perform rewriting and program transformation at scale, driven by our need to synthesize efficient, proven-correct, low-level cryptographic primitives [Erb+19]. Where **??** addresses the performance challenges of verified or proof-producing program transformation, Chapter 4 (Engineering Challenges in the Rewriter) is a deep-dive into the performance challenges of engineering the tool itself and serves as a sort-of microcosm of the performance bottlenecks previously discussed and the solutions we've proposed to them. Unlike the other chapters of this dissertation, Chapter 4 at times assumes a great deal of familiarity with the details of the Coq proof assistant. Finally, Chapter 5 (Reification by Parametricity) presents a way to efficiently, elegantly, and easily perform *reification*, the first step of proof by reflection, which is often a bottleneck in its own right. We discovered—or invented—this trick in the course of working on our library for synthesis of low-level cryptographic primitives [Erb+19; GEC18].

Part III (API Design) is devoted, by and large, to the performance bottlenecks that arise from the use of dependent types as the basis of a proof assistant as introduced in Subsection 1.3.1; in Chapter 6 (Abstraction), we discuss lessons on engineering libraries at scale drawn from our case study in formalizing category theory and augmented by our other experience. Many of the lessons presented here are generalizations of examples described in Chapter 4 (Engineering Challenges in the Rewriter). The category-theory library formalized as part of this doctoral work, available at HoTT Library Authors [HoT20], is described briefly in this chapter; a more thorough description can be found in the paper we published on our experience formalizing this library [GCS14].

Part IV (Conclusion) is in some sense the mirror image of Part I: Where Chapter 2 is a broad look at what is currently lacking and where performance bottlenecks arise, Chapter 7 (A Retrospective on Performance Improvements) takes a historical perspective on what advancements have already been made in the performance of proof assistants and Coq in particular. Finally, while the present chapter which we are now concluding has looked back on the present state and history of formal verification, Chapter 8 (Concluding Remarks) looks forward to what we believe are the most important next steps in the perhaps-nascent field of proof-assistant performance at scale.

Chapter 2

The Performance Landscape in Type-Theoretic Proof Assistants

2.1 Introduction

As alluded to in Section 1.1, when writing nonautomated proofs to verify code in a proof assistant, the number of lines of proof we need to write scales with the number of lines of code being verified, typically resulting to a 10× to 100× overhead. In this strategy, proof generation and proof checking times are reasonable, often scaling linearly with the number of lines of proof. Automating the generation of proofs resolves the issue of overhead of proof-writing time. Automation significantly decreases the marginal cost of proving theorems about new code that is similar enough to code already verified. However, it introduces massive *nonlinear* overhead in the time it takes for the computer to generate and check the proof.

The main contribution of this thesis, presented in **??**, is a tool that solves this problem of unacceptable overhead in proof-generation and proof-checking for the Fiat Cryptography project [Erb+19] and which we believe is broadly applicable to other domains.

In building this work, the author developed a deep understanding of the performance bottlenecks faced and addressed. This chapter lays out the groundwork for understanding these performance bottlenecks, where they come from, and how our solution addressed the relevant performance issues. We describe what we have seen of the landscape of performance issues in proof assistants and provide a map for navigation. Our hope is that readers will be able to apply our map to performance bottlenecks they encounter in dependently typed tactic-driven proof assistants like Coq.

2.2 Exponential Domain

We sketch out the main differences between performance issues we've encountered in dependently typed proof assistants and performance issues in other languages. Some of these differences are showcased through a palette of real performance issues that have arisen in Coq.

The widespread commonsense in performance engineering is that good performance optimization happens in a particular order: there is no use micro-optimizing code when implementing an algorithm with unacceptable performance characteristics; imagine trying to optimize the pseudorandom number generator used in bogosort [GHR07], for example.[1] Similarly, there is no use trying to find or create a better algorithm if the problem being solved is more complicated than it needs to be; consider, for example, the difference between ray tracers and physics simulators. Ray tracers determine what objects can be seen from a given point by drawing lines from the viewpoint to the objects and seeing if they pass through any other objects "in front of" them. Alternatively, it is possible to provide a source of light waves and simulate the physical interaction of light with the various objects, to determine what images remain when the light arrives at a particular point. There's no use trying to find an efficient algorithm for simulating quantum electrodynamics, though, if all that is needed is to answer "which parts of which objects need to be drawn on the screen?"

One essential ingredient for this division of concerns—between specifying the problem, picking an efficient algorithm, and optimizing the implementation of the algorithm—is knowledge of what a typical set of inputs looks like and what the scope looks like. When sorting a list, we know that the length of the list and the initial ordering matter; for sorting algorithms that work for sorting lists with any type of elements, it generally doesn't matter, though, whether we're sorting a list of integers or colors or names. Furthermore, randomized datasets tend to be reasonably representative for list ordering, though we may also care about some special cases, such as already-sorted lists, nearly sorted lists, and lists in reverse-sorted order. We can say that sorting is always possible in $\mathcal{O}(n \log n)$ time, and that's a pretty good starting point.

In Coq, and other dependently typed proof assistants, this ingredient is missing. The domain is much larger: in theory, we want to be able to check any proof anyone might write. Furthermore, in dependently typed proof assistants, the worst-case behavior is effectively unbounded, because any provably terminating computation can be run at typechecking time.

In fact, this issue already arises for compilers of mainstream programming languages. The C++ language, for example, has `constexpr` constructions that allow running arbitrary computation at compile time, and it's well-known that C++ templates can

[1] Bogosort, whose name is a portmanteau of the words bogus and sort [Ray03], sorts a list by randomly permuting the list over and over until it is sorted.

incur a large compile-time performance overhead. However, we claim that, in most languages, even as programs scale, these performance issues are the exception rather than the rule. Most code written in C or C++ does not hit unbounded compile-time performance bottlenecks. Generally, for code that compiles in a reasonable amount of time, as the codebase size is scaled up, compile time will creep up linearly.

In Coq, however, the scaling story is very different. Compile time scales superlinearly with example size. Frequently, users will cobble together code that works to prove a toy version of some theorem or to verify a toy version of some program. By virtue of the fact that humans are impatient, the code will execute in reasonable time, perhaps a couple seconds, on the toy version. The user will then apply the same proof technique on a slightly larger example, and the proof-checking time will often be pretty similar. After scaling the example a bit more, the proof-checking time will be noticeably slow—maybe it now takes a couple of minutes. Suddenly, though, scaling the example just a tiny bit more will result in the compiler not finishing even if we let it run for a day or more. This is what working in an exponential performance domain is like.

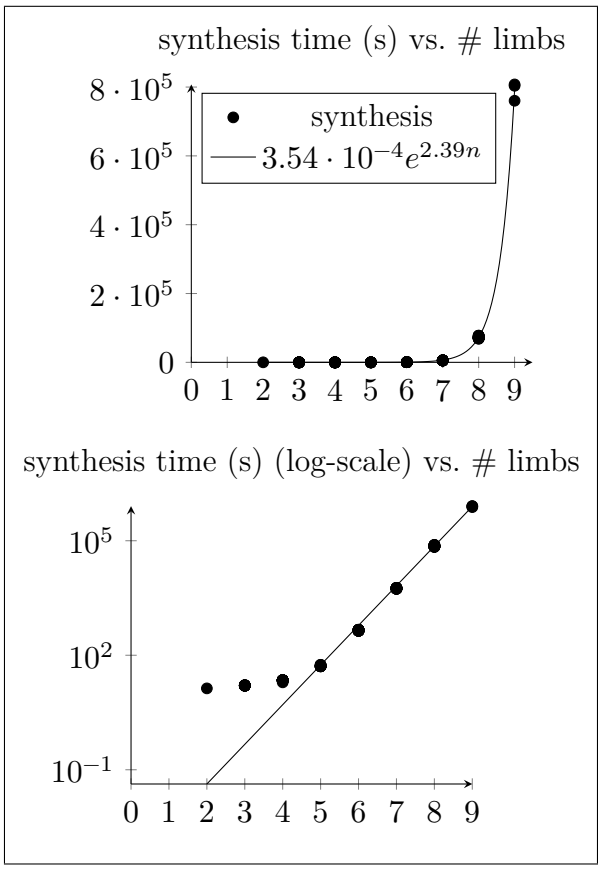


Figure 2-1: Synthesizing Subtraction

To put numbers on this, let us consider an example from Fiat Cryptography which involved generating C code to do arithmetic on very large numbers. The code generation was parameterized on the number of machine words needed to represent a single big integer. Our smallest toy example used two machine words; our largest example used 17. The smallest toy example took about 14 seconds. Based on the the compile-time performance of about a hundred examples, we expect the largest example would have taken over four thousand *millennia*! See Figure 2-1. (Our primary nontoy test example used four machine words and took just under a minute; the biggest realistic example we were targeting was twice that size, at eight machine words, and took about 20 hours.)

2.3 Motivating the Performance Map

In most performance domains, solutions to performance bottlenecks are generally either hyper specialized to the code being optimized or the domain of the algorithm, or else they are so general as to be applicable to all performance engineering. For example, solving a performance issue might involving caching the result of a particular computation; caching is a very general solution, while the particular computation being run is hyper specialized. Once factored like this, there is generally no remaining insight to be had about the particular performance bottleneck encountered. In our experience with proof assistants, most performance bottlenecks are far from the code being written and the domain being investigated and are yet also far from general performance engineering.

The example above is an instance of a performance bottleneck which is neither specific the domain (in our case, cryptographic code generation) nor general enough to apply to performance engineering outside of proof assistants.

Where *is* the bottleneck? Maybe, one might ask, were we generating unreasonable amounts of code? Each example using n machine words generated $3n$ lines of code. How can exponential performance result from linear code?

Our method involved two steps: first generate the code, then check that the generated code matches with what comes out of the verified code generator. This may seem a bit silly, but it is actually somewhat common; in a theorem that says "any code that comes out of this code generator satisfies this property", we need a proof that the code we feed into the theorem actually came out of the specified code generator, and the easiest way to prove this is, roughly, to tell the proof assistant to just check that fact for you. (It's possible to be more careful and not do the work twice, but this often makes the code a bit harder to read and understand and is oftentimes pointless; premature optimization is the root of all evil, as they say.) Furthermore, because we often don't want to fully compute results when checking that two expressions are equal—just imagine having to compute the factorial of 1000 just to check that 1000! is equal to itself—the default method for checking that the code came out of the code generator is different from the method we used to compute the code in the first place.

It turns out that the actual code generation took less than 0.002% of the total time on the largest examples we tested (just 14 seconds out of about 211 hours). The rest of the time was spent checking that the generated code in fact matched what comes out of the verified code generator.

2.4 Performance Engineering in Proof Assistants Is Hard

The fix to the example is itself quite simple, being only 21 characters long.[2] However, tracking down this solution was quite involved, requiring the following pieces:

1. A good profiling tool for proof scripts (see Subsection 7.1.3). This is a standard component of a performance engineer's toolkit, but when I started my PhD, there was no adequate profiling infrastructure for Coq. While such a tool is essential for performance engineering in all domains, what's unusual about dependently typed proof assistants, I claim, is that essentially *every* codebase that needs to scale runs into performance issues, and furthermore these issues are frequently total blockers for development because so many of them are exponential in nature.

2. Understanding the details of how Coq works under-the-hood. Conversion, the ability to check if two types or terms are the same, is one of the core components of any dependently typed proof assistant. Understanding the details of how conversion works is generally not something users of a proof assistant want to worry about; it's like asking C programmers to keep in mind the size of `gcc`'s maximum nesting level for `#include`'d files[3] when writing basic programs. It's certainly something that advanced users need to be aware of, but it's not something that comes up frequently.

3. Being able to run the proof assistant in your head. When I looked at the conversion problem, I knew immediately what the most likely cause of the performance issue was, but this is because I've managed to internalize most of how Coq runs in my head.

   This might seem reasonable at a glance; one expects to have to understand the system being optimized in order to optimize it. However, the knowledge required here is hard-won and not easily accessible. While I've managed to learn the details of what Coq is doing—including performance characteristics—basically without having to read the source code at all, the relevant performance characteristics are not documented anywhere and are not even easily interpretable from the source code of Coq. This is akin to, say, being able to learn how `gcc` represents various bits of C code, what transformations it does in what order, and what performance characteristics these transformations have, just from using `gcc` to compile C code and reading the error messages it gives you. These are details that should not need to be exposed to the user, but because dependent type theory is so complicated—complicated enough that it's generally assumed that users will get *line-by-line interactive feedback from the compiler* while developing—the numerous design decisions and seemingly

[2] `Strategy 1 [Let_In].` for those who are curious.

[3] It's 200, for those who are curious [Fre17].

reasonable defaults and heuristics lead to subtle performance issues. Note, furthermore, that this performance issue is essentially about the algorithm used to implement conversion and is not even sensible when only talking about only the spec of what it means for two terms to be convertible.

Furthermore, note that the requirement of being able to run the typechecker in one's head is essentially the statement that the entire implementation is part of the specification.[4]

4. Knowing how to tweak the built-in defaults for parts of the system which most users expect to be able to treat as black-boxes.

Note that even after this fix, the performance is *still* exponential! However, the performance is good enough that we deemed it not currently worth digging into the profile to understand the remaining bottlenecks. See Figure 2-2.

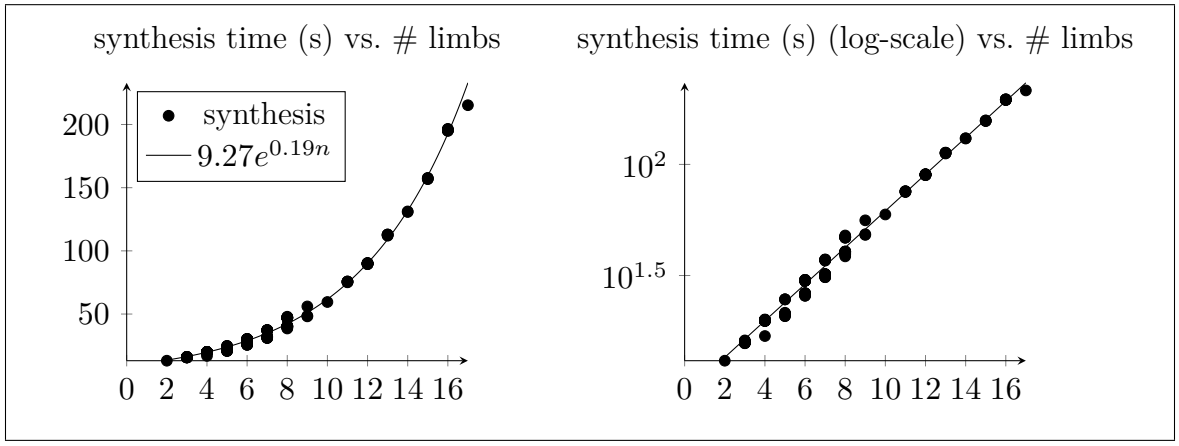


Figure 2-2: Timing of synthesizing subtraction after fixing the bottleneck

2.5 Fixing Performance Bottlenecks in the Proof Assistant Itself Is Also Hard

In many domains, the performance challenges have been studied and understood, resulting in useful decompositions of the problem into subtasks that can be optimized independently. It's rarely the case that disparate parts of a codebase must be simultaneously optimized to see any performance improvement at all.

We have not found any such study of performance challenges in proof assistants. It seems to us that there are many disparate parts of any proof assistant satisfying the de Bruijn criterion which are deeply coupled and which cannot be performance-optimized independently. There are many seemingly reasonable implementation choices that

[4]Thanks to Andres Erbsen for pointing this out to me.

can be made for the kernel—the trusted proof checker—which make performance-optimizing the proof engine, which generates the proof, next to impossible. Worse, if performance optimization is done incrementally, to avoid needless premature optimization, then it can be the case that performance-optimizing the kernel has effectively no visible impact; the most efficient proof-engine design for the slower kernel might be inefficient in ways that prevent optimizations in the kernel from showing up in actual use cases, because simple proof-engine implementations tend to avoid the performance bottlenecks of the kernel while simultaneously shadowing them with bottlenecks with similar performance characteristics.

2.6 The Four Axes of the Landscape

We've now seen what superlinear scaling in dependently typed proof assistants looks like. We've covered general arguments for why proof assistants might have such scaling and what we believe broadly underpins the challenges of performance engineering in and on proof assistants.

The rest of this chapter is devoted to mapping out the landscape of performance bottlenecks we've encountered in a way that we hope will illuminate structure in the performance bottlenecks which are neither specific to the domain of the proof being checked nor general to all performance engineering. We present a map of performance bottlenecks comprising four axes. These axes are by no means exhaustive, but, in our experience, most interesting performance bottlenecks scale as a superlinear factor of one or more of these axes.

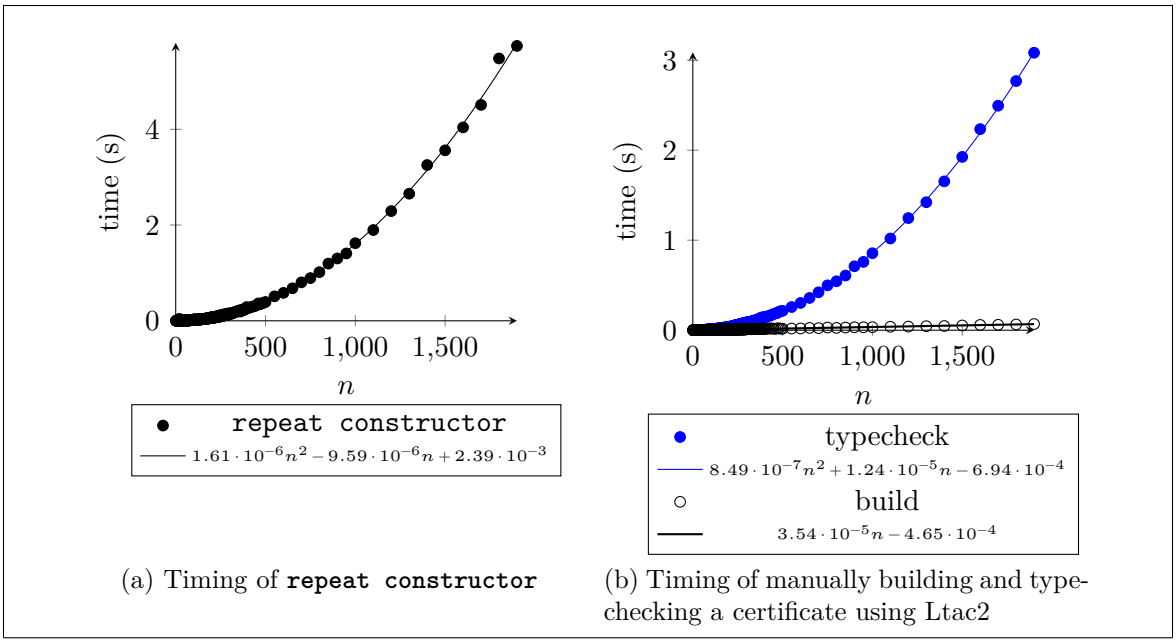
2.6.1 The Size of the Type

We start with one of the simplest axes.

Suppose we want to prove a conjunction of n propositions, say, $\texttt{True} \wedge \texttt{True} \wedge \cdots \wedge \texttt{True}$. For such a simple theorem, we want the size of the proof, and the time and memory complexity of checking it, to be linear in n.

Recall from Subsection 1.3.2 that we want a separation between the small trusted part of the proof assistant and the larger untrusted part. The untrusted part generates certificates, which in dependently typed proof assistants are called terms, which the trusted part, the kernel, checks.

The obvious certificate to prove a conjunction $A \wedge B$ is to hold a certificate a proving A and a certificate b proving B. In Coq, this certificate is called `conj` and it takes four parameters: A, B, $a : A$, and $b : B$. Perhaps the reader can already spot the problem.



(a) Timing of `repeat constructor`

(b) Timing of manually building and type-checking a certificate using Ltac2

Figure 2-3: Proving a conjunction of n `True`s

To prove a conjunction of n propositions, we end up repeating the type n times in the certificate, resulting in a term that is quadratic in the size of the type. We see in Figure 2-3a the time it takes to do this in Coq's tactic mode via **`repeat constructor`**. If we are careful to construct the certificate manually without duplicating work, we see that it takes linear time for Coq to build the certificate and quadratic time for Coq to check the certificate; see Figure 2-3b.

Note that for small and even medium-sized examples, it's pretty reasonable to do duplicative work. It's only when we reach very large examples that we start hitting nonlinear behavior.

There are two obvious solutions for this problem:

1. We can drop the type parameters from the `conj` certificates.
2. We can implement some sort of sharing, where common subterms of the type only exist once in the representation.

Dropping Type Parameters: Nominal vs. Structural Typing

The first option requires that the proof assistant implement structural typing rather than nominal typing [Pie02, 19.3 Nominal and Structural Type Systems]. Note that it doesn't actually require structural; we can do it with nominal typing if we enforce everywhere that we can only compare terms who are known to be the same type, because not having structural typing results in having a single kernel term with multiple

nonunifiable types. Morally, the reason for this is that if we have an inductive record type whose fields do not constrain the parameters of the inductive type family, then we need to consider different instantiations of the same inductive type family to be convertible. That is, if we have a phantom record such as

```
Record Phantom (A : Type) := phantom {}.
```

and our implementation does not include `A` as an argument to `phantom`, then we must consider `phantom` to be both of type `Phantom nat` and `Phantom bool`, even though `nat` and `bool` are not the same. I have requested this feature in Coq issue #5293. Note, however, that sometimes it is important for such phantom types to be considered distinct when doing type-level programming.

Sharing

The alternative to eliminating the duplicative arguments is to ensure that the duplication is at-most constant sized. There are two ways to do this: either the user can explicitly share subterms so that the size of the term is in fact linear in the size of the goal, or the proof assistant can ensure maximal sharing of subterms.

There are two ways for the user to share subterms: using `let` binders and using function abstraction. For example, rather than writing

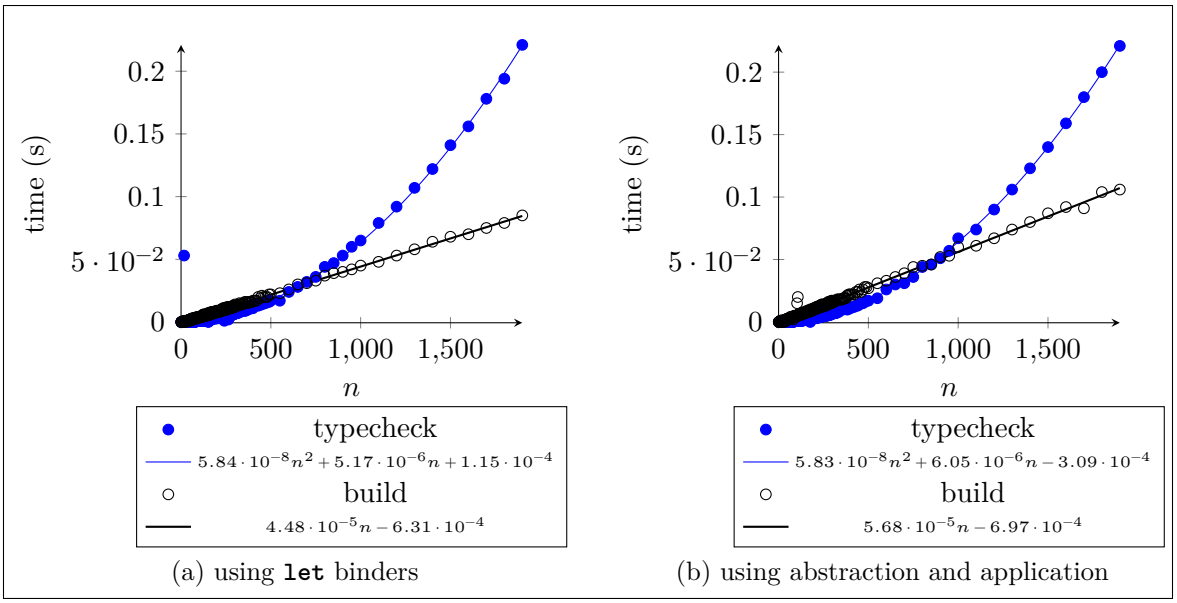
```
@conj True (and True (and True True))
      I
      (@conj True (and True True) I (@conj True True I I))
```

and having roughly n^2 occurrences[5] of `True` when we are trying to prove a conjunction of n `True`s, the user can instead write

```
let T0 : Prop := True in
let v0 : T0   := I in
let T1 : Prop := and True T0 in
let v1 : T1   := @conj True T0 I v0 in
let T2 : Prop := and True T1 in
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

which has only n occurrences of `True`. Alternatively, the user can write

[5]The exact count is $n(n+1)/2 - 1$.



(a) using `let` binders

(b) using abstraction and application

Figure 2-4: Timing of manually building and typechecking a certificate to prove a conjunction of n `True`s using Ltac2

```
(λ (T0 : Prop) (v0 : T0),
  (λ (T1 : Prop) (v1 : T1),
    (λ (T2 : Prop) (v2 : T2), @conj True T2 I v2)
      (and True T1) (@conj True T1 I v1))
    (and True T0) (@conj True T0 I v0))
  True I
```

Unfortunately, both of these incur quadratic typechecking cost, even though the size of the term is linear. See Figure 2-4.

Recall that the typing rules for λ and `let` are as follows:

$$\frac{\Gamma, x : A \vdash f : B}{\Gamma \vdash (\lambda(x : A), f) : \forall x : A, B}$$

$$\frac{\Gamma \vdash f : \forall x : A, B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A := a \vdash f : B}{\Gamma \vdash (\texttt{let}\ \ x : A := a\ \ \texttt{in}\ \ f) : B[a/x]}$$

Let us consider the inferred types for the intermediate terms when typechecking the `let` expression:

- We infer the type `and True T2` for the expression

```
@conj True T2 I v2
```

- We perform the no-op substitution of `v2` into that type to type the expression

```
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

- We substitute `T2 := and True T1` into this type to get the type `and True (and True T1)` for the expression

```
let T2 : Prop := and True T1 in
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

- We perform the no-op substitution of `v1` into this type to get the type for the expression

```
let v1 : T1   := @conj True T0 I v0 in
let T2 : Prop := and True T1 in
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

- We substitute `T1 := and True T0` into this type to get the type `and True (and True (and True T0))` for the expression

```
let T1 : Prop := and True T0 in
let v1 : T1   := @conj True T0 I v0 in
let T2 : Prop := and True T1 in
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

- We perform the no-op substitution of `v0` into this type to get the type for the expression

```
let v0 : T0   := I in
let T1 : Prop := and True T0 in
let v1 : T1   := @conj True T0 I v0 in
let T2 : Prop := and True T1 in
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

- Finally, we substitute `T0 := True` into this type to get the type `and True (and True (and True True))` for the expression

```
let T0 : Prop := True in
let v0 : T0   := I in
let T1 : Prop := and True T0 in
let v1 : T1   := @conj True T0 I v0 in
let T2 : Prop := and True T1 in
```

```
let v2 : T2   := @conj True T1 I v0 in
@conj True T2 I v2
```

Note that we have performed linearly many substitutions into linearly sized types, so unless substitution is constant-time in size of the term into which we're substituting, we incur quadratic overhead here. The story for function abstraction is similar.

We again have two choices to fix this: either we can change the typechecking rules (which work just fine for small-to-medium-sized terms), or we can adjust typechecking to deal with some sort of pending substitution data, so that we only do substitution once.

The proof assistant can also try to heuristically share subterms for us. Many proof assistants do some version of this, called *hash consing*.

However, hash consing looses a lot of its benefit if terms are not maximally shared (and they almost never are), and it can lead to very unpredictable performance when transformations unexpectedly cause a loss of sharing. Furthermore, it's an open problem how to efficiently persist full hash consing to disk in a way that allows for diamond dependencies.

2.6.2 The Size of the Term

Recall that Coq (and dependently typed proof assistants in general) have *terms* which serve as both programs and proofs. The essential function of a proof checker is to verify that a given term has a given type. We obviously cannot type-check a term in better than linear time in the size of the representation of the term.

Recall that we cannot place any hard bounds on complexity of typechecking a term, as terms as simple as `@eq_refl bool true` proving that the Boolean `true` is equal to itself can also be typechecked as proofs of arbitrarily complex decision procedures returning success. For example, suppose the function `f TM n` takes as arguments a description of a Turing machine `TM` and a number of steps `n` and outputs `false` unless `TM` halts within `n`, in which case it instead outputs `true`. Then for any concrete number `n` and any concrete description of a Turing machine `TM` which does in fact halt within `n` steps, the term `@eq_refl bool true` can be typechecked as a proof of `f TM n = true` because `f TM n` computes to `true`.

We might reasonably hope that typechecking problems which require no interesting computation can be completed in time linear in the size of the term and its type.

However, some seemingly reasonable decisions can result in typechecking taking quadratic time in the size of the term, as we saw in Section 2.6.1.

Even worse, typechecking can easily be unboundedly large in the size of the term when the typechecker chooses the wrong constants to unfold, even when very little work ought to be done.

Consider the problem of typechecking `@eq_refl nat (fact 100) : @id nat (fact 100) = fact 100`, where `fact` is the factorial function on natural numbers and `id` is the polymorphic identity function. If the typechecker either decides to unfold `id` before unfolding `fact`, or if it performs a breadth-first search, then we get speedy performance. However, if the typechecker instead unfolds `id` *last*, then we end up computing the normal form of 100!, which takes a long time and a lot of memory. See Figure 2-5.

Note that it is by no means obvious that the typechecker can meaningfully do anything about this. Breadth-first search is significantly more complicated than depth-first, is harder to write good heuristics for, can incur enormous space overheads, and can be massively slower in cases where there are many options and the standard heuristics for depth-first unfolding in conversion-checking are sufficient. Furthermore, the more heuristics there are to tune conversion-checking, the more "magic" the algorithm seems, and the harder it is to debug when the performance is inadequate.

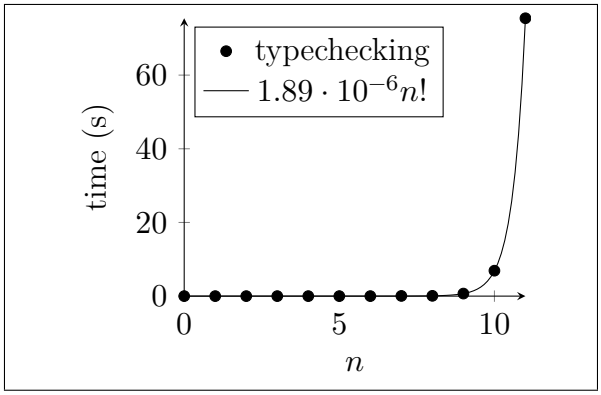


Figure 2-5: Timing of typechecking `@eq_refl nat (fact n) : @id nat (fact n) = fact n`

As described in Section 2.2, in Fiat Cryptography, we got exponential slowdown due to this issue, with an estimated overhead of over four thousand millennia of extra typechecking time in the worst examples we were trying to handle.

2.6.3 The Number of Binders

This is a particular subcase of the above sections that we call out explicitly. Often there will be some operation (for example, substitution, lifting, context creation) that needs to happen every time there is a binder and which, when done naïvely, is linear in the size of the term or the size of the context. As a result, naïve implementations will often incur quadratic—or worse—overhead in the number of binders.

Similarly, if there is any operation that is even linear rather than constant in the number of binders in the context, then any user operation in proof mode which must be done, say, for each hypothesis will incur an overall quadratic-or-worse performance penalty.

The claim of this subsection is not that any particular application is inherently constrained by a performance bottleneck in the number of binders, but instead that it's very, very easy to end up with quadratic-or-worse performance in the number of binders, and hence that this forms a meaningful cluster for performance bottlenecks in practice.

I will attempt to demonstrate this point with a palette of actual historical performance issues in Coq—some of which persist to this day—where the relevant axis was "number of binders." None of these performance issues are insurmountable, but all of them are either a result of seemingly reasonable decisions, have subtle interplay with seemingly disparate parts of the system, or else are to this day still mysterious despite the work of developers to investigate them.

Name Resolution

One key component of interactive proof assistants is figuring out which constant is referred to by a given name. It may be tempting to keep the context in an array or linked list. However, if looking up which constant or variable is referred to by a name is $\mathcal{O}(n)$, then internalizing a term with n typed binders is going to be $\mathcal{O}(n^2)$, because we need to do name lookups for each binder. See Coq bug #9582; note that Coq 8.10 and later do not show this superlinear behavior due to Coq PR #9586, and hence our plots in this section use Coq 8.9.1.

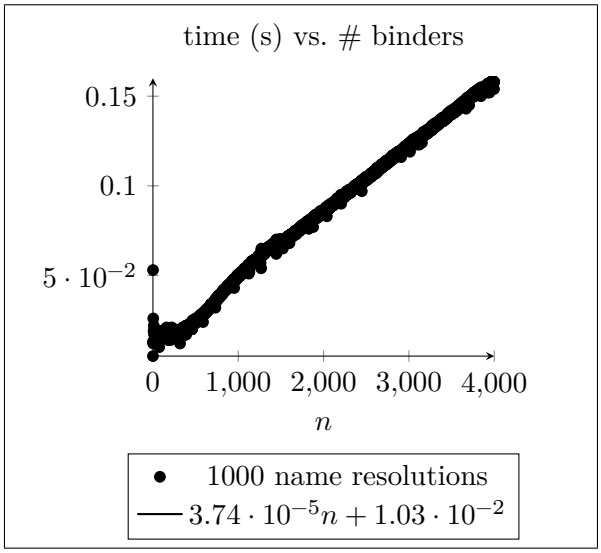


Figure 2-6: Timing of internalizing a name 1000 times under n binders

See Figure 2-6 for the timing of name resolution in Coq as a function of how many binders are in the context. In particular, this plot measures the time it takes to resolve the name `I` a thousand times in a context with a given number of binders.[6] See Figure 2-7 for the effect on internalizing a lambda with n arguments.

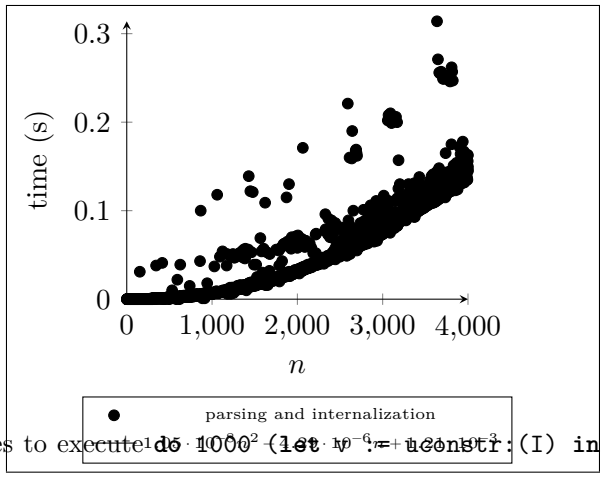


Figure 2-7: Timing of internalizing a function with n differently named arguments

[6] This is done by measuring the time it takes to execute `do 1000 (let v := constr:(I) in idtac)`.

Capture-Avoiding Substitution

If the user is presented with a proof-engine interface where all context variables are named, then in general the proof engine must implement capture-avoiding substitution. For example, if the user wants to operate inside the hole in `(λ x, let y := x in λ x, _)`, then the user needs to be able to talk about the body of `y`, which is not the same as the innermost `x`. However, if the α-renaming is even just linear in the existing context, then creating a new hole under n binders will take $\mathcal{O}(n^2)$ time in the worst case, as we may have to do n renamings, each of which take time $\mathcal{O}(n)$. See Coq bug #9582, perhaps also Coq bug #8245 and Coq bug #8237 and Coq bug #8231.

This might be the cause of the difference in Figure 2-8b between having different names (which do not need to be renamed) and having either no name (requiring name generation) or having all binders with the same name (requiring renaming in evar substitutions).

Quadratic Creation of Substitutions for Existential Variables

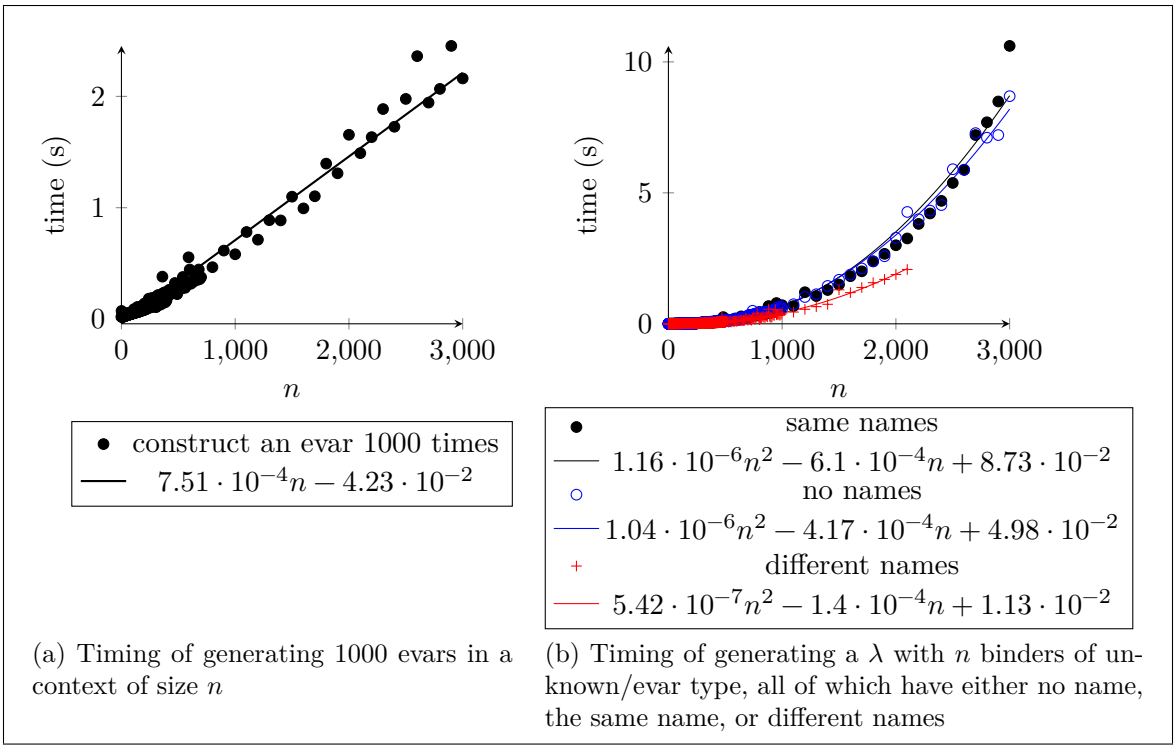
Recall that when we separate the trusted kernel from the untrusted proof engine, we want to be able to represent not-yet-finished terms in the proof engine. The standard way to do this is to enrich the type of terms with an "existential variable" node, which stands for a term which will be filled later. Such an existential variable, or evar, typically exists in a particular context. That is, when filling an evar, some hypotheses are accessible while others are not.

Sometimes, reduction results in changing the context in which an evar exists. For example, if we want to β-reduce `(λ x, ?e`$_1$`) (S y)`, then the result is the evar `?e`$_1$ with `S y` substituted for `x`.

There are a number of ways to represent substitution, and the choices are entangled with the choices of term representation.

Note that most substitutions are either identity or lifting substitutions.

One popular representation is the locally nameless representation [Cha12; Ler07], which we discuss more in Section 3.1.3. However, if we use a locally nameless term representation, then finding a compact representation for identity and lifting substitutions is quite tricky. If the substitution representation takes $\mathcal{O}(n)$ time to create in a context of size n, then having a λ with n arguments whose types are not known takes $\mathcal{O}(n^2)$ time, because we end up creating identity substitutions for n holes, with linear-sized contexts.



(a) Timing of generating 1000 evars in a context of size n

(b) Timing of generating a λ with n binders of unknown/evar type, all of which have either no name, the same name, or different names

Figure 2-8: Performance Benchmarks for Substitution

Note that fully nameless, i.e. de Bruijn, term representations do not suffer from this issue.

See Coq bug #8237 and Coq PR #11896 for a mitigation of some (but not all) issues.

See also Figure 2-8a and Figure 2-8b.

Quadratic Substitution in Function Application

Consider the case of typechecking a nondependent function applied to n arguments. If substitution is performed eagerly, following directly the rules of the type theory, then typechecking is quadratic. This is because the type of the function is $\mathcal{O}(n)$, and doing substitution n times on a term of size $\mathcal{O}(n)$ is quadratic.

If the term representation contains n-ary application nodes, it's possible to resolve this performance bottleneck by delaying the substitutions. If only unary application nodes

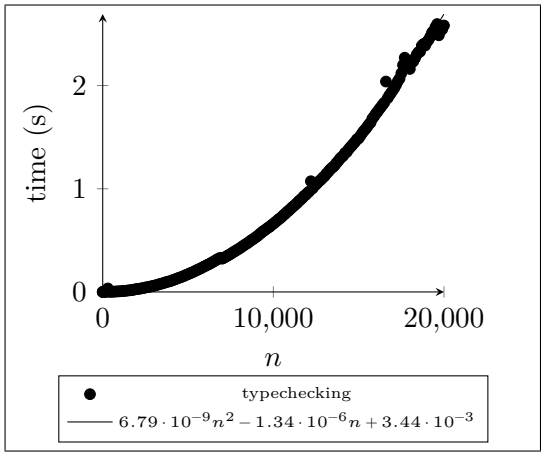


Figure 2-9: Timing of typechecking a function applied to n arguments

exist, it's much harder to solve.

Note that this is important, for example, in attempts to avoid the problem of quadratically sized certificates by making a n-ary conjunction constructor which is parameterized on a list of the conjuncts. Such a function could then be applied to the n proofs of the conjuncts.

We've reported these issues in Coq in Coq bug #8232 and Coq bug #12118, and a partial solution has been merged in Coq PR #8255.

See Figure 2-9 for timing details on a microbenchmark of this bottleneck, where we use Ltac2 to build an application of a function to n arguments of type `unit`. Ltac2 is a relatively recent successor to the $\mathcal{L}_{tac}$ tactic language, which allows more low-level operations that provide more fine-grained control over what the proof assistant is actually doing.

Quadratic Normalization by Evaluation

Normalization by evaluation (NbE) is a nifty way to implement reduction where function abstraction in the object language is represented by function abstraction in the metalanguage. We discuss the details of how to implement NbE in **??**. Coq uses NbE to implement two of its reduction machines (**`lazy`** and **`cbv`**).

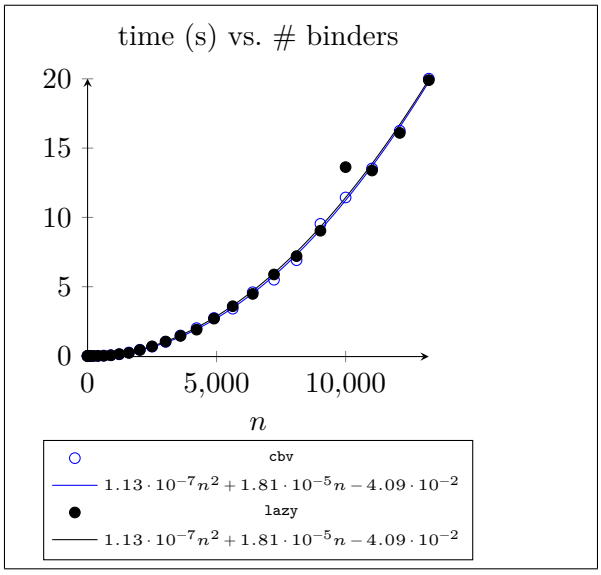


Figure 2-10: Timing of running **`cbv`** and **`lazy`** reduction on interpreting a PHOAS expression as a function of the number of binders

The details of implementing NbE depend on the term representation used. If a fancy term encoding like PHOAS, which we explain in Section 3.1.3, is used, then it's not hard to implement a good NbE algorithm. However, such fancy term representations incur unpredictable and hard-to-deal-with performance costs. Most languages do not do any reduction on thunks until they are called with arguments, which means that forcing early reduction of a PHOAS-like term representation requires round-tripping though another term representation, which can be costly on large terms if there is not much to reduce. On the other hand, other term representations need to implement either capture-avoiding substitution (for named representations) or index lifting (for de Bruijn and locally nameless representations).

The sort-of obvious way to implement this transformation is to write a function that takes a term and a binder and either renames the binder for capture-avoiding substitution or else lifts the indices of the term. The problem with this implementation is that if we call it every time we move a term under a binder, then moving a term under n binders traverses the term n times. If the term size is also proportional to n, then the result is quadratic blowup in the number of binders.

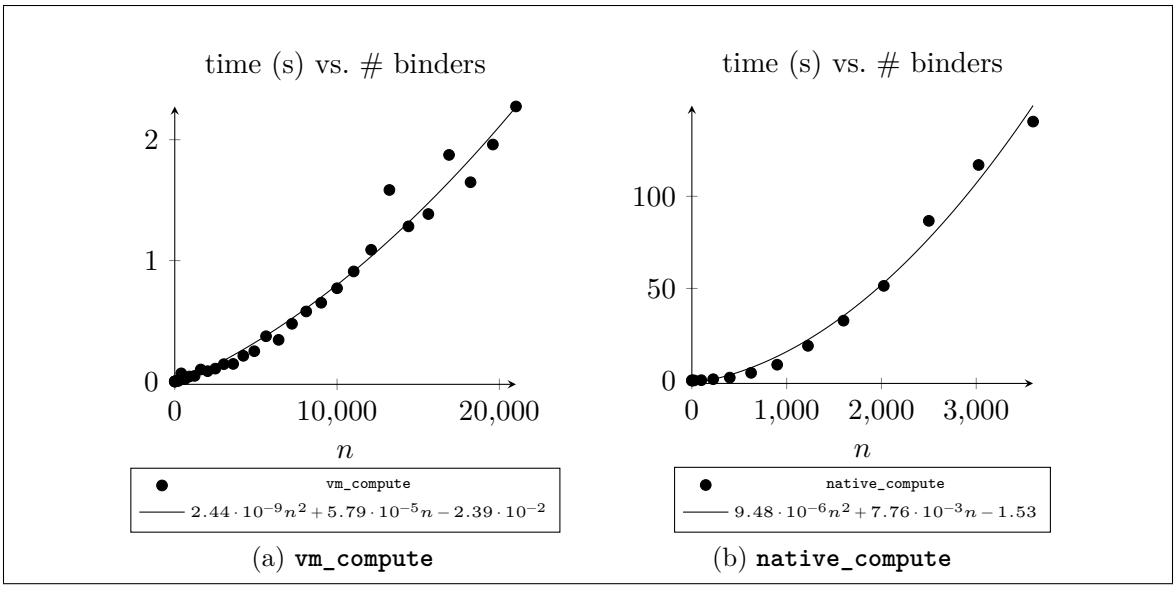
See Coq bug #11151 for an occurrence of this performance issue in the wild in Coq. See also Figure 2-10.

Quadratic Closure Compilation

It's important to be able to perform reduction of terms in an optimized way. When doing optimized reduction in an imperative language, we need to represent closures—abstraction nodes—in some way. Often this involves associating to each closure both some information about or code implementing the body of the function, as well as the values of all of the free variables of that closure [SA00]. In order to have efficient lookup, we need to know the memory location storing the value of any given variable statically at closure-compilation time. The standard way of doing this is to allocate an array of values for each closure. If variables are represented with de Bruijn indices, for example, it's then a very easy array lookup to get the value of any variable. Note that this allocation is linear in the number of free variables of a term. If we have many nested binders and use all of them underneath all the binders, then every abstraction node has as many free variables as there are total binders, and hence we get quadratic overhead.

See Coq bug #11151 and Coq bug #11964 and OCaml bug #7826 for an occurrence of this issue in the wild. Note that this issue rarely shows up in hand-written code, only in generated code, so developers of compilers such as `ocamlc` and `gcc` might be uninterested in optimizing this case. However, it's quite essential when doing metaprogramming involving large generated terms. It's especially essential if we want to chain together reflective automation passes that operate on different input languages and therefore require denotation and reification between the passes. In such cases, unless our encoding language uses named or de Bruijn variable encoding, there's no way to avoid large numbers of nested binders at compilation time while preserving code sharing. Hence if we're trying to reuse the work of existing compilers to bootstrap good performance of reduction (as is the case for the native compiler in Coq), we have trouble with cases such as this one.

See also Figure 2-11a and Figure 2-11b.



(a) **vm_compute** (b) **native_compute**

Figure 2-11: Timing of running reduction on interpreting a PHOAS expression as a function of the number of binders

2.6.4 The Number of Nested Abstraction Barriers

This axis is the most theoretical of the axes. An abstraction barrier is an interface for making use of code, definitions, and theorems. For example, we might define nonnegative integers using a binary representation and present the interface of zero, successor, and the standard induction principle, along with an equational theory for how induction behaves on zero and successor. We might use lists and nonnegative integers to implement a hash-set datatype for storing sets of hashable values and present the hash-set with methods for empty, add, remove, membership-testing, and some sort of fold. Each of these is an abstraction barrier.

There are three primary ways that nested abstraction barriers can lead to performance bottlenecks: one involving conversion missteps and two involving exponential blow-up in the size of types.

Conversion Troubles

If abstraction barriers are not perfectly opaque—that is, if the typechecker ever has to unfold the definitions making up the API in order to typecheck a term—then every additional abstraction barrier provides another opportunity for the typechecker to pick the wrong constant to unfold first. In some typecheckers, such as Coq, it's possible to provide hints to the typechecker to inform it which constants to unfold when. In such a system, it's possible to carefully craft conversion hints so that abstraction barriers are always unfolded in the right order. Alternatively, it might be possible to carefully craft a system which picks the right order of unfolding by using a dependency analysis.

However, most users don't bother to set up hints like this, and dependency analysis isn't sufficient to determine which abstraction barrier is "higher up" when there are many parts of it, only some of which are mentioned in any given part of the next abstraction barrier. The reason users don't set up hints like this is that usually it's not necessary. There's often minimal overhead, and things just work, even when the wrong path is picked—until the number of abstraction barriers or the size of the underlying term gets large enough. Then we get noticeable exponential blowup and our development no longer terminates in reasonable time. Furthermore, it's hard to know which part of conversion is incurring exponential blowup, and thus one has to basically get all of the conversion hints right, simultaneously, without any feedback, to see any performance improvement.

Type-Size Blowup: Abstraction Barrier Mismatch

When abstraction barriers are leaky or misaligned, there's a cost that accumulates in the size of the types of theorems. Consider, for example, the two different ways of using tuples: (1) we can use the projections `fst` and `snd`; or (2) we can use the eliminator `pair_rect : ∀ A B (P : A × B → Type), (∀ a b, P (a, b)) → ∀ x, P x`. The first gets us access to one element of the tuple at a time, while the second has us using all elements of the tuple simultaneously.

Suppose now there is one API defined in terms of `fst` and `snd` and another API defined in terms of `pair_rect`. To make these APIs interoperate, we need to convert explicitly from one representation to another. Furthermore, every theorem about the composition of these APIs needs to include the interoperation in talking about how they relate.

If such API mismatches are nested, or if this code-size blowup interacts with conversion missteps, then the performance issues compound.

Let us consider things a bit more generally.

Structure and Interpretation of Computer Programs defines abstraction as naming and manipulating compound elements as units [SSA96, p. 6]. An *abstraction barrier* is a collection of definitions and theorems about those definitions that together provide an interface for such a compound element. For example, we might define an interface for sorting a list, together with a proof that sorting any list results in a sorted list. Or we might define an interface for key-value maps (perhaps implemented as association lists, or hash-maps, or binary search trees, or in some other way).

Piercing an abstraction barrier is the act of manipulating the compound element by its components, rather than through the interface. For example, suppose we have implemented key-value maps as association lists, representing the map as a list of key-value pairs, and provided some interface. Any function which, for example, asks

for the first element of the association list has pierced the abstraction barrier of our interface.

We might say that an abstraction barrier is *leaky* if we ever need to pierce it, or perhaps if our program does in fact pierce the abstraction barrier, even if the piercing is needless. (Which definition we choose is not of great significance for this dissertation.)

In proof assistants like Coq, using **`unfold`**, **`simpl`**, or `cbn` can often indicate a leaky abstraction barrier, where in order to prove a property we unfold the interface we are given to see how it is implemented. This is all well and good when we are in the process of defining the abstraction barrier—unfolding the definition of sorting a list, for example, to prove that sorting the list gives back a list with all the same elements—but can be problematic when used more pervasively.

Let us look at an example from a category-theory library we implemented in Coq [GCS14], which we introduce in Section 6.3. Category theory generalizes functions and product types, and the example we present here is a category-theoretic version of the isomorphism between functions of type $C_1 \times C_2 \to D$, which take a pair of elements $c_1 \in C_1$ and $c_2 \in C_2$ and return an element of D, and functions of type $C_1 \to (C_2 \to D)$ which take a single argument $c_1 \in C_1$ and return a function from C_2 to D. We write this isomorphism as

$$(C_1 \times C_2 \to D) \cong (C_1 \to (C_2 \to D))$$

In computer science, this is known as (un)currying. The abstractions used in formalizing this example are as follows

- A *category* $\mathcal{C}$ is a collection of objects and composable arrows (called *morphisms*) between those objects, subject to some algebraic laws. The class of objects is generally denoted $\mathrm{Ob}_{\mathcal{C}}$, and the class of morphisms between $x, y \in \mathrm{Ob}_{\mathcal{C}}$ is generally denoted $\mathrm{Hom}_{\mathcal{C}}(x, y)$. Categories are a sort of generalization of sets or types.

- The *product category* $\mathcal{C} \times \mathcal{D}$ generalizes the Cartesian product of sets.

- An *isomorphism* between objects x and y in a category $\mathcal{C}$, written $x \cong y$, is a pair of morphisms from x to y and from y to x such that the composition in either direction is the identity morphism.

- A *functor* is an arrow between categories, mapping objects to objects and morphisms to morphisms, subject to some algebraic laws. The action of a functor F on an object x is often denoted $F(x)$. As the action of F on a morphism m is often also denoted $F(m)$, we will use F_0 to denote the action on objects and F_1 to denote the action on morphisms when it might otherwise be unclear.

- A *natural transformation* is an arrow between functors F and G consisting of a way of mapping from the on-object-action of F to the on-object-action of G, satisfying some algebraic laws.
- A category of functors $\mathcal{C} \to \mathcal{D}$ is the category whose objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and whose morphisms are natural transformations. This category generalizes the notion of function types or of sets of functions.
- The category of categories, generally denoted Cat, is a category whose objects are themselves categories and whose morphisms are functors. Much like the set of all sets or the type of all types, the categories in Cat are subject to size restrictions discussed further in Section 7.2.1.

Although we eventually go into a bit more of the detail of these definitions throughout Section 7.2.1, we advise the interested reader to consult the rich existing literature on category theory, including for example Awodey [Awo] and Mac Lane [Mac]. These are by no means required reading, though; most of this dissertation is unrelated to category theory, and we have aimed to make even the parts related to category theory relatively accessible to readers with no category-theoretic background.

There are only seven components of the isomorphism $(\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}) \cong (\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D}))$ which are not proofs of algebraic laws. Their definition, spelled out in Figure 2-12 and given in Gallina Coq code (with suitable notations) in Figure 2-13, is relatively trivial.

Typechecking the code that defines these components, however, takes nearly two seconds! This is more than 200× slower than it needs to be. The essential structure needed to define these components can be defined without any of the categorical indirection, taking $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{D}$ to be types and taking morphisms in these "categories" to be proofs of equality between morphism sources and targets. Defining the structure of the seven components in this way nets us a 200× speedup![7] We attribute the overhead of the categorical definition to the large types generated and the nontrivial conversion problems which require unfolding various definitions, i.e., piercing various abstraction barriers.

While two seconds is long, there is an even more serious issue that arises when attempting to prove the algebraic laws. The types here are already a bit long: The goal that going from $(C_1 \times C_2 \to D)$ to $(C_1 \to (C_2 \to D))$ and back again is the identity is only about 24 lines after β reduction (when `Set Printing All` is on, there are about 3 300 words).

However, if we pierce the abstraction barrier of functor composition, the goal blows up to about 254 lines (about 18 000 words with `Set Printing All`)! This blow-up is due to the fact that the opaque proofs that functor composition is functorial

[7] See Appendix A.1.1 for the code used to make this timing measurement.

To define currying, going from $(\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D})$ to $(\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D}))$:

1. Each functor $F : \mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}$ gets mapped to a functor which takes in an object $c_1 \in \mathrm{Ob}_{\mathcal{C}_1}$ and returns a functor which takes in an object $c_2 \in \mathrm{Ob}_{\mathcal{C}_2}$ and returns the object $F((c_1, c_2)) \in \mathrm{Ob}_{\mathcal{D}}$.

2. The action of the returned functor on morphisms in $\mathcal{C}_2$ is to first lift this morphism from $\mathcal{C}_2$ to $\mathcal{C}_1 \times \mathcal{C}_2$ by pairing with the identity morphism on c_1, and then to return the image of this morphism under F.

3. The action of the outer functor on morphisms $m_1 \in \mathrm{Hom}_{\mathcal{C}_1}$ is to return the natural transformation which, for each object $c_2 \in \mathrm{Ob}_{\mathcal{C}_2}$ first pairs the morphism m_1 with the identity on c_2 and then returns the image of this morphism in $\mathcal{C}_1 \times \mathcal{C}_2$ under F.

4. Each natural transformation $T \in \mathrm{Hom}_{\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}}$ gets mapped to the natural transformation in $\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D})$ which, after binding c_1 and c_2, returns the morphism in $\mathcal{D}$ given by the action of T on (c_1, c_2).

To define uncurrying, going from $(\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D}))$ to $(\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D})$:

5. Each functor $F : \mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D})$ gets mapped to the functor which takes in an object $(c_1, c_2) \in \mathrm{Ob}_{\mathcal{C}_1 \times \mathcal{C}_2}$ and returns $(F(c_1))(c_2)$.

6. The action of this functor on morphisms $(m_1, m_2) \in \mathrm{Hom}_{\mathcal{C}_1 \times \mathcal{C}_2}$ is to compose $F(m_1)$ applied to a suitable object of $\mathcal{C}_2$ with F applied to a suitable object of c_1 and then applied to m_2.

7. Each natural transformation $T \in \mathrm{Hom}_{\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D})}$ gets mapped to the natural transformation which maps each object $(c_1, c_2) \in \mathrm{Ob}_{\mathcal{C}_1 \times \mathcal{C}_2}$ to the morphism $(T(c_1))(c_2)$ in $\mathrm{Hom}_{\mathcal{D}}$.

While this is a mouthful, there is no insight in any of these definitions; for each component, there is exactly one choice that can be made which has the correct type.

Figure 2-12: The interesting components of $(\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}) \cong (\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D}))$.

```
(** [(C1 × C2 → D) ≅ (C1 → (C2 → D))] *)
(** We denote functors by pairs of maps on objects ([λo]) and
    morphisms ([λm]), and natural transformations as a single map
    ([λt]) *)
Time Program Definition curry_iso (C1 C2 D : Category)
  : (C1 * C2 -> D) ≅ (C1 -> (C2 -> D)) :>>> Cat
  := {| fwd
        := λo F, λo c1, λo c2, F 0 (c1, c2)
                     ; λm m , F 1 (identity c1, m)
              ; λm m1, λt c2, F 1 (m1, identity c2)
         ; λm T, λt c1, λt c2, T (c1, c2);
        bwd
        := λo F, λo '(c1, c2), (F 0 c1)0 c2
              ; λm '(m1, m2), (F 1 m1) _ ∘ (F 0 _)1 m2
         ; λm T, λt '(c1, c2), (T c1) c2 |}.
(* Finished transaction in 1.958 secs (1.958u,0.s) (successful) *)
```

Figure 2-13: The interesting components of $(\mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}) \cong (\mathcal{C}_1 \to (\mathcal{C}_2 \to \mathcal{D}))$, in Coq. The surrounding definitions and notations required for this example to type-check are given in Appendix A.1.

take the entirety of the functors being composed as arguments. Hence unfolding the composition of two functors duplicates those functors many times over. If we must compose more than two functors, we get even more blow-up.

Piercing this barrier also shows up in proof-checking time. If we first decompose the goal into the separate equalities we wish to prove and only then unfold the abstraction barrier (thereby side-stepping the issue of passing large arguments to opaque proofs), it takes less than a tenth of a second to prove each of the two algebraic laws of the isomorphism. However, if we instead unfold the definitions first and then decompose the goal into separate goals, it takes about 5× longer to check the proof.

Readers interested in the full compiling code for this example can refer to Appendix A.1.

Type Size Blowup: Packed vs. Unpacked Records

When designing APIs, especially of mathematical objects, one of the biggest choices is whether to pack the records or whether to pass arguments in as fields. That is, when defining a monoid, for example, there are five ways to go about specifying it:

1. (packed) A *monoid* consists of a type A, a binary operation $\cdot : A \to A \to A$, an identity element e, a proof that e is a left and right identity $e \cdot a = a \cdot e = a$

for all a, and a proof of associativity that $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

2. A *monoid on a carrier type* A consists of a binary operation $\cdot : A \to A \to A$, an identity element e, a proof that e is a left and right identity, and a proof of associativity.

3. A *monoid on a carrier type* A *under the binary operation* $\cdot : A \to A \to A$ consists of an identity element e, a proof that e is a left and right identity, and a proof of associativity.

4. (mostly unpacked) A *monoid on a carrier type* A *under the binary operation* $\cdot : A \to A \to A$ *with identity element* e consists of a proof that e is a left and right identity and a proof of associativity. Note that MathClasses [KSW; SW11; SW10] uses this strategy, as discussed in Garillot et al. [Gar+09b].

5. (fully unpacked) A monoid on a carrier type A under the binary operation $\cdot : A \to A \to A$ with identity element e using a proof p that e is a left and right identity and a proof of q of associativity consists of an element of the one-element unit type.

If we go with anything but the fully packed design, then we incur exponential overhead as we go up abstraction layers, as follows. A *monoid homomorphism* from a monoid A to a monoid B consists of a function between the carrier types and proofs that this function respects composition and identity. If we use an unpacked definition of monoid with n type parameters, then a similar definition of a monoid homomorphism involves at least $2n + 2$ type parameters. In higher category theory, it's common to talk about morphisms between morphisms, and every additional layer here doubles the number of type arguments, and this can quickly lead to very large terms, resulting is major performance bottlenecks. Note that number of type parameters determines the constant factor out front of the exponential growth in the number of layers of mathematical constructions.

How much is this overhead concretely? When developing a category-theory library [GCS14], described in more detail in Section 6.3, we sped up overall compilation time by approximately a factor of two, from around 16 minutes to around 8 minutes, by changing one of the two parameters to a field in the definition of a category.[8]

2.7 Conclusion of This Chapter

We hope the reader now has a sense of the landscape of superlinear performance bottlenecks we've seen in dependently typed proof assistants. In the chapters to come, we invite the reader to keep in the back of their mind the four axes we've laid

[8] See commit 209231a of JasonGross/catdb on GitHub for details.

out as scaling factors in most performance bottlenecks we've encountered—the size of the type, the size of the term, the number of binders, and the number of nested abstraction barriers.

Part II

Program Transformation and Rewriting

Chapter 3

Reflective Program Transformation

3.1 Introduction

Proof by reflection [Bou97] is an established method for employing verified proof procedures, within larger proofs [MCB14; Mal+13; Mal17; GMT16]. There are a number of benefits to using verified functional programs written in the proof assistant's logic, instead of tactic scripts. We can often prove that procedures always terminate without attempting fallacious proof steps, and perhaps we can even prove that a procedure gives logically complete answers, for instance telling us definitively whether a proposition is true or false. In contrast, tactic-based procedures may encounter runtime errors or loop forever. As a consequence, if we want to keep the trusted codebase small, as discussed in Subsection 1.3.2, these tactic procedures must output proof terms, justifying their decisions, and these terms can grow large, making for slower proving and requiring transmission of large proof terms to be checked slowly by others. A verified procedure need not generate a certificate for each invocation.

3.1.1 Proof-Script Primer

Basic Coq proofs are often written as lists of steps such as **`induction`** on some structure, **`rewrite`** using a known equivalence, or **`unfold`** of a definition. As mentioned in Section 1.1, proofs can very quickly become long and tedious, both to write and to read, and hence Coq provides $\mathcal{L}_{tac}$, a scripting language for proofs, which we first mentioned in Subsection 1.3.2. As theorems and proofs grow in complexity, users frequently run into performance and maintainability issues with $\mathcal{L}_{tac}$, some of which we've seen in Chapter 2. Consider the case where we want to prove that a large algebraic expression, involving many `let … in …` expressions, is even:

```
Inductive is_even : nat → Prop :=
| even_0 : is_even 0
| even_SS : forall x, is_even x → is_even (S (S x)).

Goal is_even (let x := 100 * 100 * 100 * 100 in
              let y := x * x * x * x in
              y * y * y * y).
```

Coq stack-overflows if we try to reduce this goal. As a workaround, we might write a lemma that talks about evenness of `let … in …`, plus one about evenness of multiplication, and we might then write a tactic that composes such lemmas.

Even on smaller terms, though, proof size can quickly become an issue. If we give a naïve proof that 7000 is even, the proof term will contain all of the even numbers between 0 and 7000, giving a proof-term-size blow-up at least quadratic in size (recalling that natural numbers are represented in unary; the challenges remain for more efficient base encodings). Clever readers will notice that Coq could share subterms in the proof tree, recovering a term that is linear in the size of the goal. However, such sharing would have to be preserved very carefully, to prevent size blow-up from unexpected loss of sharing, and today's Coq version does not do that sharing. Even if it did, tactics that rely on assumptions about Coq's sharing strategy become harder to debug, rather than easier.

3.1.2 Reflective-Automation Primer

Enter reflective automation, which simultaneously solves both the problem of performance and the problem of debuggability. Proof terms, in a sense, are traces of a proof script. They provide Coq's kernel with a term that it can check to verify that no illegal steps were taken. Listing every step results in large traces.

The idea of reflective automation is that, if we can get a formal encoding of our goal, plus an algorithm to *check* the property we care about, then we can do much better than storing the entire trace of the program. We can prove that our checker is correct once and for all, removing the need to trace its steps.

```
Fixpoint check_is_even (n : nat) : bool
  := match n with
     | 0 => true
     | 1 => false
     | S (S n) => check_is_even n
     end.
```

Figure 3-1: Evenness Checking

A simple evenness checker can just operate on the unary encoding of natural numbers (Figure 3-1). We can use its correctness theorem to prove goals much more quickly:

```
Theorem soundness : forall n, check_is_even n = true → is_even n.

Goal is_even 2000.
  Time repeat (apply even_SS || apply even_0). (* 1.8 s *)
  Undo.
  Time apply soundness; vm_compute; reflexivity. (* 0.004 s *)
```

The tactic **vm_compute** tells Coq to use its virtual machine for reduction, to compute the value of `check_is_even 2000`, after which **reflexivity** proves that `true = true`. Note how much faster this method is. In fact, even the asymptotic complexity is better; this new algorithm is linear rather than quadratic in `n`.

However, even this procedure takes a bit over three minutes to prove the goal `is_even (10 * 10 * 10 * 10 * 10 * 10 * 10 * 10 * 10)`. To do better, we need a formal representation of terms or expressions.

3.1.3 Reflective-Syntax Primer

Sometimes, to achieve faster proofs, we must be able to tell, for example, whether we got a term by multiplication or by addition, and not merely whether its normal form is 0 or a successor.[1]

A reflective automation procedure generally has two steps. The first step is to *reify* the goal into some abstract syntactic representation, which we call the *term language* or an *expression language*. The second step is to run the algorithm on the reified syntax.

What should our expression language include? At a bare minimum, we must have multiplication nodes, and we must have `nat` literals. If we encode `S` and `O` separately, a decision that will become important later in Section 5.2, we get the inductive type of Figure 3-2.

```
Inductive expr :=
| NatO : expr
| NatS (x : expr) : expr
| NatMul (x y : expr) : expr.
```

Figure 3-2: Simple Expressions

Before diving into methods of reification, let us write the evenness checker.

```
Fixpoint check_is_even_expr (t : expr) : bool
  := match t with
     | NatO => true
```

[1]Sometimes this distinction is necessary for generating a proof at all, as is the case in **nsatz** and **romega**; there is no way to prove that addition is commutative if you cannot identify what numbers you were adding in the first place.

```
    | NatS x => negb (check_is_even_expr x)
    | NatMul x y => orb (check_is_even_expr x) (check_is_even_expr y)
    end.
```

Before we can state the soundness theorem (whenever this checker returns `true`, the represented number is even), we must write the function that tells us what number our expression represents, called *denotation* or *interpretation*:

```
Fixpoint denote (t : expr) : nat
  := match t with
     | NatO => O
     | NatS x => S (denote x)
     | NatMul x y => denote x * denote y
     end.

Theorem check_is_even_expr_sound (e : expr)
  : check_is_even_expr e = true → is_even (denote e).
```

Given a tactic `Reify` to produce a reified term from a `nat`, we can time the execution of `check_is_even_expr` in Coq's VM. It is instant on the last example.

Before we proceed to reification, we will introduce one more complexity. If we want to support our initial example with `let … in …` efficiently, we must also have **`let`** expressions. Our current procedure that inlines **`let`** expressions takes 19 seconds, for example, on **`let`** `x0 := 10 * 10` **`in let`** `x1 := x0 * x0` **`in …`** **`let`** `x24 := x23 * x23` **`in`** `x24`. The choices of representation of binders, which are essential to encoding **`let`** expressions, include higher-order abstract syntax (HOAS) [PE88], parametric higher-order abstract syntax (PHOAS) [Chl08] which is also known as weak HOAS [CS13], de Bruijn indices [Bru72], nominal representations [Pit03], locally nameless representations [Cha12; Ler07], named representations, and nested abstract syntax [HM12; BP99]. A survey of a number of options for binding can be found in [Ayd+08].

Although we will eventually choose the PHOAS representation for the tools presented in s **??** and 5, we will also briefly survey some of the options for encoding binders, with an eye towards performance implications.

PHOAS

The PHOAS representation [Chl08; CS13] is particularly convenient. In PHOAS, expression binders are represented by binders in Gallina, the functional language of Coq, and the expression language is parameterized over the type of the binder. Let us define a constant and notation for **`let`** expressions as definitions (a common choice

in real Coq developments, to block Coq's default behavior of inlining `let` binders silently; the same choice will also turn out to be useful for reification later). We thus have:

```
Inductive expr {var : Type} :=
| NatO : expr
| NatS : expr → expr
| NatMul : expr → expr → expr
| Var : var → expr
| LetIn : expr → (var → expr) → expr.
Notation "'elet' x := v 'in' f" := (LetIn v (fun x => f))
    (x ident, at level 200).

Definition Let_In {A B} (v : A) (f : A → B) := let x := v in f x.
Notation "'dlet' x := v 'in' f" := (Let_In v (fun x => f))
    (x ident, at level 200).
```

Conventionally, syntax trees are parametric over the value of the `var` parameter, which we may instantiate in various ways to allow variable nodes to hold various kinds of information, and we might define a type for these parametric syntax trees:

```
Fixpoint denote (t : @expr nat) : nat
  := match t with
     | NatO => O
     | NatS x => S (denote x)
     | NatMul x y => denote x * denote y
     | Var v => v
     | LetIn v f => dlet x := denote v in denote (f x)
     end.

Fixpoint check_is_even_expr (t : @expr bool) : bool
  := match t with
     | NatO => true
     | NatS x => negb (check_is_even_expr x)
     | NatMul x y => orb (check_is_even_expr x) (check_is_even_expr y)
     | Var v_even => v_even
     | LetIn v f => let v_even := check_is_even_expr v in
                    check_is_even_expr (f v_even)
     end.
```

Figure 3-3: Two definitions using two different instantiations of the PHOAS `var` parameter.

```
Inductive related {var1 var2 : Type}
  : list (var1 * var2) → @expr var1 → @expr var2 → Prop :=
| RelatedNatO {Γ}
 : related Γ NatO NatO
| RelatedNatS {Γ e1 e2}
 : related Γ e1 e2 → related Γ (NatS e1) (NatS e2)
| RelatedNatMul {Γ x1 x2 y1 y2}
 : related Γ x1 x2 → related Γ y1 y2
   → related Γ (NatMul x1 y1) (NatMul x2 y2)
| RelatedVar {Γ v1 v2}
 : (v1, v2) ∈ Γ → related Γ (Var v1) (Var v2)
| RelatedLetIn {Γ e1 e2 f1 f2}
 : related Γ e1 e2
   → (∀ v1 v2, related ((v1, v2) :: Γ) (f1 v1) (f2 v2))
   → related Γ (LetIn e1 f1) (LetIn e2 f2).
```

Figure 3-4: A PHOAS relatedness predicate

```
Definition Expr := ∀ var, @expr var.
```

Note, importantly, that `check_is_even_expr` and `denote` will take `expr`s with *different* instantiations of the `var` parameters, as seen in Figure 3-3. This is necessary so that we can store the information about whether or not a particular **`let`**-bound expression is even (or what its denotation is) in the variable node itself. However, this means that we cannot reuse the same expression as arguments to both functions to formulate the soundness condition. Instead, we must introduce a notion of *relatedness* of expressions with different instantiations of the `var` parameter.

A PHOAS relatedness predicate has one constructor for each constructor of `expr`, essentially encoding that the two expressions have the same structure. For the `Var` case, we defer to membership in a list of "related" variables, which we extend each time we go underneath a binder. See Figure 3-4 for such an inductive predicate.

We require that all instantiations give related ASTs (in the empty context), whence we call the parametric AST *well-formed*:

```
Definition Wf (e : Expr) := ∀ var1 var2, related [] (e var1) (e var2)
```

We could then prove a modified form of our soundness theorem:

```
Theorem check_is_even_expr_sound (e : Expr) (H : Wf e)
: check_is_even_expr (e bool) = true → is_even (denote (e nat)).
```

To complete the picture, we would need a tactic `Reify` which took in a term of type `nat` and gave back a term of type `forall var, @expr var`, plus a tactic `prove_wf` which solved a goal of the form `Wf e` by repeated application of constructors. Given these, we could solve an evenness goal by writing[2]

```
match goal with
| [ |- is_even ?v ]
  => let e := Reify v in
     refine (check_is_even_expr_sound e _ _);
     [ prove_wf | vm_compute; reflexivity ]
end.
```

Multiple Types

One important point, not yet mentioned, is that sometimes we want our reflective language to handle multiple types of terms. For example, we might want to enrich our language of expressions with lists. Since expressions like "take the successor of this list" don't make sense, the natural choice is to index the inductive over codes for types.

We might write:

```
Inductive type := Nat | List (_ : type).
Inductive expr {var : type → Type} : type → Type :=
| NatO : expr Nat
| NatS : expr Nat → expr Nat
| NatMul : expr Nat → expr Nat → expr Nat
| Var {t} : var t → expr t
| LetIn {t1 t2} : expr t1 → (var t1 → expr t2) → expr t2
| Nil {t} : expr (List t)
| Cons {t} : expr t → expr (List t) → expr (List t)
| Length {t} : expr (List t) → expr Nat.
```

We would then have to adjust the definitions of the other functions accordingly. The type signatures of these functions might become

[2]Note that for the `refine` to be fast, we must issue something like `Strategy -10 [denote]` to tell Coq to unfold `denote` before `Let_In`. Alternatively, we may issue something like `Strategy 10 [Let_In]` to tell Coq to unfold `Let_In` only after unfolding any constant with no `Strategy` declaration. This invocation may look familiar to those readers who read the footnotes in Section 2.2 (Exponential Domain), as in fact this is the issue at the root cause of the exponential performance blowup which resulted in numbers like "over 4 000 millennia" in an earlier version of Fiat Cryptography.

```
Fixpoint denote_type (t : type) : Type
  := match t with
     | Nat => nat
     | List t => list (denote_type t)
     end.

Fixpoint even_data_of_type (t : type) : Type
  := match t with
     | Nat => bool (* is the nat even or not? *)
     | List t => list (even_data_of_type t)
     end.

Fixpoint denote {t} (e : @expr denote_type t) : denote_type t.

Fixpoint check_is_even_expr {t} (e : @expr even_data_of_type t)
  : even_data_of_type t.

Inductive related {var1 var2 : type → Type}
  : list { t : type & var1 t * var2 t}
    → ∀ {t}, @expr var1 t → @expr var2 t → Prop.

Definition Expr (t : type) := ∀ var, @expr var t.

Definition Wf {t} (e : Expr t)
  := ∀ var1 var2, related [] (e var1) (e var2).
```

See Chlipala [Chl08] for a fuller treatment.

de Bruijn Indices

The idea behind *de Bruijn indices* is that variables are encoded by numbers which count up starting from the nearest enclosing binder. We might write

```
Inductive expr :=
| NatO : expr
| NatS : expr → expr
| NatMul : expr → expr → expr
| Var : nat → expr
| LetIn : expr → expr → expr.

Fixpoint denote (default : nat) (Γ : list nat) (t : @expr nat) : nat
  := match t with
     | NatO => O
```

```
  | NatS x => S (denote default Γ x)
  | NatMul x y => denote default Γ x * denote default Γ y
  | Var idx => nth_default default Γ idx
  | LetIn v f => dlet x := denote default Γ v in
                 denote default (x :: Γ) f
  end.
```

If we wanted a more efficient representation, we could choose better data structures for the context Γ and variable indices than linked lists and unary-encoded natural numbers. One particularly convenient choice, in Coq, would be using the efficient `PositiveMap.t` data structure which encodes a finite map of binary-encoded `positive`s to any type.

One unfortunate result is that the natural denotation function is no longer total. Here we have chosen to give a denotation function which returns a default element when a variable reference is too large, but we could instead choose to return an `option nat`. In general, however, returning an optional result significantly complicates the denotation function when binders are involved, because the types `A → option B` and `option (A → B)` are not isomorphic. On the other hand, requiring a default denotation prevents syntax trees from being able to represent possibly empty types.

This causes further problems when dealing with an AST type which can represent terms of multiple types. In that case, we might annotate each variable node with a type code, mandate decidable equality of type codes, and then during denotation, we'd check the type of the variable node with the type of the corresponding variable in the context.

Nested Abstract Syntax

If we want a variant of de Bruijn indices which guarantees well-typed syntax trees, we can use nested abstract syntax [HM12; BP99]. On monotyped ASTs, this looks like encoding the size of the context in the type of the expressions. For example, we could use `option` types [HM12]:

```
Notation "^ V" := (option V).
Inductive expr : Type → Type :=
| NatO {V} : expr V
| NatS {V} : expr V → expr V
| NatMul {V} : expr V → expr V → expr V
| Var {V} : V → expr V
| LetIn {V} : expr V → expr (^V) → expr V.
```

This may seem a bit strange to those accustomed to encodings of terms in proof

assistants, but it generalizes to a quite familiar intrinsic encoding of dependent type theory using types, contexts, and terms [Ben+12]. Namely, when the expressions are multityped, we end up with something like

```
Inductive context :=
| emp : context
| push : type → context → context.

Inductive var : context → type → Type :=
| Var0 {t Γ} : var (push t Γ) t
| VarS {t t' Γ} : var Γ t → var (push t' Γ) t.

Inductive expr : context → type → Type :=
| Nat0 {Γ} : expr Γ Nat
| NatS {Γ} : expr Γ Nat → expr Γ Nat
| NatMul {Γ} : expr Γ Nat → expr Γ Nat → expr Γ Nat
| Var {t Γ} : var Γ t → expr Γ t
| LetIn {Γ t1 t2} : expr Γ t1 → expr (push t1 Γ) t2 → expr Γ t2.
```

Note that this generalizes nicely to codes for dependent types if the proof assistant supports induction-induction.

Although this representation enjoys both decidable equality of binders (like de Bruijn indices), as well as being well-typed-by-construction (like PHOAS), it's unfortunately unfit for coding algorithms that need to scale without massive assistance from the proof assistant. In particular, the naïve encoding of this inductive datatype incurs a quadratic overhead in representing terms involving binders, because each node stores the entire context. It is possible in theory to avoid this blowup by dropping the indices of the inductive type from the runtime representation [BMM03]. One way to simulate this in Coq would be to put `context` in **`Prop`** and then extract the code to OCaml, which erases the **`Prop`**s. Alternatively, if Coq is extended with support for dropping irrelevant subterms [Gil+19] from the term representation, then this speedup could be accomplished even inside Coq.

Nominal

Nominal representations [Pit03] use names rather than indices for binders. These representations have the benefit of being more human-readable but require reasoning about freshness of names and capture-avoiding substitution. Additionally, if the representation of names is not sufficiently compact, the overhead of storing names at every binder node can become significant.

Locally Nameless

We mention the locally nameless representation [Cha12; Ler07] because it is the term representation used by Coq itself. This representation uses de Bruijn indices for locally-bound variables and names for variables which are not bound in the current term.

Much like nominal representations, locally nameless representations also incur the overhead of generating and storing names. Naïve algorithms for generating fresh names, such as the algorithm used in Coq, can easily incur overhead that's linear in the size of the context. Generating n fresh names then incurs $\Theta(n^2)$ overhead. Additionally, using a locally nameless representation requires that evar substitutions be named. See also **??**.

3.1.4 Performance of Proving Reflective Well-Formedness of PHOAS

We saw in Section 3.1.3 that in order to prove the soundness theorem, we needed a way to relate two PHOASTs (parametric higher-order abstract syntax trees), which generalized to a notion of well-formedness for the `Expr` type.

Unfortunately, the proof that two `expr`s are `related` is quadratic in the size of the expression, for much the same reason that proving conjunctions in Subsection 2.6.1 resulted in a proof term which was quadratic in the number of conjuncts. We present two ways to encode linearly sized proofs of well-formedness in PHOAS.

Iterating Reflection

The first method of encoding linearly sized proofs of `related` is itself a good study in how using proof by reflection can compress proof terms. Rather than constructing the inductive `related` proof, we can instead write a fixed point:

```
Fixpoint is_related {var1 var2 : Type} (Γ : list (var1 * var2))
    (e1 : @expr var1) (e2 : @expr var2) : Prop :=
  match e1, e2 with
  | NatO, NatO => True
  | NatS e1, NatS e2 => is_related Γ e1 e2
  | NatMul x1 y1, NatMul x2 y2
    => is_related Γ x1 x2 /\ is_related Γ y1 y2
  | Var v1, Var v2 => List.In (v1, v2) Γ
  | LetIn e1 f1, LetIn e2 f2
    => is_related Γ e1 e2
       /\ ∀ v1 v2, is_related ((v1, v2) :: Γ) (f1 v1) (f2 v2)
```

```
  | _, _ => False
  end.
```

This unfortunately isn't quite linear in the size of the syntax tree, though it is significantly smaller. One way to achieve even more compact proof terms is to pick a more optimized representation for list membership and to convert the proposition to be an eliminator.[3] This consists of replacing $A \wedge B$ with $\forall P, (A \to B \to P) \to P$, and similar.

```
Fixpoint is_related_elim {var1 var2 : Type} (Γ : list (var1 * var2))
   (e1 : @expr var1) (e2 : @expr var2) : Prop :=
  match e1, e2 with
  | NatO, NatO => True
  | NatS e1, NatS e2 => is_related_elim Γ e1 e2
  | NatMul x1 y1, NatMul x2 y2 => forall P : Prop,
      (is_related_elim Γ x1 x2 → is_related_elim Γ y1 y2 → P) → P
  | Var v1, Var v2 => forall (P : Prop),
      (forall n, List.nth_error Γ (N.to_nat n) = Some (v1, v2) → P) → P
  | LetIn e1 f1, LetIn e2 f2 => forall P : Prop,
      (is_related_elim Γ e1 e2
       → (forall v1 v2, is_related_elim ((v1, v2) :: Γ) (f1 v1) (f2 v2))
       → P)
      → P
  | _, _ => False
  end.
```

We can now prove `is_related_elim Γ e1 e2 → is_related Γ e1 e2`.

Note that making use of the fixpoint is significantly more inconvenient than making use of the inductive; the proof of `check_is_even_expr_sound`, for example, proceeds most naturally by induction on the relatedness hypothesis. We could instead induct on one of the ASTs and destruct the other one, but this becomes quite hairy when the ASTs are indexed over their types.

Via de Bruijn

An alternative, ultimately superior, method of constructing compact proofs of relatedness involves a translation to a de Bruijn representation. Although producing

[3]The size of the proof term will still have an extra logarithmic factor in the size of the syntax tree due to representing variable indices in binary. Moreover, the size of the proof term will still be quadratic due to the fact that functions store the types of their binders. However, this representation allows proof terms that are significantly faster to construct in Coq's proof engine for reasons that are not entirely clear to us.

well-formedness proofs automatically using a verified-as-well-formed translator from de Bruijn was present already in early PHOAS papers [Chl10], we believe the trick of *round-tripping* through a de Bruijn representation is new. Additionally, there are a number of considerations that are important for achieving adequate performance which we believe are not explained elsewhere in the literature, which we discuss at the end of this subsubsection.

We can define a Boolean predicate on de Bruijn syntax representing well-formedness.

```
Fixpoint is_closed_under (max_idx : nat) (e : expr) : bool :=
  match expr with
  | NatO => true
  | NatS e => is_closed_under max_idx e
  | NatMul x y
     => is_closed_under max_idx x && is_closed_under max_idx y
  | Var n => n <? max_idx
  | LetIn v f
     => is_closed_under max_idx v && is_closed_under (S max_idx) f
  end.
Definition is_closed := is_closed_under 0.
```

Note that this check generalizes quite nicely to expressions indexed over their types—so long as type codes have decidable equality—where we can pass around a list (or more efficient map structure) of types for each variable and just check that the types are equal.

Now we can prove that whenever a de Bruijn `expr` is closed, any two PHOAS `expr`s created from that AST will be related in the empty context. Therefore, if the PHOAS `expr` we start off with is the result of converting some de Bruijn `expr` to PHOAS, we can easily prove that it's well-formed simply by running **`vm_compute`** on the `is_closed` procedure. How might we get such a de Bruijn `expr`? The easiest way is to write a converter from PHOAS to de Bruijn.

Hence we can prove the theorem `∀ e, is_closed (PHOAS_to_deBruijn e) = true ∧ e = deBruijn_to_PHOAS (PHOAS_to_deBruijn e) → Wf e`. The hypothesis of this theorem is quite easy to check; we simply run **`vm_compute`** and then instantiate it with the proof term `conj (eq_refl true) (eq_refl e)`, which is linear in the size of `e`.

Note that, unlike the initial term representation of Chlipala [Chl10], we cannot have a closed-by-construction de Bruijn representation if we want linear asymptotics. If we index each node over the size of the context—or, worse, the list of types of variables in the context—then the term representation incurs quadratic overhead in the size of the context.

```
Ltac f v term := (* reify var term *)
 lazymatch term with
 | O     => constr:(@NatO v)
 | S ?x  => let X := f v x in constr:(@NatS v X)
 | ?x*?y => let X := f v x in let Y := f v y in constr:(@NatMul v X Y)
 end.
```

Figure 3-5: Reification Without Binders in $\mathcal{L}_{tac}$

3.2 Reification

The one part of proof by reflection that we've neglected up to this point is reification. There are many ways of performing reification; in Chapter 5, we discuss 19 different ways of implementing reification, using 6 different metaprogramming facilities in the Coq ecosystem: $\mathcal{L}_{tac}$, Ltac2, Mtac2 [Gon+13b; Kai+18], type classes [SO08], canonical structures [GMT16], and reification-specific OCaml plugins (quote [Coq17b][4], template-coq [Ana+18], ours). Figure 3-5 displays the simplest case: an $\mathcal{L}_{tac}$ script to reify a tree of function applications and constants. Unfortunately, all methods we surveyed become drastically more complicated or slower (and usually both) when adapted to reify terms with variable bindings such as `let-in` or λ nodes.

We have made detailed walkthroughs and source code of these implementations available[5] in hope that they will be useful for others considering implementing reification using one of these metaprogramming mechanisms, instructive as nontrivial examples of multiple metaprogramming facilities, or helpful as a case study in Coq performance engineering. However, we do *not* recommend reading these out of general interest: most of the complexity in the described implementations strikes us as needless, with significant aspects of the design being driven by surprising behaviors, misfeatures, bugs, and performance bottlenecks of the underlying machinery as opposed to the task of reification.

There are a couple of complications that arise when reifying binders, which broadly fall into two categories. One category is the metaprogramming language's treatment of binders. In $\mathcal{L}_{tac}$, for example, the body of a function is not a well-typed term, because the variable binder refers to a nonexistent name; getting the name to actually refer to something, so that we can inspect the term, is responsible for a great deal of the complexity in reification code in $\mathcal{L}_{tac}$. The other category is any mismatch between the representation of binders in the metaprogramming language and the representation of binders in the reified syntax. If the metaprogramming language represents variables as de Bruijn indices, and we are reifying to a de Bruijn representation, then we can reuse the indices. If the metaprogramming language represents

[4]Note that this plugin was removed in Coq 8.10 [Dén18], and so our plots no longer include this plugin.

[5]`https://github.com/mit-plv/reification-by-parametricity`

variables as names, and we are reifying to a named representation, then we can reuse the names. If the representations mismatch, then we need to do extra work to align the representations, such as keeping some sort of finite map structure from binders in the metalanguage to binders in the AST.

3.3 What's Next?

Having introduced and explained proof by reflection and reflective automation, we can now introduce the main contribution of this thesis. In the next chapter we'll present the reflective framework we developed for achieving adequate performance in the Fiat Cryptography project.

Chapter 4

Engineering Challenges in the Rewriter

> [P]remature optimization is the root of all evil
>
> — Donald E. Knuth [Knu74a, p. 671]

?? discussed in detail our framework for building verified partial evaluators, going into the context, motivation, and techniques used to put the framework together. However, there was a great deal of engineering effort that went into building this tool which we glossed over. Much of the engineering effort was mundane, and we elide the details entirely. However, we believe some of the engineering effort serves as a good case study for the difficulties of building proof-based systems at scale. This chapter is about exposing the details relevant to understanding how the bottlenecks and principles identified elsewhere in this dissertation played out in designing and implementing this tool. Note that many of the examples and descriptions in this chapter are highly technical, and we expect the discussion will only be of interest to the motivated reader, familiar with Coq, who wants to see more concrete nontoy examples of the bottlenecks and principles we've been describing; other readers are encouraged to skip this chapter.

While the core rewriting engine of the framework is about 1 300 lines of code, and early simplified versions of the core engine were only about 150 lines of code[1], the correctness proofs take nearly another 8 000 lines of code! As such, this tool, developed

[1] See `https://github.com/JasonGross/fiat-crypto/blob/3b3e926e/src/Experiments/RewriteRulesSimpleNat.v` for the file `src/Experiments/RewriteRulesSimpleNat.v` from the branch `experiments-small-rewrite-rule-compilation` on `JasonGross/fiat-crypto` on GitHub.

to solve performance scaling issues in verified syntax transformation, itself serves as a good case study of some of the bottlenecks that arise when scaling proof-based engineering projects.

Our discussion in this section is organized by the conceptual structure of the normalization and pattern-matching-compilation engine; we hope that organizing the discussion in this way will make the examples more understandable, motivated, and incremental. We note, however, that many of the challenges fall into the same broad categories that are identified elsewhere in this dissertation: issues arising from the power and (mis)use of dependent types, as introduced in Subsection 1.3.1 (Dependent Types: What? Why? How?); and issues arising from API mismatches, as described in Chapter 6 (Abstraction).

4.1 Prereduction

The two biggest underlying causes of engineering challenges are expression–API mismatch, which we'll discuss in Section 4.2 (NbE vs. Pattern-Matching Compilation: Mismatched Expression APIs and Leaky Abstraction Barriers), and our desire to reduce away known computations in the rewriting engine once and for all when compiling rewriting rules, rather than again and again every time we perform a rewrite. In practice, performing this early reduction nets us an approximately 2× speed-up. We'll now discuss this early reduction and what goes into making it work.

4.1.1 What Does This Reduction Consist Of?

Recall from **??** that the core of our rewriting engine consists of three steps:

1. The first step is pattern-matching compilation: we must compile the left-hand sides of the rewrite rules to a decision tree that describes how and in what order to decompose the expression, as well as describing which rewrite rules to try at which steps of decomposition.

2. The second step is decision-tree evaluation, during which we decompose the expression as per the decision tree, selecting which rewrite rules to attempt.

3. The third and final step is to actually rewrite with the chosen rule.

The first step is performed once and for all; it depends only on the rewrite rules and not on the expression we are rewriting in. The second and third steps do, in fact, depend on the expression being rewritten, and it is in these steps that we seek to eliminate needless work early.

The key insight, which allows us to perform this precompilation at all, is that most of the decisions we seek to eliminate depend only on the *head identifier* of any application.[2] We thus augment the reduce(c) constant case of **??** in **??** by first η-expanding the identifier, before proceeding to η-expand the identifier application and perform rewriting with rewrite-head once we have an η-long form.

Now that we know what the reduction consists of, we can discuss what goes into making the reduction possible and the engineering challenges that arise.

4.1.2 CPS

Due to the pervasive use of Gallina **match** statements on terms which are not known during this compilation phase, we need to write essentially all of the decision-tree-evaluation code in continuation-passing style. This causes a moderate amount of proof-engineer overhead, distributed over the entire rewriter.

The way that CPS permits reduction under blocked **match** statements is essentially the same as the way it permits reduction of functions in the presence of unreduced **let** binders in **??** (**??**). Consider the expression

```
option_map List.length (option_map (λ x. List.repeat x 5) y)
```

where `option_map : (A → B) → option A → option B` maps a function over an option, and `List.repeat x n` creates a list consisting of `n` copies of `x`. If we fully reduce this term, we get the Gallina term

```
match
  match y with
  | Some x => Some [x; x; x; x; x]
  | None => None
  end
with
| Some x =>
    Some
      ((fix Ffix (x0 : list _) : nat :=
          match x0 with
          | [] => 0
          | _ :: x2 => S (Ffix x2)
```

[2] In order to make this simplification, we need to restrict the rewrite rules we support a little bit. In particular, we only support rewrite rules operating on η-long applications of concrete identifiers to arguments. This means that we cannot support identifiers with variable arrow structure (e.g., a variadic `curry` function) nor do we support rewriting expressions like `List.map f` to `List.map g`—we only support rewriting `List.map f xs` to `List.map g ys`.

```
          end) x)
| None => None
end
```

Consider now a CPS'd version of `option_map`:

```
Definition option_map_cps {A B} (f : A → B) (x : option A)
   : ∀ {T}, (option B → T) → T
  := λ T cont.
        match x with
        | Some x => cont (Some (f x))
        | None => cont None
        end.
```

Then we could write the somewhat more confusing term

```
option_map_cps (λ x. List.repeat x 5) y (option_map List.length)
```

whence reduction gives us

```
match y with
| Some _ => Some 5
| None => None
end
```

So we see that rewriting terms in continuation-passing style allows reduction to proceed without getting blocked on unknown terms.

Note that if we wanted to pass this list length into a further continuation, we'd need to instead write a term like

```
λ cont.
   option_map_cps (λ x. List.repeat x 5) y
     (λ ls. option_map_cps List.length ls cont)
```

which reduces to

```
λ cont. match y with
        | Some _ => cont (Some 5)
        | None => cont None
        end
```

4.1.3 Type Codes

The pattern-matching-compilation algorithm of Aehlig, Haftmann, and Nipkow [AHN08] does not deal with types. In general, unification of types is somewhat more complicated than unification of terms, because types are used as indices in terms whereas nothing gets indexed over the terms. We have two options, here:

1. We can treat terms and types as independent and untyped, simply collecting a map of unification variables to types, checking nonlinear occurrences (such as the types in `@fst ?A ?B (@pair ?A ?B ?x ?y)`) for equality, and run a typechecking pass afterwards to reconstruct well-typedness. In this case, we would consider the rewriting to have failed if the replacement is not well-typed.
2. We can perform matching on types first, taking care to preserve typing information, and then perform matching on terms afterwards, taking care to preserve typing information.

The obvious trade-off between these options is that the former option requires doing more work at runtime, because we end up doing needless comparisons that we could know in advance will always turn out a particular way. Importantly, note that Coq's reduction will not be able to reduce away these runtime comparisons; reduction alone is not enough to deduce that a Boolean equality function defined by recursion will return true when passed identical arguments, if the arguments are not also concrete terms.

Following standard practice in dependently typed languages, we chose the second option. We now believe that this was a mistake, as it's fiendishly hard to deconstruct the expressions in a way that preserves enough typing information to completely avoid the need to compare type codes for equality and cast across proofs. For example, to preserve typing information when matching for `@fst ?A ?B (@pair ?A ?B ?x ?y)`, we would have to end up with the following **match** statement. Note that the reader is not expected to understand this statement, and the author was only able to construct it with some help from Coq's typechecker.

```
| App f v =>
 let f :=
  match f in expr t return option (ident t) with
  | Ident idc => Some idc
  | _ => None
  end in
 match f with
 | Some maybe_fst =>
   match v in expr s return ident (s -> _) -> _ with
   | App f y =>
```

```
    match f in expr _s
     return
      match _s with arrow b _ => expr b | _ => unit end
      -> match _s with arrow _ ab => ident (ab -> _) | _ => unit end
      -> _
    with
    | App f x =>
      let f :=
       match f in expr t return option (ident t) with
       | Ident idc => Some idc
       | _ => None
       end in
      match f with
      | Some maybe_pair =>
        match maybe_pair in ident t
         return
          match t with arrow a _ => expr a | _ => unit end
          -> match t with arrow a (arrow b _) => expr b | _ => unit end
          -> match t with arrow a (arrow b ab) => ident (ab -> _) | _ => unit end
          -> _
        with
        | @pair a b =>
          fun (x : expr a) (y : expr b) (maybe_fst : ident _) =>
           let is_fst := match maybe_fst with fst => true | _ => false end in
           if is_fst
           then … (* now we can finally do something with a, b, x, and y *)
           else …
        | _ => …
        end x
      | None => …
      end
    | _ => …
    end y
  | _ => …
  end maybe_fst
| None => …
end
```

This is quite the mouthful.

Furthermore, there are two additional complications. First, this sort of match expression must be generated *automatically*. Since pattern-matching evaluation happens on *lists* of expressions, we'd need to know exactly what each match reveals about the types of all other expressions in the list. Additionally, in order to allow reduction to

happen where it should, we need to make sure to match the head identifier *first*, without convoying it across matches on unknown variables. Note that in the code above, we did not follow this requirement, as it would complicate the `return` clauses even more (presuming we wanted to propagate typing information as we'd have to in the general case rather than cutting corners). The convoy pattern, for those unfamiliar with it, is explained in detail in Chapter 8 ("More Dependent Types") of *Certified Programming with Dependent Types* [Chl13].

Second, trying to prove anything about functions written like this is an enormous pain. Because of the intricate dependencies in typing information involved in the convoy pattern, Coq's `destruct` tactic is useless. The `dependent destruction` tactic is sometimes able to handle such goals, but even when it can, it often introduces a dependency on the axiom `JMeq_eq`, which is equivalent to assuming *uniqueness of identity proofs* (UIP), that all proofs of equality are equal—note that this contradicts, for example, the popular univalence axiom of homotopy type theory [Uni13]. In order to prove anything about such functions without assuming UIP, the proof effectively needs to replicate the complicated `return` clauses of the function definition. However, since they are not to be replicated exactly but merely be generated from the same insights, such proof terms often have to be written almost entirely by hand. These proofs are furthermore quite hard to maintain, as even small changes in the structure of the function often require intricate changes in the proof script.

Due to a lack of foresight and an unfortunate reluctance to take the design back to the drawing board after we already had working code, we ended up mixing these two approaches, getting, not quite the worst of both worlds, but definitely a significant fraction of the pain of both worlds: We must deal with both the pain of indexing our term unification information over our type unification information, and we must still insert typecasts in places where we have lost the information that the types will line up.

4.1.4 How Do We Know What We Can Unfold?

Coq's built-in reduction is somewhat limited, especially when we want it to have reasonable performance. This is, after all, a large part of the problem this tool is intended to solve.

In practice, we make use of three reduction passes; that we cannot interleave them is a limitation of the built-in reduction:

1. First, we unfold everything except for a specific list of constants; these constants are the ones that contain computations on information not fully known at preevaluation time.
2. Next, we unfold all instances of a particular set of constants; these constants

are the ones that we make sure to only use when we know that inlining them won't incur extra overhead.

3. Finally, we use `cbn` to simplify a small set of constants in only the locations that these constants are applied to constructors.

Ideally, we'd either be able to do the entire simplification in the third step, or we'd be able to avoid the third step entirely. Unfortunately, Coq's reduction is not fast enough to do the former, and the latter requires a significant amount of effort. In particular, the strategy that we'd need to follow is to have two versions of every function which sometimes computes on known data and sometimes computes on unknown data, and we'd need to track in all locations which data is known and which data is unknown.

We already track known and unknown data to some extent (see, for example, the `known` argument to the `rIdent` constructor discussed below). Additionally, we have two versions of a couple of functions, such as the bind function of the option monad, where we decide which to use based on, e.g., whether or not the option value that we're binding will definitely be known at prereduction time.

Note that tracking this sort of information is nontrivial, as there's no help from the typechecker.

We'll come back to this in Subsection 4.4.1.

4.2 NbE vs. Pattern-Matching Compilation: Mismatched Expression APIs and Leaky Abstraction Barriers

We introduced normalization by evaluation (NbE) [BS91] in **??** and expanded on it in **??** as a way to support higher-order reduction of λ-terms. The termination argument for NbE proceeds by recursion on the type of the term we're reducing. In particular, the most natural way to define these functions in a proof assistant is to proceed by structural recursion on the type of the term being reduced. This feature suggests that using intrinsically-typed syntax is more natural for NbE, and we saw in Section 3.1.3 that denotation functions are also simpler on syntax that is well-typed by construction.

However, the pattern-matching-compilation algorithm of Maranget [Mar08] inherently operates on untyped syntax. We thus have four options:

(1) use intrinsically well-typed syntax everywhere, paying the cost in the pattern-matching compilation and evaluation algorithm;

(2) use untyped syntax in both NbE and rewriting, paying the associated costs in NbE, denotation, and in our proofs;

(3) use intrinsically well-typed syntax in most passes and untyped syntax for pattern-matching compilation;

(4) invent a pattern-matching compilation algorithm that is well-suited to type-indexed syntax.

We ultimately chose option (3). I was not clever enough to follow through on option (4), and while options (1) and (2) are both interesting, option (3) seemed to follow the well-established convention of using whichever datatype is best-suited to the task at hand. As we'll shortly see, all of these options come with significant costs, and (3) is not as obviously a good choice as it might seem at first glance.

4.2.1 Pattern-Matching Evaluation on Type-Indexed Terms

While the cost of performing pattern-matching compilation on type-indexed terms is noticeable, it's relatively insignificant compared to the cost of evaluating decision trees directly on type-indexed terms. In particular, pattern-matching compilation effectively throws away the type information whenever it encounters it; whether we do this early or late does not matter much, and we only perform this compilation once for any given set of rewrite rules.

By contrast, evaluation of the decision tree needs to produce *term ASTs* that are used in rewriting, and hence we need to preserve type information in the input. Recall from **??** that decision-tree evaluation operates on lists of terms. Here already we hit our first snag: if we want to operate on well-typed terms, we must index our lists over a list of types. This is not so bad, but recall also from **??** that decision trees contain four constructors:

- `TryLeaf k onfailure`: Try the k^{th} rewrite rule; if it fails, keep going with `onfailure`.
- `Failure`: Abort; nothing left to try.
- `Switch icases app_case default`: With the first element of the vector, match on its kind; if it is an identifier matching something in `icases`, which is a list of pairs of identifiers and decision trees, remove the first element of the vector and run that decision tree; if it is an application and `app_case` is not `None`, try the `app_case` decision tree, replacing the first element of each vector with the two elements of the function and the argument it is applied to; otherwise, do not modify the vectors and use the `default` decision tree.

- `Swap i cont`: Swap the first element of the vector with the $i^{\texttt{th}}$ element (0-indexed) and keep going with `cont`.

The first two constructors are not very interesting, as far as overhead goes, but the third and fourth constructors mandate quite involved adaptations for operating on well-typed terms.

Note that the type of `eval_decision_tree` would be something like $\forall$ `{T : Type} (d : decision_tree) (ts : list type) (es : exprlist ts) (K :` $\mathbb{N} \to$ `exprlist ts` $\to$ `option T), option T` where the $\mathbb{N}$ argument to the continuation describes which rewrite rule to invoke. Note that we are using continuation-passing style here to achieve adequate reduction behavior inside Coq. Note that this is the same reason we introduced in **??**, except one metalevel up.

We cover the `Swap` case first, because it is simpler. To perform a `Swap`, we must exchange two elements of the type-indexed list. Hence we need both to swap the elements of the list of types and then to have a separate, dependently typed swap function for the vector of expressions. Moreover, since we need to undo the swapping inside the continuation, we must have a *separate* unswap function on expression vectors which goes from a swapped type list to the original one. We could instead elide the swap node, but then we could no longer use matching, `hd`, and `tl` to operate on the expressions and would instead need special operations to do surgery in the middle of the list, in a way that preserves type indexing.

To perform a `Switch`, we must break apart the first element of our type-indexed list, determining whether it is an application, and identifier, or other. Note that even with dependent types, we cannot avoid needing a failure case for when the type-indexed list is empty, even though such a case should never occur because good decision trees will never have a `Switch` node after consuming the entire vector of expressions. This failure case cannot be avoided because there is no type-level relation between the expression vector and the decision tree. This mismatch—the need to include failure cases that one might expect to be eliminated by dependent typing information—is a sign that the amount of dependency in the types is wrong. It may be too little, whence the developer should see if there is a way to incorporate the lack of error into the typing information (which in this case would require indexing the type of the decision tree over the length of the vector[3]). It may alternatively be too much dependent typing, and the developer might be well-served by removing more dependency from

[3]Note that choosing to index the decision tree over the length of the vector severely complicates our ability to avoid separate `swap` and `unswap` functions by indexing into the middle of the vector. We'd need to use some sort of finite type to ensure the indices are not too large, and we'd need to be very careful to write dependently typed middle-of-the-vector surgery operations which are judgmentally invertible on the effects that they have on the length of the vector. Here we see an example of how dependent types introduce coupling between seemingly unrelated design decisions, which is a large part of why abstraction barriers are so essential, as we'll discuss in Section 6.2 (When and How To Use Dependent Types Painlessly).

the types and letting more things fall into the error case.

After breaking apart the first element, we must convoy the continuation across the **match** statement so that we can pass an expression vector of the correct type to the continuation K. In code, this branch might look something like

```
…
| Switch icases app_case default
  => match es in exprlist ts
       return (exprlist ts → option T) → option T
     with
     | [] => λ _, None
     | e :: es
       => match e in expr t
            return (exprlist (t :: ts) → option T) → option T
          with
          | App s d f x => λ K,
              let K' : exprlist ((s → d) :: s :: ts)
                  (* new continuation to pass on recursively *)
                := λ es', K (App (hd es') (hd (tl es')) :: tl (tl es')) in
              … (* do something with app_case *)
          | Ident t idc => λ K,
              let K' : exprlist ts
                  (* new continuation to pass on recursively *)
                := λ es', K (Ident idc :: es') in
              … (* do something with icases *)
          | _ => λ K, … (* do something with default *)
          end
     end K
…
```

Note that `hd` and `tl` *must* be type-indexed, and we *cannot* simply match on `es'` in the `App` case; there is no way to preserve the connection between the types of the first two elements of `es'` inside such a **match** statement.

This may not look too bad, but it gets worse. Since the **match** on `e` will not be known until we are actually doing the rewriting on a concrete expression, and the continuation is convoyed across this **match**, there is no way to evaluate the continuation during compilation of rewrite rules. If we don't want to evaluate the continuation early, we'd have to be very careful not to duplicate it across all of the decision-tree evaluation cases, as we might otherwise incur a superlinear runtime factor in the number of rewrite rules. As noted in Section 4.1, our early reduction nets us a 2× speedup in runtime of rewriting and is therefore relatively important to be able to do.

Here we see something interesting, which does not appear to be as much of a concern in other programming languages: the representation of our data forces our hand about how much efficiency can be gained from precomputation, even when the representation choices are relatively minor.

4.2.2 Untyped Syntax in NbE

There is no good way around the fact that NbE requires typing information to argue termination. Since NbE will be called on subterms of the overall term, even if we use syntax that is not guaranteed to be type-correct, we must still store the type information in the nodes of the AST.

Furthermore, as we say in Section 3.1.3 (de Bruijn Indices), converting from untyped syntax to intrinsically typed syntax, as well as writing a denotation function, requires either that all types be nonempty or that we carry around a proof of well-typedness to use during recursion. As discussed in Chapter 6 and specifically in Section 6.2 (When and How To Use Dependent Types Painlessly), needing to mix proofs with programs is often a big warning flag, unless the mixing can be hidden behind a well-designed API. However, if we are going to be hiding the syntax behind an API of being well-typed, it seems like we might as well just use intrinsically well-typed syntax, which naturally inhabits that API. Furthermore, unlike in many cases where the API is best treated as opaque everywhere, here the API mixing proofs and programs needs to have adequate behavior under reduction and ought to have good behavior even under partial reduction. This severely complicates the task of building a good abstraction barrier, as we not only need to ensure that the abstraction barrier does not need to be broken in the course of term-building and typechecking, but we must also ensure that the abstraction barrier can be broken in a principled way via reduction without introducing significant overhead.

4.2.3 Mixing Typed and Untyped Syntax

The third option is to use whichever datatype is most naturally suited for each pass and to convert between them as necessary. This is the option that we ultimately chose, and the one, we believe, that would be most natural to choose to engineers and developers coming from nondependently typed languages.

There are a number of considerations that arose when fleshing out this design and a number of engineering pain points that we encountered. The theme to all of these, as we will revisit in Chapter 6, is that imperfectly opaque abstraction barriers cause headaches in a nonlocal manner.

We got lucky, in some sense, that the rewriting pass *always* has a well-typed default option: do no rewriting. Hence we do not need to worry about carrying around

proofs of well-typedness, and this avoids some of the biggest issues described in Subsection 4.2.2 (Untyped Syntax in NbE).

The biggest constraint driving our design decisions is that we need conversion between the two representations to be $\mathcal{O}(1)$; if we need to walk the entire syntax tree to convert between typed and untyped representations at every rewriting location, we'll incur quadratic overhead in the size of the term being rewritten. We can actually relax this constraint a little bit: by designing the untyped representation to be completely evaluated away during the compilation of rewrite rules, we can allow conversion from the untyped syntax to the typed syntax to walk any part of the term that already needed to be revealed for rewriting, giving us amortized constant time rather than truly constant time. As such, we need to be able to embed well-typed syntax directly into the nontype-indexed representation at cost $\mathcal{O}(1)$.

As the entire purpose of the untyped syntax is to (a) allow us to perform matching on the AST to determine which rewrite rule to use, and furthermore (b) allow us to reuse the decomposition work so as to avoid needing to decompose the term multiple times, we need an inductive type which can embed PHOAS expressions and has separate nodes for the structure that we need, namely application and identifiers:

```
Inductive rawexpr : Type :=
| rIdent (known : bool) {t} (idc : ident t) {t'} (alt : expr t')
| rApp (f x : rawexpr) {t} (alt : expr t)
| rExpr {t} (e : expr t)
| rValue {t} (e : NbE_t t).
```

There are three perhaps-unexpected things to note about this inductive type, which we will discuss in later subsections:

1. The constructor `rValue` holds an NbE value of the type $\texttt{NbE}_\texttt{t}$ introduced in **??**. We will discuss this in Section 4.7 (Delayed Rewriting in Variable Nodes).
2. The constructors `rIdent` and `rExpr` hold "alternate" PHOAS expressions. We will discuss this in Subsection 4.4.2 (Revealing "Enough" Structure).
3. The constructor `rIdent` has an extra Boolean `known`. We will discuss this in Section 4.4.1 (The `known` argument).

With this inductive type in hand, it's easy to see how `rExpr` allows us $\mathcal{O}(1)$ embedding of intrinsically typed `expr`s into untyped `rawexpr`s.

While it's likely that sufficiently good abstraction barriers around this datatype would allow us to use it with relative ease, we did not succeed in designing good enough

abstraction barriers. The bright side of this failure is that we now have a number of examples for this dissertation of ways in which inadequate abstraction barriers cause overhead in terms of both the intricacy of definitions and theorems and the size of their statements and proofs as well as, to a lesser extent, the running time of proof generation.

We will discuss the many issues that arise from leaks in this abstraction barrier in the upcoming subsections.

4.2.4 Pattern-Matching Compilation Made for Intrinsically Typed Syntax

The cost of this fourth option is the cleverness required to come up with a version of the pattern-matching compilation which, rather than being hindered by types in its syntax, instead puts them to good use. Lacking this cleverness, we were unable to pay the requisite cost and hence have not much to say in this section.

4.3 Patterns with Type Variables – The Three Kinds of Identifiers

We have one final bit of infrastructure to explain and motivate before we have enough of the structure sketched out to give all of the rest of the engineering challenges: representing the identifiers. Recall from **??** (**??**) that we automatically emit an inductive type describing all available primitive functions.

When deciding how to represent identifiers, there are roughly three options we have to choose from:

1. We could use an untyped representation of identifiers, such as Coq strings (as in Anand et al. [Ana+18], for example) or integers indexing into some finite map.

2. We could index the expression type over a finite map of valid identifiers and use dependent typing to ensure that we only have well-typed identifiers.

3. We could have a fixed set of valid identifiers, using types to ensure that we have only valid expressions.

The first option results in expressions that are not always well-typed. As discussed in Chapter 6 and seen in the preceding sections, having leaky abstraction barriers is

often worse than having none at all, and we expect that having partially well-typed expressions would be no exception.

The second option is probably the way to go if we want truly extensible identifier sets. There are two issues. First, this adds a linear overhead in the number of identifiers—or more precisely, in the total size of the types of the identifiers—because every AST node will store a copy of the entire finite map. Second, because our expression syntax is simply typed, polymorphic identifiers pose a problem. To support identifiers like `fst` and `snd`, which have types `∀ A B, A * B → A` and `∀ A B, A * B → B` respectively, we must either replicate the identifiers with all of the ways they might be applied, or else we must add support in our language for dependent types or for explicit type polymorphism.

Instead, we chose to go with the third option, which we believe is the simplest. The inductive type of identifiers is indexed over the type of the identifier, and type polymorphism is expressed via metalevel arguments to the constructor. So, for example, the identifier code for `fst` takes two type-code arguments `A` and `B` and has type `ident (A * B → A)`. Hence all fully applied identifier codes have simple types (such as `A * B → A`), and our inductive type still supports polymorphic constants. An additional benefit of this approach is that unification of identifiers is just pattern matching in Gallina, and hence we can rely on the pattern-matching-compilation schemes of Coq's fast reduction machines, or the OCaml compiler itself, to further speed up our rewriting.

Aside: Why Use Pattern-Matching Compilation At All? Given the fact that, after prereduction, there is no trace of the decision tree remaining, one might ask why we use pattern-matching compilation at all, rather than just leaving it to the pattern-matching compiler of Coq or OCaml to be performant. We have three answers to this question.

The first, perhaps most honest answer is that it is a historical accident; we prematurely optimized this part of the rewriting engine when writing it.

The second answer is that pattern-matching compilation is a good abstraction barrier for factoring out the work of revealing enough structure from the work of unifying a pattern with an expression. Said another way, even though we reduce away the decision tree and its evaluation, there is basically no wasted work; removing pattern-matching compilation while preserving all the benefits would effectively just be inlining all of the functions, and there would be no dead code revealed by this inlining.

The third and final answer is that it allows us to easily prune useless work. The pattern-matching-compilation algorithm naturally prunes away patterns that can be known to not work, given the structure that we've revealed. By contrast, if we just

record what information we've already revealed as we're performing pattern unification, it's quite tricky to avoid decomposition which can be known to be useless based on only the structure that's been revealed already.

Consider, for example, rewriting with two rules whose left-hand sides are $x + (y + 1)$ and $(a + b) + (c * 2)$. When revealing structure for the first rewrite rule, the engine will first decompose the (unknown) expression into the application of the + identifier to two arguments, and then decompose the second argument into the application of the + identifier to two arguments, and then finally decompose the second inner argument into a literal identifier to check if it is the literal 1. If the decomposition succeeds, but the literal is not 1 (or if the second inner argument is not a literal at all), then rewriting will fall back to the second rewrite rule. If we are doing structure decomposition in the naïve way, we will then decompose the outer first argument (bound to x in the first rewrite rule) into the application of the identifier + to two arguments. We will then attempt to decompose the second outer argument into the application of the identifier $*$ to two arguments. Since there is no way an identifier can be both + and $*$, this decomposition will fail. However, we could have avoided doing the work of decomposing x into $a + b$ by realizing that the second rewrite rule is incompatible with the first; this is exactly what pattern-matching compilation and decision-tree evaluation does.

Pattern Matching For Rewriting We now arrive at the question of how to do pattern matching for rewriting with identifiers. We want to be able to support type variables, for example to rewrite `@fst ?A ?B (@pair ?A ?B ?x ?y)` to `x`. While it would arguably be more elegant to treat term and type variables identically, doing this would require a language supporting dependent types, and we are not aware of any extension of PHOAS to dependent types. Extensions of HOAS to dependent types are known [McB10], but the obvious modifications of such syntax that in the simply typed case turn HOAS into PHOAS result in infinite self-referential types in the dependently typed case.

As such, insofar as we are using intrinsically well-typed syntax at all, we need to treat type variables separately from term variables. We need three different sorts of identifiers:

- identifiers whose types contain no type variables, for use in external-facing expressions and the denotation function,
- identifiers whose types are permitted to contain type variables, for use in patterns, and
- identifiers with no type information, for use in pattern-matching compilation.

The first two are relatively self-explanatory. The third of these is required because

pattern-matching compilation proceeds in an untyped way; there's no obvious place to keep the typing information associated to identifiers in the decision tree, which must be computed before we do any unification, type variables or otherwise.

We could, in theory, use a single inductive type of type codes for all three of these. We could parameterize the inductive of type codes over the set of free type variables (or even just over a Boolean declaring whether or not type variables are allowed) and conventionally use the type code for unit in all type-code arguments when building decision trees.

This sort of reuse, however, is likely to introduce more problems than it solves.

The identifier codes used in pattern-matching compilation must be untyped, to match the decision we made for expressions in Section 4.2. Having them conventionally be typed pattern codes instantiated with unit types is, in some sense, just more opportunity to mess up and try to inspect the types when we really shouldn't. There is a clear abstraction barrier here, of having these identifier codes not carry types, and we might as well take advantage of that and codify the abstraction barrier in our code.

The question of type variables is more nuanced. If we are only tracking whether or not a type is allowed to have type variables, then we might as well use two different inductive types; there is not much benefit to indexing the type codes over a Boolean rather than having two copies of the inductive, for there's not much that can be done generically in whether or not type variables are allowed. Note also that we must track at least this much information, for identifiers in expressions passed to the denotation function must not have uninstantiated type variables, and identifiers in patterns must be permitted to have uninstantiated type variables.

However, there is some potential benefit to indexing over the set of uninstantiated type variables. This might allow us to write type signatures for functions that guarantee some invariants, possibly allowing for easier proofs. However, it's not clear to us where this would actually be useful; most functions already care only about whether or not we permit type variables at all. Our current code in fact performs a poor approximation of this strategy in some places: we index over the entire pattern where indexing over the free variables of the pattern would suffice.

This unneeded indexing enormously complicates the code and theorems and is yet another example of how poorly designed abstraction barriers incur outsized overhead. Rewrite-rule replacements are expressed as dependently typed towers indexed first over the type variables of a pattern and then again over the term variables. This design is a historical artifact, from when we expected to be writing rewrite rule ASTs by hand rather than reifying them from Gallina and found the curried towers more convenient to write. This design, however, is absolutely a mistake, especially given the concession we make in Subsection 4.1.3 (Type Codes) to not track enough typing

information to avoid all typechecking.

While indexing over only the set of permitted type variables would simplify proofs significantly, we'd benefit even more by indexing only over whether or not we permit type variables at all. None of our proofs are made simpler by tracking the set of permitted type variables rather than just whether or not that set is empty.

4.4 Preevaluation Revisited

Having built up enough infrastructure to give a bit more in the way of code examples, we now return to the engineering challenges posed by reducing early, first investigated in Section 4.1

4.4.1 How Do We Know What We Can Unfold?

We can now revisit Subsection 4.1.4 in a bit more detail.

The `known` argument We noted in Subsection 4.2.3 the `known` argument of the `rIdent` constructor of `rawexpr`. This argument is used to track what sorts of operations can be unfolded early. In particular, if a given identifier has no type arguments (for example, the identifier coding for addition on $\mathbb{Z}$s), and we have already matched against it, then when performing further matches to unify with other patterns, we can directly match it against pattern identifiers. By contrast, if the identifier has not yet been matched against, or if it has unknown type arguments, we cannot guarantee that **`match`**es will reduce. Tracking this information adds a not-insignificant amount of nuance and intricacy to the code.

Consider the following two cases, where we will make use of both `true` and `false` for the `known` argument.

First, let us consider the simpler case of looking for examples where `known` will be `false`. As a toy example, suppose we are rewriting with the rule `@List.map A B f (x::xs) = f x :: List.map f xs` and the rule `@List.map (option A) (option B) (option_map f) (List.map (@Some A) xs) = @List.map A (option B) (fun x => Some (f x)) xs`. When decomposing structure for the first rewrite rule, we will match on the head identifier to see if it is `List.map`. Supposing that the final argument is not a cons cell, we will fall back to the second rewrite rule. While we know that the first identifier is a `List.map`, we do not know its type arguments. Therefore, when we want to try to substitute with the second rewrite rule, we must match on the type structure of the first type argument to `List.map` to see if it is an option, and, if so, extract the underlying type to put into unification data. However, this

decomposition will block on the type arguments to `List.map`, so we don't want to unfold it fully during early reduction. Note that the first rewrite rule is not really necessary in this example; the essential point is that we don't want to be unfolding complicated recursive matches on the type structure that are not going to reduce.[4]

There are two cases where we want to reduce the `match` on an identifier. One of them is when the identifier is known from the initial η-expansion of identifiers discussed in Subsection 4.1.1 (note that this is distinct from the η-expansion of identifier applications), and the identifier has no type arguments.[5] The other case is when we have tested an identifier against a pattern identifier, and it has no type arguments. In this case, when we eventually get around to collecting unification data for this identifier, we know that we can reduce away the check on this identifier. Whether or not the overhead is worth it in this second case is unclear; the design of this part of the rewriting engine suffers from the lack of a unified picture about what, exactly, is worth reducing, and what is not.

Gratuitous Dependent Types: How much do we actually want to unfold? When computing the replacement of a given expression, how much do we want to unfold? Here we encounter a case of premature optimization being the root of, if not evil, at least headaches. The simplest path to take here would be to have unification output a map of type-variable indices to types and a map of expression-variable indices to expressions of unknown types. We could then have a function, not to be unfolded early, which substitutes the expressions into some untyped representation of terms and then performs a typechecking pass to convert back to a well-typed expression.

Instead, we decided to reduce as much as we possibly could. Following the common practice of eager students looking to use dependent types, we defined a dependently typed data structure indexed over the pattern type which holds the mapping of each pattern type variable to a corresponding type. While this mapping cannot be fully computed at rewrite-rule-compilation time—we may not know enough type structure in the `rawexpr`—we can reduce effectively all of the lookups by turning them into matches on this mapping which *can* be reduced. This, unfortunately, complicates our proofs significantly while likely not providing any measurable speedup, serving only as yet another example of the overhead induced by needless dependency at the type level.

[4] In the current codebase, removing the first rewrite rule would, unfortunately, result in unfolding of the matching on the type structure, due to an oversight in how we compute the `known` argument. See the next footnote for more details.

[5] In our current implementation we don't actually check that the identifier has no type arguments in this case. This is an oversight, and the correct design would be able to distinguish between "this identifier is known and it has no type arguments", "this identifier is known but it has unknown type arguments", and "this identifier is completely unknown". Failure to distinguish these cases does not seem to cause too much trouble, because the way the code is structured luckily ensures that we only match on the type arguments once, and because everything is CPS'd, this matching does not block further reduction.

4.4.2 Revealing "Enough" Structure

We noted in Subsection 4.2.3 that the constructors `rIdent` and `rExpr` hold "alternate" PHOAS expressions. We now discuss the reason for this.

Consider the example where we have two rewrite rules: that $(x+y)+1 = x+(y+1)$ and that $x + 0 = x$. If we have the expression $(a + b) + 0$, we would first try to match this against $(x + y) + 1$. If we didn't store the expression $a + b$ as a PHOAS expression and had it only as a `rawexpr`, then we'd have to retypecheck it, inserting casts as necessary, in order to get a PHOAS expression to return from unification of $a + b$ with x in $x + 0$.

Instead of incurring this overhead, we store the undecomposed PHOAS expression in the `rawexpr`, allowing us to reuse it when no more decomposition is needed. This does, however, complicate proofs: we need to talk about matching the revealed and unrevealed structure, sometimes just on the type level, and other times on both the term level and the type level.

4.5 Monads: Missing Abstraction Barriers at the Type Level

We introduce in `??` the `UnderLets` monad for `let` lifting, which we inline into the definition of the $\texttt{NbE}_\texttt{t}$ value type. We use two other monads in the rewriting engine: the option monad, to encode possible failure of rewrite-rule side conditions and substitutions, and the CPS monad discussed in Subsection 4.1.2.

Although we introduce a bit of syntactic sugar for monadic binds in an ad-hoc way, we do not fully commit to a monadic abstraction barrier in our code. This lack of principle incurs overhead when we have to deal with mismatched monads in different functions, especially when we haven't ordered the monadic applications in a principled way.

The simplest example of this overhead is in our mixing of the option and CPS monads in `eval_decision_tree`. The type of `eval_decision_tree` is $\forall$ `{T : Type} (es : list rawexpr) (d : decision_tree) (K :` $\mathbb{N} \to$ `list rawexpr → option T), option T`. Recall that the function of `eval_decision_tree` is to reveal structure on the list of expressions `es` by evaluating the decision tree `d`, calling `K` to perform rewriting with a given rewrite rule (referred to by index) whenever it hits a leaf node, and continuing on when `K` fails with `None`. What is the correctness condition for `eval_decision_tree`?

We need two correctness conditions. One of them is that, if `eval_decision_tree` succeeds at all, it is equivalent to calling `K` on some index with some list of expres-

sions which is appropriately equivalent to `es`. (See Subsection 4.7.1 for discussion of what, exactly, "equivalent" means in this case.) This is the interpretation correctness condition.

The other correctness condition is significantly more subtle and corresponds to the property that the rewriter must map related PHOAS expressions to related PHOAS expressions. This one is a monster. We present the code before explaining it to show just how much of a mouthful it is.

```
Lemma wf_eval_decision_tree {T1 T2} G d
 : ∀ (P : option T1 → option T2 → Prop)
     (HPNone : P None None)
     (ctx1 : list (@rawexpr var1))
     (ctx2 : list (@rawexpr var2))
     (ctxe : list { t : type & @expr var1 t * @expr var2 t }%type)
     (Hctx1 : length ctx1 = length ctxe)
     (Hctx2 : length ctx2 = length ctxe)
     (Hwf : ∀ t re1 e1 re2 e2,
         List.In ((re1, re2), existT _ t (e1, e2))
                 (List.combine (List.combine ctx1 ctx2) ctxe)
         → @wf_rawexpr G t re1 e1 re2 e2)
     cont1 cont2
     (Hcont : ∀ n ls1 ls2,
         length ls1 = length ctxe
         → length ls2 = length ctxe
         → (forall t re1 e1 re2 e2,
               List.In ((re1, re2), existT _ t (e1, e2))
                       (List.combine (List.combine ls1 ls2) ctxe)
               → @wf_rawexpr G t re1 e1 re2 e2)
         → (cont1 n ls1 = None ↔ cont2 n ls2 = None)
           ∧ P (cont1 n ls1) (cont2 n ls2)),
   P (@eval_decision_tree var1 T1 ctx1 d cont1)
     (@eval_decision_tree var2 T2 ctx2 d cont2).
```

This is one particular way to express the following meaning: Suppose that we have two calls to `eval_decision_tree` with different PHOAS `var` types, different return types `T1` and `T2`, different continuations `cont1` and `cont1`, different untyped expression lists `ctx1` and `ctx2`, and the same decision tree. Suppose further that we have two lists of PHOAS expressions and a relation relating elements of `T1` to elements of `T2`. Let us assume the following properties of the expression lists and the continuations: The two lists of untyped `rawexpr`s (`ctx1` and `ctx2`) match with each other and the two lists of typed expressions, and all of the types line up. The two continuations, when fed identical indices and fed lists of `rawexpr`s which match with the given lists of typed expressions, either both succeed with related outputs or both fail. Then we

can conclude that the calls to `eval_decision_tree` either both succeed with related outputs or both fail. Note, importantly, that we connect the lists of `rawexpr`s fed to the continuations with the lists of `rawexpr`s fed to `eval_decision_tree` only via the lists of typed expressions.

Why do we need such complication here? The `eval_decision_tree` function makes no guarantee about how much of the expression it reveals, but we must capture the fact that related PHOAS inputs result in the *same* amount of revealing, however much revealing that is. We do, however, also guarantee that the revealed expressions are both related to each other as well as to the original expressions, modulo the amount of revealing. Finally, the continuations that we use assume that enough structure is revealed and hence are not guaranteed to be independent of the level of revealing.

There are a couple of ways that this correctness condition might be simplified, all of which essentially amount to better enforcement of abstraction barriers.

The function that rewrites with a particular rule relies on the invariant that the function `eval_decision_tree` reveals enough structure. This breaks the abstraction barrier that rewriting with a particular rule is only supposed to care about the expression structure. If we enforced this abstraction barrier, we'd no longer need to talk about whether or not two `rawexpr`s had the same level of revealed structure, which would vastly simplify the definition `wf_rawexpr` (discussed more in the upcoming Subsection 4.7.2). Furthermore, we could potentially remove the lists of typed expressions, mandating only that the lists of `rawexpr`s be related to each other.

Finally, we could split apart the behavior of the continuation from the behavior of `eval_decision_tree`. Since the behavior of the continuations could be assumed to not depend on the amount of revealed structure, we could prove that invoking `eval_decision_tree` on any such "good" continuation returned a result *equal* to invoking the continuation on the same list of `rawexpr`s, rather than merely one equivalent to it modulo the amount of revealing. This would bypass the need for this lemma entirely, allowing us to merely strengthen the previous lemma used for interpretation correctness.

So here we see that a minor leak in an abstraction barrier (allowing the behavior of rewriting to depend on how much structure has been revealed) can vastly complicate correctness proofs, even forcing us to break other abstraction barriers by inlining the behavior of various monads.

4.6 Rewriting Again in the Output of a Rewrite Rule

We now come to the feature of the rewriter that took the most time and effort to deal with in our proofs and theorem statements: allowing some rules to be designated as subject to a second bottomup rewriting pass in their output. This feature is important for allowing users to express one operation (for example, `List.flat_map`) in terms of other operations (for example, `list_rect`) which are themselves subject to reduction.

The technical challenge, here, is that the PHOAS `var` type of the input of normalization by evaluation is not the same as the `var` type of the output. Hence the rewrite-rule replacement phase of rules marked for subsequent rewriting passes must change the `var` type when they do replacement. This can be done, roughly, by wrapping arguments passed in to the replacement rule in an extra layer of `Var` nodes.

However, this incurs severe cost in phrasing and proving the correctness condition of the rewriter. While most of the nitty-gritty details are beyond the scope even of this chapter, we will look at one particular implication of supporting this feature in Subsection 4.7.2 (Which Equivalence Relation?).

4.7 Delayed Rewriting in Variable Nodes

We saw in Subsection 4.2.3 that the `rawexpr` inductive has separate constructors for PHOAS expressions and for $\texttt{NbE}_\text{t}$ values. The reason for this distinction lies at the heart of fusing normalization by evaluation and pattern-matching compilation.

Consider rewriting in the expression `List.map (λ x. y + x) [0; 1]` with the rules `x + 0 = x`, and `List.map f [x ; … ; y] = [f x ; … ; f y]`. We want to get out the list `[y; y + 1]` and *not* `[y + 0; y + 1]`. In the bottomup approach, we first perform rewriting on the arguments to `List.map` before applying rewriting to `List.map` itself. Although it would seem that no rewrite rule applies to either argument, in fact what happens is that `(λ x. y + x)` becomes an $\texttt{NbE}_\text{t}$ thunk which is waiting for the structure of `x` before deciding whether or not rewriting applies. Hence when doing decision-tree evaluation, it's important to keep this thunk waiting, rather than forcing it early with a generic variable node. The `rValue` constructor allows us to do this. The `rExpr` constructor, by contrast, holds expressions which we are allowed to do further matching on.

How does the use of these different constructors show up? Recall from **??** in **??** that we put constants into η-long application form by calling reflect at the base case of reduce(c). When performing this η-expansion, we build up a `rawexpr`. When we encounter an argument with an arrow type, we drop it directly into an `rValue` constructor, marking it as not subject to structure revealing. When we encounter

an argument whose type is not an arrow, we can guarantee that there is no thunked rewriting, and so we can put the value into an `rExpr` constructor, marking it as subject to structure decomposition.

One might ask: since we distinguish the creation of `rExpr` and `rValue` on the basis of the argument's type, could we not just use the same constructor for both? The reason we cannot do this is that when revealing structure, we may decompose an expression in an `rExpr` node into an application of an expression to another expression. In this case, the first of these will have an arrow type, and both must be placed into the `rExpr` constructor and be marked as subject to further decomposition. Hence we cannot distinguish these cases just on the basis of the type, and we do in fact need two constructors.

4.7.1 Relating Expressions and Values

First, some background context: When writing PHOAS compiler passes, there are in general two correctness conditions that must be proven about them. The first is a soundness theorem. In Figure 3.1.3, we called this theorem `check_is_even_expr_sound`. For compiler passes that produce syntax trees, this theorem will relate the denotation of the input AST to the denotation of the output AST and might hence alternatively be called a *semantics-preservation* theorem, or an *interpretation-correctness* theorem. The second theorem, only applicable to compiler passes that produce ASTs (unlike our evenness checker from Subsection 3.1.2), is a syntactic well-formedness theorem. It will say that if the input AST is well-formed, then the output AST will also be well-formed. As seen in Figure 3.1.3, the definition of well-formed for PHOAS relates two expressions with different `var` arguments. Hence most PHOAS well-formedness theorems are proven by showing that a given compiler pass preserves relatedness between PHOASTs with different `var` arguments.

The fact that NbE values contain thunked rewriting creates a great deal of subtlety in relating `rawexpr`s. As the only correctness conditions on the rewriter are that it preserves denotational semantics of expressions and that it maps related expressions to related expressions, these are the only facts that hold about the $\texttt{NbE}_\texttt{t}$ values in `rValue`. Since native PHOAS expressions do not permit such thunked values, we can only relate $\texttt{NbE}_\texttt{t}$ values to the interpretations of such expressions. Even this is not straightforward, as we must use an extensional equivalence relation, saying that an $\texttt{NbE}_\texttt{t}$ value of arrow type is equivalent to an interpreted function only when equivalence between the $\texttt{NbE}_\texttt{t}$ value argument and the interpreted function argument implies equivalence of their outputs.

4.7.2 Which Equivalence Relation?

Generalizing the challenge from Subsection 4.7.1, it turns out that describing how to relate two (or more!) objects was one of the most challenging parts of the proof effort. All told, we needed approximately *two dozen* ways of relating various objects.

We begin with the equivalence relations hinted at in previous sections.

`wf_rawexpr` In Section 4.5, we introduced without definition the four-place `wf_rawexpr` relation. This relation, a beefed-up version of the PHOAS definition of `related` in Figure 3.1.3, takes in two `rawexpr`s, two PHOAS expressions (of the same type), and is parameterized over a list of pairs of allowed and related variables, much like the definition of `related`. It requires that both `rawexpr`s have the same amount of revealed structure (important only because we broke the abstraction barrier of revealed structure only mattering as an optimization); that the unrevealed structure, the "alternate" expression of the `rApp` and `rIdent` nodes, match exactly with the given expressions; and that the structure that is revealed matches as well with the given expressions. The only nontrivial case in this definition is what to say about when $\texttt{NbE}_\texttt{t}$ values match expressions. We say that an $\texttt{NbE}_\texttt{t}$ value is equivalent only to the result of calling NbE's `reify` function on that value. That this definition suffices is highly nonobvious; we refer the reader to our Coq proofs, performed without any axioms, as our justification of sufficiency. That each $\texttt{NbE}_\texttt{t}$ value must match at least the result of calling NbE's `reify` function on that value is a result of how we handle unrevealed forms when building up the arguments to an η-long identifier application as discussed briefly in Subsection 4.1.1 (What Does This Reduction Consist Of?). Namely, when forming applications of `rawexpr`s to $\texttt{NbE}_\texttt{t}$ values during η-expansion, we say that the "unrevealed" structure of an $\texttt{NbE}_\texttt{t}$ value `v` is `reify v`.

`interp_maybe_do_again` In Section 4.6, we discussed a small subset of the implications of supporting rewriting again in the output of a rewrite rule. The most easily describable intricacy and overhead caused by this feature shows up in the definition of what it means for a rewrite rule to preserve denotational semantics. At the user level, this is quite obvious: the left-hand side of the rewrite rule (prior to reification[6]) must equal the right-hand side. However, there are two subtleties to expressing the correctness condition to intermediate representations of the rewrite rule. We will discuss one of them here and the other in Section 4.8 (What's the Ground Truth: Patterns or Expressions?).

At some point in the rewriting process, the rewrite rule must be expressed in terms of a PHOAS expression whose `var` type is either the output `var` type—if this rule is

[6]Note that this reification is a tactic procedure reifying Gallina to PHOAS, *not* the `reify` function of normalization by evaluation discussed elsewhere in this chapter.

not subject to more rewriting—or else is the $\texttt{NbE}_\text{t}$ value type—if the rule is subject to more rewriting. Hence we must be able to relate an object of this type to the denotational interpretation that we are hoping to preserve. There are two subtleties here. The first is that we cannot simply "interpret" the $\texttt{NbE}_\text{t}$ values stored in `Var` nodes; we must use the extensional relation described above in Section 4.7 (Delayed Rewriting in Variable Nodes), saying that an $\texttt{NbE}_\text{t}$ value of arrow type is equivalent to an interpreted function only when equivalence between the $\texttt{NbE}_\text{t}$ value argument and the interpreted function argument implies equivalence of their outputs.

Second, we cannot simply interpret the expression which surrounds the `Var` node, and we must instead ensure that the "interpretation" of λs in the AST is extensional over all appropriately related $\texttt{NbE}_\text{t}$ values they might be passed. Note that it's not even obvious how to materialize the function they must be extensionally related to. When trying to prove that the application of $(\lambda\ \texttt{f x.}\ \texttt{v}_1\ \texttt{(f x)})$ to $\texttt{NbE}_\text{t}$ values $\texttt{v}_2$ and $\texttt{v}_3$ is appropriately related to some interpreted function g, how do we materialize the interpreted functions equivalent to $(\lambda\ \texttt{f x.}\ \texttt{v}_1\ \texttt{(f x)})$ and $\texttt{v}_2$ which when combined via application give g? The answer is that we cannot, at least if we are looking for the application to be *equal* to g. If we require that the application only be *related* to g (where "related" in this case means extensionally or pointwise equal), then we can materialize such functions by reading them off our inductive hypotheses. While we initially depended on the axiom of functional extensionality, after sinking dozens of hours into understanding the details of these relatedness functions, we were eventually able to extract the insight that two interpreted functions are extensionally equal if and only if there exists an expression to which both functions are related. See commits 4d7999e and e9b3505 in the `mit-plv/rewriter` repository on GitHub for more details on the exact changes required to implement this insight and remove the dependence on the axiom of functional extensionality.

Related Miscellanea While delving into the details of all two-dozen ways of relating objects is beyond the scope of this dissertation, we mention a couple of other nonobvious design questions that we found challenging to answer.

Recall from **??** that $\texttt{NbE}_\text{t}$ values are Gallina functions on arrow types; dropping the subtleties of the `UnderLets` monad, we had

$$\begin{aligned}\text{NbE}_t(t_1 \to t_2) &:= \text{NbE}_t(t_1) \to \text{NbE}_t(t_2)\\ \text{NbE}_t(b) &:= \texttt{expr}(b)\end{aligned}$$

The PHOAS relatedness condition of Figure 3.1.3 (PHOAS) is parameterized over a list of pairs of permitted related variables.
Design Question: What is the relation between the permitted related variables lists of the terms of types $\text{NbE}_t(t_1)$, $\text{NbE}_t(t_2)$, and $\text{NbE}_t(t_1 \to t_2)$?
Spoiler: The list for $\text{NbE}_t(t_1)$ is unconstrained and is prepended to the list for

$\mathrm{NbE}_t(t_1 \to t_2)$ (which is given) to get the list for $\mathrm{NbE}_t(t_2)$. That is, we write

$$\begin{aligned}\text{related\_NbE}_{t_1 \to t_2}(\Gamma, f_1, f_2) &:= \forall\ \Gamma'\ v_1\ v_2, \text{related\_NbE}_{t_1}(\Gamma', v_1, v_2) \\ &\quad \to \text{related\_NbE}_{t_2}(\Gamma' +\!\!+ \Gamma, f_1(v_1), f_2(v_2)) \\ \text{related\_NbE}_b(\Gamma, e_1, e_2) &:= \text{related}(\Gamma, e_1, e_2)\end{aligned}$$

Some correctness lemmas do not need full-blown relatedness conditions. For example, in some places, we do not need that a `rawexpr` is fully consistent with its alternate expression structure, only that the types match and that the top-level structure of each alternate PHOAS expression matches the node of the `rawexpr`.
Design Question: Is it better to minimize the number of relations and fold these "self-matching" or "goodness" properties into the definitions of relatedness, which are then used everywhere; or is it better to have separate definitions for goodness and relatedness and have correctness conditions which more tightly pin down the behavior of the corresponding functions?
(Non-Spoiler: We don't have an answer to this one.)

4.8 What's the Ground Truth: Patterns or Expressions?

We mentioned in Subsection 4.7.2 (Which Equivalence Relation?) that there were two subtleties to expressing the interpretation-correctness condition for intermediate representations of rewrite rules, and we proceeded to discuss only one of them. We discuss the other one here.

We must answer the question, in proving our rewriter correct: What denotational semantics do we use for a rewrite rule?

In our current framework, we talk about rewrite rules in terms of patterns, which are special ASTs which contain extra pattern variables in both the types and the terms, and in terms of a replacement function, which takes in unification data and returns either failure or else a PHOAST with the data plugged in. While this design is sort-of a historical accident of originally intending to write rewrite rules by hand, there is also a genuine question of how to relate patterns to replacement functions. While we could, in theory, in a better-designed rewriter, indirect through the expressions that each of these came from, the functions turning expressions into patterns and replacement rules are likely to be quite complicated, especially with the support for rewriting again described in Section 4.6 (Rewriting Again in the Output of a Rewrite Rule).

The way we currently relate these is that we write an interpretation function for patterns, parameterized over unification data, and relate this to the interpretation of the replacement function applied to unification data, suitably restricted to just the type variables of the pattern in question to make various dependent types line up. Note that this restriction of the unification data would likely be unnecessary if we stripped out all of the dependent types that we don't actually need; c.f. Subsection 4.1.3 (Type Codes). This interpretation function is itself also severely complicated by the use of dependent types in talking about unification data.

4.9 What's the Takeaway?

This chapter has been a brief survey of the engineering challenges we encountered in designing and implementing a framework for building verified partial evaluators with rewriting. We hope that this deep dive into the details of our framework has fleshed out some of the design principles and challenges we've discussed in previous sections.

If the reader wishes to take only one thing from this chapter, we invite it to be a sense and understanding of just how important good abstraction barriers and API design are to engineering at scale in verified and dependently typed settings, which we will come back to in Chapter 6.

Chapter 5

Reification by Parametricity

Fast Setup for Proof by Reflection, in Two Lines of $\mathcal{L}_{tac}$

5.1 Introduction

We introduced reification in Section 3.2 as the starting point for proof by reflection. Reification consists of translating a "native" term of the logic into an explicit abstract syntax tree, which we may then feed to verified procedures or any other functional programs in the logic. As mentioned in **??**, the method of reification used in our framework for reflective partial evaluation and rewriting presented in **??** was not especially optimized and can be a bottleneck for large terms, especially those with many binders. Popular methods turn out to be surprisingly slow, often to the point where, counterintuitively, the majority of proof-execution time is spent in reification – unless the proof engineer invests in writing a plugin directly in the proof assistant's metalanguage (e.g., OCaml for Coq).

In this chapter, we present a new strategy discovered by Andres Erbsen and me during my doctoral work, originally presented and published as [GEC18], showing that reification can be both simpler and faster than with standard methods. Perhaps surprisingly, we demonstrate how to reify terms almost entirely through reduction in the logic, with a small amount of tactic code for setup and no ML programming. We have already summarized our survey into prior approaches to reification Section 3.2, providing high-quality implementations and documentation for them, serving a tutorial function independent of our new contributions. We will begin in Section 5.2 with an explanation of our alternative technique. We benchmark our approach against 18 competitors in Section 5.3.

5.2 Reification by Parametricity

We propose factoring reification into two passes, both of which essentially have robust, built-in implementations in Coq: *abstraction* or *generalization*, and *substitution* or *specialization*.

The key insight to this factoring is that the shape of a reified term is essentially the same as the shape of the term that we start with. We can make precise the way these shapes are the same by abstracting over the parts that are different, obtaining a function that can be specialized to give either the original term or the reified term.

That is, we have the commutative triangle in Figure 5-1.

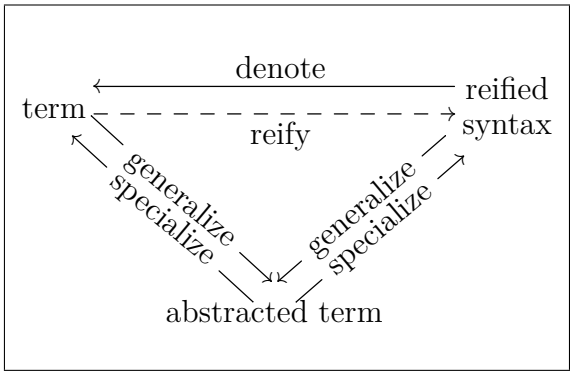


Figure 5-1: Abstraction and Reification

5.2.1 Case-By-Case Walkthrough

Function Applications and Constants.

Consider the example of reifying 2×2. In this case, the *term* is 2×2 or (mul (S (S O)) (S (S O))).

To reify, we first *generalize* or *abstract* the term 2×2 over the successor function S, the zero constructor O, the multiplication function mul, and the type $\mathbb{N}$ of natural numbers. We get a function taking one type argument and three value arguments:

$$\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)\ (\mathrm{O} : N)\ (\mathrm{S} : N \to N).\ \textsc{Mul}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{O}))\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{O}))$$

We can now specialize this term in one of two ways: we may substitute $\mathbb{N}$, mul, O, and S, to get back the term we started with; or we may substitute `expr`, `NatMul`, `NatO`, and `NatS` to get the reified syntax tree

```
NatMul (NatS (NatS NatO)) (NatS (NatS NatO))
```

This simple two-step process is the core of our algorithm for reification: abstract over all identifiers (and key parts of their types) and specialize to syntax-tree constructors for these identifiers.

Wrapped Primitives: `let` Binders, Eliminators, Quantifiers.

The above procedure can be applied to a term that contains `let` binders to get a PHOAS tree that represents the original term, but doing so would not capture sharing. The result would contain native `let` bindings of subexpressions, not PHOAS let expressions. Call-by-value evaluation of any procedure applied to the reification result would first substitute the let-bound subexpressions – leading to potentially exponential blowup and, in practice, memory exhaustion.

The abstraction mechanisms in all proof assistants (that we know about) only allow abstracting over terms, not language primitives. However, primitives can often be wrapped in explicit definitions, which we *can* abstract over. For example, we already used a wrapper for `let` binders, and terms that use it can be reified by abstracting over that definition. If we start with the expression

$$\texttt{dlet}\ a := 1\ \texttt{in}\ a \times a$$

and abstract over (`@Let_In` $\mathbb{N}$ $\mathbb{N}$), S, O, mul, and $\mathbb{N}$, we get a function of one type argument and four value arguments:

$$\Lambda N.\, \lambda\ (\textsc{Mul} : N \to N \to N).\ \lambda(\textsc{O} : N).\ \lambda(\textsc{S} : N \to N).\\ \lambda(\textsc{LetIn} : N \to (N \to N) \to N).\ \textsc{LetIn}\ (\textsc{S}\ \textsc{O})\ (\lambda a.\ \textsc{Mul}\ a\ a)$$

We may once again specialize this term to obtain either our original term or the reified syntax. Note that to obtain reified PHOAS, we must include a `Var` node in the `LetIn` expression; we substitute $(\lambda x\ f.\ \texttt{LetIn}\ x\ (\lambda v.\ f\ (\texttt{Var}\ v)))$ for LetIn to obtain the PHOAS tree

```
LetIn (NatS NatO) (λv. NatMul (Var v) (Var v))
```

Wrapping a metalanguage primitive in a definition in the code to be reified is in general sufficient for reification by parametricity. Pattern matching and recursion cannot be abstracted over directly, but if the same code is expressed using eliminators, these can be handled like other functions. Similarly, even though $\forall/\Pi$ cannot be abstracted over, proof automation that itself introduces universal quantifiers before reification can easily wrap them in a marker definition (`_forall T P := forall (x:T), P x`) that can be. Existential quantifiers are not primitive in Coq and can be reified directly.

Lambdas.

While it would be sufficient to require that, in code to be reified, we write all lambdas with a named wrapper function, that would significantly clutter the code. We can do better by making use of the fact that a PHOAS object-language lambda (`Abs` node) consists of a metalanguage lambda that binds a value of type `var`, which can

be used in expressions through constructor `Var` : `var` → `expr`. Naïve reification by parametricity would turn a lambda of type $N \to N$ into a lambda of type `expr` → `expr`. A reification procedure that explicitly recurses over the metalanguage syntax could just precompose this recursive-call result with `Var` to get the desired object-language encoding of the lambda, but handling lambdas specially does not fit in the framework of abstraction and specialization.

First, let us handle the common case of lambdas that appear as arguments to higher-order functions. One easy approach: while the parametricity-based framework does not allow for special-casing lambdas, it is up to us to choose how to handle functions that we expect will take lambdas as arguments. We may replace each higher-order function with a metalanguage lambda that wraps the higher-order arguments in object-language lambdas, inserting `Var` nodes as appropriate. Code calling the function `sum_upto` $n\ f := f(0)+f(1)+\cdots+f(n)$ can be reified by abstracting over relevant definitions and substituting ($\lambda n\ f.$ `SumUpTo` n (`Abs` ($\lambda v.\ f$ (`Var` v)))) for `sum_upto`. Note that the expression plugged in for `sum_upto` differs from the one plugged in for `Let_In` only in the use of a deeply embedded abstraction node. If we wanted to reify `LetIn` as just another higher-order function (as opposed to a distinguished wrapper for a primitive), the code would look identical to that for `sum_upto`.

It would be convenient if abstracting and substituting for functions that take higher-order arguments were enough to reify lambdas, but here is a counterexample. Starting with

$$\lambda\ x\ y.\ x \times ((\lambda\ z.\ z \times z)\ y),$$

abstraction gives

$$\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N).\ \lambda\ (x\ y : N).\ \texttt{Mul}\ x\ ((\lambda\ (z : N).\ \texttt{Mul}\ z\ z)\ y),$$

and specialization and reduction give

$$\lambda\ (x\ y : \texttt{expr}).\ \texttt{NatMul}\ x\ (\texttt{NatMul}\ y\ y).$$

The result is not even a PHOAS expression. We claim a desirable reified form is

$$\texttt{Abs}(\lambda\ x.\ \texttt{Abs}(\lambda\ y.\ \texttt{NatMul}\ (\texttt{Var}\ x)\ (\texttt{NatMul}\ (\texttt{Var}\ y)\ (\texttt{Var}\ y))))$$

Admittedly, even our improved form is not quite precise: $\lambda\ z.\, z \times z$ has been lost. However, as almost all standard Coq tactics silently reduce applications of lambdas, working under the assumption that functions not wrapped in definitions will be arbitrarily evaluated during scripting is already the norm. Accepting that limitation, it remains to consider possible occurrences of metalanguage lambdas in normal forms of outputs of reification as described so far. As lambdas in `expr` nodes that take metalanguage functions as arguments (`LetIn`, `Abs`) are handled by the rules for these nodes, the remaining lambdas must be exactly at the head of the expression. Manipulating these is outside of the power of abstraction and specialization; we recommend

postprocessing using a simple recursive tactic script.

5.2.2 Commuting Abstraction and Reduction

Sometimes, the term we want to reify is the result of reducing another term. For example, we might have a function that reduces to a term with a variable number of `let` binders.[1] We might have an inductive type that counts the number of `let … in …` nodes we want in our output.

```
Inductive count := none | one_more (how_many : count).
```

It is important that this type be syntactically distinct from $\mathbb{N}$ for reasons we will see shortly.

We can then define a recursive function that constructs some number of nested `let` binders:

```
Fixpoint big (x:nat) (n:count)
  : nat
  := match n with
     | none => x
     | one_more n'
      => dlet x' := x * x in
         big x' n'
     end.
```

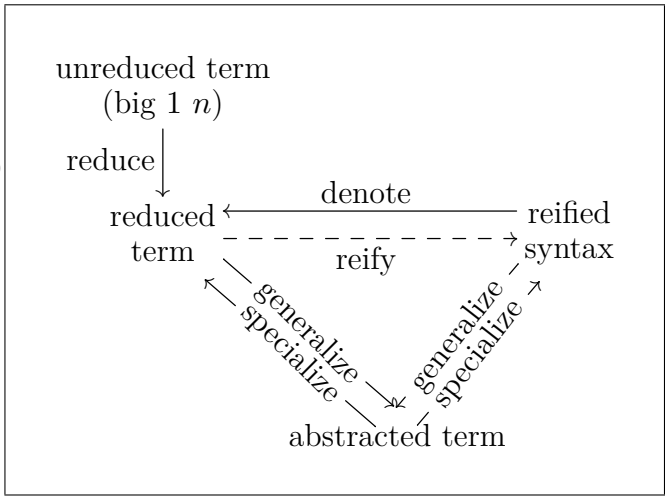
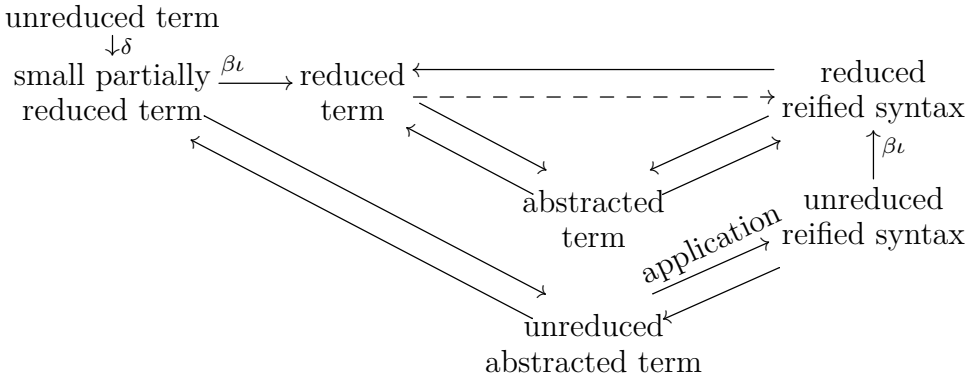


Figure 5-2: Abstraction, Reification, Reduction

Our commutative diagram in Figure 5-1 now has an additional node, becoming Figure 5-2. Since generalization and specialization are proportional in speed to the size of the term begin handled, we can gain a significant performance boost by performing generalization before reduction. To explain why, we split apart the commutative diagram a bit more; in reduction, there is a δ or unfolding step, followed by a $\beta\iota$ step that reduces applications of λs and evaluates recursive calls. In specialization, there is an application step, where the λ is applied to arguments, and a β-reduction step, where the arguments are substituted. To obtain reified syntax, we may perform generalization after δ-reduction (before $\beta\iota$-reduction), and we are not required to perform the final β-reduction step of specialization to get a well-typed term. It is important that unfolding `big` results in exposing the body for generalization, which we accomplish in Coq by exposing the anonymous recursive function; in other languages, the result may be a primitive eliminator applied to the body of the

[1] More realistically, we might have a function that represents big numbers using multiple words of a user-specified width. In this case, we may want to specialize the procedure to a couple of different bitwidths, then reify the resulting partially reduced terms.

fixpoint. Either way, our commutative diagram thus becomes



Let us step through this alternative path of reduction using the example of the unreduced term `big 1 100`, where we take 100 to mean the term represented by $\underbrace{(\texttt{one\_more} \cdots (\texttt{one\_more}}_{100}\ \texttt{none}\underbrace{)\cdots)}_{100}$.

Our first step is to unfold `big`, rendered as the arrow labeled δ in the diagram. In Coq, the result is an anonymous fixpoint; here we will write it using the recursor `count_rec` of type $\forall T.\ T \to (\texttt{count} \to T \to T) \to \texttt{count} \to T$. Performing δ-reduction, that is, unfolding `big`, gives us the small partially reduced term

$$\begin{array}{l}(\lambda(x : \mathbb{N}).\ \lambda(n : \texttt{count}).\\ \texttt{count\_rec}\ (\mathbb{N} \to \mathbb{N})\ (\lambda x.\ x)\ (\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \texttt{dlet}\ x' := x \times x\ \texttt{in}\ \texttt{big}_{n'}\ x'))\ 1\ 100\end{array}$$

We call this term small, because performing $\beta\iota$ reduction gives us a much larger reduced term:

$$\texttt{dlet}\ x_1 := 1 \times 1\ \texttt{in}\ \cdots\ \texttt{dlet}\ x_{100} := x_{99} \times x_{99}\ \texttt{in}\ x_{100}$$

Abstracting the small partially reduced term over (`@Let_In` $\mathbb{N}$ $\mathbb{N}$), S, O, mul, and $\mathbb{N}$ gives us the abstracted unreduced term

$$\begin{array}{l}\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)(\textsc{O} : N)(\textsc{S} : N \to N)(\textsc{LetIn} : N \to (N \to N) \to N).\\ \quad (\lambda(x : N).\ \lambda(n : \texttt{count}).\ \texttt{count\_rec}\ (N \to N)\ (\lambda x.\ x)\\ \qquad (\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \textsc{LetIn}\ (\textsc{Mul}\ x\ x)\ (\lambda x'.\ \texttt{big}_{n'}\ x')))\\ \quad (\textsc{S}\ \textsc{O})\ \ 100\end{array}$$

Note that it is essential here that `count` is not syntactically the same as $\mathbb{N}$; if they were the same, the abstraction would be ill-typed, as we have not abstracted over `count_rec`. More generally, it is essential that there is a clear separation between

types that we reify and types that we do not, and we must reify *all* operations on the types that we reify.

We can now apply this term to `expr`, `NatMul`, `NatS`, `NatO`, and, finally, to the term $(\lambda v\ f.\ \texttt{LetIn}\ v\ (\lambda x.\ f\ (\texttt{Var}\ x)))$. We get an unreduced reified syntax tree of type `expr`. If we now perform $\beta\iota$ reduction, we get our fully reduced reified term.

We take a moment to emphasize that this technique is not possible with any other method of reification. We could just as well have not specialized the function to the `count` of `100`, yielding a function of type `count` $\rightarrow$ `expr`, despite the fact that our reflective language knows nothing about `count`!

This technique is especially useful for terms that will not reduce without concrete parameters but which should be reified for many different parameters. Running reduction once is slightly faster than running OCaml reification once, and it is more than twice as fast as running reduction followed by OCaml reification. For sufficiently large terms and sufficiently many parameter values, this performance beats even OCaml reification.[2]

5.2.3 Implementation in $\mathcal{L}_{tac}$

`ExampleMoreParametricity.v` in the associated codebase, available at `https://github.com/mit-plv/reification-by-parametricity`, mirrors the development of reification by parametricity in Subsection 5.2.1.

Unfortunately, Coq does not have a tactic that performs abstraction.[3] However, the `pattern` tactic suffices; it performs abstraction followed by application, making it a sort of one-sided inverse to β-reduction. By chaining `pattern` with an $\mathcal{L}_{tac}$-`match` statement to peel off the application, we can get the abstracted function.

```
Ltac Reify x :=
match (eval pattern nat, Nat.mul, S, O, (@Let_In nat nat) in x) with
| ?rx _ _ _ _ _ =>
 constr:( fun var => rx (@expr var) NatMul NatS NatO
                        (fun v f => LetIn v (fun x => f (Var x))) )
end.
```

Note that if `@expr var` lives in `Type` rather than `Set`, we must `pattern` over `(nat : Type)` rather than `nat`. In older versions of Coq, an additional step involving retyping the term with the $\mathcal{L}_{tac}$ primitive `type of` is needed; we refer the reader to

[2]We discovered this method in the process of needing to reify implementations of cryptographic primitives [Erb+19] for a couple hundred different choices of numeric parameters (e.g., prime modulus of arithmetic). A couple hundred is enough to beat the overhead.

[3]The `generalize` tactic returns $\forall$ rather than λ, and it only works on types.

`Parametricity.v` in the code supplement.

The error messages returned by the `pattern` tactic can be rather opaque at times; in `ExampleParametricityErrorMessages.v`, we provide a procedure for decoding the error messages.

Open Terms.

At some level it is natural to ask about generalizing our method to reify open terms (i.e., with free variables), but we think such phrasing is a red herring. Any lemma statement about a procedure that acts on a representation of open terms would need to talk about how these terms would be closed. For example, solvers for algebraic goals without quantifiers treat free variables as implicitly universally quantified. The encodings are invariably ad-hoc: the free variables might be assigned unique numbers during reification, and the lemma statement would be quantified over a sufficiently long list that these numbers will be used to index into. Instead, we recommend directly reifying the natural encoding of the goal as interpreted by the solver, e.g. by adding new explicit quantifiers. Here is a hypothetical goal and a tactic script for this strategy:

```
       (a b : nat) (H : 0 < b) |- ∃ q r, a = q × b + r ∧ r < b

repeat match goal with
       | n : nat |- ?P =>
         match eval pattern n in P with
         | ?P' _ => revert n; change (_forall nat P')
         end
       | H : ?A  |- ?B => revert H; change (impl A B)
       | |- ?G => (* ∀ a b, 0 < b -> ∃ q r, a = q × b + r ∧ r < b *)
         let rG := Reify G in
         refine (nonlinear_integer_solver_sound rG _ _);
         [ prove_wf | vm_compute; reflexivity ]
       end.
```

Briefly, this script replaced the context variables `a` and `b` with universal quantifiers in the conclusion, and it replaced the premise `H` with an implication in the conclusion. The syntax-tree datatype used in this example can be found in the Coq source file `ExampleMoreParametricity.v`.

5.2.4 Advantages and Disadvantages

This method is faster than all but Ltac2 and OCaml reification, and commuting reduction and abstraction makes this method faster even than the low-level Ltac2

reification in many cases. Additionally, this method is much more concise than nearly every other method we have examined, and it is very simple to implement.

We will emphasize here that this strategy shines when the initial term is small, the partially computed terms are big (and there are many of them), and the operations to evaluate are mostly well-separated by types (e.g., evaluate all of the `count` operations and none of the `nat` ones).

This strategy is not directly applicable for reification of `match` (rather than eliminators) or `let … in …` (rather than a definition that unfolds to `let … in …`), `forall` (rather than a definition that unfolds to `forall`), or when reification should not be modulo $\beta\iota\zeta$-reduction.

5.3 Performance Comparison

We have done a performance comparison of the various methods of reification to the PHOAS language `@expr var` from Section 3.1.3 in Coq 8.12.2. A typical reification routine will obtain the term to be reified from the goal, reify it, run `transitivity (denote reified_term)` (possibly after normalizing the reified term), and solve the side condition with something like `lazy [denote]; reflexivity`. Our testing on a few samples indicated that using `change` rather than `transitivity; lazy [denote]; reflexivity` can be around 3X slower; note that we do not test the time of `Defined`.

There are two interesting metrics to consider: (1) how long does it take to reify the term? and (2) how long does it take to get a normalized reified term, i.e., how long does it take both to reify the term and normalize the reified term? We have chosen to consider (1), because it provides the most fine-grained analysis of the actual reification method.

5.3.1 Without Binders

We look at terms of the form `1 * 1 * 1 * …` where multiplication is associated to create a balanced binary tree. We say that the *size of the term* is the number of 1s. We refer the reader to the attached code for the exact test cases and the code of each reification method being tested.

We found that the performance of all methods is linear in term size.

Sorted from slowest to fastest, most of the labels in Figure 5-3 should be self-explanatory and are found in similarly named `.v` files in the associated code; we call out a few potentially confusing ones:

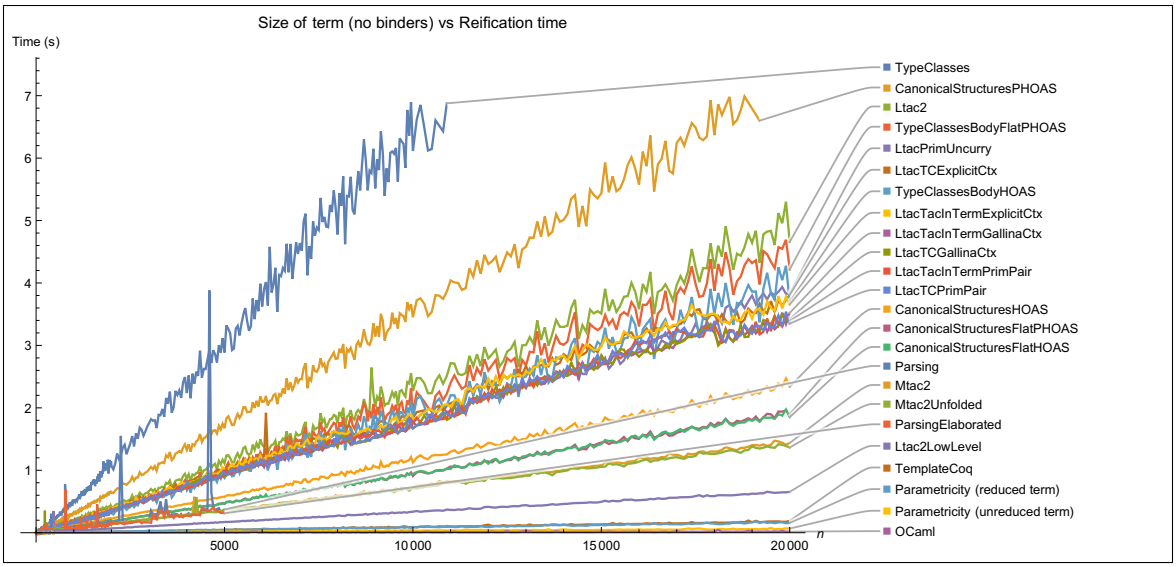


Figure 5-3: Performance of Reification without Binders

- The "Parsing" benchmark is "reification by copy-paste": a script generates a `.v` file with notation for an already-reified term; we benchmark the amount of time it takes to parse and typecheck that term. The "ParsingElaborated" benchmark is similar, but instead of giving notation for an already-reified term, we give the complete syntax tree, including arguments normally left implicit. Note that these benchmarks cut off at around 5000 rather than at around 20 000, because on large terms, Coq crashes with a stack overflow in parsing.

- We have four variants starting with "CanonicalStructures" here. The Flat variants reify to `@expr nat` rather than to `forall var, @expr var` and benefit from fewer function binders and application nodes. The HOAS variants do not include a case for `let … in …` nodes, while the PHOAS variants do. Unlike most other reification methods, there is a significant cost associated with handling more sorts of identifiers in canonical structures.

We note that on this benchmark our method is slightly faster than template-coq, which reifies to de Bruijn indices, and slightly slower than the quote plugin in the standard library[4] and the OCaml plugin we wrote by hand.

5.3.2 With Binders

We look at terms of the form `dlet` a_1 `:= 1 * 1 in dlet` a_2 `:=` a_1 `*` a_1 `in … dlet` a_n `:=` a_{n-1} `*` a_{n-1} `in` a_n, where n is the size of the term. The first graph shown here includes all of the reification variants at linear scale, while the next step zooms in on the highest-performance variants at log-log scale.

[4]This plugin no longer appears in this graph because it was removed in Coq 8.10 [Dén18], though

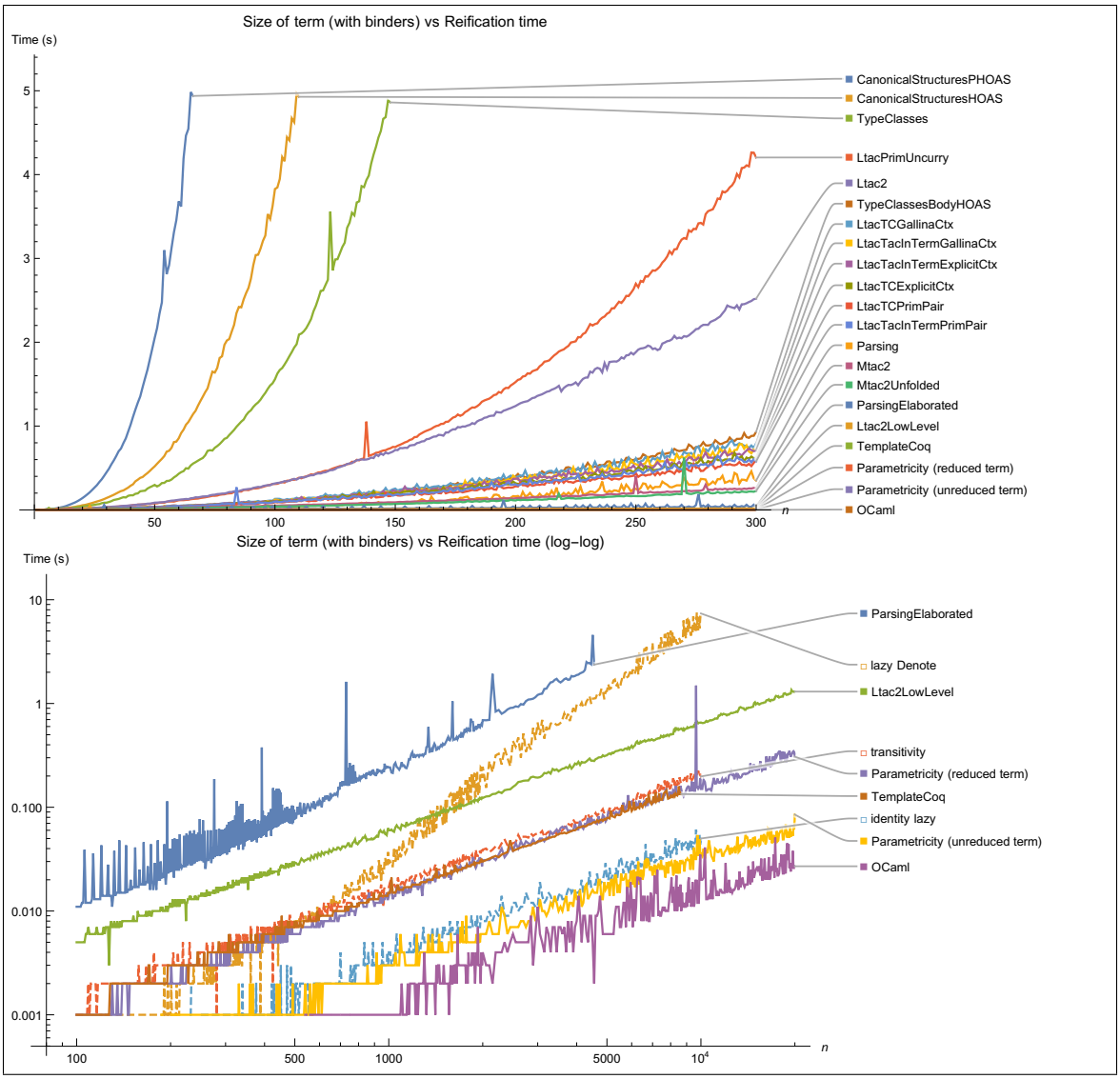


Figure 5-4: Performance of Reification with Binders

In addition to reification benchmarks, the graph in Figure 5-4 includes as a reference (1) the time it takes to run `lazy` reduction on a reified term already in normal form ("identity lazy") and (2) the time it takes to check that the reified term matches the original native term ("lazy Denote"). The former is just barely faster than OCaml reification; the latter often takes longer than reification itself. The line for the template-coq plugin cuts off at around 10 000 rather than around 20 000 because at that point template-coq starts crashing with stack overflows.

it appears in the graph in Gross, Erbsen, and Chlipala [GEC18].

5.4 Future Work, Concluding Remarks

We identify one remaining open question with this method that has the potential of removing the next largest bottleneck in reification: using reduction to show that the reified term is correct.

Recall our reification procedure and the associated diagram, from Figure 5.2.2. We perform δ on an unreduced term to obtain a small, partially reduced term; we then perform abstraction to get an abstracted, unreduced term, followed by application to get unreduced reified syntax. These steps are all fast. Finally, we perform $\beta\iota$-reduction to get reduced, reified syntax and perform $\beta\iota\delta$ reduction to get back a reduced form of our original term. These steps are slow, but we must do them if we are to have verified reflective automation.

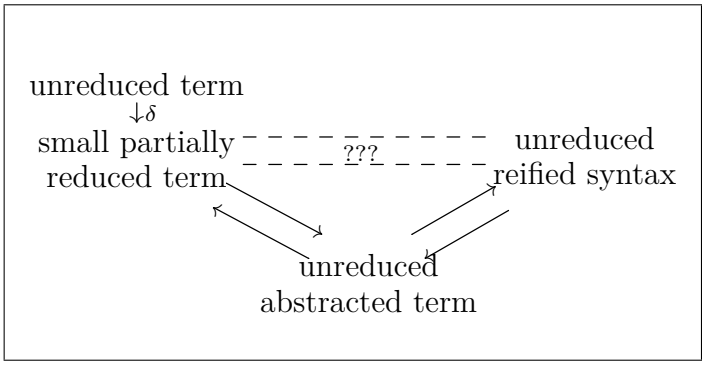


Figure 5-5: Completing the commutative triangle

It would be nice if we could prove this equality without ever reducing our term. That is, it would be nice if we could have the diagram in Figure 5-5.

The question, then, is how to connect the small partially reduced term with `denote` applied to the unreduced reified syntax. That is, letting F denote the unreduced abstracted term, how can we prove, without reducing F, that

$$F\ \mathbb{N}\ \text{Mul O S (@Let\_In }\mathbb{N}\ \ \mathbb{N}) = \texttt{denote}\ (F\ \texttt{expr NatMul NatO NatS LetIn})$$

We hypothesize that a form of internalized parametricity would suffice for proving this lemma. In particular, we could specialize the type argument of F with $\mathbb{N} \times \texttt{expr}$. Then we would need a proof that for any function F of type

$$\forall (T : \texttt{Type}), (T \to T \to T) \to T \to (T \to T) \to (T \to (T \to T) \to T) \to T$$

and any types A and B, and any terms $f_A : A \to A \to A$, $f_B : B \to B \to B$, $a : A$, $b : B$, $g_A : A \to A$, $g_B : B \to B$, $h_A : A \to (A \to A) \to A$, and $h_B : B \to (B \to B) \to B$, using $f \times g$ to denote lifting a pair of functions to a function over pairs:

$$\texttt{fst}\ (F\ (A \times B)\ (f_A \times f_B)\ (a, b)\ (g_A \times g_B)\ (h_A \times h_B)) = F\ A\ f_A\ a\ g_A\ h_A \wedge$$
$$\texttt{snd}\ (F\ (A \times B)\ (f_A \times f_B)\ (a, b)\ (g_A \times g_B)\ (h_A \times h_B)) = F\ B\ f_B\ b\ g_B\ h_B$$

This theorem is a sort of parametricity theorem.

Despite this remaining open question, we hope that our performance results make a

strong case for our method of reification; it is fast, concise, and robust.

Part III

API Design

Chapter 6

Abstraction

6.1 Introduction

In Chapters 1 and 2, we discussed two different fundamental sources of performance bottlenecks in proof assistants: the power that comes from having dependent types, in Subsection 1.3.1; and the de Bruijn criterion of having a small trusted kernel, in Subsection 1.3.2. In this chapter, we will dive further into the performance issues arising from the first of these design decisions, expanding on Subsection 2.6.4 (The Number of Nested Abstraction Barriers) and proposing some general guidelines for handling these performance bottlenecks.

This chapter is primarily geared at the users of proof assistants and especially at proof-assistant library developers.

We saw in The Number of Nested Abstraction Barriers three different ways that design choices for abstraction barriers can impact performance: We saw in Type-Size Blowup: Abstraction Barrier Mismatch that API mismatch results in type-size blowup; we saw a particularly striking example of this in Section 4.5 (Monads: Missing Abstraction Barriers at the Type Level) where an API mismatch resulted in vastly more complicated theorem statements. We saw in Conversion Troubles that imperfectly opaque abstraction barriers result in slowdown due to needless calls to the conversion checker. We saw in Type Size Blowup: Packed vs. Unpacked Records how the choice of whether to use packed or unpacked records impacts performance.

In this chapter, we will focus primarily on the first of these three ways that design choices for abstraction barriers can impact performance; while it might seem like a simple question of good design, it turns out that good API design in dependently typed programming languages is significantly harder than in nondependently typed programming languages. We will additionally weave in ways that abstraction

barriers have helped us develop our tools and libraries, though this use of abstraction barriers (sometimes called data abstraction) is already well-known in software engineering [SSA96]. Mitigating the second source of performance bottlenecks, imperfectly opaque abstraction barriers, on the other hand, is actually just a question of meticulous tracking of how abstraction barriers are defined and used and designing them so that all unfolding is explicit. However, we will present an exception to the rule of opaque abstraction barriers in Section 6.5 in which deliberate breaking of all abstraction barriers in a careful way can result in performance gains of up to a factor of two: Section 6.5 presents one of our favorite design patterns for categorical constructions—a way of coaxing Coq's definitional equality into implementing *proof by duality*, one of the most widely known ideas in category theory. Finally, the question of whether to use packed or unpacked records is actually a genuine trade-off in both design space and performance, as far as I can tell; the nonperformance design considerations have been discussed before in Garillot et al. [Gar+09b], while the performance implications are relatively straightforward. As far as I'm aware, there's not really a good way to get the best of all worlds.

Much of this chapter will draw on examples and experience from a category-theory library we implemented in Coq [GCS14], which we introduce in Section 6.3.

The only prior work we've been able to find where abstraction barriers are mentioned for proof development performance is Gu et al. [Gu+15]. Though this paper suggests that good abstraction barriers resulted in simpler invariants which alleviated proof burden on the developer, we suspect that their use of abstraction barriers also dodged the proof-generation and proof-checking performance bottlenecks of large types, large terms, large goals, and excessive unfolding that plague developments with leaky abstraction barriers.

6.2 When and How To Use Dependent Types Painlessly

Though abstraction barriers have been studied in the context of nondependently typed languages [SSA96], we're not aware of any systematic investigation of abstraction in dependently typed languages. Hence we provide in this section some rules of thumb that we've learned for developing good abstraction in dependently typed languages. Following these guidelines, in our experience, tends both to alleviate proof burden by decoupling and simplifying theorem statements and also to improve the performance of individual proofs, sometimes by an order of magnitude or more, by avoiding the superlinear scaling laid out in Section 2.6 (The Four Axes of the Landscape).

The extremes of using dependent types are relatively easy:

- Total separation between proofs and programs, so that programs are nondependently typed, works relatively well.

- Preexisting mathematics, where objects are fully bundled with proofs and never need to be separated from them, also works relatively well.

We present a rule of thumb for being in the middle: it incurs enormous overhead—both proof-authoring time and often proof-checking time—to recombine proofs and programs after separating them; if this separation and recombination is being done to define an opaque transformation that acts on proof-carrying code, that is okay, but if the abstraction barrier cannot be constructed, enormous overhead results.

For example, if we have length-indexed lists and want to index into them with elements of a finite type, things are fine until we need to divorce the index from its proof of finiteness. If, for example, we want to index into the concatenation of two lists with an index into the first of the lists, then we will likely run into trouble, because we are trying to consider the index separately from its proof of finitude, but we have to recombine them to do the indexing.

We saw in footnote 3 in Subsection 4.2.1 (Pattern-Matching Evaluation on Type-Indexed Terms) how dependent types cause coupling between otherwise-unrelated design decisions. This is due to the fact that every single operation needs to declare how it interacts with all of the various indices and must effectively include a proof for each of these interactions. In the example of Subsection 4.2.1, we considered indexing the list of terms over a list of types and indexing both this list of types and the decision tree over a natural-number length. In this case, the choice of whether we operate in the middle of the list or at the front of the list is severely complicated by the length index. If we need to insert an unknown number of elements into the middle of a length-indexed list, the length of the resulting list is not judgmentally the sum of the lengths, because addition is not judgmentally commutative.

6.3 A Brief Introduction to Our Category-Theory Library

Category theory [Mac] is a popular all-encompassing mathematical formalism that casts familiar mathematical ideas from many domains in terms of a few unifying concepts. A *category* can be described as a directed graph plus algebraic laws stating equivalences between paths through the graph. Because of this spartan philosophical grounding, category theory is sometimes referred to in good humor as "formal abstract nonsense." Certainly the popular perception of category theory is quite far from pragmatic issues of implementation. Our implementation of category theory ran squarely into issues of design and efficient implementation of type theories, proof assistants, and developments within them.

One might presume that it is a routine exercise to transliterate categorical concepts

from the whiteboard to Coq. Most category theorists would probably be surprised to learn that standard constructions "run too slowly", but in our experience that is exactly the result of experimenting with naïve first Coq implementations of categorical constructs. It is important to tune the library design to minimize the cost of manipulating terms and proving interesting theorems.

Category theory, said to be "notoriously hard to formalize" [Har96b], provides a good stress test of any proof assistant, highlighting problems in usability and efficiency.

Formalizing the connection between universal morphisms and adjunctions provides a typical example of our experience with performance. A *universal morphism* is a construct in category theory generalizing extrema from calculus. An *adjunction* is a weakened notion of equivalence. In the process of rewriting our library to be compatible with homotopy type theory, we discovered that cleaning up this construction conceptually resulted in a significant slow-down, because our first attempted rewrite resulted in a leaky abstraction barrier and, most importantly, large goals (Subsection 7.2.3). Plugging the holes there reduced goal sizes by two orders of magnitude[1], which led to a factor of ten speedup in that file (from 39s to 3s) but incurred a factor of three slow-down in the file where we defined the abstraction barriers (from 7s to 21s).[2] Working around slow projections of Σ types (Subsection 6.4.2) and being more careful about code reuse each gave us back half of that lost time.[3]

Although preexisting formalizations of category theory in proof assistants abound [Meg; AKS13; OKe04; Pee+; Saï; Sim; SW10; KKR06; Gro14; Ahrb; Ahra; CM98; Cha; Ish; Pou; Soza; Niq10; Pot; Ahr10; Web02; Cap; HS00; AP90; Kat10; KSW; AKS; Moh95; Spi11; CW01; Acz93; Wil05; Miq01; Dyc85; Wil12; Har96b; Age95; Nuo13], we chose to implement our library [HoT20] from scratch. Beginning from scratch allowed me to familiarize myself with both category theory and Coq, without simultaneously having to familiarize myself with a large preexisting code base.

6.4 A Sampling of Abstraction Barriers

We acknowledge that the concept of performance issues arising from choices of abstraction barriers may seem a bit counterintuitive. After all, abstraction barriers generally live in the mind of the developer, in some sense, and it seems a bit insane to say that performance of the code depends on the mental state of the programmer.

Therefore, we will describe a sampling of abstraction barriers and the design choices that went into them, drawn from real examples, as well as the performance issues

[1]The word count of the larger of the two relevant goals went from 7,312 to 191.

[2]See commit eb00990 in HoTT/HoTT on GitHub for more details.

[3]See commits c1e7ae3, 93a1258, bab2b34, and 3b0932f in HoTT/HoTT on GitHub for more details.

that arose from these choices. We will also discuss ways that various design choices increased or reduced the effort required of us to define objects and prove theorems.

6.4.1 Abstraction in Limits and Colimits

In many projects, choosing the right abstraction barriers is essential to reducing mistakes, improving maintainability and readability of code, and cutting down on time wasted by programmers trying to hold too many things in their heads at once. This project was no exception; we developed an allergic reaction to constructions with more than four or so arguments, after making one too many mistakes in defining limits and colimits. Limits are a generalization, to arbitrary categories, of subsets of Cartesian products. Colimits are a generalization, to arbitrary categories, of disjoint unions modulo equivalence relations.

Our original flattened definition of limits involved a single definition with 14 nested binders for types and algebraic properties. After a particularly frustrating experience hunting down a mistake in one of these components, we decided to factor the definition into a larger number of simpler definitions, including familiar categorical constructs like terminal objects and comma categories. This refactoring paid off even further when some months later we discovered the universal morphism definition of adjoint functors [Wik20a; nCa12a]. With a little more abstraction, we were able to reuse the same decomposition to prove the equivalence between universal morphisms and adjoint functors, with minimal effort.

Perhaps less typical of programming experience, we found that picking the right abstraction barriers could drastically reduce compile time by keeping details out of sight in large goal formulas. In the instance discussed in the introduction, we got a factor of ten speed-up by plugging holes in a leaky abstraction barrier![4]

6.4.2 Nested Σ Types

In Coq, there are two ways to represent a data structure with one constructor and many fields: as a single inductive type with one constructor (records) or as a nested Σ type. For instance, consider a record type with two type fields A and B and a function f from A to B. A logically equivalent encoding would be $\Sigma A.\ \Sigma B.\ A \to B$. There are two important differences between these encodings in Coq.

The first is that while a theorem statement may abstract over all possible Σ types, it may not abstract over all record types, which somehow have a less first-class status. Such a limitation is inconvenient and leads to code duplication.

[4]See commit eb00990 in HoTT/HoTT on GitHub for the exact change.

The far-more-pressing problem, overriding the previous point, is that nested Σ types have horrendous performance and are sometimes a few orders of magnitude slower. The culprit is projections from nested Σ types, which, when unfolded (as they must be, to do computation), each take almost the entirety of the nested Σ type as an argument and so grow in size very quickly.

Let's consider a toy example to see the asymptotic performance. To construct a nested Σ type with three fields of type `unit`, we can write the type:

```
{ _ : unit & { _ : unit & unit }}
```

If we want to project out the final field, we must write `projT2 (projT2 x)` which, when implicit arguments are included, expands to

```
@projT2 unit (λ _ : unit, unit) (@projT2 unit (λ _ : unit, { _ : unit
                                & unit }) x)
```

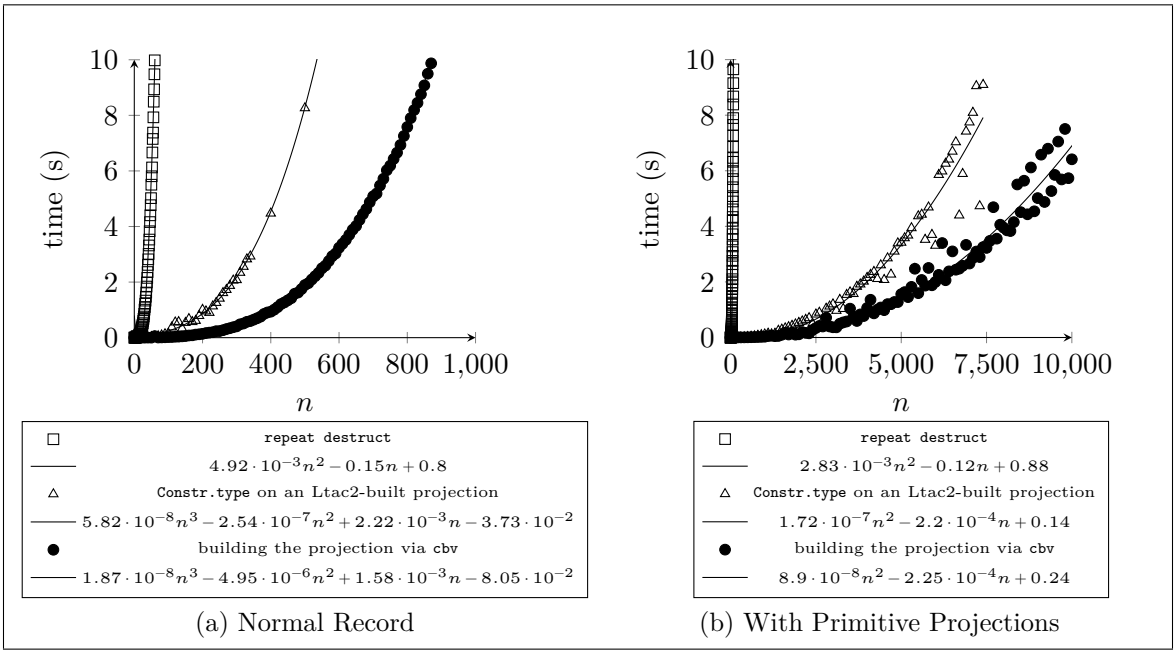
This term grows quadratically in the number of projections because the type of the n^{th} field is repeated approximately $2n$ times. This is even more of a problem when we need to **`destruct x`** to prove something about the projections, as we need to **`destruct`** it as many times as there are fields, which adds another factor of n to the performance cost of building the proof from scratch; in Coq, this cost is either avoided due to sharing or else is hidden by a quadratic factor with a much larger constant coefficient. Note that this is a sort-of dual to the problem of Subsection 2.6.1; there, we encountered quadratic overhead in applying the constructors (which is also a problem here), whereas right now we are discussing quadratic overhead in applying the eliminators. See Figure 6-1 for the performance details.

We can avoid much of the cost of building the projection term by using *primitive projections* (see Subsection 7.1.6 for more explanation of this feature). Note that this feature is a sort-of dual to the proposed feature of dropping constructor parameters described in Section 2.6.1. This does drastically reduce the overhead of building the projection term but only cuts in half the constant factor in destructing the variable so as to prove something about the projection. See Figure 6-1b for performance details.

There are two solutions to this issue:

1. use built-in *record* types
2. carefully define intermediate abstraction barriers to avoid the quadratic overhead

Both of these essentially solve the issue of quadratic overhead in projecting out the fields. This is the benefit of good abstraction barriers.



(a) Normal Record (b) With Primitive Projections

Figure 6-1: There are two ways we look at the performance of building a term like `projT1 (projT2 ... (projT2 x))` with n `projT2`s: we can define a recursive function that computes this term and then use **`cbv`** to reduce away the recursion and time how long this takes; or we can build the term using Ltac2 and then typecheck it. These plots display both of these methods and in addition display the time it takes to run **`destruct`** to break x into its component fields, as a lower bound for how long it takes to prove anything about a nested Σ type with n fields. The second graph displays the timing with primitive projections turned on. Note that the x-axis is $10\times$ larger on this plot.

In Coq 8.11, **destruct** is unfortunately still quadratic due to issues with name generation, but the constant factor is much smaller; see Figure 6-2 and Coq bug #12271.

We now come to the question: how much do we pay for using this abstraction barrier? That is, how much is the one-time cost of defining the abstraction barrier? Obviously, we can just make definitions for each of the projections and for the eliminator and pay the cubic (or perhaps even quartic; see the leading term in Figure 6-1) overhead once. There's an interesting question, though, of if we can avoid this overhead altogether.

As seen in Figure 6-2, using records partially avoids the overhead. Defining the record type, though, still incurs a quadratic factor due to hash consing the projections; see Coq bug #12270.

If our proof assistant does not support records out-of-the-box, or we want to avoid using them for whatever reason[5], we can instead define intermediate abstraction barriers by hand. Here is what code that almost works looks like for four fields:

```
Local Set Implicit Arguments.
Record sigT {A} (P : A -> Type) := existT { projT1 : A ; projT2 : P projT1 }.
Definition sigT_eta {A P} (x : @sigT A P) : x = existT P (projT1 x) (projT2 x).
Proof. destruct x; reflexivity. Defined.
Definition _T0 := unit.
Definition _T1 := @sigT unit (fun _ : unit => _T0).
Definition _T2 := @sigT unit (fun _ : unit => _T1).
Definition _T3 := @sigT unit (fun _ : unit => _T2).
Definition T := _T3.
Definition Build_T0 (x0 : unit) : _T0 := x0.
Definition Build_T1 (x0 : unit) (rest : _T0) : _T1
  := @existT unit (fun _ : unit => _T0) x0 rest.
Definition Build_T2 (x0 : unit) (rest : _T1) : _T2
  := @existT unit (fun _ : unit => _T1) x0 rest.
Definition Build_T3 (x0 : unit) (rest : _T2) : _T3
  := @existT unit (fun _ : unit => _T2) x0 rest.
Definition Build_T (x0 : unit) (x1 : unit) (x2 : unit) (x3 : unit) : T
  := Build_T3 x0 (Build_T2 x1 (Build_T1 x2 (Build_T0 x3))).

Definition _T0_proj (x : _T0) : unit := x.
Definition _T1_proj1 (x : _T1) : unit := projT1 x.
Definition _T1_proj2 (x : _T1) : _T0 := projT2 x.
Definition _T2_proj1 (x : _T2) : unit := projT1 x.
Definition _T2_proj2 (x : _T2) : _T1 := projT2 x.
Definition _T3_proj1 (x : _T3) : unit := projT1 x.
Definition _T3_proj2 (x : _T3) : _T2 := projT2 x.
```

[5]Note that the UniMath library [Voe15; VAG+20; Gra18] does this.

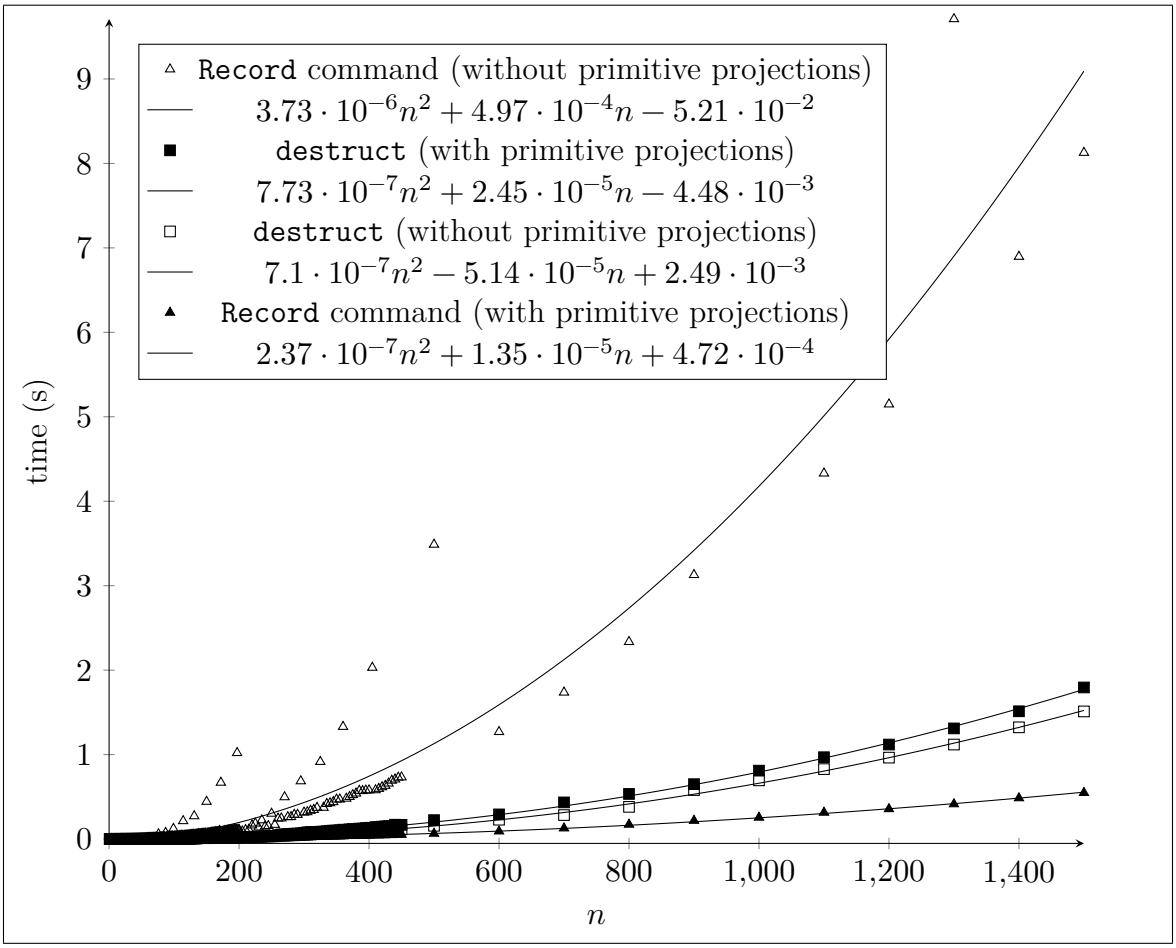


Figure 6-2: Timing of running a **`Record`** command to define a record with n fields and the time to **`destruct`** such a record. Note that building the goal involving projecting out the last field takes less than 0.001s for all numbers of fields that we tested. (Presumably for large enough numbers of fields, we'd start getting a logarithmic overhead from parsing the name of the final field, which, when represented as x followed by the field number in base 10, does grow in size as $\log_{10} n$.) Note that the nonmonotonic timing is *reproducible* and seems to be due to hitting garbage collection; see Coq issue #12270 for more details.

```
Definition proj_T_1 (x : T) : unit := _T3_proj1 x.
Definition proj_T_1_rest (x : T) : _T2 := _T3_proj2 x.
Definition proj_T_2 (x : T) : unit := _T2_proj1 (proj_T_1_rest x).
Definition proj_T_2_rest (x : T) : _T1 := _T2_proj2 (proj_T_1_rest x).
Definition proj_T_3 (x : T) : unit := _T1_proj1 (proj_T_2_rest x).
Definition proj_T_3_rest (x : T) : _T0 := _T1_proj2 (proj_T_2_rest x).
Definition proj_T_4 (x : T) : unit := _T0_proj (proj_T_3_rest x).

Definition _T0_eta (x : _T0) : x = Build_T0 (_T0_proj x) := @eq_refl _T0 x.
Definition _T1_eta (x : _T1) : x = Build_T1 (_T1_proj1 x) (_T1_proj2 x)
  := @sigT_eta unit (fun _ : unit => _T0) x.
Definition _T2_eta (x : _T2) : x = Build_T2 (_T2_proj1 x) (_T2_proj2 x)
  := @sigT_eta unit (fun _ : unit => _T1) x.
Definition _T3_eta (x : _T3) : x = Build_T3 (_T3_proj1 x) (_T3_proj2 x)
  := @sigT_eta unit (fun _ : unit => _T2) x.

Definition T_eta (x : T)
 : x = Build_T (proj_T_1 x) (proj_T_2 x) (proj_T_3 x) (proj_T_4 x)
  := let lhs3 := x in
     let lhs2 := _T3_proj2 lhs3 in
     let lhs1 := _T2_proj2 lhs2 in
     let lhs0 := _T1_proj2 lhs1 in
     let final := _T0_proj lhs0 in
     let rhs0 := Build_T0 final in
     let rhs1 := Build_T1 (_T1_proj1 lhs1) rhs0 in
     let rhs2 := Build_T2 (_T2_proj1 lhs2) rhs1 in
     let rhs3 := Build_T3 (_T3_proj1 lhs3) rhs2 in
     (((@eq_trans _T3)
         lhs3 (Build_T3 (_T3_proj1 lhs3) lhs2) rhs3
         (_T3_eta lhs3)
         ((@f_equal _T2 _T3 (Build_T3 (_T3_proj1 lhs3)))
            lhs2 rhs2
            ((@eq_trans _T2)
               lhs2 (Build_T2 (_T2_proj1 lhs2) lhs1) rhs2
               (_T2_eta lhs2)
               ((@f_equal _T1 _T2 (Build_T2 (_T2_proj1 lhs2)))
                  lhs1 rhs1
                  ((@eq_trans _T1)
                     lhs1 (Build_T1 (_T1_proj1 lhs1) lhs0) rhs1
                     (_T1_eta lhs1)
                     ((@f_equal _T0 _T1 (Build_T1 (_T1_proj1 lhs1)))
                        lhs0 rhs0
                        (_T0_eta lhs0)))))))
      : x = Build_T (proj_T_1 x) (proj_T_2 x) (proj_T_3 x) (proj_T_4 x)).
```

```
Import EqNotations.
Definition T_rect (P : T -> Type)
           (f : forall (x0 : unit) (x1 : unit) (x2 : unit) (x3 : unit),
               P (Build_T x0 x1 x2 x3))
           (x : T)
  : P x
  := rew <- [P] T_eta x in
     f (proj_T_1 x) (proj_T_2 x) (proj_T_3 x) (proj_T_4 x).
```

It only almost works because, although the overall size of the terms, even accounting for implicits, is linear in the number of fields, we still incur a quadratic number of unfoldings in the final cast node in the proof of `T_eta`. Note that this cast node is only present to make explicit the conversion problem that must happen; removing it does not break anything, but then the quadratic cost is hidden in nontrivial substitutions of the **`let`** binders into the types. It might be possible to avoid this quadratic factor by being even more careful, but I was unable to find a way to do it.[6] Worse, though, due to the issue with nested **`let`** binders described in Section 2.6.1, we would still incur a quadratic typechecking cost.

We can, however, avoid this cost by turning on primitive projections via `Set Primitive Projections` at the top of this block of code: this enables judgmental η-conversion for primitive records, whence we can prove `T_eta` with the proof term `@eq_refl T x`.

At least, so says the theoretical analysis. Our best stab at implementing this still resulted in at least quadratic asymptotic performance, if not worse.

6.5 Internalizing Duality Arguments in Type Theory

In general, we tried to design our library so that trivial proofs on paper remain trivial when formalized. One of Coq's main tools to make proofs trivial is the definitional

[6] Note that even reflective automation (see Chapter 3) is not sufficient to solve this issue. Essentially, the bottleneck is that at the bottom of the chain of **`let`** binders in the η proof, we have two different types for the η principle. One of them uses the globally defined projections out of `T`, while the other uses the projections of `x` defined in the local context. We need to convert between these two types in linear time. Converting between two differently defined projections takes time linear in the number of under-the-hood projections, i.e., linear in the number of fields. Doing this once for each projection thus takes quadratic time. Using a reflective representation of nested Σ types, and thus being able to prove the η principle once and for all in constant time, would not help here, because it takes quadratic time to convert between the type of the η principle in reflective-land and the type that we want. One thing that might help would be to have a version of conversion checking that was both memoized and could perform in-place reduction; see Coq issue #12269.

equality, where some facts follow by computational reduction of terms. We came up with some small tweaks to core definitions that allow a common family of proofs by *duality* to follow by computation.

This is an exception to the rule of opaque abstraction barriers. Here, deliberate breaking of all abstraction barriers in a careful way can result in performance gains of up to a factor of two!

Proof by duality is a common idea in higher mathematics: sometimes, it is productive to flip the directions of all the arrows. For example, if some fact about least upper bounds is provable, chances are that the same kind of fact about greatest lower bounds will also be provable in roughly the same way, by replacing "greater than"s with "less than"s and vice versa.

Concretely, there is a dualizing operation on categories that inverts the directions of the morphisms:

```
Notation "C ^op" := ({| Ob := Ob C; Hom x y := Hom C y x; ... |}).
```

Dualization can be used, roughly, for example, to turn a proof that Cartesian product is an associative operation into a proof that disjoint union is an associative operation; products are dual to disjoint unions.

One of the simplest examples of duality in category theory is initial and terminal objects. In a category $\mathcal{C}$, an initial object 0 is one that has a unique morphism $0 \to x$ to every object x in $\mathcal{C}$; a terminal object 1 is one that has a unique morphism $x \to 1$ from every object x in $\mathcal{C}$. Initial objects in $\mathcal{C}$ are terminal objects in $\mathcal{C}^{\text{op}}$. The initial object of any category is unique up to isomorphism; for any two initial objects 0 and $0'$, there is an isomorphism $0 \cong 0'$. By flipping all of the arrows around, we can prove, by duality, that the terminal object is unique up to isomorphism. More precisely, from a proof that an initial object of $\mathcal{C}^{\text{op}}$ is unique up to isomorphism, we get that any two terminal objects $1'$ and 1 in $\mathcal{C}$, which are initial in $\mathcal{C}^{\text{op}}$, are isomorphic in $\mathcal{C}^{\text{op}}$. Since an isomorphism $x \cong y$ in $\mathcal{C}^{\text{op}}$ is an isomorphism $y \cong x$ in $\mathcal{C}$, we get that 1 and $1'$ are isomorphic in $\mathcal{C}$.

It is generally straightforward to see that there is an isomorphism between a theorem and its dual, and the technique of dualization is well-known to category theorists, among others. We discovered that, by being careful about how we defined constructions, we could make theorems be judgmentally equal to their duals! That is, when we prove a theorem

```
initial_ob_unique : ∀ C(x y : Ob C),
                      is_initial_ob x → is_initial_ob y → x ≅ y,
```

we can define another theorem

```
terminal_ob_unique : ∀ C(x y : Ob C),
                        is_terminal_ob x → is_terminal_ob y → x ≅ y
```

as

```
terminal_ob_unique C x y H H' := initial_ob_unique C^op y x H' H.
```

Interestingly, we found that in proofs with sufficiently complicated types, it can take a few seconds or more for Coq to accept such a definition; we are not sure whether this is due to peculiarities of the reduction strategy of our version of Coq, or speed dependency on the size of the normal form of the type (rather than on the size of the unnormalized type), or something else entirely.

In contrast to the simplicity of witnessing the isomorphism, it takes a significant amount of care in defining concepts, often to get around deficiencies of Coq, to achieve *judgmental* duality. Even now, we were unable to achieve this ideal for some theorems. For example, category theorists typically identify the functor category $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$ (whose objects are functors $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$ and whose morphisms are natural transformations) with $(\mathcal{C} \to \mathcal{D})^{\mathrm{op}}$ (whose objects are functors $\mathcal{C} \to \mathcal{D}$ and whose morphisms are flipped natural transformations). These categories are canonically isomorphic (by the dualizing natural transformations), and, with the univalence axiom [Uni13], they are equal as categories! However, to make these categories definitionally equal, we need to define functors as a structural record type (see Section 2.6.1) rather than a nominal one.

6.5.1 Duality Design Patterns

One of the simplest theorems about duality is that it is involutive; we have that $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}} = \mathcal{C}$. In order to internalize proof by duality via judgmental equality, we sometimes need this equality to be judgmental. Although it is impossible in general in Coq 8.4 (see dodging judgmental η on records below), the latest version of Coq available when we were creating this library, we want at least to have it be true for any explicit category (that is, any category specified by giving its objects, morphisms, etc., rather than referred to via a local variable).

Removing Symmetry

Taking the dual of a category, one constructs a proof that $f \circ (g \circ h) = (f \circ g) \circ h$ from a proof that $(f \circ g) \circ h = f \circ (g \circ h)$. The standard approach is to apply symmetry. However, because applying symmetry twice results in a judgmentally different proof, we decided instead to extend the definition of `Category` to require both a proof of

$f \circ (g \circ h) = (f \circ g) \circ h$ and a proof of $(f \circ g) \circ h = f \circ (g \circ h)$. Then our dualizing operation simply swaps the proofs. We added a convenience constructor for categories that asks only for one of the proofs and applies symmetry to get the other one. Because we formalized 0-truncated category theory, where the type of morphisms is required to have unique identity proofs, asking for this other proof does not result in any coherence issues.

Dualizing the Terminal Category

To make everything work out nicely, we needed the terminal category, which is the category with one object and only the identity morphism, to be the dual of itself. We originally had the terminal category as a special case of the discrete category on n objects. Given a type T with uniqueness of identity proofs, the discrete category on T has as objects inhabitants of T and has as morphisms from x to y proofs that $x = y$. These categories are not judgmentally equal to their duals, because the type $x = y$ is not judgmentally the same as the type $y = x$. As a result, we instead used the indiscrete category, which has `unit` as its type of morphisms.

Which Side Does the Identity Go On?

The last tricky obstacle we encountered was that when defining a functor out of the terminal category, it is necessary to pick whether to use the right identity law or the left identity law to prove that the functor preserves composition; both will prove that the identity composed with itself is the identity. The problem is that dualizing the functor leads to a road block where either concrete choice turns out to be "wrong," because the dual of the functor out of the terminal category will not be judgmentally equal to another instance of itself. To fix this problem, we further extended the definition of category to require a proof that the identity composed with itself is the identity.

Dodging Judgmental η on Records

The last problem we ran into was the fact that sometimes, we really, really wanted judgmental η on records. The η rule for records says any application of the record constructor to all the projections of an object yields exactly that object; e.g. for pairs, $x \equiv (x_1, x_2)$ (where x_1 and x_2 are the first and second projections, respectively). For categories, the η rule says that given a category $\mathcal{C}$, for a "new" category defined by saying that its objects are the objects of $\mathcal{C}$, its morphisms are the morphisms of $\mathcal{C}$, …, the "new" category is judgmentally equal to $\mathcal{C}$.

In particular, we wanted to show that any functor out of the terminal category is the opposite of some other functor; namely, any $F : 1 \to \mathcal{C}$ should be equal to $(F^{\mathrm{op}})^{\mathrm{op}} : 1 \to (\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$. However, without the judgmental η rule for records, a local

variable $\mathcal{C}$ cannot be judgmentally equal to $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$, which reduces to an application of the constructor for a category, unless the η rule is built into the proof assistant. To get around the problem, we made two variants of dual functors: given $F : \mathcal{C} \to \mathcal{D}$, we have $F^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, and given $F : C^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, we have $F^{\mathrm{op}'} : \mathcal{C} \to \mathcal{D}$. There are two other flavors of dual functors, corresponding to the other two pairings of $^{\mathrm{op}}$ with domain and codomain, but we have been glad to avoid defining them so far. As it was, we ended up having four variants of dual natural transformation and are very glad that we did not need sixteen. When Coq 8.5 was released, we no longer needed to pull this trick, as we could simply enable the η rule for records judgmentally.

6.5.2 Moving Forward: Computation Rules for Pattern Matching

While we were able to work around most of the issues that we had in internalizing proof by duality, the experience would have been far nicer if we had more η rules. The η rule for records is explained above. The η rule for equality says that the identity function is judgmentally equal to the function $f : \forall x\, y, x = y \to x = y$ defined by pattern matching on the first proof of equality; this rule is necessary to have any hope that applying symmetry twice is judgmentally the identity transformation.

Homotopy type theory provides a framework that systematizes reasoning about proofs of equality, turning a seemingly impossible task into a manageable one. However, there is still a significant burden associated with reasoning about equalities, because so few of the rules are judgmental.

We have spent some time attempting to divine the appropriate computation rules for pattern-matching constructs, in the hopes of making reasoning with proofs of equality more pleasant.[7]

[7]See Coq issue #3179 and Coq issue #3119.

Part IV

Conclusion

Chapter 7

A Retrospective on Performance Improvements

Throughout this dissertation, we've looked at the problem of performance in proof assistants, especially those based on dependent type theory, with Coq as our primary tool under investigation. Part I aimed to convince the reader that this problem is interesting, important, challenging, and understudied, as it differs in nontrivial ways from performance bottlenecks in nondependently typed languages. Part II took a deep dive into a particular set of performance bottlenecks and presented a tool and, we hope, exposed the underlying design methodology, which allows eliminating asymptotic bottlenecks in one important part of proof-assistant systems. Part III zoomed back out to discuss design principles to avoid performance pitfalls.

In this chapter, we will look instead at the successes of the past decade[1], ways in which performance has improved in major ways. Section 7.1 will discuss specific improvements in the implementation of Coq which resulted in performance gains, paying special attention to the underlying bottlenecks being addressed. Those without special interest in the low-level details of proof-assistant implementation may want to skip to Section 7.2, which will discuss changes to the underlying type theory of Coq which make possible drastic performance improvements. While we will again have our eye on Coq in Section 7.2, we will broaden our perspective in Section 7.3 to discuss new discoveries of the past decade or so in dependent type theory which enable performance improvements but have not yet made their way into Coq.

[1] Actually, the time span we're considering is the course of the author's experience with Coq, which is a bit less than a decade.

7.1 Concrete Performance Advancements in Coq

In this section, we dive into the minutiae: concrete changes to Coq that have measurably increased performance.

7.1.1 Removing Pervasive Evar Normalization

Back when I started using Coq, in version 8.4, almost every single tactic was at least linear in performance in the size of the goal. This included tactics like "add new hypothesis to the context of type `True`" (`pose proof I`) and tactics like "give me the type of the most recently added hypothesis" (`match goal with H : ?T |- _ => T end`). The reason for this was pervasive *evar normalization*.

Let us review some details of the way Coq handles proof scripts. In Coq the current state of a proof is represented by a partial proof term, where not-yet-given subterms are *existential variables*, or evars, which may show up as goals. For example, when proving the goal `True ∧ True`, after running `split`, the proof term would be `conj ?Goal1 ?Goal2`, where `?Goal1` and `?Goal2` are evars. There are two subtleties:

1. Evars may be under binders. Coq uses a locally nameless representation of terms (c.f. Section 3.1.3), where terms use de Bruijn indices to refer to variables bound in the term but use names to refer to variables bound elsewhere. Thus terms generated in the context of a proof goal refer to all context variables by name and evars too refer to all variables by name. Hence each evar carries with it a named context, which causes a great deal of trouble as described in Section 2.6.3 (Quadratic Creation of Substitutions for Existential Variables).

2. Coq supports backtracking, so we must remember the history of partial proof terms. In particular, we cannot simply mutate partial proof terms to instantiate the evars, and copying the entire partial proof term just to update a small part of it would also incur a great deal of overhead. Instead, Coq never mutates the terms and instead simply keeps a map of which evars have been instantiated with which terms, called the *evar map*.

There is an issue with the straightforward implementation of evars and evar maps. When walking terms, care must be taken with the evar case, to check whether or not the evar has in fact been instantiated or not. Subtle bugs in unification and other areas of Coq resulted from some functions being incorrectly sensitive to whether or not a term had been built via evar instantiation or given directly.[2] The fast-and-easy solution used in older versions of Coq was to simply evar-normalize the goal before walking it. That is, every tactic that had to walk the goal for any reason whatsoever would create a copy of the type of the goal—and sometimes the proof context as

[2]See the discussion at Pédrot [Péd17b] for more details.

well—replacing all instantiated evars with their instantiations. Needless to say, this was very expensive when the size of the goal was large.

As of Coq 8.7, most tactics no longer perform useless evar normalization and instead walk terms using a dedicated API which does on-the-fly normalization as necessary [Péd17b]. This brought speedups of over 10% to some developments and improved asymptotic performance of some tactic scripts and interactive proof development.

7.1.2 Delaying the Externalization of Application Arguments

Coq has many representations of terms. There is `constr_expr`, the AST produced by Coq's parser. Internalization turns `constr_expr` into the untyped `glob_constr` representation of terms by performing name resolution, bound-variable checks, notation desugaring, and implicit-argument insertion [Coqb]. Type inference fills in the holes in untyped `glob_constr`s to turn them into typed `constr`s, possibly with remaining existential variables [Coqe]. In order to display proof goals, this process must be reversed. The internal representation of `constr` must be "detyped" into `glob_constr`s, which involves primarily just turning de Bruijn indices into names [Coqc]. Finally, implicit arguments must be erased and notations must be resugared when externalizing `glob_constr`s into `constr_expr`s, which can be printed relatively straightforwardly [Coqa; Coqd].

In old versions, Coq would externalize the entire goal, including subterms that were never printed due to being hidden by notations and implicit arguments. Starting in version 8.5pl2, lazy externalization of function arguments was implemented [Péd16b]. This resulted in massive speed-ups to interactive development involving large goals whose biggest subterms were mostly hidden.

Changes like this one can be game-changers for interactive proof development. The kind of development that can happen when it takes a tenth of a second to see the goal after executing a tactic is vastly different from the kind of development that can happen when it takes a full second or two. In the former case, the proof engine can almost feel like an extension of the coder's mind, responding to thoughts about strategies to try almost as fast as they can be typed. In the latter case, development is significantly more clunky and involves much more friction.

In the same vein, bugs such as #3691 and #4819, where Coq crawled the entire evar map in `-emacs` mode (used for ProofGeneral/Emacs) looking at all instantiated evars, resulted in interactive proof times of up to half a second for every goal display, even when the goal was small and there was nothing in the context. Fixed in Coq 8.6, these bugs, too, got in the way of seamless proof development.

7.1.3 The $\mathcal{L}_{tac}$ Profiler

> If you blindly optimize without profiling, you will likely waste your time on the 99% of code that isn't actually a performance bottleneck and miss the 1% that is.
>
> — Charles E. Leiserson[3] [Lei20]

In old versions of Coq, there was no good way to profile tactic execution. Users could wrap some invocations in `time` to see how long a given tactic took or could regularly print some output to see where execution hung. Both of these are very low-tech methods of performance debugging and work well enough for small tactics. For debugging hundreds or thousands of lines of $\mathcal{L}_{tac}$ code, though, these methods are insufficient.

A genuine profiler for $\mathcal{L}_{tac}$ was developed in 2015 and integrated into Coq itself in version 8.6 [TG15].

For those interested in amusing quirks of implementation details, the profiler itself was relatively easy to implement. If I recall correctly, Tobias Tebbi, after hearing of my $\mathcal{L}_{tac}$ performance woes, mentioned to me the profiler he implemented over the course of a couple of days. Since $\mathcal{L}_{tac}$ already records backtraces for error reporting, it was a relatively simple matter to hook into the stack-trace recorder and track how much time was spent in each call stack. With some help from the Coq development team, I was able to adapt the patch to the new tactic engine of Coq $\geq$ 8.5 and shepherded it into Coq's codebase.

7.1.4 Compilation to Native Code

Starting in version 8.5, Coq allows users to compile their functional Gallina programs to native code and fully reduce them to determine their output [BDG11; Dén13a]. In some cases, the native compiler is almost 10× faster[4] than the optimized call-by-value evaluation bytecode-based virtual machine described in Grégoire and Leroy [GL02].

[3] Although this quote comes from the class I took at MIT, 6.172 — Performance Engineering of Software Systems, the inspiration for the quote is an extended version of Donald Knuth's "premature optimization is the root of all evil" quote:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
>
> — Donald E. Knuth [Knu74b, p. 268]

[4] `https://github.com/coq/coq/pull/12405#issuecomment-633612308`

The native compiler shines most at optimizing algorithmic and computational bottlenecks. For example, computing the number of primes less than n via the Sieve of Eratosthenes is about $2\times$ to $5\times$ faster in the native compiler than in the VM. By contrast, when the input term is very large compared to the amount of computation, the compilation time can dwarf the running time, eating up any gains that the native compiler has over the VM. This can be seen by comparing the times it takes to get the head of the explicit list of all unary-encoded natural numbers less than, say, 3000, on which the native compiler (1.7s) is about 5% slower than the VM (1.6s) which itself is about $2\times$ slower than built-in call-by-value reduction machine (0.79s) which requires no translation. Furthermore, when the output is large, both the VM and the native compiler suffer from inefficiencies in the readback code.

7.1.5 Primitive Integers and Arrays

Primitive 31-bit integer-arithmetic operations were added to Coq in 2007 [Spi07; Arm+10]. Although most of Coq merely used an inductive representation of 31-bit integers, the VM included code for compiling these constants to native machine integers.[5] After hitting memory limits in storing the inductive representations in proofs involving proof traces from SMT solvers, work was started to allow the use of primitive datatypes that would be stored efficiently in proof terms [Dén13b].

Some of this work has since been merged into Coq, including IEEE 754-2008 binary64 floating-point numbers merged in Coq 8.11 [MBR19], 63-bit integers merged in Coq 8.10 [DG18], and persistent arrays [CF07] merged into Coq 8.13 [Dén20b]. Work enabling primitive recursion over these native datatypes is still underway [Dén20a], and the actual use of these primitive datatypes to reap the performance benefits is still to come as of the writing of this dissertation.

7.1.6 Primitive Projections for Record Types

Since version 8.5, Coq has had the ability to define record types with projections whose arguments are not stored in the term representation [Soz14]. This allows asymptotic speedups, as discussed in Subsection 6.4.2 (Nested Σ Types).

Note that this is a specific instance of a more general theory of implicit arguments [Miq01; BB08], and there has been other work on how to eliminate useless arguments from term representations [BMM03].

[5]The integer arithmetic is 31-bit rather than 32-bit because OCaml reserves the lowest bit for tagging whether a value is a pointer address to a tagged value or an integer.

7.1.7 Fast Typing of Application Nodes

In Section 2.6.3 (Quadratic Substitution in Function Application), we discussed how the typing rule for function application resulted in quadratic performance behavior when there was in fact only linear work that needed to be done. As of Coq 8.10, when typechecking applications in the kernel, substitution is delayed so as to achieve linear performance [Péd18]. Unfortunately, the pretyping and type-inference algorithm is still quadratic, due to the type-theory rules used for type inference.

7.2 Performance-Enhancing Advancements in the Type Theory of Coq

While some of the above performance enhancements touch the trusted kernel of Coq, they do not fundamentally change the type theory. Some performance enhancements require significant changes to the type theory. In this section we will review a couple of particularly important changes of this kind.

7.2.1 Universe Polymorphism

Recall that the main case study of Chapter 6 was our implementation of a category-theory library. Recall also from Type Size Blowup: Packed vs. Unpacked Records how the choice of whether to use packed or unpacked records impacts performance; while unpacked records are more friendly for developing algebraic hierarchies, packed records achieve significantly better performance when large towers of dependent concepts (such as categories, functors between categories, and natural transformations between functors) are formalized.

This section addresses a particular feature which allows an entire-library 2× speed-up when using fully packed records. How is such a large performance gain achievable? Without this feature, called *universe polymorphism*, encoding some mathematical objects requires *duplicating* the entire library! Removing this duplication of code will halve the compile time.

What Are Universes?

Universes are type theory's answer to Russell's paradox [ID16]. Russell's paradox, a famous paradox discovered in 1901, proceeds as follows. A *set* is an unordered collection of distinct objects. Since each *set* is an object, we may consider the set of all sets. Does this set contain itself? It must, for by definition it contains all sets.

So we see by example that some sets contain themselves, while others (such as the

empty set with no objects) do not. Let us consider now the set consisting of exactly the sets that do not contain themselves. Does this set contain itself? If it does not, then it fails to live up to its description as the set of *all* sets that do not contain themselves. However, if it does contain itself, then it also fails to live up to its description as a set consisting *only* of sets that do not contain themselves. Paradox!

The resolution to this paradox is to forbid sets from containing themselves. The collection of all sets is too big to be a set, so let's call it (and collections of its size) a proper class. We can nest this construction, as type theory does: We have $\texttt{Type}_0$, the $\texttt{Type}_1$ of all small types, and we have $\texttt{Type}_1$, the $\texttt{Type}_2$ of all $\texttt{Type}_1$s, etc. These subscripts are called *universe levels*, and the subscripted **Type**s are sometimes called *universes*.

Most constructions in Coq work just fine if we simply place them in a single, high-enough universe. In fact, the entire standard library in Coq effectively uses only three universes. Most of the standard library in fact only needs one universe. We need a second universe for the few constructions that talk about equality between types, and a third for the encoding of a variant of Russell's paradox in Coq.

However, one universe is not sufficient for category theory, even if we don't need to talk about equality of types nor prove that `Type : Type` is inconsistent.

The reason is that category theory, much like set theory, talks about itself.

Complications from Categories of Categories

In standard mathematical practice, a category $\mathcal{C}$ can be defined [Awo] to consist of:

- a class $\mathrm{Ob}_{\mathcal{C}}$ of *objects*
- for all objects $a, b \in \mathrm{Ob}_{\mathcal{C}}$, a class $\mathrm{Hom}_{\mathcal{C}}(a, b)$ of *morphisms from a to b*
- for each object $x \in \mathrm{Ob}_{\mathcal{C}}$, an *identity morphism* $1_x \in \mathrm{Hom}_{\mathcal{C}}(x, x)$
- for each triple of objects $a, b, c \in \mathrm{Ob}_{\mathcal{C}}$, a *composition function* $\circ : \mathrm{Hom}_{\mathcal{C}}(b, c) \times \mathrm{Hom}_{\mathcal{C}}(a, b) \to \mathrm{Hom}_{\mathcal{C}}(a, c)$

satisfying the following axioms:

- associativity: for composable morphisms f, g, h, we have $f \circ (g \circ h) = (f \circ g) \circ h$.
- identity: for any morphism $f \in \mathrm{Hom}_{\mathcal{C}}(a, b)$, we have $1_b \circ f = f = f \circ 1_a$

Some complications arise in applying this definition of categories to the full range of common constructs in category theory. One particularly prominent example formalizes the structure of a collection of categories, showing that this collection itself may be considered as a category.

The morphisms in such a category are *functors*, maps between categories consisting of a function on objects, a function on hom-types, and proofs that these functions respect composition and identity [Mac; Awo; Uni13].

The naïve concept of a "category of all categories", which includes even itself, leads into mathematical inconsistencies which manifest as universe-inconsistency errors in Coq, much as with the set of all sets discussed above.

The standard resolution, as with sets, is to introduce a hierarchy of categories, where, for instance, most intuitive constructions are considered *small* categories, and then we also have *large* categories, one of which is the category of small categories. Both definitions wind up with literally the same text in Coq, giving:

```
Definition SmallCat : LargeCategory :=
  {| Ob := SmallCategory;
     Hom C D := SmallFunctor C D;
     ⋮
  |}.
```

It seems a shame to copy-and-paste this definition (and those of `Category`, `Functor`, etc.) n times to define an n-level hierarchy.

Universe polymorphism is a feature that allows definitions to be quantified over their universes. While Coq 8.4 supports a restricted flavor of universe polymorphism that allows the universe of a definition to vary as a function of the universes of its arguments, Coq 8.5 and later [Soz14] support an established kind of more general universe polymorphism [HP91], previously implemented only in NuPRL [Con+86]. In these versions of Coq, any definitions declared polymorphic are parametric over their universes.

While judicious use of universe polymorphism can reduce code duplication, careless use can lead to tens of thousands of universe variables which then become a performance bottleneck in their own right.[6]

[6]See, for example, the commit message of a445bc3 in the HoTT/HoTT library on GitHub, where moving from Coq 8.5β2 to 8.5β3 incurred a 4× slowdown in the file `hit/V.v`, entirely due to performance regressions in universe handling, which were later fixed. This slowdown is likely the one of Coq bug #4537.

See also commit d499ef6 in the HoTT/HoTT library on GitHub, where reducing the number of polymorphic universes in some constants used by `rewrite` resulted in an overall 2× speedup, with

7.2.2 Judgmental η for Record Types

The same commit that introduced universe polymorphism in Coq 8.5 also introduced judgmental η conversion for records with primitive projections [Soz14]. We have already discussed the advantages of primitive projections in Subsection 7.1.6, and we have talked a bit about judgmental η in Section 6.5.1 (Dodging Judgmental η on Records) and Section 6.5 (Internalizing Duality Arguments in Type Theory).

The η conversion rule for records says that every term x of record type T is convertible with the constructor of T applied to the projections of T applied to x. For example, if `x` has type `A * B`, then the η rule equates `x` with `(fst x, snd x)`.

As discussed in Section 6.5, having records with judgmental η conversion allows deduplicating code that would otherwise have to be duplicated.

7.2.3 SProp: The Definitionally Proof-Irrelevant Universe

Coq is slow at dealing with large terms. For goals around 175,000 words long[7], we have found that simple tactics like `apply f_equal` take around 1 second to execute, which makes interactive theorem proving very frustrating.[8] Even more frustrating is the fact that the largest contribution to this size is often arguments to irrelevant functions, i.e., functions that are provably equal to all other functions of the same type. (These are proofs related to algebraic laws like associativity, carried inside many constructions.)

Opacification helps by preventing the type checker from unfolding some definitions, but it is not enough: the type checker still has to deal with all of the large arguments to the opaque function. Hash-consing might fix the problem completely.

Alternatively, it would be nice if, given a proof that all of the inhabitants of a type were equal, we could forget about terms of that type, so that their sizes would not impose any penalties on term manipulation. One solution might be irrelevant fields, like those of Agda, or implemented via the Implicit CiC [BB08; Miq01]. While there is as-yet no way to erase these arguments, Coq versions 8.10 and later have the

speedups reaching 10× in some `rewrite`-heavy files.

Coq actually had an implementation of full universe polymorphism between versions 8.3 and 8.4, implemented in commit d98dfbc and reverted mere minutes later in commit 60bc3cb. In-person discussion, either with Matthieu himself or with Bob Harper, revealed that Matthieu abandoned this initial attempt after finding that universe polymorphism was too slow, and it was only by implementing the algorithm of Harper and Pollack [HP91] that universe polymorphism with typical ambiguity [Shu12; Spe66; HP91], where users need not write universe variables explicitly, was able to be implemented in a way that was sufficiently performant.

[7]When we had objects as arguments rather than fields in our category-theory library (see Section 2.6.4), we encountered goals of about 219,633 words when constructing pointwise Kan extensions.

[8]See also Coq bug #3280.

ability to define types as judgmentally irrelevant, paving the way for more aggressive erasure [Gil18; Gil+19].

7.3 Performance-Enhancing Advancements in Type Theory at Large

We come now to discoveries and inventions of the past decade or so which have not yet made it into Coq but which show great promise for significant performance improvements.

7.3.1 Higher Inductive Types: Setoids for Free

Recall again that the main case study of Chapter 6 was our implementation of a category-theory library.

Equality

Equality, which has recently become a very hot topic in type theory [Uni13] and higher category theory [Lei], provides another example of a design decision where most usage is independent of the exact implementation details. Although the question of what it means for objects or morphisms to be equal does not come up much in classical 1-category theory, it is more important when formalizing category theory in a proof assistant, for reasons seemingly unrelated to its importance in higher category theory. We consider some possible notions of equality.

Setoids A setoid [Bis67] is a carrier type equipped with an equivalence relation; a map of setoids is a function between the carrier types and a proof that the function respects the equivalence relations of its domain and codomain. Many authors [Pee+; KSW; Meg; HS00; Ahrb; Ahr10; Ish; Pot; Soza; CM98; Wil12] choose to use a setoid of morphisms, which allows for the definition of the category of set(oid)s, as well as the category of (small) categories, without assuming functional extensionality, and allows for the definition of categories where the objects are quotient types. However, there is significant overhead associated with using setoids everywhere, which can lead to slower compile times. Every type that we talk about needs to come with a relation and a proof that this relation is an equivalence relation. Every function that we use needs to come with a proof that it sends equivalent elements to equivalent elements. Even worse, if we need an equivalence relation on the universe of "types with equivalence relations", we need to provide a transport function between equivalent types that respects the equivalence relations of those types.

Propositional Equality An alternative to setoids is propositional equality, which carries none of the overhead of setoids but does not allow an easy formulation of quotient types and requires assuming functional extensionality to construct the category of sets.

Intensional type theories like Coq's have a built-in notion of equality, often called definitional equality or judgmental equality, and denoted as $x \equiv y$. This notion of equality, which is generally internal to an intensional type theory and therefore cannot be explicitly reasoned about inside of that type theory, is the equality that holds between $\beta\delta\iota\zeta\eta$-convertible terms.

Coq's standard library defines what is called *propositional equality* on top of judgmental equality, denoted $x = y$. One is allowed to conclude that propositional equality holds between any judgmentally equal terms.

Using propositional equality rather than setoids is convenient because there is already significant machinery made for reasoning about propositional equalities, and there is much less overhead. However, we ran into significant trouble when attempting to prove that the category of sets has all colimits, which amounts to proving that it is closed under disjoint unions and quotienting; quotient types cannot be encoded without assuming a number of other axioms.

Higher Inductive Types The recent emergence of higher inductive types allows the best of both worlds. The idea of higher inductive types [Uni13] is to allow inductive types to be equipped with extra proofs of equality between constructors. They originated as a way to allow homotopy type theorists to construct types with nontrivial higher paths. A very simple example is the interval type, from which functional extensionality can be proven [Shu]. The interval type consists of two inhabitants `zero : Interval` and `one : Interval`, and a proof `seg : zero = one`. In a hypothetical type theory with higher inductive types, the type checker does the work of carrying around an equivalence relation on each type for us and forbids users from constructing functions that do not respect the equivalence relation of any input type. For example, we can, hypothetically, prove functional extensionality as follows:

```
Definition f_equal {A B x y} (f : A → B) : x = y → f x = f y.
Definition functional_extensionality {A B} (f g : A → B)
    : (∀ x, f x = g x) → f = g
  := λ (H : ∀ x, f x = g x)
       ⇒ f_equal (λ (i : Interval) (x : A)
                    ⇒ match i with
                        | zero ⇒ f x
                        | one  ⇒ g x
                        | seg  ⇒ H x
```

```
      end)
seg.
```

Had we neglected to include the branch for `seg`, the type checker should complain about an incomplete match; the function λ `i : Interval` $\Rightarrow$ **`match i with zero`** $\Rightarrow$ `true | one` $\Rightarrow$ `false` **`end`** of type `Interval` $\rightarrow$ `bool` should not typecheck for this reason.

The key insight is that most types do not need special equivalence relations, and, moreover, if we are not explicitly dealing with a type with a special equivalence relation, then it is impossible (by parametricity) to fail to respect the equivalence relation. Said another way, the only way to construct a function that might fail to respect the equivalence relation would be by some eliminator like pattern matching, so all we have to do is guarantee that direct invocations of the eliminator result in functions that respect the equivalence relation.

As with the choice involved in defining categories, using propositional equality with higher inductive types rather than setoids derives many of its benefits from not having to deal with all of the overhead of custom equivalence relations in constructions that do not need them. In this case, we avoid the overhead by making the type checker or the metatheory deal with the parts we usually do not care about. Most of our definitions do not need custom equivalence relations, so the overhead of using setoids would be very large for very little gain.

7.3.2 Univalence and Isomorphism Transport

When considering higher inductive types, the question "when are two types equivalent?" arises naturally. The standard answer in the past has been "when they are syntactically equal". The result of this is that two inductive types that are defined in the same way, but with different names, will not be equal. Voevodsky's univalence principle gives a different answer: two types are equal when they are isomorphic. This principle, encoded formally as the *univalence axiom*, allows reasoning about isomorphic types as easily as if they were equal.

Tabareau et al. built a framework on top of the insights of univalence, combined with parametricity [Rey83; Wad89], for automatically porting definitions and theorems to equivalent types [TTS18; TTS19].

What is the application to performance? As we saw, for example, in Section 4.2 (NbE vs. Pattern-Matching Compilation: Mismatched Expression APIs and Leaky Abstraction Barriers), the choice of representation of a datatype can have drastic consequences on how easy it is to encode algorithms and write correctness proofs. These design choices can also be intricately entwined with both the compile-time

and run-time performance characteristics of the code. One central message of both Chapter 4 and Chapter 6 is that picking the right API really matters when writing code with dependent types. The promise of univalence, still in its infancy, is that we could pick the right API for each algorithmic chunk, prove the APIs isomorphic, and use some version of univalence to compose the APIs and reason about the algorithms as easily as if we had used the same interface everywhere.

7.3.3 Cubical Type Theory

One important detail we elided in the previous subsections is the question of computation. Higher inductive types and univalence are much less useful if they are opaque to the type checker. The proof of function extensionality, for example, relies on the elimination rule for the interval having a judgmental computation rule.[9]

Higher inductive types whose eliminators compute on the point constructors can be hacked into dependently typed proof assistants by adding inconsistent axioms and then hiding them behind opaque APIs so that inconsistency cannot be proven [Lic11; Ber13]. This is unsatisfactory, however, on two counts:

1. The eliminators do not compute on path constructors. For example, the interval eliminator would compute on `zero` and `one` but not on `seg`.
2. Adding these axioms compromises the trust story.

Cubical type theory is the solution to both of these problems, for both higher inductive types and univalence [Coh+18]. Unlike most other type theories, computation in cubical type theory is implemented by appealing to the category-theoretic model, and the insights that allow such computation are slowly making their way into more mainstream dependently typed proof assistants [VMA19].

[9] We leave it as a fun exercise for the advanced reader to figure out why the Church encoding of the interval, where `Interval := ∀ P (zero : P) (one : P) (seg : zero = one), P`, does not yield a proof a functional extensionality.

Chapter 8

Concluding Remarks

We spent Part I mapping out the landscape of the problems of performance we encountered in dependently typed proof assistants. In Part II and Part III, we laid out more-or-less systematic principles and tools for avoiding these performance bottlenecks. In the last chapter, Chapter 7, we looked back on the concrete performance improvements in Coq over time.

We look now to the future.

The clever reader might have noticed something that we swept under the rug in Parts II and III. In Section 1.3 we laid out two basic design choices—dependent types and the de Bruijn criterion—which are responsible for much of the power and much of the trust we can have in a proof assistant like Coq. We then spent the next chapters of this dissertation investigating the performance bottlenecks that can perhaps be said to result from these choices and how to ameliorate these performance issues.

If the strategies we laid out in Parts II and III for how to use dependent types and untrusted tactics in a performant way are to be summed up in one word, that word is: "don't!" To avoid the performance issues resulting from tactics being untrusted, the source of much of the trust in proof assistants like Coq, we suggest in Part II that users effectively *throw away the entire tactic engine* and instead code tactics reflectively. To avoid the performance issues incurred by unpredictable computation at the type level, the source of much of the power of dependent type theory, we broadly suggest in Part III to *avoid using the computation at all* (except in the rare cases where the entire proof can be moved into computation at the type level, such as proof by duality (Section 6.5) and proof by reflection (Chapter 3)).

This is a sorry state of affairs: we are effectively advising users to basically avoid using most of the power and infrastructure of the proof assistant.

We admit that we are not sure what an effective resolution to the performance issue of computation at the type level would look like. While Chapter 6 lays out in Section 6.2 (When and How To Use Dependent Types Painlessly) principles for how and when to use dependent types that allow us to recover much of the power of dependent types without running into issues of slow conversion, even at scale, this is nowhere near a complete roadmap for actually using partial computation at the type level.

On the question of using tactics, however, we do know what a resolution would look like, and hence we conclude this dissertation with such a call for future research.

As far as we can tell, no one has yet laid out a theory of what are the necessary basic building blocks of a usable tactic engine for proofs. Such a theory should include:

- a list of basic operations
- with necessary asymptotic performance,
- justification that these building blocks are sufficient for constructing all the proof automation users might want to construct, and
- justification that the asymptotic performance does not incur needless overhead above and beyond the underlying algorithm of proof construction.

What is *needless* overhead, though? How can we say what the performance of the "underlying algorithm" is?

A first stab might be thus: we want a proof engine which, for any heuristic algorithm A that can sometimes determine the truth of a theorem statement (and will otherwise answer "I don't know") in time $\mathcal{O}(f(n))$, where n is some parameter controlling the size of the problem, we can construct a proof script which generates proofs of these theorem statements in time not worse than $\mathcal{O}(f(n))$, or perhaps in time that is not much worse than $\mathcal{O}(f(n))$.

This criterion, however, is both useless and impossible to meet.

Useless: In a dependently typed proof assistant, if we can prove that A is sound, i.e., that when it says "yes" the theorem is in fact true, then we can simply use reflection to create a proof by appeal to computation. This is not useful when what we are trying to do is describe how to identify a proof engine which gives adequate building blocks *aside* from appeal to computation.

Impossible to meet: Moreover, even if we could modify this criterion into a useful one, perhaps by requiring that it be possible to construct such a proof script without any appeal to computation, meeting the criterion would still be impossible. Taking inspiration from Garrabrant et al. [Gar+16, pp. 24–25], we ask the reader to consider

a program $\texttt{prg}(x)$ which searches for proofs of absurdity (i.e., `False`) in Coq which have length less than 2^x characters and which can be checked by Coq's kernel in less than 2^x CPU cycles. If such a proof of absurdity is found, the program outputs `true`. If no such proof is found under the given computational limits, the program outputs `false`. Assuming that Coq is, in fact, consistent, then we can recognize true theorems of the form $\texttt{prg}(x) = \texttt{false}$ for all x in time $\mathcal{O}(\log x)$. (The running time is logarithmic, rather than linear or constant, because representing the number x in any place-value system, such as decimal or binary, requires $\log n$ space.) At the same time, by Gödel's incompleteness theorem, there is no hope of proving $\forall x, \texttt{prg}(x) = \texttt{false}$, and hence we cannot prove this simple $\mathcal{O}(\log x)$-time theorem recognizer correct. We almost certainly will be stuck running the program, which will take time $\Omega(2^x)$, which is certainly not an acceptable overhead over $\mathcal{O}(\log x)$.

We do not believe that all hope is lost, though! Gödelian incompleteness did not prove to be a fatal obstacle to verification and automation of proofs, as we saw in Section 1.1, and we hope that it proves to be surmountable here as well.

We can take a second stab at specifying what it might mean to avoid needless overhead: Suppose we are given some algorithm A which can sometimes determine the truth of a theorem statement (and will otherwise answer "I don't know") in time $\mathcal{O}(f(n))$, and suppose we are given a proof that A is sound, i.e., a proof that whenever A claims a theorem statement is true, that statement is in fact true. Then we would like a proof engine which permits the construction of proofs, without any appeal to computation, of theorems that A claims are true in time $\mathcal{O}(f(n))$, or perhaps time that is not much worse than $\mathcal{O}(f(n))$. Said another way, we want a proof engine for which reflective proof scripts can be turned into nonreflective proof scripts without incurring overhead, or at least without incurring too much overhead.

Is such a proof engine possible? Is such a proof engine sufficient? Is this criterion necessary? Or is there perhaps a better criterion? We leave all of these questions for future work in this field, noting that there may be some inspiration to be drawn from the extant research on the overhead of using a functional language over an imperative one [Cam10; BG92; Ben96; BJD97; Oka96; Oka98; Pip97]. This body of work shows that we can always turn an imperative program into a strict functional program with at most $\mathcal{O}(\log n)$ overhead, and often we get no overhead at all.[1]

We hope the reader leaves this dissertation with an improved understanding of the performance landscape of engineering of proof-based software systems and perhaps goes on to contribute new insight to this nascent field themselves.

[1]Note that if we are targeting a lazy functional language rather than a strict one, it may in fact always be possible to achieve a transformation without any overhead [Cam10].

Bibliography

[Acz93] Peter Aczel. "Galois: a theory development project". In: (1993). URL: http://www.cs.man.ac.uk/~petera/galois.ps.gz.

[Age95] Sten Agerholm. "Experiments in Formalizing Basic Category Theory in Higher Order Logic and Set Theory". In: *Draft manuscript* (Dec. 1995). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.8437&rep=rep1&type=pdf.

[AGN95] Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. *The Bologna Optimal Higher-order Machine*. Technical report. University of Bologna, Mar. 1995. DOI: 10.1017/s0956796800001994. URL: https://pdfs.semanticscholar.org/3517/03af066fd2e65ad64c63108672d960b9d8fb.pdf.

[AHN08] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. "A Compiled Implementation of Normalization by Evaluation". In: *Proc. TPHOLs*. 2008. DOI: 10.1007/978-3-540-71067-7_8.

[Ahra] Benedikt Ahrens. *benediktahrens/Foundations typesystems*. URL: https://github.com/benediktahrens/Foundations/tree/typesystems.

[Ahrb] Benedikt Ahrens. *Coinductives*. URL: https://github.com/benediktahrens/coinductives.

[Ahr10] Benedikt Ahrens. *Categorical semantics of programming languages (in COQ)*. 2010. URL: http://math.unice.fr/~ahrens/edsfa/ahrens_edsfa.pdf.

[AKS] Benedikt Ahrens, Chris Kapulkin, and Michael Shulman. *benediktahrens/rezk_completion*. URL: https://github.com/benediktahrens/rezk_completion.

[AKS13] Benedikt Ahrens, Chris Kapulkin, and Michael Shulman. "Univalent categories and the Rezk completion". In: *ArXiv e-prints* (Mar. 2013). DOI: 10.1017/s0960129514000486. arXiv: 1303.0584 [math.CT].

[AL94] Andrea Asperti and Cosimo Laneve. "Interaction Systems I: The theory of optimal reductions". In: *Mathematical Structures in Computer Science* 4.4 (1994), pages 457–504. DOI: 10.1017/s0960129500000566. URL: https://hal.inria.fr/docs/00/07/69/88/PDF/RR-1748.pdf.

[AM06] Thorsten Altenkirch and Conor McBride. "Towards observational type theory". In: *Manuscript, available online* (2006). URL: http://www.strictlypositive.org/ott.pdf.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM. 2007, pages 57–68. DOI: 10.1145/1292597.1292608. URL: http://www.strictlypositive.org/obseqnow.pdf.

[Ana+17] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Matthew Weaver, Matthieu Sozeau, Olivier Savary Belanger, Randy Pollack, and Zoe Paraskevopoulou. "CertiCoq: A verified compiler for Coq". In: *CoqPL*. 2017. URL: http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.

[Ana+18] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. "Towards Certified Meta-Programming with Typed Template-Coq". In: *Proc. ITP*. 2018. DOI: 10.1007/978-3-319-94821-8_2.

[AP90] James A. Altucher and Prakash Panangaden. "A mechanically assisted constructive proof in category theory". In: *10th International Conference on Automated Deduction*. Springer. 1990, pages 500–513. DOI: 10.1007/3-540-52885-7_110.

[AR14] Abhishek Anand and Vincent Rahli. "Towards a Formally Verified Proof Assistant". In: *Interactive Theorem Proving*. Edited by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pages 27–44. ISBN: 978-3-319-08970-6. URL: http://www.nuprl.org/html/Nuprl2Coq/verification.pdf.

[AR17] Nada Amin and Tiark Rompf. "LMS-Verify: Abstraction without Regret for Verified Systems Programming". In: *Proc. POPL*. 2017. DOI: 10.1145/3093333.3009867.

[Arm+10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. "Extending Coq with Imperative Features and Its Application to SAT Verification". In: *Interactive Theorem Proving*. Edited by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer, 2010, pages 83–98. ISBN: 978-3-642-14052-5. DOI: 10.1007/978-3-642-14052-5_8. URL: https://hal.inria.fr/inria-00502496v2/document.

[Asp+07] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. "User Interaction with the Matita Proof Assistant". In: *Journal of Automated Reasoning* 39.2 (2007), pages 109–139.

[Asp+11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. "The Matita Interactive Theorem Prover". In: *Automated Deduction – CADE-23*. Edited by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer, 2011, pages 64–69. ISBN: 978-3-642-22438-6. DOI: 10.1007/978-3-642-22438-6_7.

[Asp95] Andrea Asperti. "$\delta o!\epsilon$=1 Optimizing optimal λ-calculus implementations". In: *International Conference on Rewriting Techniques and Applications*. Edited by Jieh Hsiang. Springer. Berlin, Heidelberg: Springer, 1995, pages 102–116. ISBN: 978-3-540-49223-8.

[Awo] Steve Awodey. *Category theory*. Second Edition. Oxford University Press. DOI: 10.1093/acprof:oso/9780198568612.001.0001.

[Ayd+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. "Mechanized Metatheory for the Masses: The POPLMark Challenge". In: *Theorem Proving in Higher Order Logics*. Edited by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer, 2005, pages 50–65. ISBN: 978-3-540-31820-0. DOI: 10.1007/11541868_4.

[Ayd+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. "Engineering Formal Metatheory". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pages 3–15. ISBN: 9781595936899. DOI: 10.1145/1328438.1328443. URL: https://www.cis.upenn.edu/~sweirich/papers/popl08-binders.pdf.

[Bar00] Bruno Barras. "Programming and Computing in HOL". In: *Theorem Proving in Higher Order Logics*. Edited by Mark Aagaard and John Harrison. Berlin, Heidelberg: Springer, 2000, pages 17–37. ISBN: 978-3-540-44659-0. DOI: 10.1007/3-540-44659-1_2.

[Bau13] Andrej Bauer. *Is rigour just a ritual that most mathematicians wish to get rid of if they could?* Version 2013-05-08. May 8, 2013. eprint: https://mathoverflow.net/q/130125. URL: https://mathoverflow.net/q/130125.

[BB08] Bruno Barras and Bruno Bernardo. "The implicit calculus of constructions as a programming language with dependent types". In: *FoSSaCS*. 2008. DOI: 10.1007/978-3-540-78499-9_26. URL: http://hal.archives-ouvertes.fr/docs/00/43/26/58/PDF/icc_barras_bernardo-tpr07.pdf.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *Proc. CPP*. 2011. DOI: 10.1007/978-3-642-25379-9_26.

[Ben+12] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. "Strongly Typed Term Representations in Coq". In: *Journal of Automated Reasoning* 49.2 (2012), pages 141–159. ISSN: 1573-0670. DOI: 10.1007/s10817-011-9219-0. URL: https://sf.snu.ac.kr/publications/typedsyntaxfull.pdf.

[Ben89] Dan Benanav. "Recognizing Unnecessary Inference". In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'89. Detroit, Michigan: Morgan Kaufmann Publishers Inc., 1989, pages 366–371.

[Ben96] Amir M. Ben-Amram. "Notes on Pippenger's Comparison of Pure and Impure LISP". DIKU, University of Copenhagen, Denmark, 1996. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.3024.

[Ber13] Yves Bertot. *Private Inductive Types: Proposing a language extension*. Apr. 2013. URL: http://coq.inria.fr/files/coq5_submission_3.pdf.

[BG01] Henk P. Barendregt and Herman Geuvers. "Proof-Assistants using Dependent Type Systems". In: *Handbook of Automated Reasoning*. Edited by Alan Robinson and Andrei Voronkov. NLD: Elsevier Science Publishers B. V., 2001, pages 1149–1238. ISBN: 0444508120. DOI: 10.1016/b978-044450813-3/50020-5.

[BG05] Bruno Barras and Benjamin Grégoire. "On the Role of Type Decorations in the Calculus of Inductive Constructions". In: *Computer Science Logic*. Edited by Luke Ong. Berlin, Heidelberg: Springer, 2005, pages 151–166. ISBN: 978-3-540-31897-2. DOI: 10.1007/11538363_12.

[BG92] Amir M. Ben-Amram and Zvi Galil. "On Pointers versus Addresses". In: *Journal of the ACM* 39.3 (July 1992), pages 617–648. DOI: 10.1145/146637.146666. URL: https://www2.mta.ac.il/~amirben/downloadable/jacm.ps.gz.

[BH18] Sara Baase and Timothy Henry. *A Gift of Fire: Social, Legal, and Ethical Issues for Computing Technology*. 5th edition. Pearson, 2018. ISBN: 9780134615271. URL: https://books.google.com/books?id=izaqAQAACAAJ.

[Bis67] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1967. DOI: 10.2307/2314383. URL: http://books.google.com/books?id=o2mmAAAAIAAJ.

[BJD97] Richard Bird, Geraint Jones, and Oege De Moor. "More Haste, Less Speed: Lazy versus Eager Evaluation". In: *Journal of Functional Programming* 7.5 (Sept. 1997), pages 541–547. ISSN: 0956-7968. DOI: 10.1017/S0956796897002827.

[BMM03] Edwin Brady, Conor McBride, and James McKinna. "Inductive Families Need Not Store Their Indices". In: *International Workshop on Types for Proofs and Programs*. Springer. 2003, pages 115–129. DOI: 10.1007/978-3-540-24849-1_8. URL: https://eb.host.cs.st-andrews.ac.uk/writings/types2003.pdf.

[Bon+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, Rustan Leino, Jay Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code". In: *Proc. USENIX Security*. 2017. URL: http://www.cs.cornell.edu/~laurejt/papers/vale-2017.pdf.

[Bou94] Richard Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. Computer Laboratory Cambridge: Technical report. University of Cambridge, Computer Laboratory, Nov. 1994. URL: https://books.google.com/books?id=7DAkAQAAIAAJ.

[Bou97] Samuel Boutin. "Using reflection to build efficient and certified decision procedures". In: *Theoretical Aspects of Computer Software*. Edited by Martín Abadi and Takayasu Ito. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pages 515–529. ISBN: 978-3-540-69530-1. DOI: 10.1007/bfb0014565.

[Boy07] Robert S. Boyer. *Nqthm, the Boyer-Moore prover*. May 13, 2007. URL: https://www.cs.utexas.edu/users/boyer/ftp/nqthm/.

[BP99] Richard S. Bird and Ross Paterson. "de Bruijn notation as a nested datatype". In: *Journal of Functional Programming* 9.1 (1999), pages 77–91. DOI: 10.1017/S0956796899003366. URL: http://www.cs.ox.ac.uk/people/richard.bird/online/BirdPaterson99DeBruijn.pdf.

[BR70] *Software Engineering Techniques. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th–11th October 1968*. Rome, Italy, Apr. 1970. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[Bra13] Edwin Brady. "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation". In: *Journal of Functional Programming* 23.5 (2013), pages 552–593. DOI: 10.1017/S095679681300018X. URL: https://eb.host.cs.st-andrews.ac.uk/drafts/impldtp.pdf.

[Bra20] Edwin Brady. *Why is Idris 2 so much faster than Idris 1?* May 20, 2020. URL: https://www.type-driven.org.uk/edwinb/why-is-idris-2-so-much-faster-than-idris-1.html.

[Bru70] N. G. de Bruijn. "The mathematical language AUTOMATH, its usage, and some of its extensions". In: *Symposium on Automatic Demonstration*. Edited by M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger. Berlin, Heidelberg: Springer, 1970, pages 29–61. ISBN: 978-3-540-36262-3. DOI: 10.1007/bfb0060623.

[Bru72] 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. 5. Elsevier. 1972, pages 381–392. DOI: 10.1016/1385-7258(72)90034-0. URL: http://www.sciencedirect.com/science/article/pii/1385725872900340.

[Bru94] N. G. de Bruijn. “A Survey of the Project Automath”. In: *Selected Papers on Automath*. Edited by R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. Volume 133. Studies in Logic and the Foundations of Mathematics. Elsevier, 1994, pages 141–161. DOI: 10.1016/S0049-237X(08)70203-9. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08702039.

[BS91] U. Berger and H. Schwichtenberg. “An inverse of the evaluation functional for typed λ–calculus”. In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. July 1991, pages 203–211. DOI: 10.1109/LICS.1991.151645. URL: http://www.mathematik.uni-muenchen.de/~schwicht/papers/lics91/paper.pdf.

[BW05] Henk Barendregt and Freek Wiedijk. “The challenge of computer mathematics”. In: *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 363.1835 (Oct. 12, 2005), pages 2351–2375. ISSN: 1364-503X. DOI: 10.1098/rsta.2005.1650.

[Cam10] Brian Campbell. *Efficiency of purely functional programming*. May 23, 2010. eprint: https://stackoverflow.com/a/1990580. URL: https://stackoverflow.com/a/1990580.

[Cap] Paolo Capriotti. *pcapriotti/agda-categories*. URL: https://github.com/pcapriotti/agda-categories/.

[CF07] Sylvain Conchon and Jean-Christophe Filliâtre. “A Persistent Union-Find Data Structure”. In: *Proceedings of the 2007 Workshop on Workshop on ML*. ML ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pages 37–46. ISBN: 9781595936769. DOI: 10.1145/1292535.1292541. URL: https://www.lri.fr/~filliatr/puf/.

[CH88] Thierry Coquand and Gérard Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2 (1988), pages 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3. URL: https://hal.inria.fr/inria-00076024/document.

[Cha] James Chapman. *jmchapman/restriction-categories*. URL: https://github.com/jmchapman/restriction-categories.

[Cha12] Arthur Charguéraud. “The Locally Nameless Representation”. English. In: *Journal of Automated Reasoning* 49.3 (Oct. 2012), pages 363–408. DOI: 10.1007/s10817-011-9225-2. URL: https://www.chargueraud.org/research/2009/ln/main.pdf.

[Chl08] Adam Chlipala. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics". In: *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, British Columbia, Canada, Sept. 2008. DOI: 10.1145/1411204.1411226. URL: http://adam.chlipala.net/papers/PhoasICFP08/.

[Chl10] Adam Chlipala. "A Verified Compiler for an Impure Functional Language". In: *POPL'10: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Madrid, Spain, Jan. 2010. URL: http://adam.chlipala.net/papers/ImpurePOPL10/.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, Dec. 2013. ISBN: 9780262026659. DOI: 10.7551/mitpress/9153.001.0001. URL: http://adam.chlipala.net/cpdt/.

[Chl15] Adam Chlipala. "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pages 609–622. ISBN: 9781450333009. DOI: 10.1145/2676726.2677003. URL: http://adam.chlipala.net/papers/BedrockPOPL15/.

[Chu36] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pages 345–363. ISSN: 00029327, 10806377. DOI: 10.2307/2371045. URL: http://www.jstor.org/stable/2371045.

[CM98] Alexandra Carvalho and Paulo Mateus. *Category Theory in Coq*. Technical report. 1049-001 Lisboa, Portugal, 1998. URL: http://sqig.math.ist.utl.pt/pub/CarvalhoA/98-C-DiplomaThesis/maintext.ps.

[Coh+18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Edited by Tarmo Uustalu. Volume 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5. arXiv: 1611.02108v1 [cs.LO].

[Con+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Dec. 1986. ISBN: 9780134518329. URL: http://www.nuprl.org/book/.

[Coqa] Coq Developers. *Constrextern (coq.Constrextern)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Constrextern/index.html.

[Coqb] Coq Developers. *Constrintern (coq.Constrintern)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Constrintern/index.html.

[Coqc] Coq Developers. *Detyping (coq.Detyping)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Detyping/index.html.

[Coqd] Coq Developers. *Ppconstr (coq.Ppconstr)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Ppconstr/index.html.

[Coqe] Coq Developers. *Pretyping (coq.Pretyping)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Pretyping/index.html.

[Coq17a] Coq Development Team. "The Coq Proof Assistant Reference Manual". In: 8.7.1. INRIA, 2017. Chapter 2.1.1 Extensions of Gallina, Record Types (Primitive Projections). URL: https://coq.inria.fr/distrib/V8.7.1/refman/gallina-ext.html#sec65.

[Coq17b] Coq Development Team. "The Coq Proof Assistant Reference Manual". In: 8.7.1. INRIA, 2017. Chapter 10.3 Detailed examples of tactics (quote). URL: https://coq.inria.fr/distrib/V8.7.1/refman/tactic-examples.html#quote-examples.

[Coq20] The Coq Development Team. *The Coq Proof Assistant*. Version 8.12.0. INRIA. July 2020. URL: https://coq.inria.fr/.

[CP88] Thierry Coquand and Christine Paulin. "Inductively Defined Types". In: *International Conference on Computer Logic*. Springer. 1988, pages 50–66. DOI: 10.1007/3-540-52335-9_47.

[CPG17] Ahmet Celik, Karl Palmskog, and Milos Gligoric. "iCoq: Regression Proof Selection for Large-Scale Verification Projects". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pages 171–182. DOI: 10.1109/ase.2017.8115630. URL: https://users.ece.utexas.edu/~gligoric/papers/CelikETAL17iCoq.pdf.

[CS13] Alberto Ciaffaglione and Ivan Scagnetto. "A weak HOAS approach to the POPLmark Challenge". In: *EPTCS 113, 2013, pp. 109-124* (Mar. 29, 2013). DOI: 10.4204/EPTCS.113.11. arXiv: 1303.7332v1 [cs.LO].

[CW01] Mario Jose Cáccamo and Glynn Winskel. "A Higher-Order Calculus for Categories". English. In: *Theorem Proving in Higher Order Logics*. Edited by Richard J. Boulton and Paul B. Jackson. Volume 2152. Lecture Notes in Computer Science. Springer Berlin Heidelberg, June 2001, pages 136–153. ISBN: 978-3-540-42525-0. DOI: 10.1007/3-540-44755-5_11. URL: ftp://ftp.daimi.au.dk/BRICS/Reports/RS/01/27/BRICS-RS-01-27.pdf.

[Dar19] Ashish Darbari. *A Brief History of Formal Verification*. EEWeb. Mar. 8, 2019. URL: https://www.eeweb.com/profile/adarbari/articles/a-brief-history-of-formal-verification.

[Dav01] Martin Davis. "The Early History of Automated Deduction. Dedicated to the memory of Hao Wang". In: *Handbook of Automated Reasoning*. Elsevier, 2001, pages 3–15. URL: http://cs.nyu.edu/cs/faculty/davism/early.ps.

[Del+15] Ben Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. Jan. 2015. DOI: 10.1145/2676726.2677006. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2015-adt-synthesis.pdf.

[Dén13a] Maxime Dénès. *New implementation of the conversion test, using normalization by evaluation to native OCaml code · coq/coq@6b908b5*. Jan. 22, 2013. URL: https://github.com/coq/coq/commit/6b908b5185a55a27a82c2b0fce47138

[Dén13b] Maxime Dénès. "Towards primitive data types for COQ. 63-bits integers and persistent arrays". In: *The Coq Workshop 2013*. Apr. 6, 2013. URL: https://coq.inria.fr/files/coq5_submission_2.pdf.

[Dén18] Maxime Dénès. *Remove quote plugin by maximedenes · Pull Request #7894 · coq/coq*. Sept. 14, 2018. URL: https://github.com/coq/coq/pull/7894.

[Dén20a] Maxime Dénès. *Comment by maximedenes on Primitive integers by maximedenes · Pull Request #6914 · coq/coq*. June 5, 2020. URL: https://github.com/coq/coq/pull/11604%5C#issuecomment-639566223.

[Dén20b] Maxime Dénès. *Primitive persistent arrays by maximedenes · Pull Request #11604 · coq/coq*. Feb. 14, 2020. URL: https://github.com/coq/coq/pull/11604.

[Des17] Jeff Desjardins. *How Many Millions of Lines of Code Does It Take?* Visual Capitalist. Feb. 8, 2017. URL: https://www.visualcapitalist.com/millions-lines-of-code/ (visited on 11/08/2020).

[DG18] Maxime Dénès and Benjamin Grégoire. *Primitive integers by maximedenes · Pull Request #6914 · coq/coq*. Mar. 5, 2018. URL: https://github.com/coq/coq/pull/6914.

[Dow97] Mark Dowson. "The Ariane 5 Software Failure". In: *ACM SIGSOFT Software Engineering Notes* 22.2 (Mar. 1997), page 84. ISSN: 0163-5948. DOI: 10.1145/251880.251992.

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3 (July 1960), pages 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[Dyc85] Roy Dyckhoff. *Category Theory as an Extension of Martin-Löf Type Theory*. Technical report. 1985. URL: http://rd.host.cs.st-andrews.ac.uk/publications/CTMLTT.pdf.

[Ebn+17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. "A Metaprogramming Framework for Formal Verification". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pages 1–29. DOI: 10.1145/3110278. URL: https://leanprover.github.io/papers/tactic.pdf.

[Erb+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises". In: *IEEE Security & Privacy*. San Francisco, CA, USA, May 2019. DOI: 10.1109/sp.2019.00005. URL: http://adam.chlipala.net/papers/FiatCryptoSP19/.

[Fai+16] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. "Coqoon. An IDE for Interactive Proof Development in Coq". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer, 2016, pages 316–331. ISBN: 978-3-662-49674-9. DOI: 10.1007/s10009-017-0457-2. URL: https://hal.inria.fr/hal-01242295/document.

[Fre17] Free Software Foundation. *The C Preprocessor: Implementation limits*. 2017. URL: https://gcc.gnu.org/onlinedocs/gcc-7.5.0/cpp/Implementation-limits.html.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. *The geometry of optimal lambda reduction*. 1992. DOI: 10.1145/143165.143172.

[Gar+09a] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-03359-9_23. URL: http://hal.inria.fr/docs/00/36/84/03/PDF/main.pdf.

[Gar+09b] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Volume 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pages 327–342. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_23. URL: https://hal.inria.fr/inria-00368403.

[Gar+16] Scott Garrabrant, Tsvi Benson-Tilsen, Andrew Critch, Nate Soares, and Jessica Taylor. *Logical Induction*. Sept. 12, 2016. arXiv: 1609.03543 [cs.AI].

[Gaw14] Nicole Gawron. *Infamous Software Bugs: FDIV Bug*. Aug. 23, 2014. URL: https://www.olenick.com/blog/articles/infamous-software-bugs-fdiv-bug.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pages 57–76. ISBN: 9781595937865. DOI: 10.1145/1297027.1297033.

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. "Experience Implementing a Performant Category-Theory Library in Coq". In: *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP'14)*. July 2014. DOI: 10.1007/978-3-319-08970-6_18. eprint: 1401.7694. URL: https://people.csail.mit.edu/jgross/personal-website/papers/category-coq-experience-itp-submission-final.pdf.

[GEC18] Jason Gross, Andres Erbsen, and Adam Chlipala. "Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac". In: *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP'18)*. July 2018. DOI: 10.1007/978-3-319-94821-8_17. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2018-reification-by-parametricity-itp-camera-ready.pdf.

[Geu09] Herman Geuvers. "Proof assistants: History, ideas and future". In: *Sādhanā* 34.1 (2009), pages 3–25. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0001-5. URL: https://www.ias.ac.in/article/fulltext/sadh/034/01/0003-0025.

[GHR07] Hermann Gruber, Markus Holzer, and Oliver Ruepp. "Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms". In: *Fun with Algorithms*. Edited by Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci. Berlin, Heidelberg: Springer, 2007, pages 183–197. ISBN: 978-3-540-72914-3. DOI: 10.1007/978-3-540-72914-3_17. URL: http://www.hermann-gruber.com/pdf/fun07-final.pdf.

[Gil+19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. "Definitional Proof-Irrelevance without K". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019). DOI: 10.1145/3290316.

[Gil18] Gaëtan Gilbert. *SProp: the definitionally proof irrelevant universe by SkySkimmer · Pull Request #8817 · coq/coq*. Oct. 25, 2018. URL: https://github.com/coq/coq/pull/8817.

[GL02] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *ICFP 2002: International Conference on Functional Programming*. ACM, 2002, pages 235–246. DOI: 10.1145/581478.581501.

[GM05] Benjamin Grégoire and Assia Mahboubi. "Proving Equalities in a Commutative Ring Done Right in Coq". In: *Theorem Proving in Higher Order Logics*. Edited by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer, 2005, pages 98–113. ISBN: 978-3-540-31820-0. DOI: 10.1007/11541868_7.

[GMT16] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Technical report. Inria Saclay Ile de France, Nov. 2016. URL: https://hal.inria.fr/inria-00258384/.

[GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. "Edinburgh LCF: A Mechanized Logic of Computation". In: *Springer-Verlag Berlin* 10 (1979), pages 11–25.

[Gon+11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. "How to Make Ad Hoc Proof Automation Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: Association for Computing Machinery, 2011, pages 163–175. ISBN: 9781450308656. DOI: 10.1145/2034773.2034798. URL: http://www.mpi-sws.org/~beta/lessadhoc/lessadhoc.pdf.

[Gon+13a] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *International Conference on Interactive Theorem Proving*. Springer. 2013, pages 163–179. DOI: 10.1007/978-3-642-39634-2_14. eprint: hal-00816699. URL: https://hal.inria.fr/file/index/docid/816699/filename/main.pdf.

[Gon+13b] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. "How to Make Ad Hoc Proof Automation Less Ad Hoc". In: *Journal of Functional Programming* 23.4 (2013), pages 357–401. DOI: 10.1017/S0956796813000051. URL: https://people.mpi-sws.org/~beta/lessadhoc/lessadhoc-extended.pdf.

[Gon08] Georges Gonthier. "Formal Proof–The Four-Color Theorem". In: *Notices of the AMS* 55.11 (2008), pages 1382–1393. URL: https://www.ams.org/notices/200811/tx081101382p.pdf.

[Gor+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. "A Metalanguage for Interactive Proof in LCF". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1978, pages 119–130.

[Gor00] Michael John Caldwell Gordon. "From LCF to HOL: A Short History". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pages 169–185. ISBN: 0262161885. URL: https://www.cl.cam.ac.uk/archive/mjcg/papers/HolHistory.pdf.

[Gor15] Michael John Caldwell Gordon. "Tactics for mechanized reasoning: a commentary on Milner (1984) 'The use of machines to assist in rigorous proof"'. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373 (2039 Apr. 13, 2015). ISSN: 1471-2962. DOI: 10.1098/rsta.2014.0234.

[Gra18] Daniel R. Grayson. "An introduction to univalent foundations for mathematicians". In: *Bulletin of the American Mathematical Society* (55 Mar. 5, 2018), pages 427–450. ISSN: 1088-9485. DOI: 10.1090/bull/1616.

[Gro14] Jason Gross. *Formalizations of category theory in proof assistants*. MathOverflow. Jan. 19, 2014. URL: http://mathoverflow.net/questions/152497/formalizations-of-category-theory-in-proof-assistants.

[Gro15a] Jason Gross. "An Extensible Framework for Synthesizing Efficient, Verified Parsers". Master's thesis. Massachusetts Institute of Technology, Sept. 2015. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2015-jgross-thesis.pdf.

[Gro15b] Jason Gross. *Coq Bug Minimizer*. Presented at The First International Workshop on Coq for PL (CoqPL'15). Jan. 2015. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2015-coq-bug-minimizer.pdf.

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pages 595–608. ISBN: 9781450333009. DOI: 10.1145/2676726.2676975. URL: https://flint.cs.yale.edu/flint/publications/dscal.pdf.

[Güç18] Osman Gazi Güçlütürk. *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*. Medium. Aug. 1, 2018. URL: https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562.

[Hal+14] Thomas Hales, Alexey Solovyevand, Hoang Le Truong, and the Flyspeck Team. *flyspeck - AnnouncingCompletion.wiki*. Aug. 10, 2014. URL: https://code.google.com/archive/p/flyspeck/wikis/AnnouncingCompletion.wiki.

[Hal06] Thomas C. Hales. "Introduction to the Flyspeck Project". In: *Mathematics, Algorithms, Proofs*. Edited by Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy. Dagstuhl Seminar Proceedings 05021. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. URL: http://drops.dagstuhl.de/opus/volltexte/2006/432.

[Hal95] Tom R. Halfhill. "The Truth Behind the Pentium Bug". In: *BYTE* (Mar. 1995). URL: https://web.archive.org/web/20060209005434/http://www.byte.com/art/9503/sec13/art1.htm.

[Har01] John Harrison. *The LCF Approach to Theorem Proving*. Intel Corporation. Sept. 12, 2001. URL: https://www.cl.cam.ac.uk/~jrh13/slides/manchester-12sep01/slides.pdf.

[Har96a] John Harrison. "A Mizar Mode for HOL". In: *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*. Edited by Joakim von Wright, Jim Grundy, and John Harrison. Volume 1125. Lecture Notes in Computer Science. Turku, Finland: Springer-Verlag, Aug. 1996, pages 203–220. URL: https://www.cl.cam.ac.uk/~jrh13/papers/mizar.html.

[Har96b] John Harrison. *Formalized mathematics*. TUCS technical report. Turku Centre for Computer Science, 1996. ISBN: 9789516508132. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.8842&rep=rep1&type=pdf.

[Har96c] John Harrison. "HOL Light: A Tutorial Introduction". In: *Lecture Notes in Computer Science* (1996), pages 265–269.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A Framework for Defining Logics". In: *JACM* (1993).

[HM12] André Hirschowitz and Marco Maggesi. "Nested Abstract Syntax in Coq". In: *Journal of Automated Reasoning* 49.3 (Oct. 1, 2012), pages 409–426. ISSN: 1573-0670. DOI: 10.1007/s10817-010-9207-9. URL: https://math.unice.fr/~ah/div/fsubwf.pdf.

[HN07] Florian Haftmann and Tobias Nipkow. "A Code Generator Framework for Isabelle/HOL". In: *Proc. TPHOLs*. 2007.

[HoT20] HoTT Library Authors. *HoTT/HoTT Categories*. Feb. 23, 2020. URL: https://github.com/HoTT/HoTT/tree/V8.12/theories/Categories.

[HP03] Gérard Huet and Christine Paulin-Mohring. "Forward". In: Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, Nov. 2003, pages IX–XVI. ISBN: 9783662079645. URL: https://books.google.com/books?id=FeklBQAAQBAJ.

[HP91] Robert Harper and Robert Pollack. "Type checking with universes". In: *Theoretical Computer Science* 89.1 (1991), pages 107–136. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90108-T. URL: http://www.sciencedirect.com/science/article/pii/030439759090108T.

[HS00] Gérard Huet and Amokrane Saïbi. “Constructive category theory”. In: *Proof, language, and interaction*. MIT Press. 2000, pages 239–275. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.4193.

[HS98] Martin Hofmann and Thomas Streicher. “The groupoid interpretation of type theory”. In: *Twenty-five years of constructive type theory (Venice, 1995)* 36 (Aug. 1998), pages 83–111. URL: http://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/agroupoidinterpretationoftypetheroy.pdf.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. “History of Interactive Theorem Proving”. In: *Computational Logic*. 2014, pages 135–214. DOI: 10.1016/b978-0-444-51624-4.50004-6. URL: https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf.

[ID16] Andrew David Irvine and Harry Deutsch. “Russell’s Paradox”. In: *The Stanford Encyclopedia of Philosophy*. Edited by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016. URL: https://plato.stanford.edu/archives/win2016/entries/russell-paradox/.

[Ish] Hiromi Ishii. *konn/category-agda*. URL: https://github.com/konn/category-agda.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993. ISBN: 0-13-020249-5.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. “Design by Contract: The Lessons of Ariane”. In: *Computer* 30.1 (Jan. 1997). Edited by Bertrand Meyer, pages 129–130. ISSN: 1558-0814. DOI: 10.1109/2.562936. URL: http://se.ethz.ch/~meyer/publications/computer/ariane.pdf.

[Kai+18] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. “Mtac2: Typed Tactics for Backward Reasoning in Coq”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236773.

[Kam02] Fairouz Kamareddine. *Thirty Five years of Automath*. Apr. 2002. URL: http://www.cedar-forest.org/forest/events/automath2002/.

[Kat10] Alexander Katovsky. “Category Theory”. In: *Archive of Formal Proofs* (June 2010). http://afp.sf.net/entries/Category2.shtml, Formal proof development. ISSN: 2150-914x.

[KKL] Vladimir Komendantsky, Alexander Konovalov, and Steve Linton. *Connecting Coq theorem prover to GAP*. SCIEnce/CICM’10; University of St Andrews, UK. URL: http://www.symcomp.org/sciencehome-view/images/e/e9/CICM_2010_Komendantsky.pdf.

[KKR06] Dexter Kozen, Christoph Kreitz, and Eva Richter. "Automating Proofs in Category Theory". In: *Automated Reasoning*. Springer, 2006, pages 392–407. DOI: 10.1007/11814771_34. URL: http://www.cs.uni-potsdam.de/ti/kreitz/PDF/06ijcar-categories.pdf.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proc. SOSP*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pages 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596.

[Kle11] Dirk Kleeblatt. "On a Strongly Normalizing STG Machine. With an Application to Dependent Type Checking". PhD thesis. Technischen Universität Berlin, 2011. URL: https://depositonce.tu-berlin.de/bitstream/11303/3095/1/Dokument_9.pdf.

[KM20a] Matt Kaufmann and J. Strother Moore. *ACL2 User's Manual — Interesting-applications*. 2020. URL: https://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/index.html?topic=ACL2____INTERESTING-APPLICATIONS.

[KM20b] Matt Kaufmann and J. Strother Moore. *ACL2 Version 8.3*. Apr. 14, 2020. URL: https://www.cs.utexas.edu/users/moore/acl2/.

[Knu74a] Donald E. Knuth. "Computer Programming as an Art". In: *Communications of the ACM* 17.12 (Dec. 1974), pages 667–673. ISSN: 0001-0782. DOI: 10.1145/361604.361612. URL: http://www.paulgraham.com/knuth.html.

[Knu74b] Donald E. Knuth. "Structured Programming with *go to* Statements". In: *ACM Computing Surveys* 6.4 (Dec. 1974), pages 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[Kova] András Kovács. *Non-deterministic normalization-by-evaluation in Olle Fredriksson's flavor*. URL: https://gist.github.com/AndrasKovacs/a0e0938113b193d6b9c1c0620d853784.

[Kovb] András Kovács. *Normalization Bench*. URL: https://github.com/AndrasKovacs/normalization-bench.

[Kovc] András Kovács. *smalltt: Demo for high-performance type theory elaboration*. URL: https://github.com/AndrasKovacs/smalltt.

[Kov18] András Kovács. *Sharing-Preserving Elaboration with Precisely Scoped Metavariables*. Agda Implementors' Meeting XXVI, Jan. 2018. URL: https://github.com/AndrasKovacs/elaboration-zoo/blob/0c7f8a676c/AIMprez/AIMprez.pdf.

[Kov19a] András Kovács. *Fast Elaboration for Dependent Type Theories*. EUTypes WG Meeting, Krakow, Feb. 24, 2019. URL: https://github.com/AndrasKovacs/smalltt/blob/fb56723b098cb1a95e8a5f3f9f5fce30bbcc67da/krakow-pres.pdf.

[Kov19b] András Kovács. *Online reference book for *implementing* concepts in type theory*. Dec. 10, 2019. URL: https://math.stackexchange.com/a/3468022/22982.

[KSW] Robbert Krebbers, Bas Spitters, and Eelis van der Weegen. *Math Classes*.

[Kum+14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pages 179–191. ISBN: 9781450325448. DOI: 10.1145/2535838.2535841.

[KV17] Vishal Kasliwal and Andrey Vladimirov. *A Performance-Based Comparison of C/C++ Compilers*. Colfax International. Nov. 19, 2017. URL: https://colfaxresearch.com/compiler-comparison/.

[Lam89] John Lamping. "An algorithm for optimal lambda calculus reduction". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pages 16–30. DOI: 10.1145/96709.96711.

[Lan14] Adam Langley. *Correct bounds in 32-bit code. · agl/curve25519-donna@2647eeb*. June 15, 2014. URL: https://github.com/agl/curve25519-donna/commit/2647eeba59fb628914c79ce691df794a8edc799f.

[Lee+96] Jonathan P. Leech, Larry Klaes, Matthew Wiener, and Yoshiro Yamada. *Space FAQ 08/13 - Planetary Probe History*. Sept. 17, 1996. URL: http://www.faqs.org/faqs/space/probe/.

[Lei] Tom Leinster. *Higher Operads, Higher Categories*. Cambridge Univ. Press. DOI: 10.1017/cbo9780511525896. arXiv: math/0305049.

[Lei20] Charles E. Leiserson. *Re: Quoting you in my PhD Thesis?* personal correspondence. Aug. 28, 2020.

[Ler07] Xavier Leroy. *A locally nameless solution to the POPLmark challenge*. Research Report RR-6098. INRIA, 2007, page 54. URL: https://hal.inria.fr/inria-00123945.

[Ler09] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pages 363–446. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9155-4. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[Lév80] Jean-Jacques Lévy. "Optimal reductions in the lambda-calculus". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Edited by J. P. Seldin and J. R. Hindley. Academic Press, 1980. URL: http://pauillac.inria.fr/~levy/pubs/80curry.pdf.

[Lic11] Dan Licata. *Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute*. Apr. 23, 2011. URL: https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/.

[Llo99] Robin Lloyd. "Metric mishap caused loss of NASA orbiter". In: *CNN* (Sept. 30, 1999). URL: http://www.cnn.com/TECH/space/9909/30/mars.metric.02/index.html.

[LP99] Leslie Lamport and Lawrence C. Paulson. "Should Your Specification Language Be Typed?" In: *ACM Transactions on Programming Languages and Systems* 21.3 (May 1999), pages 502–526. ISSN: 0164-0925. DOI: 10.1145/319301.319317. URL: https://lamport.azurewebsites.net/pubs/lamport-types.pdf.

[LT93] Nancy G. Leveson and Clark S. Turner. "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7 (July 1993), pages 18–41. ISSN: 1558-0814. DOI: 10.1109/MC.1993.274940. URL: https://web.archive.org/web/20041128024227/http://www.cs.umd.edu/class/spring2003/cmsc838p/Misc/therac.pdf.

[Luc+79] David C. Luckham, Steven M. German, F. W. V. Henke, Richard A. Karp, and P. W. Milne. *Stanford Pascal Verifier User Manual*. Technical report. Stanford University of California Department of Computer Science, 1979. URL: http://i.stanford.edu/pub/cstr/reports/cs/tr/79/731/CS-TR-79-731.pdf.

[Mac] Saunders Mac Lane. *Categories for the working mathematician*. DOI: 10.1007/978-1-4612-9839-7. URL: http://books.google.com/books?id=MXboNPdTv7QC.

[Mak11] Henning Makholm. *Are there statements that are undecidable but not provably undecidable*. Sept. 17, 2011. eprint: https://math.stackexchange.com/q/65302. URL: https://math.stackexchange.com/q/65302 (visited on 06/12/2020).

[Mal+13] Gregory Malecha, Adam Chlipala, Thomas Braibant, Patrick Hulin, and Edward Z. Yang. "MirrorShard: Proof by Computational Reflection with Verified Hints". In: *CoRR* abs/1305.6543 (2013). arXiv: 1305.6543.

[Mal14] Gregory Michael Malecha. "Extensible Proof Engineering in Intensional Type Theory". PhD thesis. Harvard University, Nov. 2014. URL: http://gmalecha.github.io/publication/2015/02/01/extensible-proof-engineering-in-intensional-type-theory.html.

[Mal17] Gregory Malecha. *Speeding Up Proofs with Computational Reflection*. June 5, 2017. URL: https://gmalecha.github.io/reflections/2017/speeding-up-proofs-with-computational-reflection.

[Mal18] Gregory Malecha. *To Be Typed and Untyped*. Feb. 20, 2018. URL: https://gmalecha.github.io/reflections/2018/to-be-typed-or-untyped.

[Mar08] Luc Maranget. "Compiling Pattern Matching to Good Decision Trees". In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM. 2008, pages 35–46. DOI: 10.1145/1411304.1411311. URL: http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf.

[Mar18] Erik Martin-Dorel. *Implementing primitive floats (binary64 floating-point numbers) - Issue #8276 - coq/coq*. Aug. 2018. URL: https://github.com/coq/coq/issues/8276.

[Mar75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Edited by H.E. Rose and J.C. Shepherdson. Volume 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pages 73–118. DOI: 10.1016/S0049-237X(08)71945-1. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08719451.

[Mar82] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI*. Edited by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Volume 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pages 153–175. DOI: 10.1016/S0049-237X(09)70189-2. URL: http://www.sciencedirect.com/science/article/pii/S0049237X09701892.

[MB16] Gregory Malecha and Jesper Bengtson. "Extensible and Efficient Automation Through Reflective Tactics". In: *Programming Languages and Systems*. Edited by Peter Thiemann. Berlin, Heidelberg: Springer, 2016, pages 532–559. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_21.

[MBR19] Erik Martin-Dorel, Guillaume Bertholon, and Pierre Roux. *Add primitive floats (binary64 floating-point numbers) by erikmd · Pull Request #9867 · coq/coq*. Mar. 29, 2019. URL: https://github.com/coq/coq/pull/9867.

[McB10] Conor McBride. "Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation". In: *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*. WGP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pages 1–12. ISBN: 9781450302517. DOI: 10.1145/1863495.1863497. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/DepRep/DepRep.pdf.

[MCB14] Gregory Malecha, Adam Chlipala, and Thomas Braibant. "Compositional Computational Reflection". In: *ITP'14: Proceedings of the 5th International Conference on Interactive Theorem Proving*. 2014. DOI: 10.1007/978-3-319-08970-6_24. URL: http://adam.chlipala.net/papers/MirrorShardITP14/.

[Meg] Adam Megacz. *Category Theory Library for Coq*. Coq. URL: http://www.cs.berkeley.edu/~megacz/coq-categories/.

[Miq01] Alexandre Miquel. "The Implicit Calculus of Constructions. Extending Pure Type Systems with an Intersection Type Binder and Subtyping". In: *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*. Volume 2044. TLCA'01. Springer. Kraków, Poland: Springer-Verlag, 2001, pages 344–359. ISBN: 3540419608. URL: http://www.pps.univ-paris-diderot.fr/~miquel/publis/tlca01.pdf.

[Moh95] Takahisa Mohri. "On formalization of category theory". Master's thesis. University of Tokyo, 1995. DOI: 10.1007/bfb0028395. URL: http://aleteya.cs.buap.mx/~jlavalle/papers/categorias/ST.ps.

[Moo07] J. S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series. Springer Netherlands, 2007. ISBN: 9780585336541. URL: https://books.google.com/books?id=Y09c047gV10C.

[Moo19] J. Strother Moore. "Milestones from The Pure Lisp Theorem Prover to ACL2". In: *Formal Aspects of Computing* 31.6 (2019), pages 699–732. ISSN: 1433-299X. DOI: 10.1007/s00165-019-00490-3.

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Edited by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pages 378–388. ISBN: 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_26. URL: https://leanprover.github.io/papers/system.pdf.

[MR05] Roman Matuszewski and Piotr Rudnicki. "MIZAR: the first 30 years". In: *Mechanized Mathematics and Its Applications* 4.1 (Mar. 2005). URL: http://mizar.org/people/romat/MatRud2005.pdf.

[MS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Volume 17. Bibliopolis, 1984. URL: http://www.cs.cmu.edu/afs/cs/Web/People/crary/819-f09/Martin-Lof80.pdf.

[MW13] J. Strother Moore and Claus-Peter Wirth. "Automation of Mathematical Induction as part of the History of Logic". In: *IfCoLog Journal of Logics and their Applications, Vol. 4, number 5, pp. 1505-1634 (2017)* (Sept. 24, 2013). arXiv: 1309.6226v5 [cs.AI].

[Myt+09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. "Producing Wrong Data Without Doing Anything Obviously Wrong!" In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: Association for Computing Machinery, 2009, pages 265–276. ISBN: 9781605584065. DOI: 10.1145/1508244.1508275. URL: https://users.cs.northwestern.edu/~robby/courses/322-2013-spring/mytkowicz-wrong-data.pdf.

[nCa12a] nCatLab Authors. *adjoint functor: in terms of universal arrows / universal factorization through unit and counit*. nCatLab. Nov. 2012. URL: http://ncatlab.org/nlab/show/adjoint+functor#UniversalArrows.

[nCa12b] nCatLab Authors. *subobject classifier*. nLab. Sept. 2012. URL: http://ncatlab.org/nlab/show/subobject+classifier.

[Nic11] Thomas R. Nicely. *Pentium FDIV flaw FAQ*. Aug. 19, 2011. URL: https://web.archive.org/web/20190618044444/http://www.trnicely.net/pentbug/pentbug.html.

[Niq10] Milad Niqui. *Coalgebras, bisimulation and lambda-coiteration*. Jan. 2010. URL: http://coq.inria.fr/pylons/pylons/contribs/view/Coalgebras/v8.4.

[NL98] George C. Necula and Peter Lee. "Efficient Representation and Validation of Proofs". In: *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. LICS '98. USA: IEEE Computer Society, June 1998, page 93. ISBN: 0818685069. DOI: 10.1109/lics.1998.705646. URL: https://people.eecs.berkeley.edu/~necula/Papers/lfi_lics98.ps.

[Nog02] Aleksey Yuryevich Nogin. *Theory and Implementation of an Efficient Tactic-Based Logical Framework*. Cornell University, 2002. URL: http://www.nuprl.org/documents/Nogin/thesis-nogin.pdf.

[Nor09] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Edited by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer, 2009, pages 230–266. ISBN: 978-3-642-04652-0. DOI: 10.1007/978-3-642-04652-0_5.

[Nor11] Ulf Norell. *Agda performance improvements*. Aug. 2011. URL: https://lists.chalmers.se/pipermail/agda/2011/003266.html.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283. LNCS. Springer-Verlag, 2002.

[Nuo13] Li Nuo. *Second-Year Annual Report*. July 2013. URL: http://www.cs.nott.ac.uk/~nzl/Home_Page/Homepage_files/AR2-8Jul2013.pdf.

[Oka96] Chris Okasaki. “Purely Functional Data Structures”. PhD thesis. Carnegie Mellon University, Sept. 1996. URL: https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. ISBN: 9781139811019. DOI: 10.1017/cbo9780511530104. URL: https://books.google.com/books?id=IV8hAwAAQBAJ.

[OKe04] Greg O’Keefe. “Towards a readable formalisation of category theory”. In: *Electronic Notes in Theoretical Computer Science* 91 (2004), pages 212–228. DOI: 10.1016/j.entcs.2003.12.014. URL: http://users.cecs.anu.edu.au/~okeefe/work/fcat4cats04.pdf.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. “PVS: A Prototype Verification System”. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*. CADE-11. Berlin, Heidelberg: Springer-Verlag, 1992, pages 748–752. ISBN: 3540556028. DOI: 10.1007/3-540-55602-8_217.

[Pau18] Lawrence C. Paulson. “Formalising Mathematics In Simple Type Theory”. In: *Computing Research Repository* abs/1804.07860 (Apr. 20, 2018). DOI: 10.1007/978-3-030-15655-8_20. arXiv: 1804.07860v1 [cs.LO].

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Volume 828. Springer, 1994.

[PCG18] Karl Palmskog, Ahmet Celik, and Milos Gligoric. “piCoq: Parallel regression proving for large-scale verification projects”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pages 344–355. DOI: 10.1145/3213846.3213877. URL: http://users.ece.utexas.edu/~gligoric/papers/PalmskogETAL18piCoq.pdf.

[PE88] Frank Pfenning and Conal Elliot. “Higher-order abstract syntax”. In: *Proc. PLDI*. Atlanta, Georgia, United States, 1988, pages 199–208. DOI: 10.1145/53990.54010. URL: https://www.cs.cmu.edu/~fp/papers/pldi88.pdf.

[Péd16a] Pierre-Marie Pédrot. *CoqHoTT-minute: Ticking like a Clockwork: the New Coq Tactics*. Feb. 14, 2016. URL: http://coqhott.gforge.inria.fr/blog/coq-tactic-engine/.

[Péd16b] Pierre-Marie Pédrot. *Fix bug #4777: Printing time is impacted by large terms that don’t print · coq/coq@87f9a31*. June 7, 2016. URL: https://github.com/coq/coq/commit/87f9a317b020554abef358aec614dad1fdc0bd98.

[Péd17a] Pierre-Marie Pédrot. *Fast rel lookup by ppedrot · Pull Request #6506 · coq/coq*. Dec. 2017. URL: https://github.com/coq/coq/pull/6506.

[Péd17b] Pierre-Marie Pédrot. *Introducing evar-insensitive constrs by ppedrot · Pull Request #379 · coq/coq*. Apr. 10, 2017. URL: https://github.com/coq/coq/pull/379.

[Péd18] Pierre-Marie Pédrot. *Fast typing of application nodes by ppedrot · Pull Request #8255 · coq/coq*. Aug. 15, 2018. URL: https://github.com/coq/coq/pull/8255.

[Pee+] Daniel Peebles, James Deikun, Andrea Vezzosi, and James Cook. *copumpkin/categories*. URL: https://github.com/copumpkin/categories.

[Pfe02] Frank Pfenning. "Logical Frameworks—A Brief Introduction". In: *Proof and System-Reliability*. Edited by Helmut Schwichtenberg and Ralf Steinbrüggen. Dordrecht: Springer Netherlands, 2002, pages 137–166. ISBN: 978-94-010-0413-8. DOI: 10.1007/978-94-010-0413-8_5. URL: https://www.cs.cmu.edu/~fp/papers/mdorf01.pdf.

[Pfe91] F. Pfenning. "Logic Programming in the LF Logical Framework". In: *Logical Frameworks* (1991).

[Pie] B. Pierce. *A taste of category theory for computer scientists*. Technical report. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=2846&context=compsci.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, 2002. ISBN: 9780262162098. URL: https://www.cis.upenn.edu/~bcpierce/tapl/.

[Pie90] William Pierce. "Toward Mechanical Methods for Streamlining Proofs". In: *10th International Conference on Automated Deduction*. Edited by Mark E. Stickel. Berlin, Heidelberg: Springer, 1990, pages 351–365. ISBN: 978-3-540-47171-4.

[Pip97] Nicholas Pippenger. "Pure versus Impure Lisp". In: *ACM Transactions on Programming Languages and Systems* 19.2 (Mar. 1997), pages 223–238. ISSN: 0164-0925. DOI: 10.1145/244795.244798.

[Pit03] Andrew M. Pitts. "Nominal logic, a first order theory of names and binding". In: *Information and Computation* 186.2 (2003). Theoretical Aspects of Computer Software (TACS 2001), pages 165–193. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00138-X. URL: https://www.sciencedirect.com/science/article/pii/S089054010300138X.

[PNW19] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. "From LCF to Isabelle/HOL". In: (July 5, 2019). DOI: 10.1007/s00165-019-00492-1. arXiv: 1907.02836v2 [cs.LO].

[Pol94] Robert Pollack. "The Theory of LEGO. A Proof Checker for the Extended Calculus of Constructions". PhD thesis. University of Edinburgh, 1994.

[Pot] Loïc Pottier. *Algebra*. URL: http://coq.inria.fr/pylons/pylons/contribs/view/Algebra/v8.4.

[Pou] Nicolas Pouillard. *crypto-agda/crypto-agda*. URL: https://github.com/crypto-agda/crypto-agda/tree/master/FunUniverse.

[Pre29] Mojżesz Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen". German. In: *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves* (1929). see Stansifer [Sta84] for an English translation, pages 92–101.

[PS99] Frank Pfenning and Carsten Schürmann. "System Description: Twelf – A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings*. CADE-16. Berlin, Heidelberg: Springer-Verlag, 1999, pages 202–206. ISBN: 3540662227. DOI: 10.1007/3-540-48660-7_14.

[Raa20] Panu Raatikainen. "Gödel's Incompleteness Theorems". In: *The Stanford Encyclopedia of Philosophy*. Edited by Edward N. Zalta. Fall 2020. Metaphysics Research Lab, Stanford University, 2020. URL: https://plato.stanford.edu/archives/fall2020/entries/goedel-incompleteness/.

[Ray03] "bogo-sort". In: *The Jargon File 4.4.7*. Edited by Eric Raymond. Dec. 29, 2003. URL: http://www.catb.org/jargon/html/B/bogo-sort.html.

[Rey83] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 1983, pages 513–523.

[Rin+20] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends in Programming Languages, Vol. 5, No. 2-3 (Sept. 2019), pp. 102-281* (Mar. 13, 2020). DOI: 10.1561/2500000045. arXiv: 2003.06458 [cs.LO].

[RO10] Tiark Rompf and Martin Odersky. "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs". In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010* (2010). DOI: 10.1145/2184319.2184345. URL: https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf.

[Rob65] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *Journal of the ACM* 12.1 (Jan. 1965), pages 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[Rud92] Piotr Rudnicki. "An Overview of the MIZAR Project". In: *University of Technology, Bastad*. June 30, 1992, pages 311–332. URL: http://mizar.org/project/MizarOverview.pdf.

[SA00] Zhong Shao and Andrew W. Appel. "Efficient and Safe-for-Space Closure Conversion". In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pages 129–161. ISSN: 0164-0925. DOI: 10.1145/345099.345125. URL: https://flint.cs.yale.edu/shao/papers/escc.html.

[Saï] Amokrane Saïbi. *Constructive Category Theory*. URL: http://coq.inria.fr/pylons/pylons/contribs/view/ConCaT/v8.4.

[Sco93] Dana S. Scott. "A Type-Theoretical Alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121.1&2 (1993), pages 411–440. DOI: 10.1016/0304-3975(93)90095-B. URL: https://www.cs.cmu.edu/~kw/scans/scott93tcs.pdf.

[Sha94] N. Shankar. *Metamathematics, Machines and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994. ISBN: 9780511569883. DOI: 10.1017/CBO9780511569883.

[Sha96] N. Shankar. "PVS: Combining Specification, Proof Checking, and Model Checking". In: *Formal Methods in Computer-Aided Design*. Edited by Mandayam Srivas and Albert Camilleri. Berlin, Heidelberg: Springer, 1996, pages 257–264. ISBN: 978-3-540-49567-3.

[SHM20] Daniel Selsam, Simon Hudon, and Leonardo de Moura. "Sealing Pointer-Based Optimizations Behind Pure Functions". In: (Mar. 3, 2020). arXiv: 2003.01685v1 [cs.PL].

[Shu] Michael Shulman. *An Interval Type Implies Function Extensionality*. URL: http://homotopytypetheory.org/2011/04/04.

[Shu12] Mike Shulman. *Universe Polymorphism and Typical Ambiguity*. Dec. 9, 2012. URL: https://golem.ph.utexas.edu/category/2012/12/universe_polymorphism_and_typi.html.

[Sim] Carlos Simpson. *CatsInZFC*. URL: http://coq.inria.fr/pylons/pylons/contribs/view/CatsInZFC/v8.4.

[Sli10] Konrad Slind. *Trusted Extensions of Interactive Theorem Provers: Workshop Summary*. Cambridge, England, Aug. 2010. URL: http://www.cs.utexas.edu/users/kaufmann/itp-trusted-extensions-aug-2010/summary/summary.pdf.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. "Implementing Typed Intermediate Languages". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pages 313–323. ISBN: 1581130244. DOI: 10.1145/289423.289460.

[SN08] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics*. Edited by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer, 2008, pages 28–32. ISBN: 978-3-540-71067-7.

[SO08] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics*. Edited by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer, 2008, pages 278–293. ISBN: 978-3-540-71067-7. DOI: 10.1007/978-3-540-71067-7_23. URL: https://www.irif.fr/~sozeau/research/publications/First-Class_Type_Classes.pdf.

[Soza] Matthieu Sozeau. *Cat*. URL: http://mattam.org/repos/coq/cat/.

[Sozb] Matthieu Sozeau. *mattam82/coq polyproj*. URL: https://github.com/mattam82/coq/tree/polyproj.

[Soz+] Matthieu Sozeau, Hugo Herbelin, Pierre Letouzey, Jean-Christophe Filliâtre, Matthieu Sozeau, anonymous, Pierre-Marie Pédrot, Bruno Barras, Jean-Marc Notin, Pierre Boutillier, Enrico Tassi, Stéphane Glondu, Arnaud Spiwack, Claudio Sacerdoti Coen, Christine Paulin, Olivier Desmettre, Yves Bertot, Julien Forest, David Delahaye, Pierre Corbineau, Julien Narboux, Matthias Puech, Benjamin Monate, Elie Soubiran, Pierre Courtieu, Vincent Gross, Judicaël Courant, Lionel Elie Mamane, Clément Renard, Evgeny Makarov, Claude Marché, Guillaume Melquiond, Micaela Mayero, Yann Régis-Gianas, Benjamin Grégoire, Vincent Siles, Frédéric Besson, Laurent Théry, Florent Kirchner, Maxime Dénès, Xavier Clerc, Loïc Pottier, Russel O'Connor, Assia Mahboubi, Benjamin Werner, xclerc, Huang Guan-Shieng, Jason Gross, Tom Hutchinson, Cezary Kaliszyk, Pierre, Daniel De Rauglaudre, Alexandre Miquel, Damien Doligez, Gregory Malecha, Stephane Glondu, and Andrej Bauer. *HoTT/coq*. URL: https://github.com/HoTT/coq.

[Soz+19] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *Proceedings of the ACM on Programming Languages* 4.POPL (POPL Dec. 2019). DOI: 10.1145/3371076.

[Soz14] Matthieu Sozeau. *This commit adds full universe polymorphism and fast projections to Coq. · coq/coq@a404360*. May 6, 2014. URL: https://github.com/coq/coq/commit/a4043608f704f026de7eb5167a109ca48e00c221.

[Spe66] Ernst Specker. "Typical Ambiguity". In: *Logic, Methodology and Philosophy of Science*. Edited by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Volume 44. Studies in Logic and the Foundations of Mathematics. Elsevier, 1966, pages 116–124. DOI: 10.1007/978-3-0348-9259-9_17. URL: http://www.sciencedirect.com/science/article/pii/S0049237X09705762.

[Spi07] Arnaud Spiwack. *Processor integers + Print assumption (see coqdev mailing list for the details). · coq/coq@2dbe106*. May 11, 2007. URL: https://github.com/coq/coq/commit/2dbe106c09b606.

[Spi11] Arnaud Spiwack. "Verified Computing in Homological Algebra". PhD thesis. École Polytechnique, 2011. URL: http://assert-false.net/arnaud/papers/thesis.spiwack.pdf.

[SSA96] Gerald Jay Sussman, Julie Sussman, and Harold Abelson. *Structure and Interpretation of Computer Programs*. English. 2nd edition. MIT Press, 1996. URL: http://mitpress.mit.edu/sicp/.

[Sta13] Antonios Michael Stampoulis. "VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants". PhD thesis. Yale University, 2013. URL: https://astampoulis.github.io/veriml/dissertation.pdf.

[Sta84] Ryan Stansifer. *Presburger's Article on Integer Airthmetic: Remarks and Translation*. Technical report TR84-639. Cornell University, Computer Science Department, Sept. 1984. URL: https://cs.fit.edu/~ryan/papers/presburger.pdf.

[Stu05] Aaron Stump. "POPLmark 1a with Named Bound Variables". In: (Dec. 30, 2005). URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.521.5740&rep=rep1&type=pdf.

[SUM20] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. "Tabled Typeclass Resolution". In: *Computing Research Repository* abs/2001.04301 (Jan. 13, 2020). arXiv: 2001.04301v2 [cs.PL].

[SW10] Bas Spitters and Eelis van der Weegen. "Developing the algebraic hierarchy with type classes in Coq". In: *Interactive Theorem Proving*. Springer, 2010. DOI: 10.1007/978-3-642-14052-5_35. URL: http://www.eelis.net/research/math-classes/mathclasses-diamond.pdf.

[SW11] Bas Spitters and Eelis van der Weegen. "Type Classes for Mathematics in Type Theory". In: (Feb. 7, 2011). DOI: 10.1017/s0960129511000119. arXiv: 1102.1323 [cs.LO].

[TG15] Tobias Tebbi and Jason Gross. *A Profiler for Ltac*. Presented at The First International Workshop on Coq for PL (CoqPL'15). Jan. 2015. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2015-ltac-profiler.pdf.

[Tho84] Ken Thompson. "Reflections on Trusting Trust". In: *Communications of the ACM* 27.8 (Aug. 1984), pages 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210.

[TTS18] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. "Equivalences for Free: Univalent Parametricity for Effective Transport". In: *Proc. ACM Program. Lang.* 2 (July 2018). DOI: 10.1145/3236787.

[TTS19] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. "The Marriage of Univalence and Parametricity". In: *ArXiv* (2019). arXiv: 1909.05027 [cs.PL].

[Tur37] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pages 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Aug. 3, 2013. arXiv: 1308.0729v1 [math.LO]. URL: http://homotopytypetheory.org/book/.

[VAG+20] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics*. available at https://github.com/UniMath/UniMath. 2020. URL: https://github.com/UniMath/UniMath.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proc. ACM Program. Lang.* 3 (July 2019). DOI: 10.1145/3341691.

[Voe15] Vladimir Voevodsky. “An experimental library of formalized Mathematics based on the univalent foundations”. In: *Mathematical Structures in Computer Science* 25.5 (2015), pages 1278–1294. DOI: 10.1017/S0960129514000577.

[Wad89] Philip Wadler. “Theorems for free!” In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. June 1989, pages 347–359.

[Web02] Tjark Weber. “Program Transformations in Nuprl”. Master’s thesis. Laramie, WY: University of Wyoming, Aug. 2002. URL: http://user.it.uu.se/~tjawe125/publications/weber02program.html.

[Wen02] Markus M. Wenzel. “Isabelle/Isar — a versatile environment for human-readable formal proof documents”. Dissertation. München: Technische Universität München, 2002. URL: https://mediatum.ub.tum.de/601724.

[Wie09] Freek Wiedijk. “Formalizing Arrow’s theorem”. In: *Sādhanā* 34.1 (2009), pages 193–220. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0005-1. URL: http://www.cs.ru.nl/~freek/pubs/arrow.pdf.

[Wik] Wikipedia contributors. *Therac-25*. URL: https://en.wikipedia.org/wiki/Therac-25.

[Wik20a] Wikipedia contributors. *Adjoint functors: Formal definitions: Definition via universal morphisms*. Wikipedia, the free encyclopedia. Sept. 24, 2020. URL: https://en.wikipedia.org/w/index.php?title=Adjoint_functors&oldid=980061872#Definition_via_universal_morphisms.

[Wik20b] Wikipedia contributors. *Automath — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-August-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Automath&oldid=968530682.

[Wik20c] Wikipedia contributors. *Nqthm — Wikipedia, The Free Encyclopedia*. [Online; accessed 2020-08-25]. Wikipedia, the free encyclopedia. Aug. 25, 2020. URL: https://en.wikipedia.org/w/index.php?title=Nqthm&oldid=956139282.

[Wil05] Olov Wilander. *An E-bicategory of E-categories exemplifying a type-theoretic approach to bicategories*. Technical report. University of Uppsala, 2005. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.5498&rep=rep1&type=pdf.

[Wil12] Olov Wilander. "Constructing a small category of setoids". In: *Mathematical Structures in Computer Science* 22.1 (2012), pages 103–121. URL: http://www.diva-portal.org/smash/get/diva2:399799/FULLTEXT01.pdf.

[ZC09] M. Zhivich and R. K. Cunningham. "The Real Cost of Software Errors". In: *IEEE Security & Privacy Magazine* 7.2 (2009). Edited by Sean W. Smith, pages 87–90. DOI: 10.1109/msp.2009.56. URL: http://hdl.handle.net/1721.1/74607.

[Zil+15] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. "Mtac: A Monad for Typed Tactic Programming in Coq". In: *Journal of Functional Programming* 25 (2015). URL: http://plv.mpi-sws.org/mtac/journal-draft.pdf.

[ZS17] Beta Ziliani and Matthieu Sozeau. "A Comprehensible Guide to a New Unifier for CIC Including Universe Polymorphism and Overloading". In: *Journal of Functional Programming* 27 (2017). URL: https://people.mpi-sws.org/~beta/papers/unicoq-journal.pdf.

Appendix A

Appendices for Chapter 2, The Performance Landscape in Type-Theoretic Proof Assistants

A.1 Full Example of Nested-Abstraction-Barrier Performance Issues

In Section 2.6.4, we discussed an example where unfolding nested abstraction barriers caused performance issues. Here we include the complete code for that example.[1]

```
Require Import Coq.Program.Basics.
Require Import Coq.Program.Tactics.

Set Primitive Projections.
Set Implicit Arguments.
Set Universe Polymorphism.
Set Printing Width 50.

Obligation Tactic := cbv beta; trivial.

Record prod (A B:Type) : Type := pair { fst : A ; snd : B }.
Infix "*" := prod : type_scope.
Add Printing Let prod.
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) : core_scope.
Arguments pair {A B} _ _.
```

[1] This code is also available in the file `fragments/CategoryExponentialLaws.v` on GitHub in the `JasonGross/doctoral-thesis` repository.

```
Arguments fst {A B} _.
Arguments snd {A B} _.

Reserved Notation "g '∘' f" (at level 40, left associativity).
Reserved Notation "F '₀' x"
         (at level 10, no associativity, format "'[' F '₀' ']' x").
Reserved Notation "F '₁' m"
         (at level 10, no associativity, format "'[' F '₁' ']' m").
Reserved Infix "≅" (at level 70, no associativity).
Reserved Notation "x ≅ y :>>> T" (at level 70, no associativity).

Record Category :=
  {
    object :> Type;
    morphism : object -> object -> Type;

    identity : forall x, morphism x x;
    compose : forall s d d',
        morphism d d'
        -> morphism s d
        -> morphism s d'
    where "f '∘' g" := (compose f g);

    associativity : forall x1 x2 x3 x4
                           (m1 : morphism x1 x2)
                           (m2 : morphism x2 x3)
                           (m3 : morphism x3 x4),
        (m3 ∘ m2) ∘ m1 = m3 ∘ (m2 ∘ m1);

    left_identity : forall a b (f : morphism a b), identity b ∘ f = f;
    right_identity : forall a b (f : morphism a b), f ∘ identity a = f;

  }.

Declare Scope category_scope.
Declare Scope object_scope.
Declare Scope morphism_scope.
Bind Scope category_scope with Category.
Bind Scope object_scope with object.
Bind Scope morphism_scope with morphism.
Delimit Scope morphism_scope with morphism.
Delimit Scope category_scope with category.
Delimit Scope object_scope with object.

Arguments identity {_} _.
```

```
Arguments compose {_ _ _ _} _ _.

Infix "∘" := compose : morphism_scope.
Notation "1" := (identity _) : morphism_scope.
Local Open Scope morphism_scope.

Record isomorphic {C : Category} (s d : C) :=
  {
    fwd : morphism C s d
    ; bwd : morphism C d s
    ; iso1 : fwd ∘ bwd = 1
    ; iso2 : bwd ∘ fwd = 1
  }.

Notation "s ≅ d :>>> C" := (@isomorphic C s d) : morphism_scope.
Infix "≅" := isomorphic : morphism_scope.

Declare Scope functor_scope.
Delimit Scope functor_scope with functor.

Local Open Scope morphism_scope.

Record Functor (C D : Category) :=
  {
    object_of :> C -> D;
    morphism_of : forall s d, morphism C s d
                              -> morphism D (object_of s) (object_of d);
    composition_of : forall s d d'
                             (m1 : morphism C s d) (m2: morphism C d d'),
        morphism_of _ _ (m2 ∘ m1)
        = (morphism_of _ _ m2) ∘ (morphism_of _ _ m1);
    identity_of : forall x, morphism_of _ _ (identity x)
                            = identity (object_of x)
  }.

Arguments object_of {C D} _.
Arguments morphism_of {C D} _ {s d}.

Bind Scope functor_scope with Functor.

Notation "F ' 0 ' x" := (object_of F x) : object_scope.
Notation "F ' 1 ' m" := (morphism_of F m) : morphism_scope.

Declare Scope natural_transformation_scope.
Delimit Scope natural_transformation_scope with natural_transformation.
```

```
Module Functor.
  Program Definition identity (C : Category) : Functor C C
    := {| object_of x := x
          ; morphism_of s d m := m |}.

  Program Definition compose (s d d' : Category)
          (F1 : Functor d d') (F2 : Functor s d)
    : Functor s d'
    := {| object_of x := F1 (F2 x)
          ; morphism_of s d m := F1 _1 (F2 _1 m) |}.
  Next Obligation. Admitted.
  Next Obligation. Admitted.
End Functor.

Infix "∘" := Functor.compose : functor_scope.
Notation "1" := (Functor.identity _) : functor_scope.

Local Open Scope morphism_scope.
Local Open Scope natural_transformation_scope.

Record NaturalTransformation {C D : Category} (F G : Functor C D) :=
  {
    components_of :> forall c, morphism D (F c) (G c);
    commutes : forall s d (m : morphism C s d),
        components_of d ∘ F _1 m = G _1 m ∘ components_of s
  }.

Bind Scope natural_transformation_scope with NaturalTransformation.

Module NaturalTransformation.
  Program Definition identity {C D : Category} (F : Functor C D)
  : NaturalTransformation F F
    := {| components_of x := 1 |}.
  Next Obligation. Admitted.

  Program Definition compose {C D : Category} (s d d' : Functor C D)
          (T1 : NaturalTransformation d d') (T2 : NaturalTransformation s d)
    : NaturalTransformation s d'
    := {| components_of x := T1 x ∘ T2 x |}.
  Next Obligation. Admitted.
End NaturalTransformation.

Infix "∘" := NaturalTransformation.compose
             : natural_transformation_scope.
```

```
Notation "1" := (NaturalTransformation.identity _)
                : natural_transformation_scope.

Program Definition functor_category (C D : Category) : Category
  := {| object := Functor C D
        ; morphism := @NaturalTransformation C D
        ; identity x := 1
        ; compose s d d' m1 m2 := m1 ∘ m2 |}%natural_transformation.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.

Notation "C -> D" := (functor_category C D) : category_scope.

Program Definition prod_category (C D : Category) : Category
  := {| object := C * D
        ; morphism s d
          := morphism C (fst s) (fst d) * morphism D (snd s) (snd d)
        ; identity x := (1, 1)
        ; compose s d d' m1 m2 := (fst m1 ∘ fst m2, snd m1 ∘ snd m2)
     |}%type%morphism.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.

Infix "*" := prod_category : category_scope.

Program Definition Cat : Category :=
  {|
    object := Category
    ; morphism := Functor
    ; compose s d d' m1 m2 := m1 ∘ m2
    ; identity x := 1
  |}%functor.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.

Local Open Scope functor_scope.
Local Open Scope natural_transformation_scope.
Local Open Scope object_scope.
Local Open Scope morphism_scope.
Local Open Scope category_scope.

Arguments Build_Functor _ _ & .
```

```
Arguments Build_isomorphic _ _ _ & .
Arguments Build_NaturalTransformation _ _ _ _ & .
Arguments pair _ _ & .
Canonical Structure default_eta {A B} (v : A * B) : A * B := (fst v, snd v).
Canonical Structure pair' {A B} (a : A) (b : B) : A * B := pair a b.

Declare Scope functor_object_scope.
Declare Scope functor_morphism_scope.
Declare Scope natural_transformation_components_scope.
Arguments Build_Functor (C D)%category_scope
& _%functor_object_scope _%functor_morphism_scope (_ _)%function_scope.
Arguments Build_NaturalTransformation [C D]%category_scope (F G)%functor_scope
& _%natural_transformation_components_scope _%function_scope.

Notation "x : A ↦_o f"
  := (fun x : A%category => f) (at level 70) : functor_object_scope.
Notation "x ↦_o f"
  := (fun x => f) (at level 70) : functor_object_scope.
Notation "' x ↦_o f"
  := (fun '(x%category) => f) (x strict pattern, at level 70)
     : functor_object_scope.
Notation "m @ s --> d ↦_m f"
  := (fun s d m => f) (at level 70) : functor_morphism_scope.
Notation "' m @ s --> d ↦_m f"
  := (fun s d 'm => f) (at level 70, m strict pattern)
     : functor_morphism_scope.
Notation "m : A ↦_m f"
  := (fun s d (m : A%category) => f) (at level 70)
     : functor_morphism_scope.
Notation "m ↦_m f"
  := (fun s d m => f) (at level 70) : functor_morphism_scope.
Notation "' m ↦_m f"
  := (fun s d '(m%category) => f) (m strict pattern, at level 70)
     : functor_morphism_scope.
Notation "x : A ↦_t f"
  := (fun x : A%category => f) (at level 70)
     : natural_transformation_components_scope.
Notation "' x ↦_t f"
  := (fun '(x%category) => f) (x strict pattern, at level 70)
     : natural_transformation_components_scope.
Notation "x ↦_t f"
  := (fun x => f) (at level 70)
     : natural_transformation_components_scope.

Notation "⟨ fo ; mo ⟩"
```

```
  := (@Build_Functor _ _ fo mo _ _) (only parsing) : functor_scope.
Notation "⟨ f ⟩"
  := (@Build_NaturalTransformation _ _ _ _ f _) (only parsing)
     : natural_transformation_scope.


Notation "'λ' ⟨ fo ; mo ⟩"
  := (@Build_Functor _ _ fo mo _ _) (only parsing) : functor_scope.
Notation "'λ' ⟨ f ⟩"
  := (@Build_NaturalTransformation _ _ _ _ f _) (only parsing)
     : natural_transformation_scope.

Notation "'λ_o' x1 .. xn , fo ; 'λ_m' m1 .. mn , mo"
  := (@Build_Functor

        _ _
        (fun x1 => .. (fun xn => fo) .. )
        (fun s d => (fun m1 => .. (fun mn => mo) .. ))
        _ _)
       (only parsing, x1 binder, xn binder, m1 binder, mn binder, at level 70)
     : functor_scope.
Notation "'λ_t' x1 .. xn , f"
  := (@Build_NaturalTransformation

        _ _ _ _
        (fun x1 => .. (fun xn => f) .. )
        _)
       (only parsing, x1 binder, xn binder, at level 70)
     : natural_transformation_scope.

(** [(C1 × C2 → D) ≅ (C1 → (C2 → D))] *)
(** We denote functors by pairs of maps on objects ([↦o]) and
    morphisms ([↦m]), and natural transformations as a single map
    ([↦t]) *)
Time Program Definition curry_iso1 (C1 C2 D : Category)
  : (C1 * C2 -> D) ≅ (C1 -> (C2 -> D)) :>>> Cat
  := {| fwd
        := ⟨ F ↦o ⟨ c1 ↦o ⟨ c2 ↦o F _0  (c1, c2)
                          ; m  ↦m F _1  (identity c1, m) ⟩
                  ; m1 ↦m ⟨ c2 ↦t F _1  (m1, identity c2) ⟩ ⟩
           ; T ↦m ⟨ c1 ↦t ⟨ c2 ↦t T (c1, c2) ⟩ ⟩ ⟩;
        bwd
        := ⟨ F ↦o ⟨ '(c1, c2) ↦o (F _0  c1)_0 c2
                  ; '(m1, m2) ↦m (F _1  m1) _ ∘ (F _0  _)_1 m2 ⟩
           ; T ↦m ⟨ '(c1, c2) ↦t (T c1) c2 ⟩ ⟩ |}.

(** [(C1 × C2 → D) ≅ (C1 → (C2 → D))] *)
```

```
(** We denote functors by pairs of maps ([λ]) on objects ([↦ₒ]) and
    morphisms ([↦ₘ]), and natural transformations as a single map
    ([λ ⟨ ... ↦ₜ ... ⟩]) *)
Time Program Definition curry_iso2 (C₁ C₂ D : Category)
  : (C₁ * C₂ -> D) ≅ (C₁ -> (C₂ -> D)) :>>> Cat
  := {| fwd
        := λ ⟨ F ↦ₒ λ ⟨ c₁ ↦ₒ λ ⟨ c₂ ↦ₒ F _0 (c₁, c₂)
                                 ; m ↦ₘ F _1 (identity c₁, m) ⟩
                       ; m₁ ↦ₘ λ ⟨ c₂ ↦ₜ F _1 (m₁, identity c₂) ⟩ ⟩
             ; T ↦ₘ λ ⟨ c₁ ↦ₜ λ ⟨ c₂ ↦ₜ T (c₁, c₂) ⟩ ⟩ ⟩;
        bwd
        := λ ⟨ F ↦ₒ λ ⟨ '(c₁, c₂) ↦ₒ (F _0 c₁)₀ c₂
                       ; '(m₁, m₂) ↦ₘ (F _1 m₁) _ ∘ (F _0 _)₁ m₂ ⟩
             ; T ↦ₘ λ ⟨ '(c₁, c₂) ↦ₜ (T c₁) c₂ ⟩ ⟩ |}.

(** [(C₁ × C₂ → D) ≅ (C₁ → (C₂ → D))] *)
(** We denote functors by pairs of maps on objects ([λₒ]) and
    morphisms ([λₘ]), and natural transformations as a single map
    ([λₜ]) *)
Time Program Definition curry_iso3 (C₁ C₂ D : Category)
  : (C₁ * C₂ -> D) ≅ (C₁ -> (C₂ -> D)) :>>> Cat
  := {| fwd
        := λₒ F, λₒ c₁, λₒ c₂, F _0 (c₁, c₂)
                       ; λₘ m , F _1 (identity c₁, m)
              ; λₘ m₁, λₜ c₂, F _1 (m₁, identity c₂)
         ; λₘ T, λₜ c₁, λₜ c₂, T (c₁, c₂);
        bwd
        := λₒ F, λₒ '(c₁, c₂), (F _0 c₁)₀ c₂
              ; λₘ '(m₁, m₂), (F _1 m₁) _ ∘ (F _0 _)₁ m₂
         ; λₘ T, λₜ '(c₁, c₂), (T c₁) c₂ |}.

(** [(C₁ × C₂ → D) ≅ (C₁ → (C₂ → D))] *)
(** We provide the action of functors on objects ([object_of]) and on
    morphisms ([morphism_of]), and we provide the action of natural
    transformations on object ([components_of] *)
Time Program Definition curry_iso (C₁ C₂ D : Category)
  : (C₁ * C₂ -> D) ≅ (C₁ -> (C₂ -> D)) :>>> Cat
  := {| fwd
        := {| object_of F
              := {| object_of c₁
                    := {| object_of      c₂ := F _0 (c₁, c₂);
                          morphism_of _ _ m := F _1 (identity c₁, m) |};
                  morphism_of _ _ m₁
                    := {| components_of  c₂ := F _1 (m₁, identity c₂) |} |};
              morphism_of _ _ T
```

```
            := {| components_of c1
                  := {| components_of  c2 := T (c1, c2) |} |} |};
      bwd
      := {| object_of F
            := {| object_of                          '(c1, c2)
                  := (F 0 c1)0 c2;
                  morphism_of '(s1, s2) '(d1, d2) '(m1, m2)
                  := (F 1 m1) d2 ∘ (F 0 s1)1 m2 |};
            morphism_of s d T
            := {| components_of '(c1, c2) := (T c1) c2 |} |}; |}.
(* Finished transaction in 1.958 secs (1.958u,0.s) (successful) *)
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation. Admitted.
Next Obligation.
(**
1 subgoal (ID 1464)

  ============================
  forall C1 C2 D : Category,
  {|
  object_of := F
               ↦o {|
                  object_of := pat
                               ↦o
                               (F0 (fst pat))0
                               (snd pat);
                  morphism_of := pat1 @ pat -->
                                 pat0
                                 ↦m
                                 (F1 (fst pat1))
                                 (snd pat0)
                                 ∘
                                 (F0 (fst pat))1
```

```
                              (snd pat1);
                  composition_of := curry_iso_obligation_7
                              F;
                  identity_of := curry_iso_obligation_8
                              F |};
morphism_of := T @ s --> d
                 ↦m {|
                    components_of := pat
                              ↦t T (fst pat)
                              (snd pat);
                    commutes := curry_iso_obligation_9
                              T |};
composition_of := curry_iso_obligation_11
                    (D:=D);
identity_of := curry_iso_obligation_10 (D:=D) |}
∘ {|
  object_of := F
                 ↦o {|
                    object_of := c1
                              ↦o
                              {|
                              object_of := c2
                              ↦o F0 (c1, c2);
                              morphism_of := m @
                              s --> d
                              ↦m F1 (1, m);
                              composition_of := curry_iso_obligation_1
                              F c1;
                              identity_of := curry_iso_obligation_2
                              F c1 |};
                    morphism_of := m1 @ s --> d
                              ↦m {|
                              components_of := c2
                              ↦t F1 (m1, 1);
                              commutes := curry_iso_obligation_3
                              F s d m1 |};
                    composition_of := curry_iso_obligation_5
                              F;
                    identity_of := curry_iso_obligation_4
                              F |};
  morphism_of := T @ s --> d
                  ↦m {|
                     components_of := c1
                              ↦t {|
                              components_of := c2
```

```
                                  ↦t T (c1 , c2 );
                                   commutes := curry_iso_obligation_6
                                   T c1 |};
                          commutes := curry_iso_obligation_12
                                   T |};
    composition_of := curry_iso_obligation_14
                        (D:=D);
    identity_of := curry_iso_obligation_13 (D:=D) |} =
  1
 *)
  (** About 48 lines *)
  cbv [compose Cat Functor.compose NaturalTransformation.compose].
  (**
1 subgoal (ID 1469)

  ============================
  forall C1 C2 D : Category,
  {|
  object_of := x
               ↦o {|
                  object_of := F
                               ↦o
                               {|
                               object_of := pat
                                ↦o (F0 (fst pat))0
                                (snd pat);
                               morphism_of := pat1 @
                                pat --> pat0
                                ↦m (F1
                                (fst pat1))
                                (snd pat0)
                                ∘ (F0 (fst pat))1
                                (snd pat1);
                               composition_of := curry_iso_obligation_7
                                F;
                               identity_of := curry_iso_obligation_8
                                F |};
                  morphism_of := T @ s --> d
                                ↦m
                                {|
                                components_of := pat
                                ↦t T (fst pat)
                                (snd pat);
                                commutes := curry_iso_obligation_9
                                T |};
```

```
    composition_of := curry_iso_obligation_11
                      (D:=D);
    identity_of := curry_iso_obligation_10
                      (D:=D) |}_0
  ({|
   object_of := F
                  ↦_o {|
                      object_of := c_1
                      ↦_o {|
                      object_of := c_2
                      ↦_o F_0 (c_1, c_2);
                      morphism_of := m @
                      s --> d
                      ↦_m F_1 (1, m);
                      composition_of := curry_iso_obligation_1
                      F c_1;
                      identity_of := curry_iso_obligation_2
                      F c_1 |};
                      morphism_of := m_1 @
                      s --> d
                      ↦_m {|
                      components_of := c_2
                      ↦_t F_1 (m_1, 1);
                      commutes := curry_iso_obligation_3
                      F s d m_1 |};
                      composition_of := curry_iso_obligation_5
                      F;
                      identity_of := curry_iso_obligation_4
                      F |};
   morphism_of := T @ s --> d
                    ↦_m
                    {|
                    components_of := c_1
                     ↦_t {|
                     components_of := c_2
                     ↦_t T (c_1, c_2);
                     commutes := curry_iso_obligation_6
                     T c_1 |};
                    commutes := curry_iso_obligation_12
                     T |};
   composition_of := curry_iso_obligation_14
                     (D:=D);
   identity_of := curry_iso_obligation_13
                     (D:=D) |}_0 x);
morphism_of := m @ s --> d
```

```
↦m {|
   object_of := F
                ↦o
                {|
                object_of := pat
                ↦o (F0 (fst pat))0
                (snd pat);
                morphism_of := pat1 @
                pat --> pat0
                ↦m (F1
                (fst pat1))
                (snd pat0)
                ∘ (F0 (fst pat))1
                (snd pat1);
                composition_of := curry_iso_obligation_7
                F;
                identity_of := curry_iso_obligation_8
                F |};
   morphism_of := T @ s0 --> d0
                ↦m {|
                components_of := pat
                ↦t T (fst pat)
                (snd pat);
                commutes := curry_iso_obligation_9
                T |};
   composition_of := curry_iso_obligation_11
                (D:=D);
   identity_of := curry_iso_obligation_10
                (D:=D) |}1
({|
 object_of := F
              ↦o
              {|
              object_of := c1
               ↦o {|
               object_of := c2
               ↦o F0 (c1, c2);
               morphism_of := m0 @
               s0 --> d0
               ↦m F1 (1, m0);
               composition_of := curry_iso_obligation_1
               F c1;
               identity_of := curry_iso_obligation_2
               F c1 |};
              morphism_of := m1 @
```

```
                                s0 --> d0
                                ↦m {|
                                components_of := c2
                                ↦t F1 (m1, 1);
                                commutes := curry_iso_obligation_3
                                F s0 d0 m1 |};
                               composition_of := curry_iso_obligation_5
                                F;
                               identity_of := curry_iso_obligation_4
                                F |};
                  morphism_of := T @ s0 --> d0
                                ↦m
                                {|
                                components_of := c1
                                ↦t {|
                                components_of := c2
                                ↦t T (c1, c2);
                                commutes := curry_iso_obligation_6
                                T c1 |};
                                commutes := curry_iso_obligation_12
                                T |};
                  composition_of := curry_iso_obligation_14
                                (D:=D);
                  identity_of := curry_iso_obligation_13
                                (D:=D) |}1 m);
composition_of := Functor.compose_obligation_1
                      {|
                      object_of := F
                                ↦o {|
                                object_of := pat
                                ↦o (F0 (fst pat))0
                                (snd pat);
                                morphism_of := pat1 @
                                pat --> pat0
                                ↦m (F1
                                (fst pat1))
                                (snd pat0)
                                ∘ (F0 (fst pat))1
                                (snd pat1);
                                composition_of := curry_iso_obligation_7
                                F;
                                identity_of := curry_iso_obligation_8
                                F |};
                      morphism_of := T @ s --> d
                                ↦m {|
```

```
          components_of := pat
          ↦t T (fst pat)
          (snd pat);
          commutes := curry_iso_obligation_9
          T |};
composition_of := curry_iso_obligation_11
          (D:=D);
identity_of := curry_iso_obligation_10
          (D:=D) |}
{|
object_of := F
          ↦o {|
          object_of := c1
          ↦o {|
          object_of := c2
          ↦o F0 (c1, c2);
          morphism_of := m @
          s --> d
          ↦m F1 (1, m);
          composition_of := curry_iso_obligation_1
          F c1;
          identity_of := curry_iso_obligation_2
          F c1 |};
          morphism_of := m1 @
          s --> d
          ↦m {|
          components_of := c2
          ↦t F1 (m1, 1);
          commutes := curry_iso_obligation_3
          F s d m1 |};
          composition_of := curry_iso_obligation_5
          F;
          identity_of := curry_iso_obligation_4
          F |};
morphism_of := T @ s --> d
          ↦m {|
          components_of := c1
          ↦t {|
          components_of := c2
          ↦t T (c1, c2);
          commutes := curry_iso_obligation_6
          T c1 |};
          commutes := curry_iso_obligation_12
          T |};
composition_of := curry_iso_obligation_14
```

```
                                (D:=D);
                        identity_of := curry_iso_obligation_13
                                (D:=D) |};
identity_of := Functor.compose_obligation_2
                    {|
                    object_of := F
                                 ↦_o
                                 {|
                                 object_of := pat
                                 ↦_o (F_0 (fst pat))_0
                                 (snd pat);
                                 morphism_of := pat1 @
                                 pat --> pat0
                                 ↦_m (F_1
                                 (fst pat1))
                                 (snd pat0)
                                 ∘ (F_0 (fst pat))_1
                                 (snd pat1);
                                 composition_of := curry_iso_obligation_7
                                 F;
                                 identity_of := curry_iso_obligation_8
                                 F |};
                    morphism_of := T @ s --> d
                                 ↦_m
                                 {|
                                 components_of := pat
                                 ↦_t T (fst pat)
                                 (snd pat);
                                 commutes := curry_iso_obligation_9
                                 T |};
                    composition_of := curry_iso_obligation_11
                                 (D:=D);
                    identity_of := curry_iso_obligation_10
                                 (D:=D) |}
                    {|
                    object_of := F
                                 ↦_o
                                 {|
                                 object_of := c_1
                                 ↦_o {|
                                 object_of := c_2
                                 ↦_o F_0 (c_1, c_2);
                                 morphism_of := m @
                                 s --> d
                                 ↦_m F_1 (1, m);
```

```
                                composition_of := curry_iso_obligation_1
                                F c1 ;
                                identity_of := curry_iso_obligation_2
                                F c1 |};
                                morphism_of := m1 @
                                s --> d
                                ↦m {|
                                components_of := c2
                                ↦t F1 (m1 , 1);
                                commutes := curry_iso_obligation_3
                                F s d m1 |};
                                composition_of := curry_iso_obligation_5
                                F;
                                identity_of := curry_iso_obligation_4
                                F |};
                  morphism_of := T @ s --> d
                                ↦m
                                {|
                                components_of := c1
                                ↦t {|
                                components_of := c2
                                ↦t T (c1 , c2);
                                commutes := curry_iso_obligation_6
                                T c1 |};
                                commutes := curry_iso_obligation_12
                                T |};
                  composition_of := curry_iso_obligation_14
                                (D:=D);
                  identity_of := curry_iso_obligation_13
                                (D:=D) |} |} = 1
   *)
  (** About 254 lines *)
  cbn [object_of morphism_of components_of].
  (**
1 subgoal (ID 1471)

  ============================
  forall C1 C2 D : Category,
  {|
  object_of := x
               ↦o {|
                  object_of := pat
                               ↦o
                               x0
                               (fst pat,
```

```
                snd pat);
morphism_of := pat1 @ pat -->
                pat0
                ↦m
                x1
                (fst pat1, 1)
                ∘
                x1
                (1, snd pat1);
composition_of := curry_iso_obligation_7
                {|
                object_of := c1
                ↦o {|
                object_of := c2
                ↦o x0 (c1, c2);
                morphism_of := m @
                s --> d
                ↦m x1 (1, m);
                composition_of := curry_iso_obligation_1
                x c1;
                identity_of := curry_iso_obligation_2
                x c1 |};
                morphism_of := m1 @
                s --> d
                ↦m {|
                components_of := c2
                ↦t x1 (m1, 1);
                commutes := curry_iso_obligation_3
                x s d m1 |};
                composition_of := curry_iso_obligation_5
                x;
                identity_of := curry_iso_obligation_4
                x |};
identity_of := curry_iso_obligation_8
                {|
                object_of := c1
                ↦o {|
                object_of := c2
                ↦o x0 (c1, c2);
                morphism_of := m @
                s --> d
                ↦m x1 (1, m);
                composition_of := curry_iso_obligation_1
                x c1;
                identity_of := curry_iso_obligation_2
```

```
                                x c1 |};
                                morphism_of := m1 @
                                s --> d
                                ↦m {|
                                components_of := c2
                                ↦t x1 (m1, 1);
                                commutes := curry_iso_obligation_3
                                x s d m1 |};
                                composition_of := curry_iso_obligation_5
                                x;
                                identity_of := curry_iso_obligation_4
                                x |} |};
morphism_of := m @ s --> d
                ↦m {|
                    components_of := pat
                                ↦t m
                                (
                                fst pat,
                                snd pat);
                    commutes := curry_iso_obligation_9
                                {|
                                components_of := c1
                                ↦t {|
                                components_of := c2
                                ↦t m (c1, c2);
                                commutes := curry_iso_obligation_6
                                m c1 |};
                                commutes := curry_iso_obligation_12
                                m |} |};
composition_of := Functor.compose_obligation_1
                    {|
                    object_of := F
                                ↦o {|
                                object_of := pat
                                ↦o (F0 (fst pat))0
                                (snd pat);
                                morphism_of := pat1 @
                                pat --> pat0
                                ↦m (F1
                                (fst pat1))
                                (snd pat0)
                                ∘ (F0 (fst pat))1
                                (snd pat1);
                                composition_of := curry_iso_obligation_7
                                F;
```

```
          identity_of := curry_iso_obligation_8
          F |};
morphism_of := T @ s --> d
          ↦_m {|
          components_of := pat
          ↦_t T (fst pat)
          (snd pat);
          commutes := curry_iso_obligation_9
          T |};
composition_of := curry_iso_obligation_11
          (D:=D);
identity_of := curry_iso_obligation_10
          (D:=D) |}
{|
object_of := F
          ↦_o {|
          object_of := c_1
          ↦_o {|
          object_of := c_2
          ↦_o F_0 (c_1, c_2);
          morphism_of := m @
          s --> d
          ↦_m F_1 (1, m);
          composition_of := curry_iso_obligation_1
          F c_1;
          identity_of := curry_iso_obligation_2
          F c_1 |};
          morphism_of := m_1 @
          s --> d
          ↦_m {|
          components_of := c_2
          ↦_t F_1 (m_1, 1);
          commutes := curry_iso_obligation_3
          F s d m_1 |};
          composition_of := curry_iso_obligation_5
          F;
          identity_of := curry_iso_obligation_4
          F |};
morphism_of := T @ s --> d
          ↦_m {|
          components_of := c_1
          ↦_t {|
          components_of := c_2
          ↦_t T (c_1, c_2);
          commutes := curry_iso_obligation_6
```

```
                                T c1 |};
                                commutes := curry_iso_obligation_12
                                T |};
                        composition_of := curry_iso_obligation_14
                                (D:=D);
                        identity_of := curry_iso_obligation_13
                                (D:=D) |};
identity_of := Functor.compose_obligation_2
                {|
                object_of := F
                                ↦o
                                {|
                                object_of := pat
                                ↦o (F0 (fst pat))0
                                (snd pat);
                                morphism_of := pat1 @
                                pat --> pat0
                                ↦m (F1
                                (fst pat1))
                                (snd pat0)
                                ∘ (F0 (fst pat))1
                                (snd pat1);
                                composition_of := curry_iso_obligation_7
                                F;
                                identity_of := curry_iso_obligation_8
                                F |};
                morphism_of := T @ s --> d
                                ↦m
                                {|
                                components_of := pat
                                ↦t T (fst pat)
                                (snd pat);
                                commutes := curry_iso_obligation_9
                                T |};
                composition_of := curry_iso_obligation_11
                                (D:=D);
                identity_of := curry_iso_obligation_10
                                (D:=D) |}
                {|
                object_of := F
                                ↦o
                                {|
                                object_of := c1
                                ↦o {|
                                object_of := c2
```

```
                                  ↦o F0 (c1, c2);
                                  morphism_of := m @
                                  s --> d
                                  ↦m F1 (1, m);
                                  composition_of := curry_iso_obligation_1
                                  F c1;
                                  identity_of := curry_iso_obligation_2
                                  F c1 |};
                                  morphism_of := m1 @
                                  s --> d
                                  ↦m {|
                                  components_of := c2
                                  ↦t F1 (m1, 1);
                                  commutes := curry_iso_obligation_3
                                  F s d m1 |};
                                  composition_of := curry_iso_obligation_5
                                  F;
                                  identity_of := curry_iso_obligation_4
                                  F |};
                        morphism_of := T @ s --> d
                                  ↦m
                                  {|
                                  components_of := c1
                                  ↦t {|
                                  components_of := c2
                                  ↦t T (c1, c2);
                                  commutes := curry_iso_obligation_6
                                  T c1 |};
                                  commutes := curry_iso_obligation_12
                                  T |};
                        composition_of := curry_iso_obligation_14
                                  (D:=D);
                        identity_of := curry_iso_obligation_13
                                  (D:=D) |} |} = 1
     *)
    (** About 200 lines *)
Abort.

Import EqNotations.
Axiom to_arrow1_eq
  : forall C1 C2 D (F G : Functor C1 (C2 -> D))
           (Hoo : forall c1 c2, F c1 c2 = G c1 c2)
           (Hom : forall c1 s d (m : morphism _ s d),
               (rew [fun s => morphism D s _] (Hoo c1 s) in
                rew [morphism D _] (Hoo c1 d) in
```

```
                (F c1)1 m) = (G c1)1 m)
            (Hm : forall s d (m : morphism _ s d) c2,
                (rew [fun s => morphism D s _] Hoo s c2 in
                 rew Hoo d c2 in
                 (F 1 m) c2)
                = (G 1 m) c2),
    F = G.
Axiom to_arrow2_eq
  : forall C1 C2 C3 D (F G : Functor C1 (C2 -> (C3 -> D)))
           (Hooo : forall c1 c2 c3, F c1 c2 c3 = G c1 c2 c3)
           (Hoom : forall c1 c2 s d (m : morphism _ s d),
               (rew [fun s => morphism D s _] (Hooo c1 c2 s) in
                rew [morphism D _] (Hooo c1 c2 d) in
                (F c1 c2)1 m) = (G c1 c2)1 m)
           (Hom : forall c1 s d (m : morphism _ s d) c2,
               (rew [fun s => morphism D s _] Hooo c1 s c2 in
                rew Hooo c1 d c2 in
                ((F c1)1 m) c2)
               = ((G c1)1 m) c2)
           (Hm : forall s d (m : morphism _ s d) c2 c3,
               (rew [fun s => morphism D s _] Hooo s c2 c3 in
                rew Hooo d c2 c3 in
                (F 1 m) c2 c3)
               = ((G 1 m) c2 c3)),
    F = G.

Local Ltac unfold_stuff
  := intros;
     cbv [Cat compose prod_category
          Functor.compose NaturalTransformation.compose];
     cbn [object_of morphism_of components_of].

Local Ltac fin_t
  := repeat first [ progress intros
                  | reflexivity
                  | progress cbn
                  | rewrite left_identity
                  | rewrite right_identity
                  | rewrite identity_of
                  | rewrite <- composition_of ].

Next Obligation.
Proof.
  Time solve [ intros; unshelve eapply to_arrow1_eq; unfold_stuff; fin_t ].
  (* Finished transaction in 0.061 secs (0.061u,0.s) (successful) *)
```

```
  Undo.
  Time solve [ intros; unfold_stuff; unshelve eapply to_arrow1_eq; fin_t ].
  (* Finished transaction in 0.176 secs (0.176u,0.s) (successful) *)
Qed.

Next Obligation.
Proof.
  Time solve [ intros; unshelve eapply to_arrow2_eq; unfold_stuff; fin_t ].
  (* Finished transaction in 0.085 secs (0.085u,0.s) (successful) *)
  Undo.
  Time solve [ intros; unfold_stuff; unshelve eapply to_arrow2_eq; fin_t ].
  (* Finished transaction in 0.485 secs (0.475u,0.007s) (successful) *)
Qed.
```

A.1.1 Example in the Category of Sets

We include here the code for the components defined in the category of sets.[2]

```
Time
  Definition curry_iso_components_set {C₁ C₂ D : Set}
:= ((fun (F : C₁ * C₂ -> D)
     => (fun c₁ c₂ => F (c₁, c₂)) : C₁ -> C₂ -> D),
    (fun (F : C₁ * C₂ -> D)
     => (fun c₁ c₂s c₂d (m₂ : c₂s = c₂d)
         => f_equal F (f_equal2 pair (eq_refl c₁) m₂))),
    (fun (F : C₁ * C₂ -> D)
     => (fun c₁s c₁d (m₁ : c₁s = c₁d) c₂
         => f_equal F (f_equal2 pair m₁ (eq_refl c₂)))),
    (fun F G (T : forall x : C₁ * C₂, F x = G x :> D)
     => (fun c₁ c₂ => T (c₁, c₂))),
    (fun (F : C₁ -> C₂ -> D)
     => (fun '(c₁, c₂) => F c₁ c₂) : C₁ * C₂ -> D),
    (fun (F : C₁ -> C₂ -> D)
     => (fun s d (m : s = d :> C₁ * C₂)
         => eq_trans (f_equal (F _) (f_equal (@snd _ _) m))
                     (f_equal (fun F => F _) (f_equal F (f_equal (@fst _ _) m)))
            : F (fst s) (snd s) = F (fst d) (snd d))),
    (fun F G (T : forall (c₁ : C₁) (c₂ : C₂), F c₁ c₂ = G c₁ c₂ :> D)
     => (fun '(c₁, c₂) => T c₁ c₂)
        : forall '((c₁, c₂) : C₁ * C₂), F c₁ c₂ = G c₁ c₂)).
(* Finished transaction in 0.009 secs (0.009u,0.s) (successful) *)
```

[2]This code is also available in the file `fragments/CategoryExponentialLawsSet.v` on GitHub in the `JasonGross/doctoral-thesis` repository.

Appendix B

Notes on the Benchmarking Setup

We performed most benchmarks in this dissertation on a 3.5 GHz Intel Haswell running Linux and Coq 8.12.2 with OCaml 4.06.1. We describe in this appendix exceptions to this benchmarking setup.

Excepting the benchmarks in s ?? and 5 and ??, all of the benchmarks can be found in the GitHub repository `https://github.com/JasonGross/doctoral-thesis` in the folder `performance-experiments` and the folder `performance-experiments-8-9`. Most benchmarks in this dissertation, excepting those that depend on external plugins or very large codebases, have been ported to `https://github.com/coq-community/coq-performance-tests`, where we expect they will continue to be updated to work with the latest version of Coq.

B.1 Plots in Chapter 1, Background

We collected data for Figure 1-2 with Coq 8.8.2. Due to various changes in notation printing, the code-printing pipeline for this old version of Fiat Cryptography does not work correctly with Coq versions $\geq$ 8.9.

B.2 Plots in Chapter 2, The Performance Landscape in Type-Theoretic Proof Assistants

Figure 2-1 is the same as Figure 1-2 which was discussed in Appendix B.1. As in Figure 2-1 and for the same reasons, we collected data for Figure 2-2 with Coq 8.8.2.

We collected data for Figures 2-6 and 2-7 with Coq 8.9.1 because Coq 8.10 and later

do not show the relevant superlinear behavior due to Coq PR #9586.

B.3 Plots in ??, ??

All plots in **??** and its appendix (**??**) were constructed using measurements from Coq 8.10.0. Gathering the data for these plots takes over a week, and we were loathe to repeatedly rerun the benchmarks using newer versions of Coq as they came out.

[**TODO:** Run ‘make update-thesis’ before submission to update the date on the cover page]

[**TODO:** Update resume submodule before submission of forms]

[**TODO:** change \finalfalseto \finaltrue]