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

# Performance Engineering of Proof-Based Software Systems at Scale

by

Jason S. Gross

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
August 19, 2015

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

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Leslie A. Kolodziejski
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 August 19, 2015, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

## Abstract

Formally verified proofs are important. Unfortunately, large-scale proofs, especially automated ones, in Coq, can be quite slow.

This thesis aims to be a partial guide to resolving the issue of slowness. We present a survey of the landscape of slowness in Coq, with a number of micro- and macro-benchmarks. We describe various metrics that allow prediction of slowness, such as term size, goal size, and number of binders, and note the occasional surprise-lack-of-bottleneck for some factors, such as total proof term size.

We identify three main categories of workarounds and partial solutions to slowness: design of APIs of Gallina libraries; changes to Coq's type theory, implementation, or tooling; and automation design patterns, including proof by reflection. We present lessons drawn from the case-studies of a category-theory library, a proof-producing parser-generator, and a verified compiler and code generator for low-level cryptographic primitives.

The central new contribution presented by this thesis, beyond hopefully providing a roadmap to avoid slowness in large Coq developments, is a reflective framework for partial evaluation and rewriting which, in addition to being used to compile a code-generator for field arithmetic cryptographic primitives which generates the code currently used in Google Chrome, can serve as a template for a possibly replacement for tactics such as `rewrite`, `rewrite_strat`, `autorewrite`, `simpl`, and `cbn` which achieves much better performance by running in Coq's VM while still allowing the flexibility of equational reasoning.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

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

# Acknowledgments

Thank you, Mom, for encouraging me from my youth and supporting me in all that I do. Last, and most of all, thank you, Adam Chlipala, for your patience, guidance, advice, and wisdom, during the writing of this thesis, and through my research career.

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 2.2.1 and those of subsections Name Resolution, Capture-Avoiding Substitution, Quadratic Creation of Substitutions for Existential Variables, and Quadratic Substitution in Function Application in Subsection 2.2.3.

This work was supported in part by the MIT bigdata@CSAIL initiative, NSF grant CCF-1253229, ONR grant N000141310260, and AFOSR grant FA9550-14-1-0031. We also thank Benedikt Ahrens, Daniel R. Grayson, Robert Harper, Bas Spitters, and Edward Z. Yang for feedback on "Experience Implementing a Performant Category-Theory Library in Coq" [GCS14].

A significant fraction of the text of this thesis 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 thesis. In particular: Sections 3.3, 3.4, 3.5.1, 3.5.2 and 3.5.3 are taken from "Experience Implementing a Performant Category-Theory Library in Coq" [GCS14]. Some of the text in Subsections 7.2.1 and 7.3.1 comes from "Experience Implementing a Performant Category-Theory Library in Coq" [GCS14]

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

Sections 6.1 and 6.2 is based on the introduction to Gross, Erbsen, and Chlipala [GEC18]. Chapter 6 is largely taken from Gross, Erbsen, and Chlipala [GEC18], with some new text for this thesis.

We would like to thank read and cite Karl Palmskog for pointing us at Lamport and Paulson [LP99] and Paulson [Pau18].[1]

Acknowledgements from the category theory paper: This work was supported in part by the MIT bigdata@CSAIL initiative, NSF grant CCF-1253229, ONR grant N000141310260, and AFOSR grant FA9550-14-1-0031. We also thank Benedikt Ahrens, Daniel R. Grayson, Robert Harper, Bas Spitters, and Edward Z. Yang for feedback on Gross, Chlipala, and Spivak [GCS14], on which we based Chapter 3.

Acknowledgements from the reification by parametricity paper: We would like to thank Hugo Herbelin for sharing the trick with `type of` to propagate universe con-

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

straints[2] as well as useful conversations on Coq's bug tracker that allowed us to track down performance issues.[3] We would like to thank Jonathan Leivent for sharing a trick which obsoletes the aforementioned `type of` trick[4]. We 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. We 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. We also thank John Wiegley for feedback on Gross, Erbsen, and Chlipala [GEC18], which is included in slightly-modified form distributed between Chapter 6 and various sections of Chapter 4.

For those interested in history, our method of reification by parametricity 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; $\beta$-reduction; reification, we could move $\beta$-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.

[2] `https://github.com/coq/coq/issues/5996#issuecomment-338405694`
[3] `https://github.com/coq/coq/issues/6252`
[4] `https://github.com/coq/coq/issues/5996#issuecomment-670955273`

# 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]

premature optimization is the root of all evil

— Donald E. Knuth [Knu74a, p. 671]

# Chapter 1

# Introduction

## 1.1 Introduction

As we come to increasingly rely on large and complex software systems, bugs in these systems have an ever more severe and detrimental impact on our world. While small bugs can cost hours or days of developer time, other bugs have more severe costs in money or even lives [ZC09]. Intel lost about $475 million when it recalled chips impacted by a minor bug in the lookup table implementing floating-point division [Gaw14; Hal95; Nic11]. NASA and other space agencies lost hardware costing billions of US dollars due to bugs in rocket software that created unrecoverable errors [Llo99; JM97; Dow97; Lee+96]. Finally, the Therac-25 radiation therapy machine killed three people and injured more by delivering dangerous doses of radiation [BH18, p. 434; LT93].

While testing of various kinds is the de facto standard for finding bugs, it is both quite costly and does not guarantee a lack of bugs. In adversarial contexts, fixing even 99% of bugs is not sufficient; there is a strong incentive for the remaining 1% of bugs to be found and exploited. In some such contexts, random testing cannot hope to eliminate all bugs. For example, some bugs in cryptographic code only occur in as few as 19 out of $2^{255}$ cases [Lan14]. Let us envision what numbers this big actually mean. Suppose you wanted to catch this bug in a "modest" twenty years of continuous random testing. You'd need to have over a thousand times as many computers as there are atoms in the solar system! While making our computers sufficiently faster might seem to solve this problem, in fact it gets us nowhere! Ensuring security would require scaling up the size of the cryptographic problem by a similar amount—otherwise attackers could use these faster computers to brute force the secrets—and we would be no closer to being able to exhaustively test for bugs.

An appealing solution to this problem is to *prove* critical software correct. We do this by specifying in formal mathematics the intended behavior of our software. We then show a correspondence between our math and our code. Ideally, the specification will be relatively simple, and much easier to trust, than the millions of lines of code making up the software we use every day.

While proofs of algorithms tend to be done with pen-and-paper (consider the ubiquitous proofs that various sorting algorithms are correct in introductory algorithms classes), proofs of actual code are much harder. Code in the real-world tends to be much more complicated than code in an algorithms textbook. This difference could be due to reasons of performance, historical accident, or interfacing with real users, among others. Proofs of code-correctness thus tend to be filled with tedious case-analysis with 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 programs in the first place.

Enter machine-checked proofs, a partial solution to the problem of human fallibility. Foundational tools allow users to write down theorems and proofs that are then checked by machine. These tools free the proof author from error-prone case analysis, as the computer will check that no cases were missed.

Even with machine-checked proofs, it is impossible to gain complete confidence in your software; the layer underneath may be untrustworthy in ways you cannot see [Tho84], you can never prove that the mathematical system you're working in is correct [Raa20], and most proof-checkers are not yet themselves proven correct.

However, machine-checked proofs allow a drastic reduction in how much code needs to be trusted. Rather than trusting the millions of lines of code of the software being verified, you only need to trust the specifications of the software, and the likely-much-smaller codebase of the proof checker. Furthermore, it's very unlikely that bugs in the proof checker will overlap with bugs in the proof in just the right way to allow errors to slip through only in one particular piece of software. Said another way, we no longer need to write tests for every piece of software separately; writing tests for the proof-checker helps eliminate bugs in *all* software proven by that checker.

We come now to the main thrust of this thesis. Tools for machine-checked proofs have already 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]. 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 focused 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 1.1.1, 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.

Why is this problem worthy of study? This thesis argues that the problem of *compile-time performance* or *proof-checking performance* is both significantly different from the problem of performance of typical programs and that it is nontrivial. 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.

Unlike many other performance domains, in our experience, the time it takes to check proofs as we scale the size of the input is almost always super-linear—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 2.1 for more details. 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 *typical*. [**QUESTION FOR READERS:** Is this paragraph in particular readable? Any comments on design choices here?]

Drawing experience from case studies in category theory [GCS14], parsers [Gro15a], and generation of low-level cryptographic code [Erb+19], we investigate and seek to explicate a broad swath of performance issues and bottlenecks. While we propose some design principles for avoiding these performance issues in Part II and present a research prototype of a tool and methodology for achieving acceptable performance at scale in Part III—and we claim both of these are original contributions of this PhD—the story that this thesis aims to tell is that in order for proof assistants to scale to industrial uses, *we must get the basics of asymptotic performance right.*

### 1.1.1 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 $\iff$), 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 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, however. 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; Wik20b], 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; KM20], 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; Wik20a]. 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 1.2.1. The legacy of Automath also includes de Bruijn indices, a method for encoding function arguments which we describe in Section 4.1.3, dependent types, which we explain in Subsection 1.2.1, and the *de Bruijn principle*—stating that proof checkers should be as small and as simple as possible—which we discuss in Subsection 1.2.2 [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 thesis, 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].

Major accomplishments of verification in Coq [Coq20] 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 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 had already 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], 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; HTt14]. 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 [KM20; ]. Other proof assistants include LF [Pfe91; HHP93; Pfe02], Twelf [PS99], Matita [Asp+07; Asp+11], PVS [Sha96; ORS92], LEGO [Pol94], and NuPRL [Con+86].

## 1.2 Basic Design Choices

Although the design-space of proof assistants is quite large, as we've touched on in Subsection 1.1.1, 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 [Wik20a; 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.

### 1.2.1 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 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 types with the set of terms of that type, identifiers propositions 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 of objects/terms/proofs | 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 ($\Pi$) | | universal quantification ($\forall$) |
| dependent pair type ($\Sigma$) | | 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 non-trivial 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 type depends on the *value* of the function's 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 non-dependently-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 functions take in one argument at a time, and return 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 pairs of numbers $n$ paired with proofs 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 4.

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].

### 1.2.2 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 unreasonable 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 thesis, 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.

## 1.3 Look ahead: Layout and contributions of the thesis

In the remainder of Part I, we will finish laying out the landscape of performance bottlenecks 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 super-linear performance bottlenecks in Section 2.2 (The Four Axes).

Part II (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 1.2.1; in Chapter 3 (Design-based fixes), we discuss lessons on engineering libraries at scale drawn from our case study in formalizing category theory and augmented by our other experience. The category theory library formalized as part of this doctoral work, available at *HoTT/HoTT Categories* [], 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 III (Program Transformation and Rewriting) is devoted, in some sense, to performance bottlenecks that arise from the de Bruijn criterion of Subsection 1.2.2. We investigate one particular method for avoiding these performance bottlenecks. We introduce this method, variously called *proof by reflection* or *reflective automation*, in Chapter 4 (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 5 (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 thesis, Chapter 5 at times assumes a great deal of familiarity with the details of the Coq proof assistant. Finally, Chapter 6 (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 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 The Story

The purpose of this chapter is to convince the reader that the issue of performance in proof assistants is non-trivial in ways that differ from performance bottlenecks in non-dependently-typed languages. I intend to do this by first sketching out what I see as the main difference between performance issues in dependently-typed proof assistants vs performance issues in other languages, and then supporting this claim with a palette of real performance issues that have arisen in Coq.

The widespread commonsense in performance engineering [**commonsense-perf-engineering-order-o** is that good performance optimization happens in a particular order: there is no use micro-optimizing code if you are 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 you're solving 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 essentially by drawing lines from the viewpoint to the object and seeing if it passes through any other object "in front of" it. Alternatively, one could 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 you need to know is "which parts of which objects need to be drawn on the screen?"

[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.

One essential ingredient to allowing 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 input looks like, and what the scope looks like. In Coq, and other dependently-typed proof assistants, this ingredient is missing. 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 proof assistants, 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 incur a large compile-time performance overhead. However, I claim that, in most languages, even as you scale your program, 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 if you write code that compiles in a reasonable amount of time, as you scale up your codebase, your compile time will slowly creep up as well.

In Coq, however, the scaling story is very different. 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 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 input size a bit more, the proof-checking time will be noticeably slow—maybe it now takes a couple of minutes. Scaling the input just a tiny bit more, though, will result in the compiler not finishing even if you let it run for a day or more. This is what working in an exponential performance domain is like.

To put numbers on this, a project I was working on [Erb+19] 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—slightly unrealistic—example used 17. The smallest toy example—two machine words—took about 14 seconds. Based on the the compile-time performance of about a hundred examples, we expect the largest

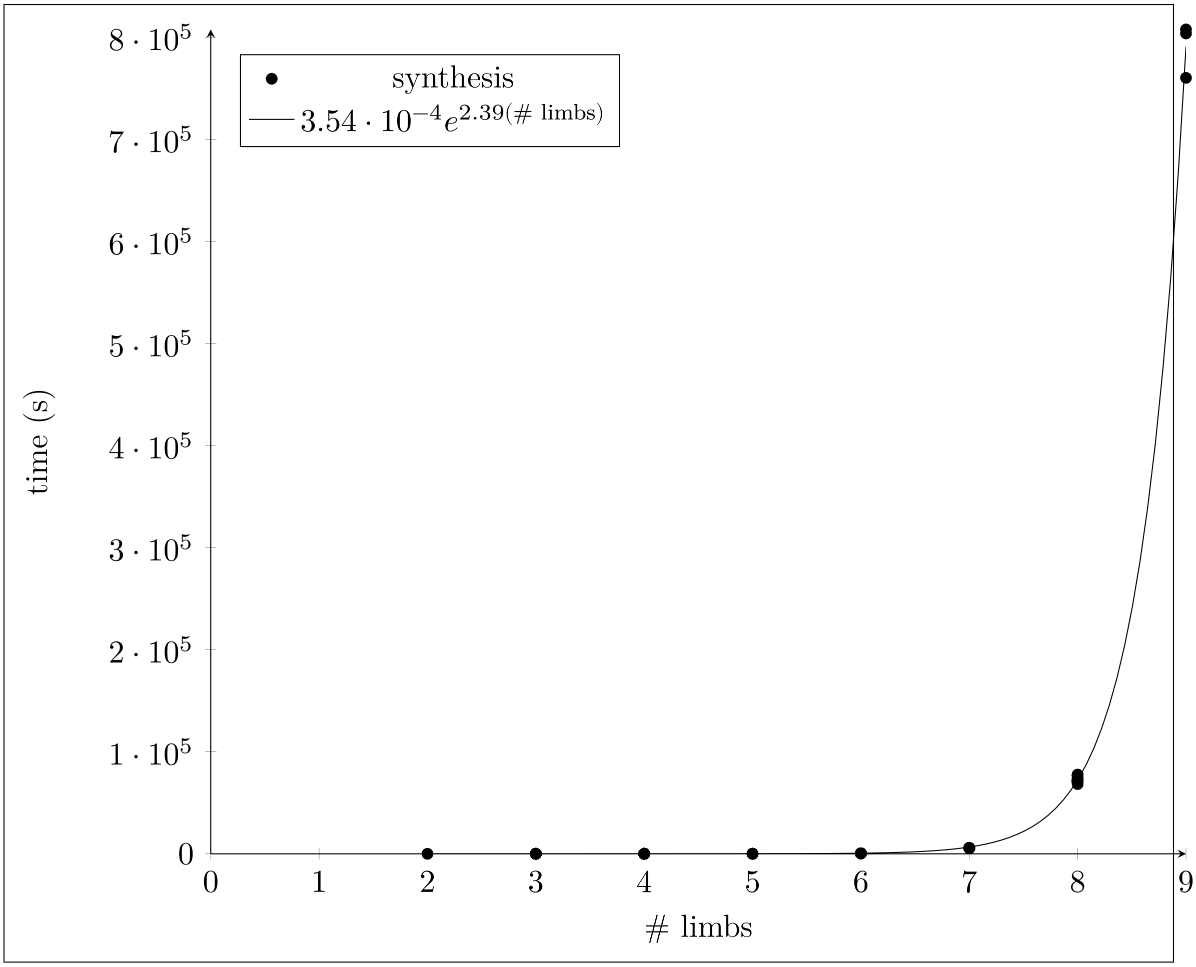



Figure 2-1: Timing of synthesizing subtraction

example—17 machine words—would have taken over four thousand *millennia*! See Figure 2-1 and Figure 2-2. (Our primary non-toy 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.)

Maybe, you might ask, were we generating unreasonable amounts of code? Each example using $n$ machine words generated $3n$ lines of code. Furthermore, 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). How can this be?

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 this is actually somewhat common; if you have a theorem that says "any code that comes out of this code generator satisfies this property", you need a proof that the code you 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

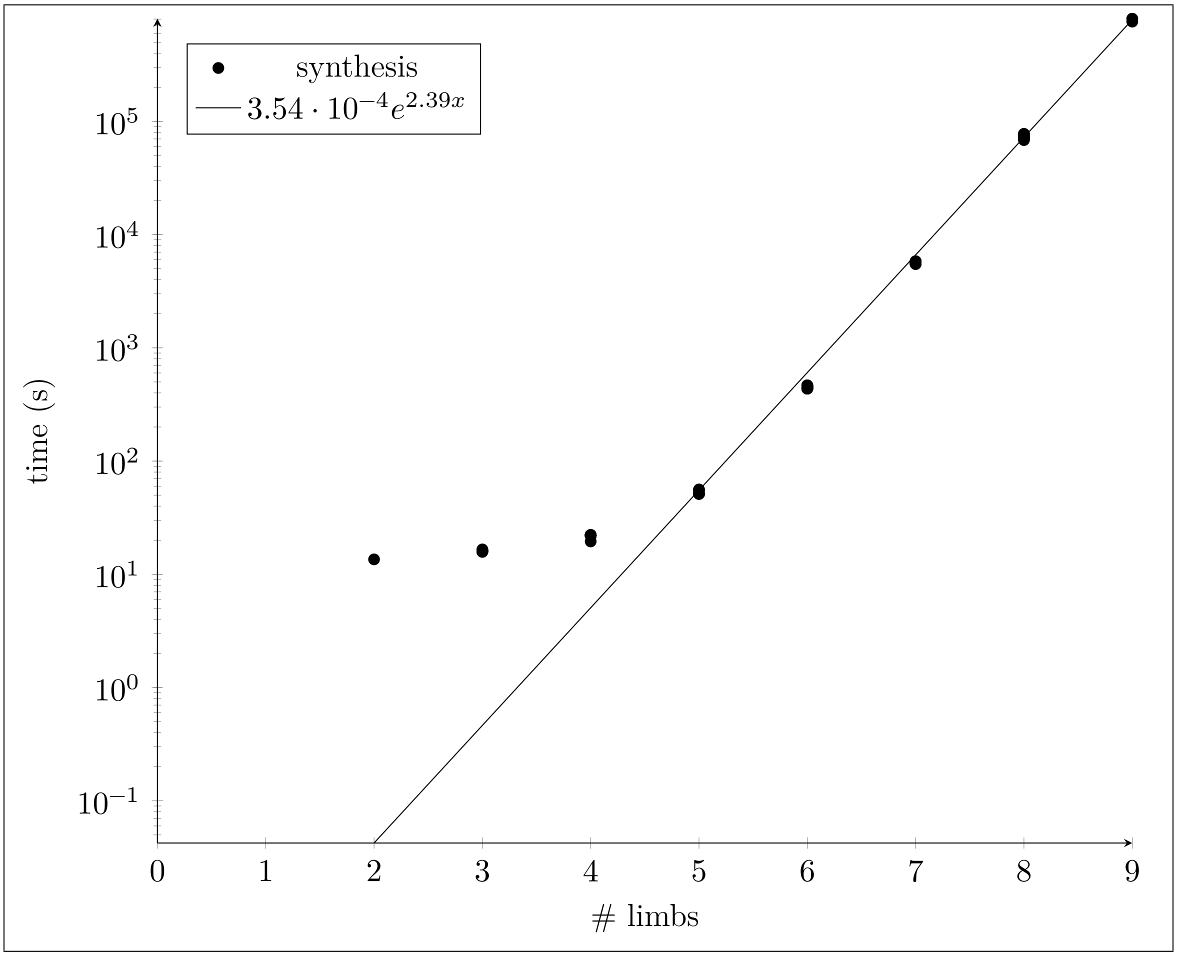



Figure 2-2: Timing of synthesizing subtraction (log-scale)

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 you often don't want to fully compute results when checking that two things 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.

The fix itself is quite simple, 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. But 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! 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

[2]`Strategy 1 [Let_In].` for those who are curious.
[3]It's 200, for those who are curious [Fre17].

it's generally assumed that users will get *line-by-line interactive feedback from the compiler* while developing, the numerous design decisions and seemingly 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 2-3 and Figure 2-4.

[**TODO:** Some sort of summary of argument-so-far here]

To finish off the argument about slowness in dependently-typed proof assistants, I want to present four axes of performance bottlenecks. These axes are by no means exhaustive, but, in my experience, most interesting performance bottlenecks scale as a super-linear factor of one or more of these axes.

# Misc Fragments

[**TODO:** Find a place for this (h/t conversation with Andres)]: because we have a kernel and a proof engine on top of it, you need to simultaneously optimize the kernel and the proof engine to see performance improvements; if the kernel API doesn't give you good enough performance on primitives, then there's no hope to optimizing the proof engine, but at the same time if the proof engine is not optimized right, improvements in the performance of the kernel API don't have noticeable impact.

[**TODO:** find a place for this:] In many domains, good performance optimization can be done locally. It's rarely the case that disparate parts of the codebase must be simultaneously optimized to see any performance improvement. However, in proof assistants satisfying the de Bruijn criterion, there are many seemingly reasonable implementation choices that can be made for the kernel which make performance-optimizing the proof engine 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

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

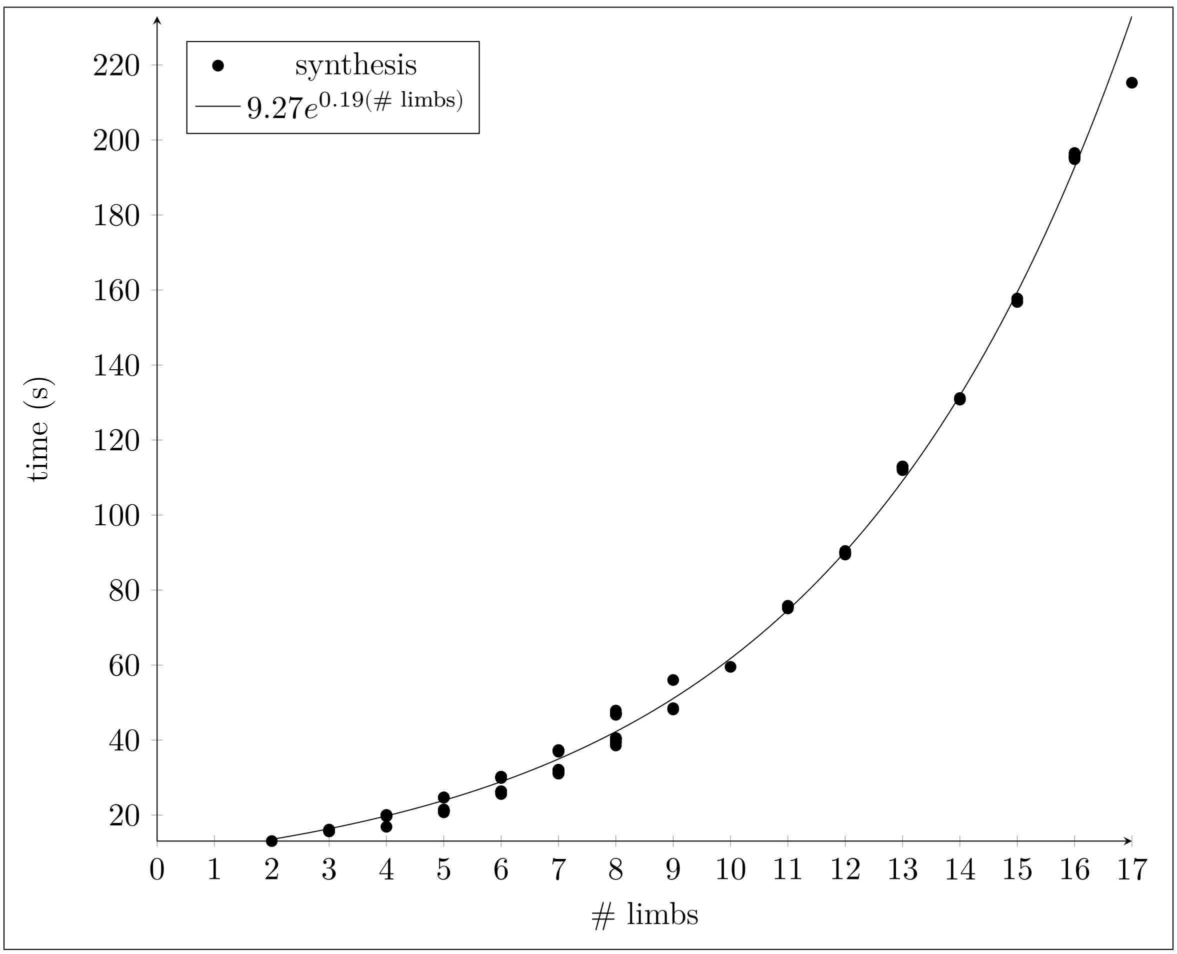



Figure 2-3: Timing of synthesizing subtraction after fixing the bottleneck

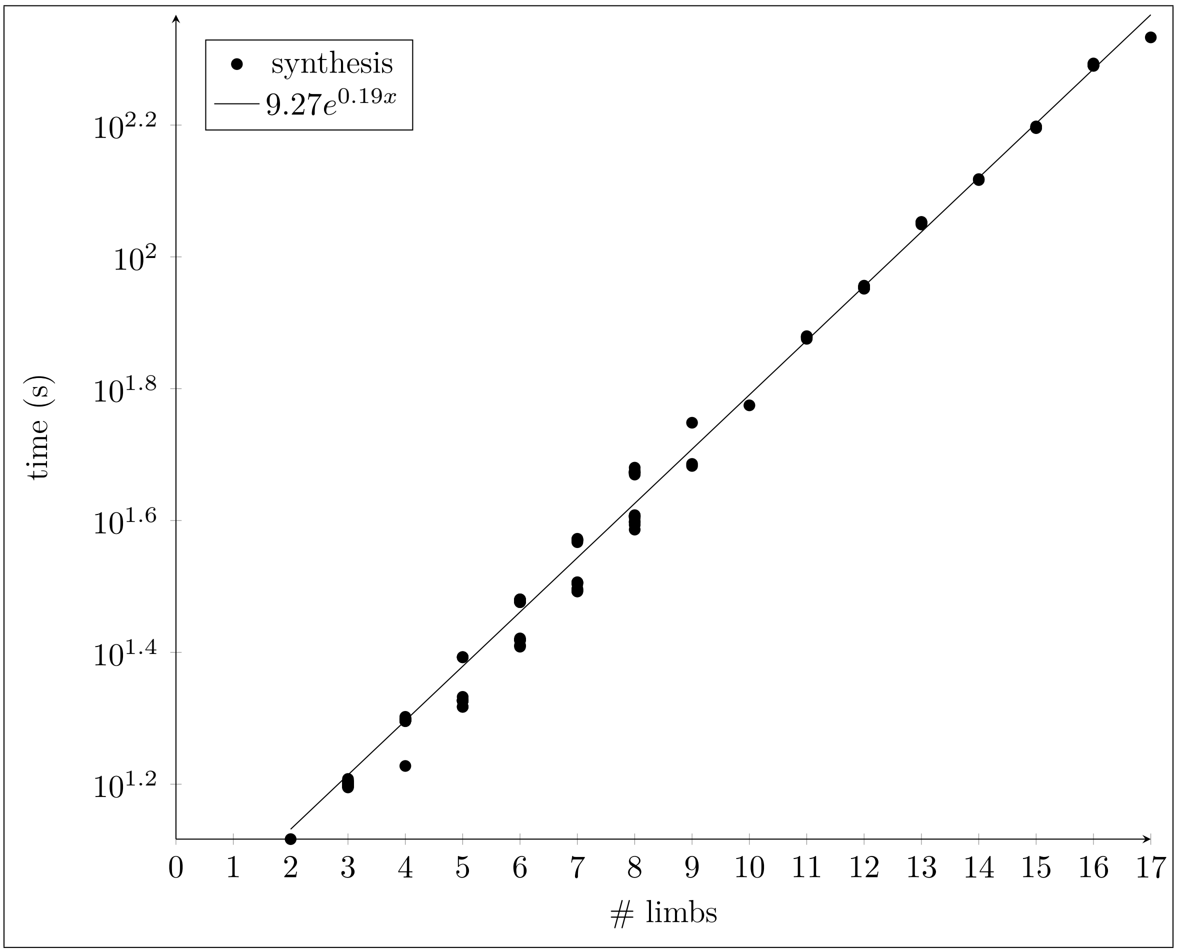



Figure 2-4: Timing of synthesizing subtraction after fixing the bottleneck (log-scale)

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.

[**TODO:** incorporate Andres' suggestions] I like the last two sentences. I would instead lead with something along the lines of "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." "in proof assistants, it doesn't look like anyone has even tried" :P. but e g signal processing was a huge mess too before the fast Fourier transform. coq abstractions are mostly accidents of history. no other system has a clear performance-conscious story for how these interfaces should be designed either.

## 2.2 The Four Axes

I now present four major axes of performance. These are not comprehensive, but after extensive experience with Coq, most performance bottlenecks scaled super-linearly as a function of at least one of these axes.

### 2.2.1 The Size of the Type

We start with one of the simplest axes.

Suppose we want to prove a conjunction of $n$ things, say, `True` $\wedge$ `True` $\wedge \cdots \wedge$ `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 1.2.2 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 you can already spot the problem.

To prove a conjunction of $n$ things, 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 2-5 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 2-6.

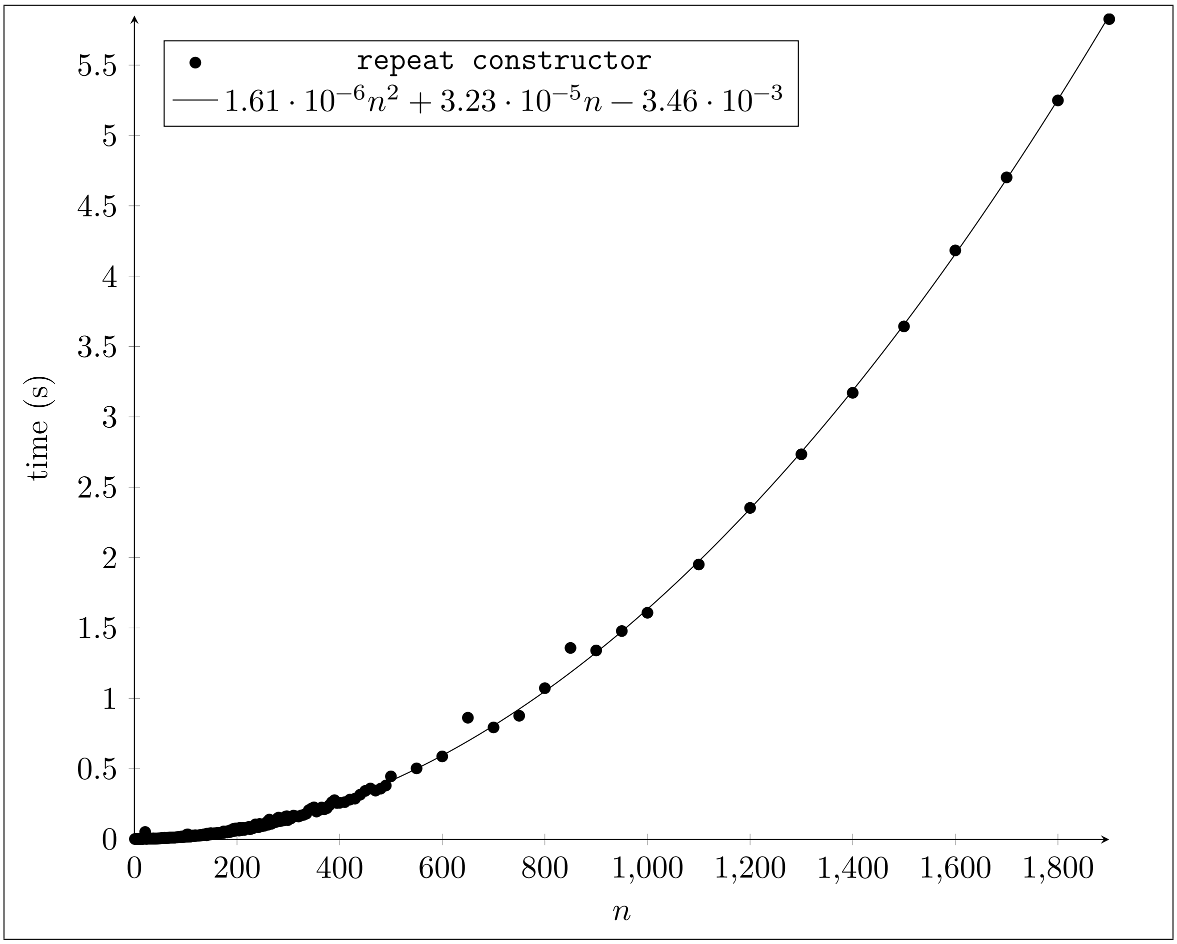



Figure 2-5: Timing of **`repeat constructor`** to prove a conjunction of $n$ `True`s

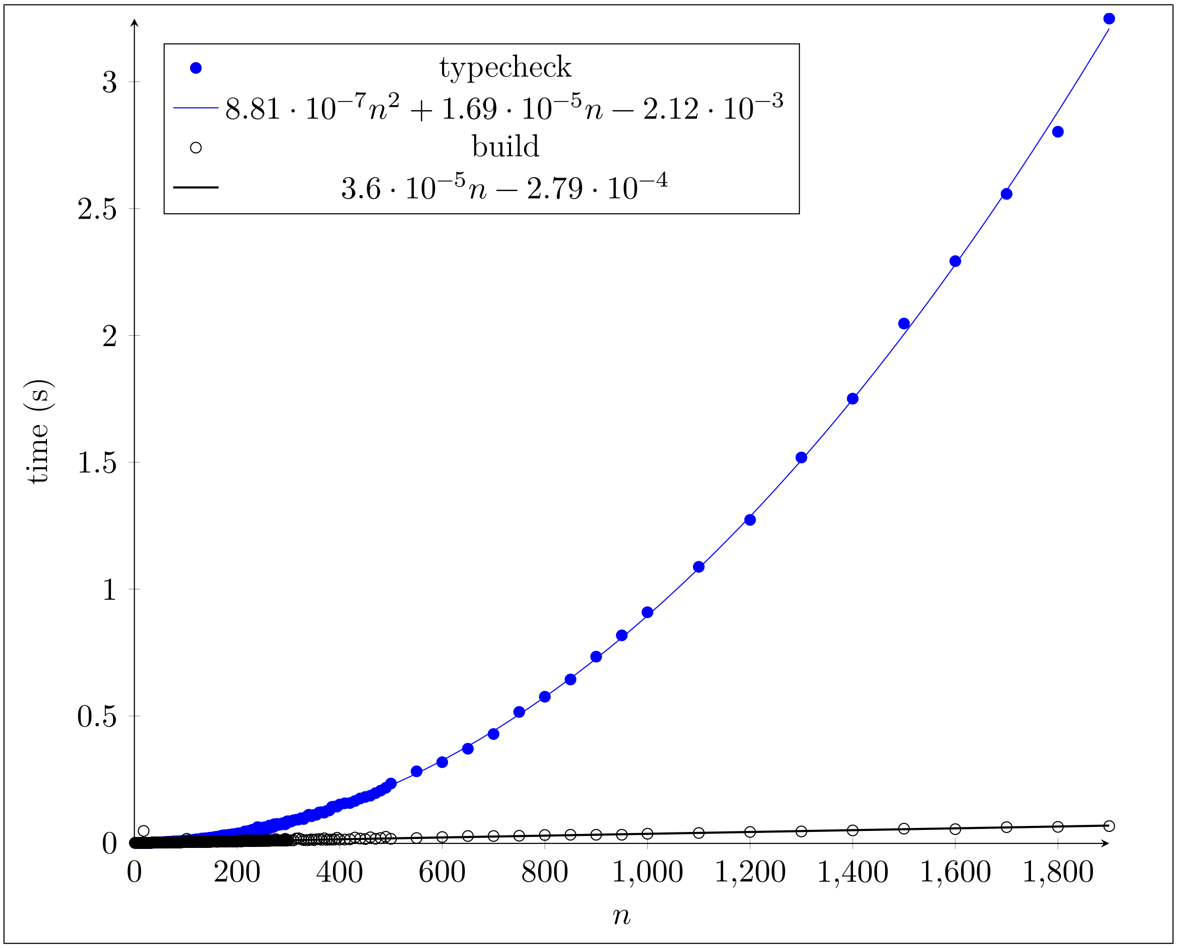



Figure 2-6: Timing of manually building and typechecking a certificate to prove a conjunction of $n$ `True`s using Ltac2

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 non-linear 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 non-unifiable 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 [**https://github.com/coq/coq/issues**
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 Tru
```

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

```
(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 2-7 and Figure 2-8.

Recall that the typing rules for $\lambda$ 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`

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

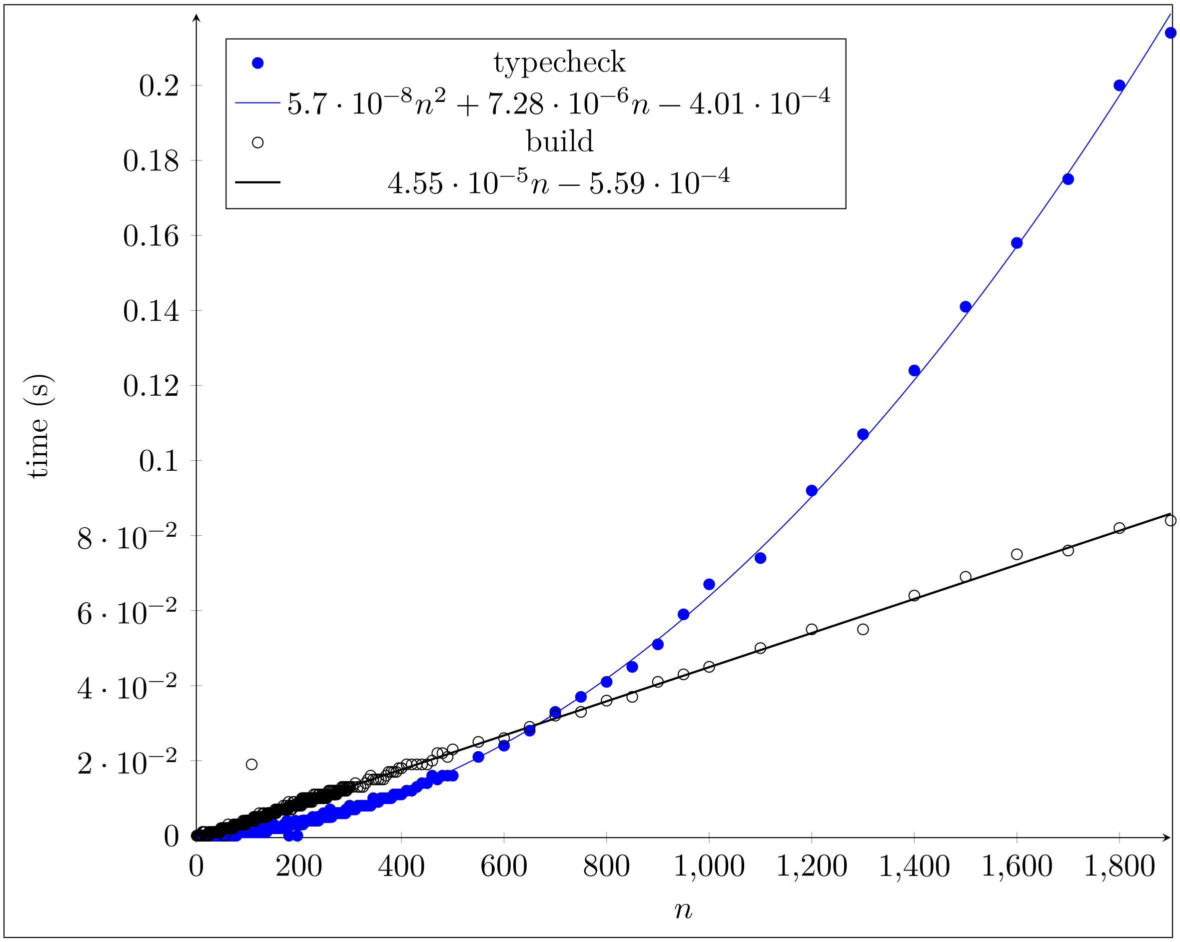



Figure 2-7: Timing of manually building and typechecking a certificate to prove a conjunction of $n$ `True`s using **`let`**-binders using Ltac2

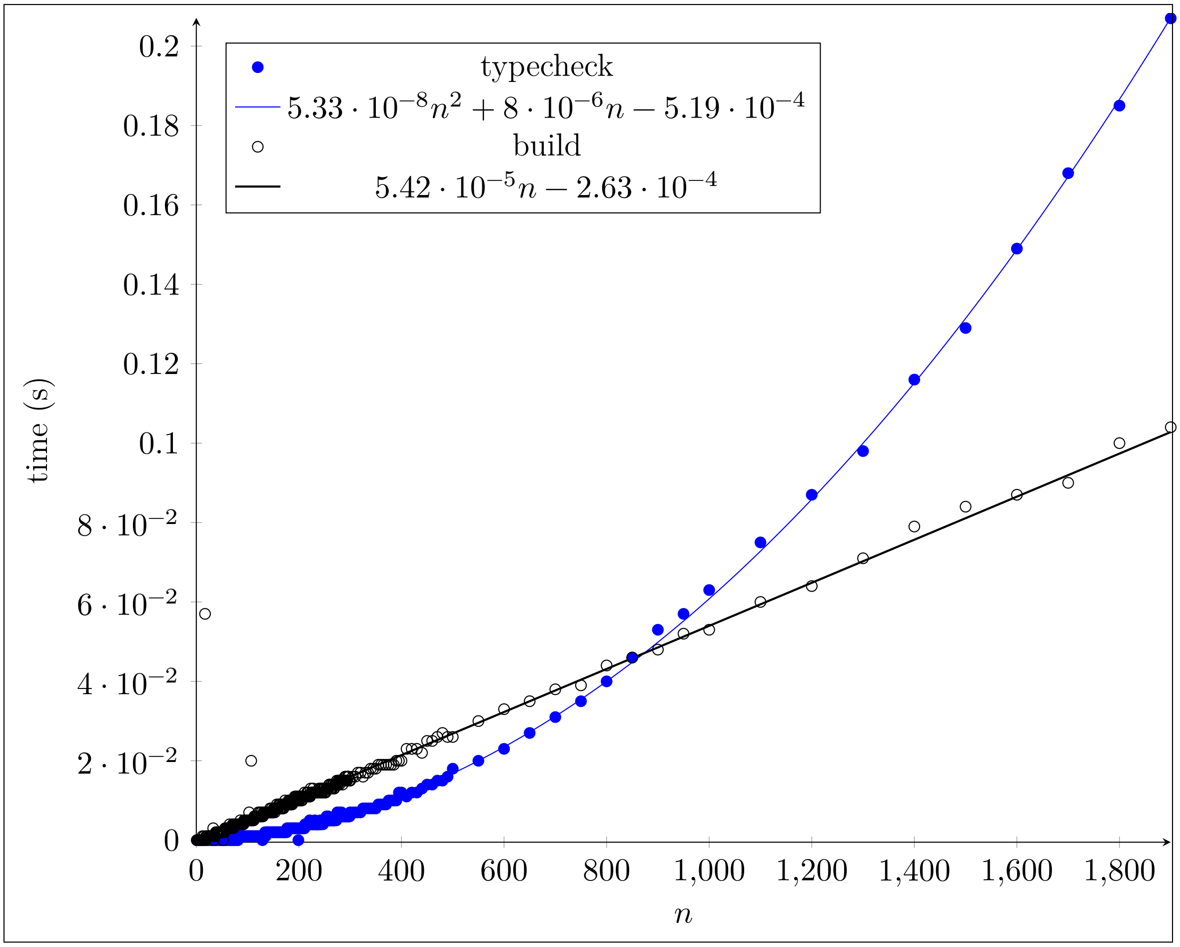



Figure 2-8: Timing of manually building and typechecking a certificate to prove a conjunction of $n$ `True`s using abstraction and application using Ltac2

- 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 being substituted, 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 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.

### 2.2.2 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.

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 2.2.1.

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 breath-first search, then we get speedy performance. However, if the typechecker instead unfolds `id` *last*, then we end up

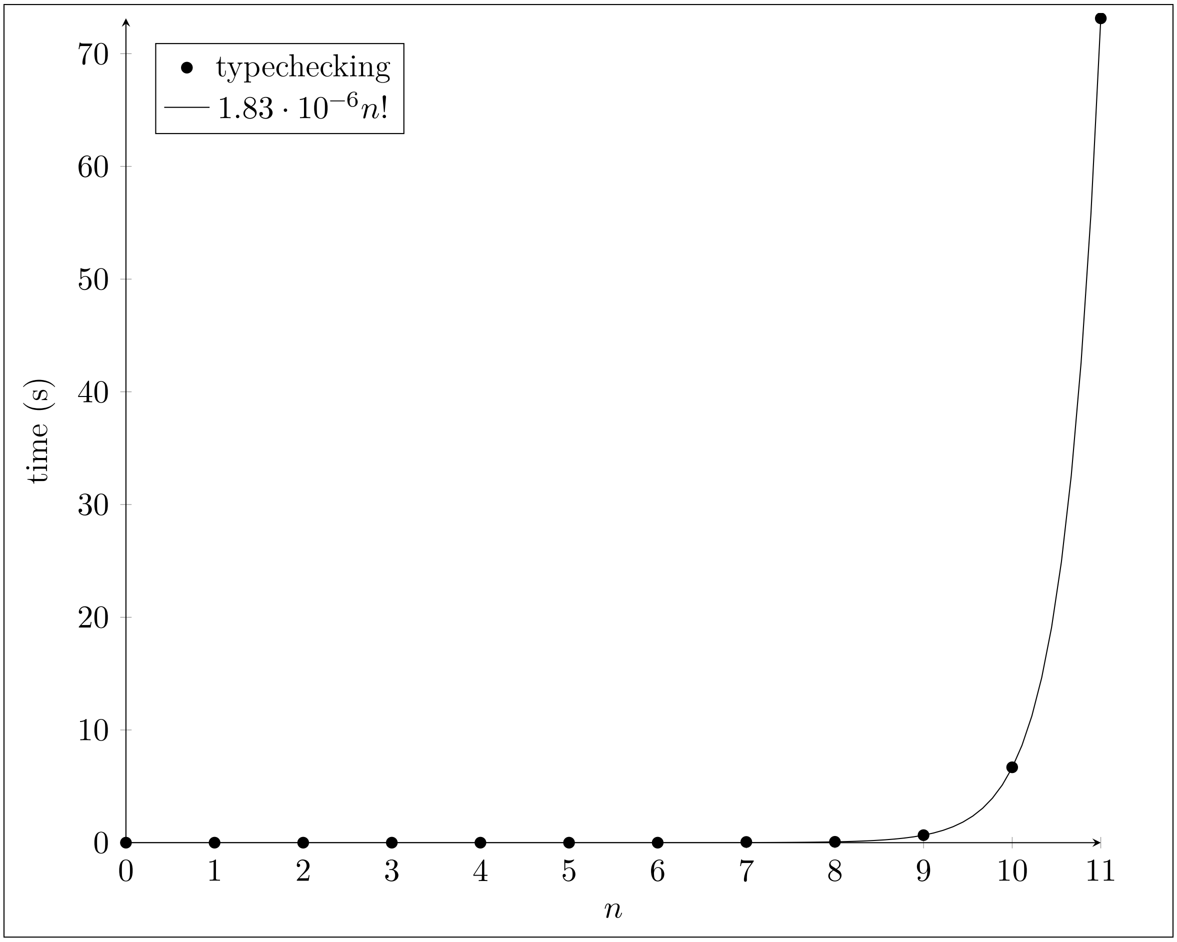


Figure 2-9: Timing of typechecking `@eq_refl nat (fact n) : @id nat (fact n) = fact n`

computing the normal form of 100!, which takes a long time and a lot of memory. See Figure 2-9.

Note that it is by no means obvious that the typechecker can meaningfully do anything about this. Breath-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.

As described in Section 2.1, in fiat-crypto, 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.

### 2.2.3 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 naively, 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 and 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 #9582 and #9586.

See Figure 2-10 for the timing of name resolution in Coq. See Figure 2-11 for the effect on internalizing a lambda with $n$ arguments.

#### 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 $\alpha$-renaming is even just linear in the existing context, then creating a new hole under $n$ binders will take $\mathcal{O}(n^2)$ time

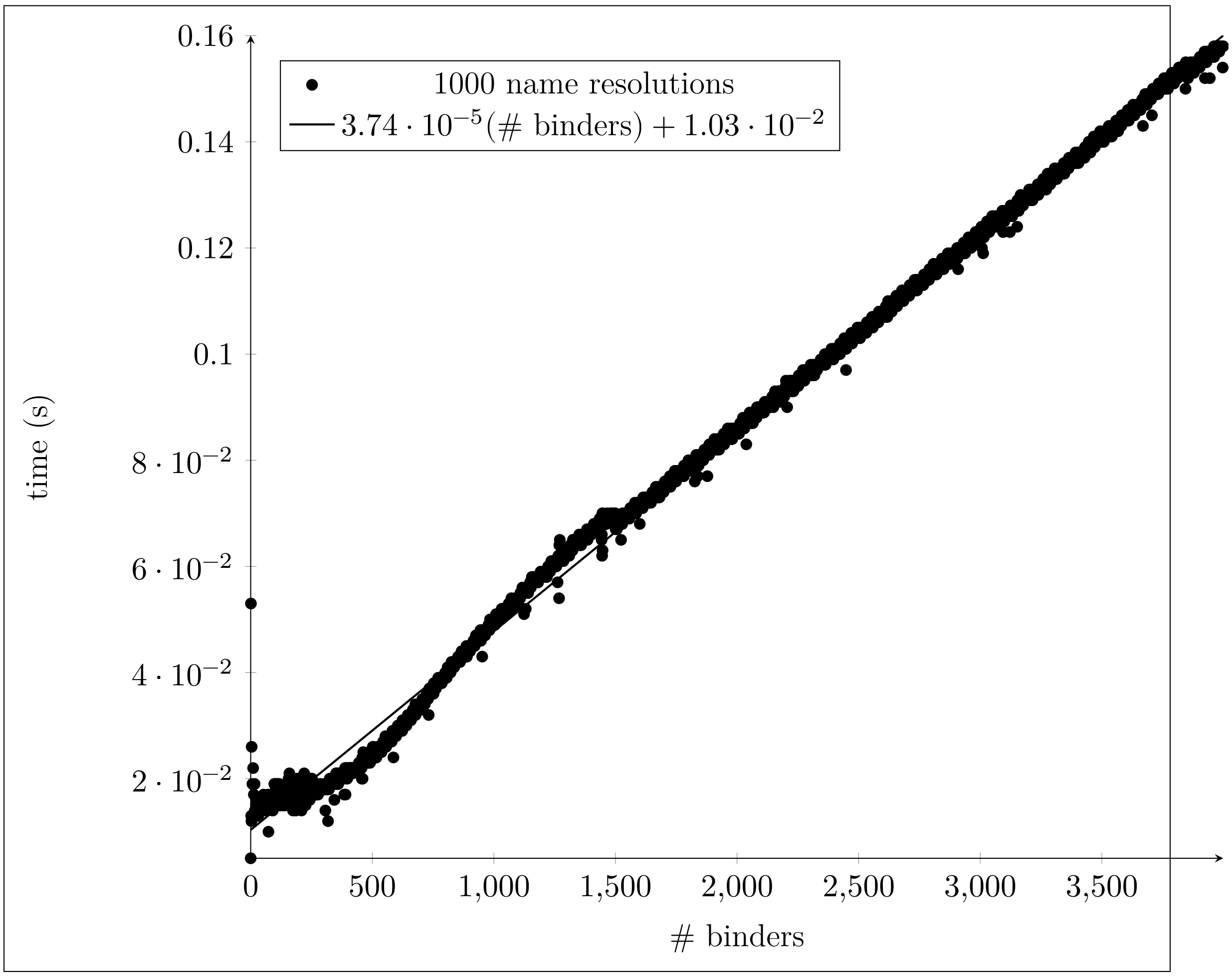



Figure 2-10: Timing of internalizing a name 1000 times under $n$ binders

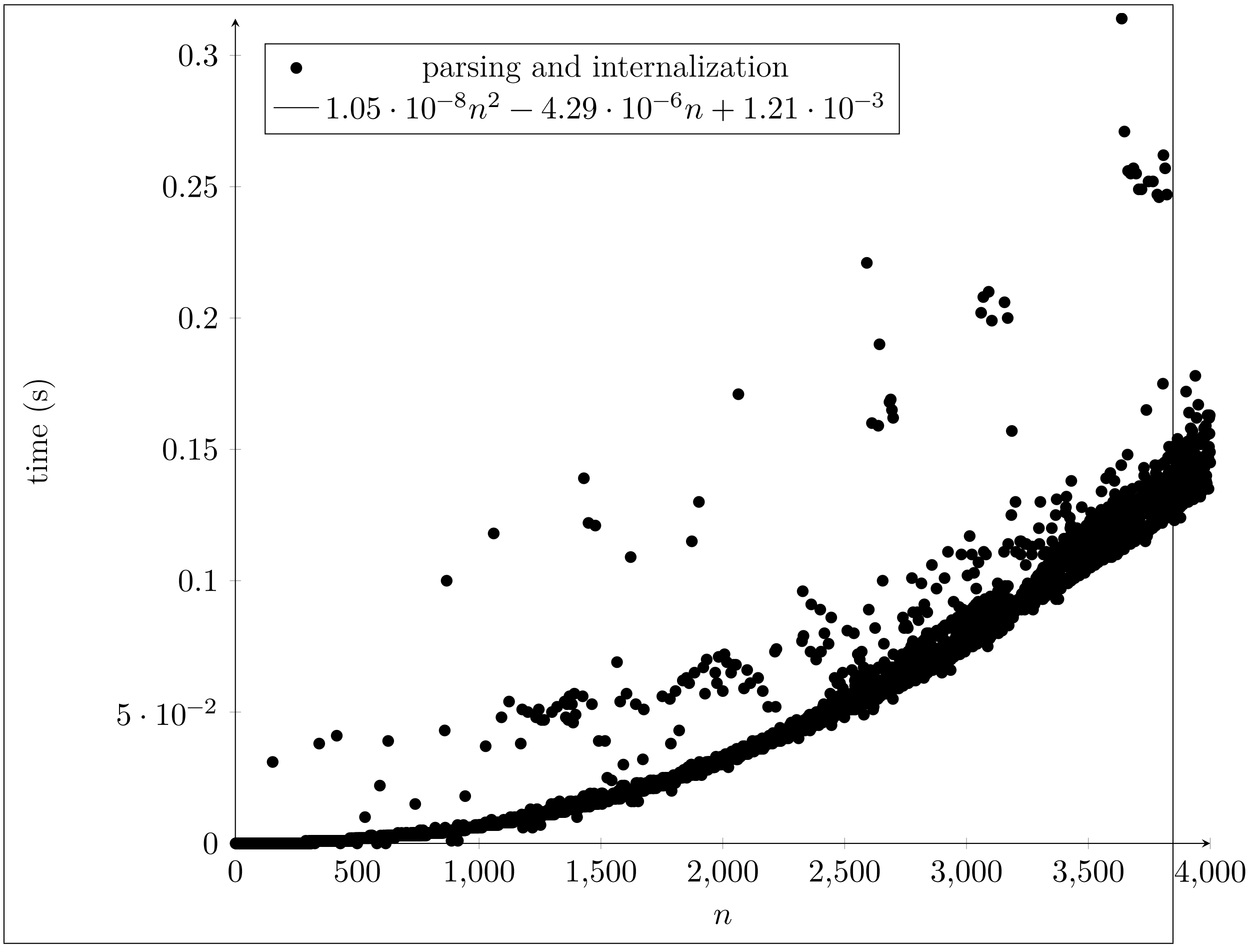



Figure 2-11: Timing of internalizing a function with $n$ differently-named arguments of type `True`

in the worst case, as we may have to do $n$ renamings, each of which take time $\mathcal{O}(n)$. See #9582, perhaps also #8245 and #8237 and #8231.

This might be the cause of the difference in Figure 2-13 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 existential variables, or evars, typically exist in a particular context. That is, you have access to some hypotheses but not others when filling an evar.

Sometimes, reduction results in changing the context in which an evar exists. For example, if we want to $\beta$-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 4.1.3. 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 $\lambda$ 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.

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

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

See also Figure 2-12 and Figure 2-13.

#### Quadratic Substitution in Function Application

Consider the case of typechecking a non-dependent function applied to $n$ arguments. If substitution is performed eagerly, following directly the rules of the type theory,

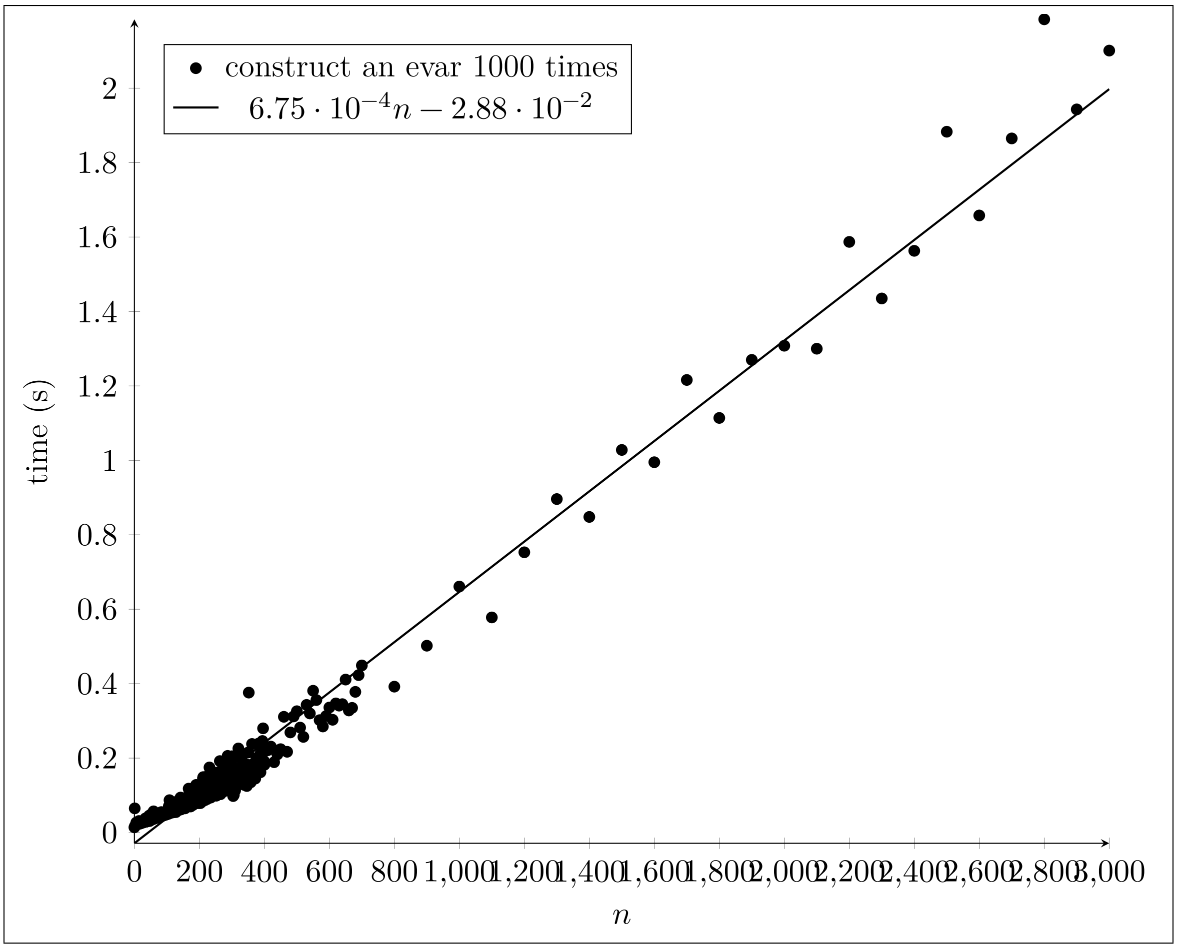



Figure 2-12: Timing of generating 1000 evars in a context of size $n$

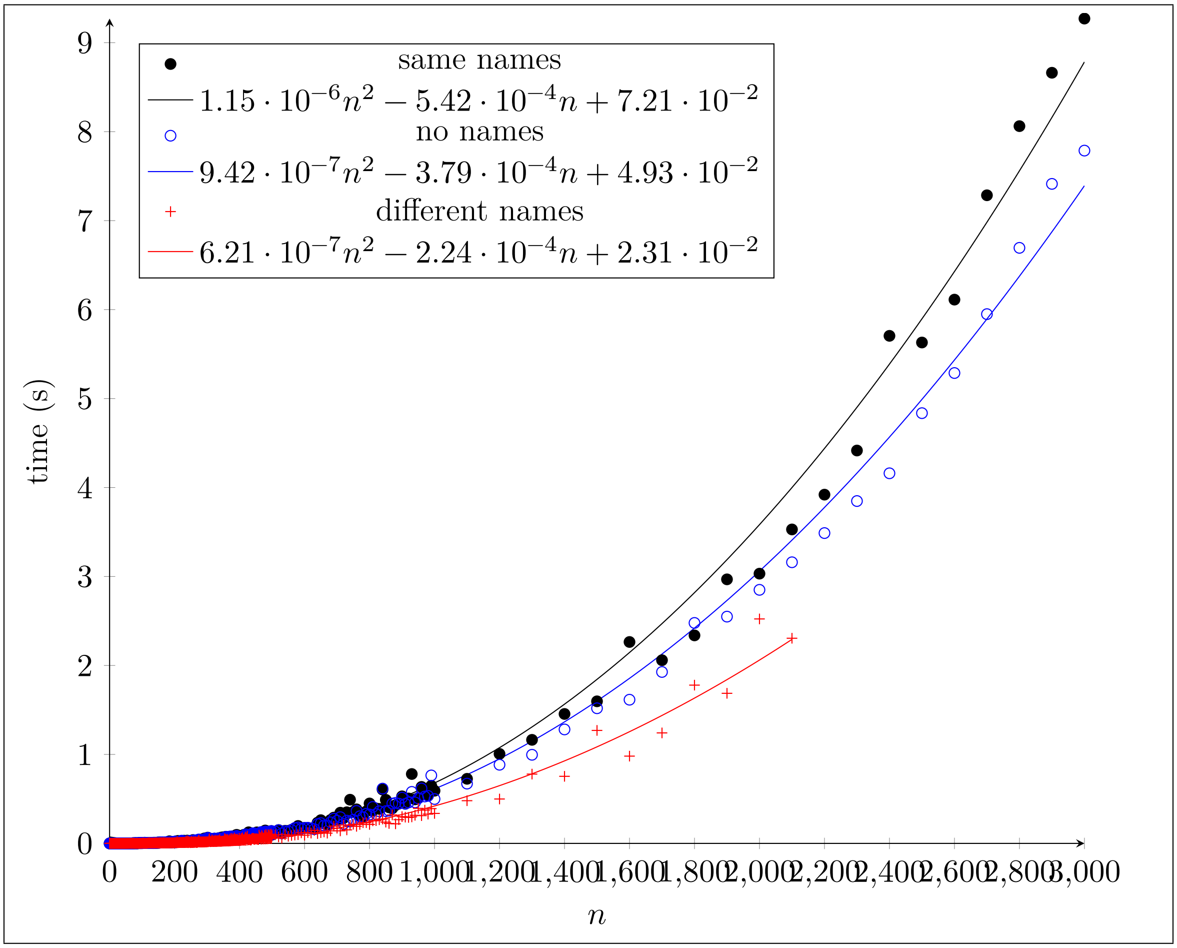



Figure 2-13: Timing of generating a $\lambda$ with $n$ binders of unknown/evar type, all of which have either no name, the same name, or different names

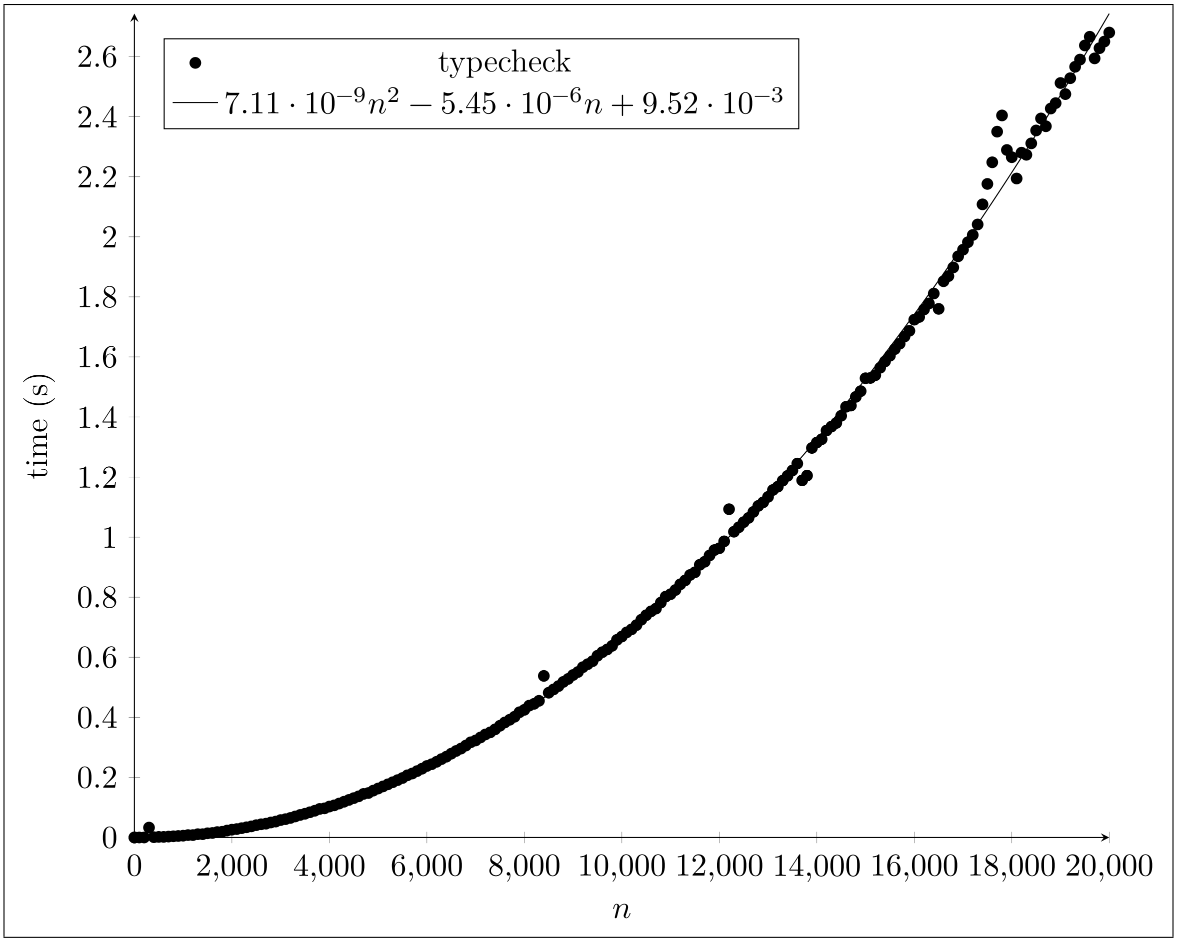


Figure 2-14: Timing of typechecking a function applied to $n$ arguments

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 exist, it's much harder to solve.

Note that this is important, for example, if you want 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.

See #8232 and #12118 and #8255.

See Figure 2-14.

### 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`**).

The details of implementing NbE depend on the term representation used. If a fancy term encoding like PHOAS, which we explain in Section 4.1.3, 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 you call it every time you 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 #11151 for an occurrence of this performance issue in the wild in Coq. See also Figure 2-15.

### 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 #11151 and #11964 and #7826 for an occurrence of this issue in the wild. Note

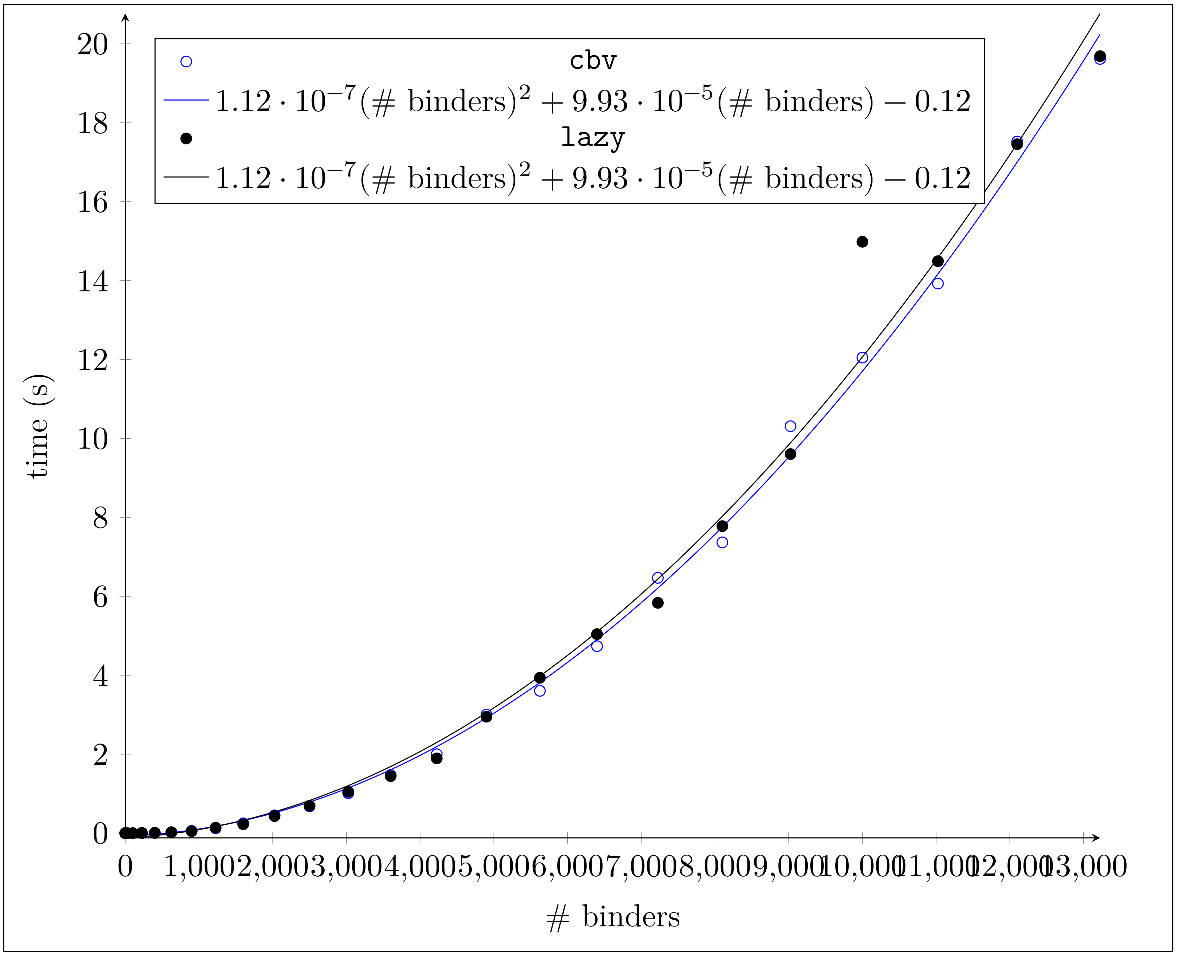



Figure 2-15: Timing of running **cbv** and **lazy** reduction on interpreting a PHOAS expression as a function of the number of binders

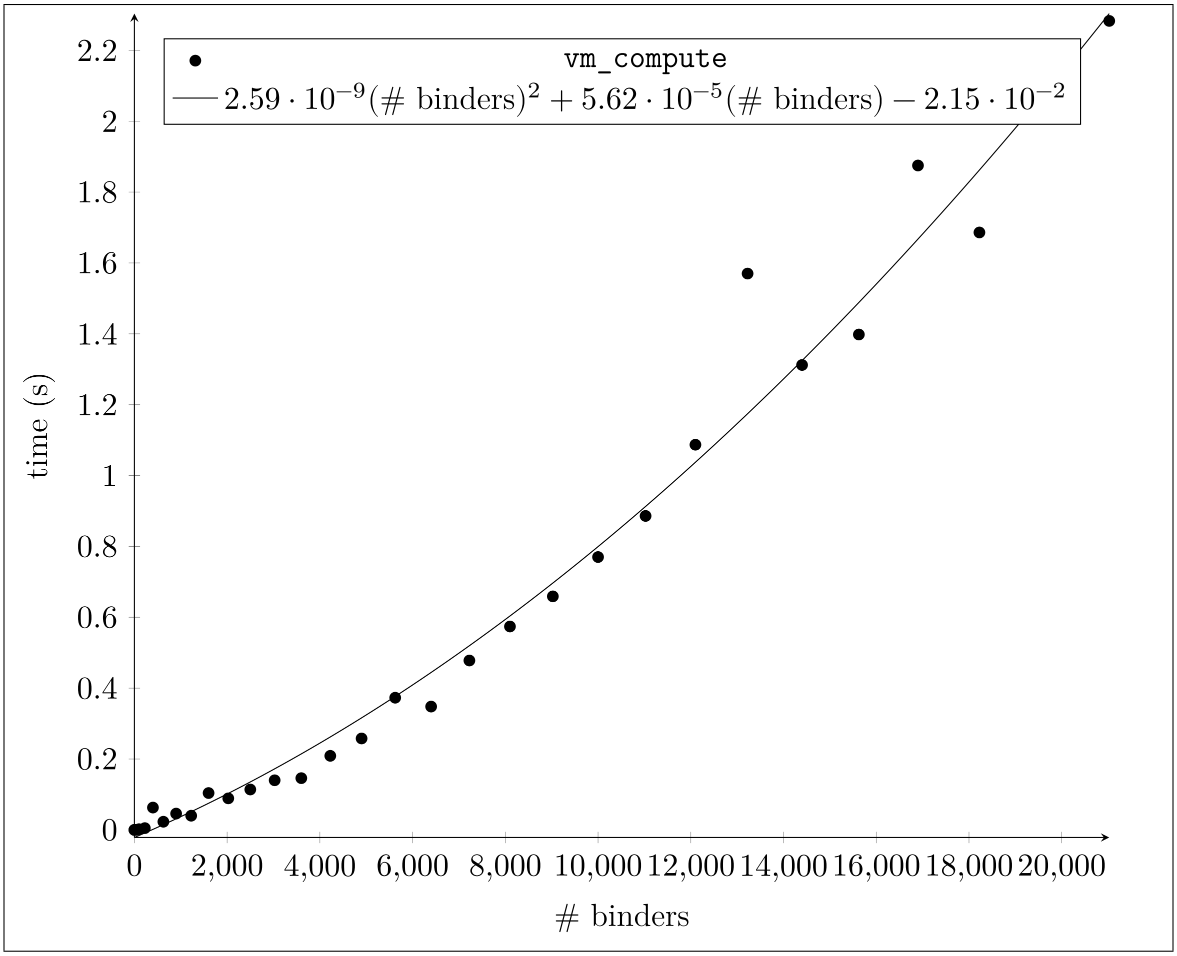



Figure 2-16: Timing of running **vm_compute** reduction on interpreting a PHOAS expression as a function of the number of binders

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 meta-programming 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 2-16 and Figure 2-17.

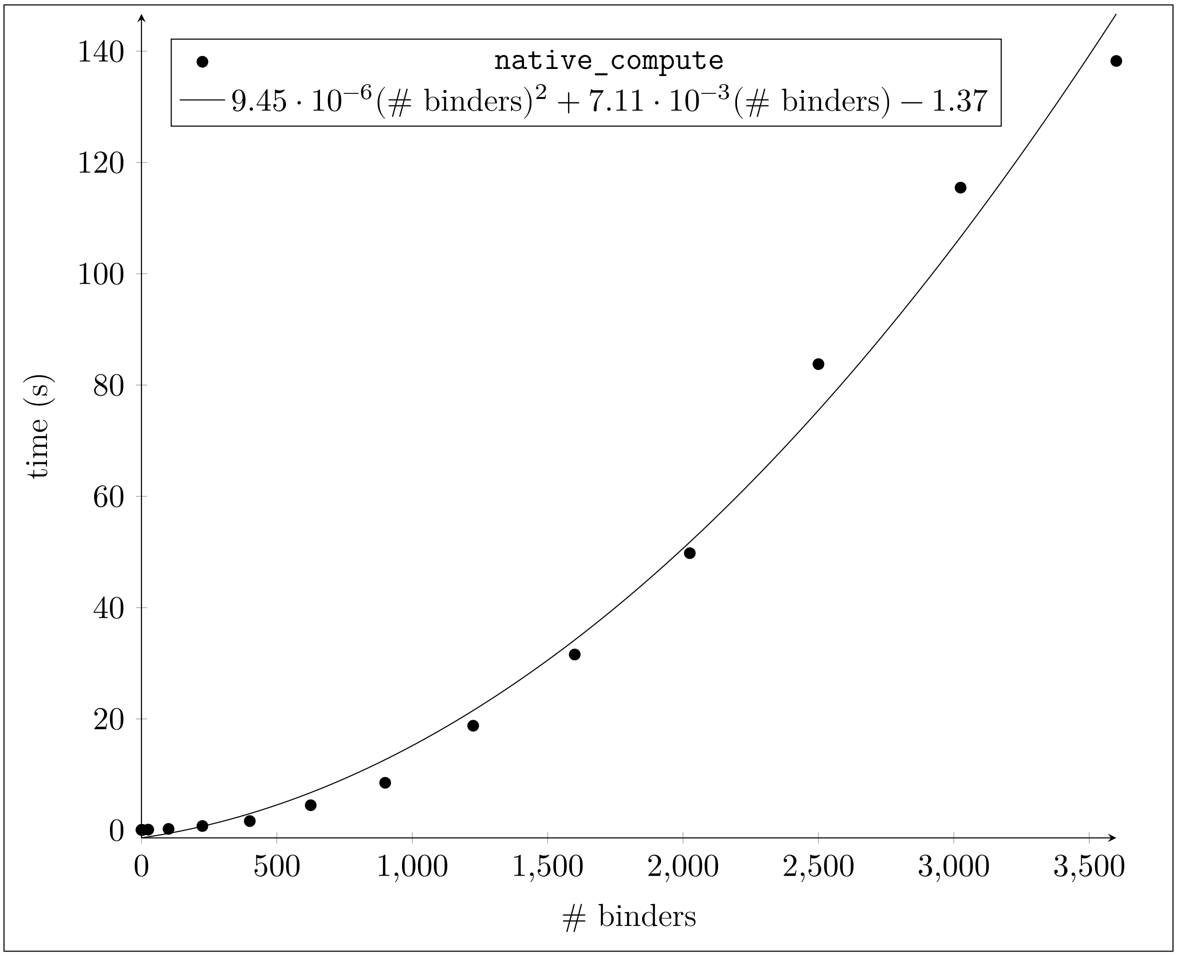


Figure 2-17: Timing of running **`native_compute`** reduction on interpreting a PHOAS expression as a function of the number of binders

### 2.2.4 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, you might define non-negative 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. You might use lists and non-negative 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 everything is sad. 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 explicitly convert 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 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 thesis.)

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 3.3. 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 and again in 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].

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 **??** and given in Gallina Coq code (with suitable notations) in **??**, is relatively trivial.

Typechecking the code that defines these components, however, takes nearly two seconds! This is more than 200× slower than defining this data in the particular case of the category of sets.[6] We attribute this to the large types generated and the non-trivial 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 $\beta$ 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 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 .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

[6] See Appendix .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 2-18: 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 2-19: 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 typecheck are given in Appendix .1.

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] 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], 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.[7]

## 2.3 Conclusion of this Chapter

[**TODO:** How should this chapter be concluded?] [**TODO:** Maybe another look-forward at what comes next?]

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

# Part II

# API Design

# Chapter 3

# Design-based fixes

## 3.1 Introduction

In Chapters 1 and 2, we talked about two different fundamental sources of performance bottlenecks in proof assistants: the power that comes from having dependent types, in Subsection 1.2.1; and the de Bruijn criterion of having a small trusted kernel, in Subsection 1.2.2. In this chapter, we will dive further into the performance issues arising from the first of these design decisions, expanding on Subsection 2.2.4 (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 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 ways; 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 simply-typed programming languages. 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 they all unfolding is explicit. However, we will present an exception to the rule of opaque abstraction barriers in Section 3.4 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 3.4 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 non-performance 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 3.3.

## 3.2 When And How To Use Dependent Types Painlessly

The extremes are relatively easy:

- Total separation between proofs and programs, so that programs are simply typed, works relatively well
- Pre-existing mathematics, where objects are fully bundled with proofs and never need to be separated from them, also works relatively well
- The rule of thumb in the middle: it is painful to recombine proofs and programs after you separate them; if you are doing it to define an opaque transformation that acts on proof-carrying code, that is okay, but if you cannot make that abstraction barrier, enormous pain results.
- For example, if you have length-indexed lists and want to index into them with elements of a finite type, things are fine until you need to divorce the index from its proof of finiteness. If you, for example, want to index into, say, the concatenation of two lists, with an index into the first of the lists, then you will likely run into trouble, because you are trying to consider the index separately from its proof of finitude, but you have to recombine them to do the indexing.

## 3.3 A Brief Introduction To Our Category Theory Library

### 3.3.1 Introduction

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 has run 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 3.5.2). 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 $\Sigma$ types (Subsection 3.5.4) and being more careful about code reuse each gave us back half of that lost time.[3]

Although pre-existing formalizations of category theory in proof assistants abound [AKS13; Meg; OKe04; Pee+; Saï; SW10; ], we chose to implement our library [] 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 pre-existing code base.

We begin our discussion in **??** considering a mundane aspect of type definitions that has large consequences for usability and performance. With the expressive power of Coq's logic Gallina, we often face a choice of making *parameters* of a type family explicit arguments to it, which looks like universal quantification; or of including them within values of the type, which looks like existential quantification. As a general principle, we found that the universal or *outside* style improves the user experience

[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.

modulo performance, while the existential or *inside* style speeds up type checking. The rule that we settled on was: *inside* definitions for pieces that are usually treated as black boxes by further constructions, and *outside* definitions for pieces whose internal structure is more important later on.

Section 3.4 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. In **??**, we describe a few other design choices that had large impacts on usability and performance, often of a few orders of magnitude.

## 3.4 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 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.

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 $\mathcal{C}^{\mathrm{op}}$. 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 $\mathcal{C}^{\mathrm{op}}$ 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 things, 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 Cᵒᵖ 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 $\mathcal{C}^{\text{op}} \to \mathcal{D}^{\text{op}}$ (whose objects are functors $\mathcal{C}^{\text{op}} \to \mathcal{D}^{\text{op}}$ and whose morphisms are natural transformations) with $(\mathcal{C} \to \mathcal{D})^{\text{op}}$ (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 2.2.1) rather than a nominal one.

### 3.4.1 Duality Design Patterns

One of the simplest theorems about duality is that it is involutive; we have that $(\mathcal{C}^{\text{op}})^{\text{op}} = \mathcal{C}$. 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 $\eta$ 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 $\eta$ on Records

The last problem we ran into was the fact that sometimes, we really, really wanted judgmental $\eta$ on records. The $\eta$ 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 $\eta$ 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 $\eta$ 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 $\eta$ 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 $\eta$ rule for records judgmentally.

### 3.4.2 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, things would have been far nicer if we had more $\eta$ rules. The $\eta$ rule for records is explained above. The $\eta$ 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.

Subsection 3.5.1 will give more examples of the pain of manipulating pattern matching on equality. 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 are currently attempting to divine the appropriate computation rules for pattern matching constructs, in the hopes of making reasoning with proofs of equality more pleasant.[4]

## 3.5 A Sampling of Abstraction Barriers

We acknowledge that the concept of performance issues arising from choices of abstraction barriers may seem a bit counter-intuitive. After all, abstraction barriers

[4]See Coq Issue #3179 and Coq Issue #3119.

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 that arose from these choices.

A few other pervasive strategies made non-trivial differences for proof performance or simplicity.

### 3.5.1 Identities vs. Equalities; Associators

There are a number of constructions that are provably equal, but which we found more convenient to construct transformations between instead, despite the increased verbosity of such definitions. This is especially true of constructions that strayed towards higher category theory. For example, when constructing the Grothendieck construction of a functor to the category of categories, we found it easier to first generalize the construction from functors to pseudofunctors. The definition of a pseudofunctor results from replacing various equalities in the definition of a functor with isomorphisms (analogous to bijections between sets or types), together with proofs that the isomorphisms obey various coherence properties. This replacement helped because there are fewer operations on isomorphisms (namely, just composition and inverting), and more operations on proofs of equality (pattern matching, or anything definable via induction); when we were forced to perform all of the operations in the same way, syntactically, it was easier to pick out the operations and reason about them.

Another example was defining the (co)unit of adjunction composition, where instead of a proof that $F \circ (G \circ H) = (F \circ G) \circ H$, we used a natural transformation, a coherent mapping between the actions of functors. Where equality-based constructions led to computational reduction getting stuck at casts, the constructions with natural transformations reduce in all of the expected contexts.

### 3.5.2 Opacity; Linear Dependence of Speed on Term Size

Coq is slow at dealing with large terms. For goals around 175,000 words long[5], we have found that simple tactics like `apply f_equal` take around 1 second to execute, which makes interactive theorem proving very frustrating.[6] 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.)

[5]When we had objects as arguments rather than fields (see ??), we encountered goals of about 219,633 words when constructing pointwise Kan extensions.

[6]See also `https://coq.inria.fr/bugs/show_bug.cgi?id=3280`.

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].

### 3.5.3 Abstraction Barriers

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. 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![7]

### 3.5.4 Nested $\Sigma$ 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 $\Sigma$ 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.

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

The first is that while a theorem statement may abstract over all possible $\Sigma$ 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.

The far more pressing problem, overriding the previous point, is that nested $\Sigma$ types have horrendous performance, and are sometimes a few orders of magnitude slower. The culprit is projections from nested $\Sigma$ types, which, when unfolded (as they must be, to do computation), each take almost the entirety of the nested $\Sigma$ 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 $\Sigma$ 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 $n^{\text{th}}$ 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 their 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 which a much larger constant factor. Note that this is a sort-of dual to the problem of Subsection 2.2.1; 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 3-1 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 2.2.1. 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 3-2 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.

In Coq 8.11, **destruct** is unfortunately still quadratic due to issues with name generation, but the constant factor is much smaller; see **??** and #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 3-1) overhead once. There's an interesting question, though, of if we can avoid this overhead all-together.

As seen in **??**, using records partially avoids the overhead. Defining the record type, though, still incurs a quadratic factor due to hash consing the projections; see #12270.

If your proof assistant does not support records out-of-the-box, or you want to avoid using them for whatever reason[8], you 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.
```

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

```
Definition _T3_proj1 (x : _T3) : unit := projT1 x.
Definition _T3_proj2 (x : _T3) : _T2 := projT2 x.

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 non-trivial 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.[9] Worse, though, due to the issue with nested **`let`**-binders described in Section 2.2.1, 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 $\eta$-conversion for primitive records, whence we can prove `T_eta` with the proof term `@eq_refl T x`. See Figures 3-4, 3-5, 3-6 and 3-7 for performance details.

[9]Note that even reflective automation (see Chapter 4) is not sufficient to solve this issue. Essentially, the bottleneck is that at the bottom of the chain of **`let`**-binders in the $\eta$ proof, we have two different types for the $\eta$-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 $\Sigma$ types, and thus being able to prove the $\eta$ principle once and for all in constant time, would not help here, because it takes quadratic time to convert between the type of the $\eta$ 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 #12269.

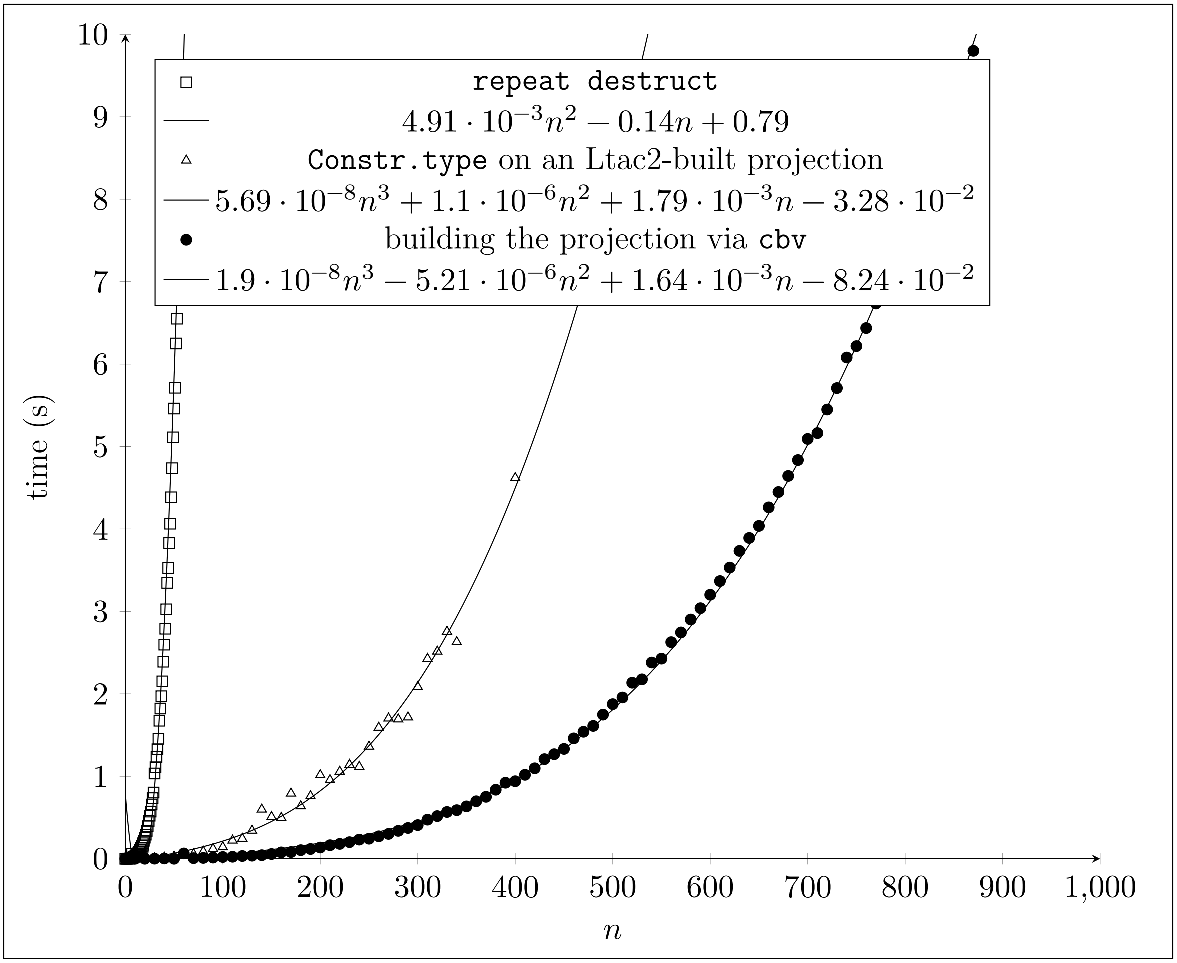



Figure 3-1: There are two ways we look at the performance of building a term like `projT1 (projT2 ... (projT2 x))` with $n$ `projT2`s: we can define a recursive function that computes this term and then use **`cbv`** to reduce away the recursion, and time how long this takes; or we can build the term using Ltac2 and then typecheck it. This plot displays both of these methods, and in addition displays the time it takes to run **`destruct`** to break $x$ into its component fields, as lower bound for how long it takes to prove anything about a nested $\Sigma$ type with $n$ fields.

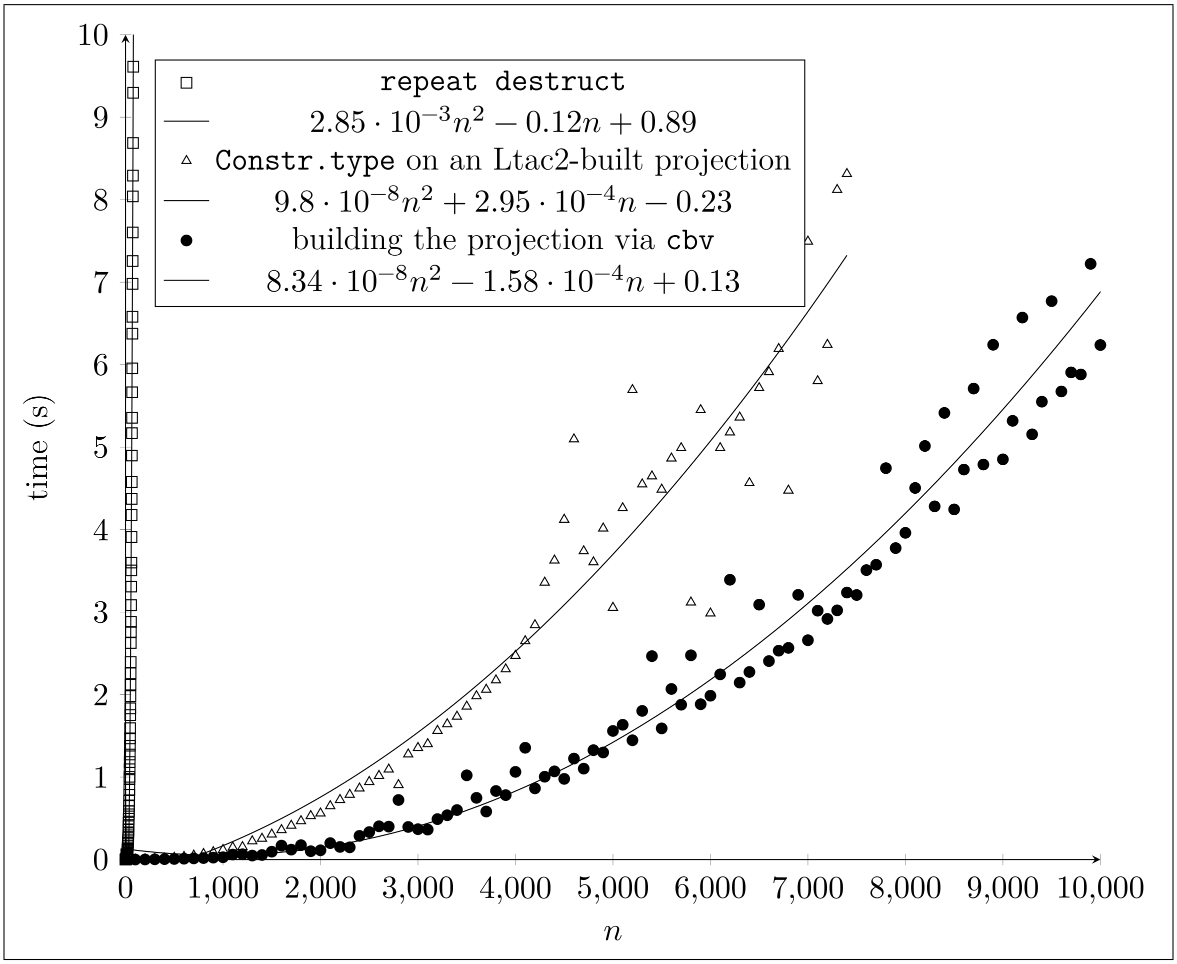



Figure 3-2: The same graph as Figure 3-1, but with primitive projections turned on. Note that the $x$-axis is 10× larger on this plot.

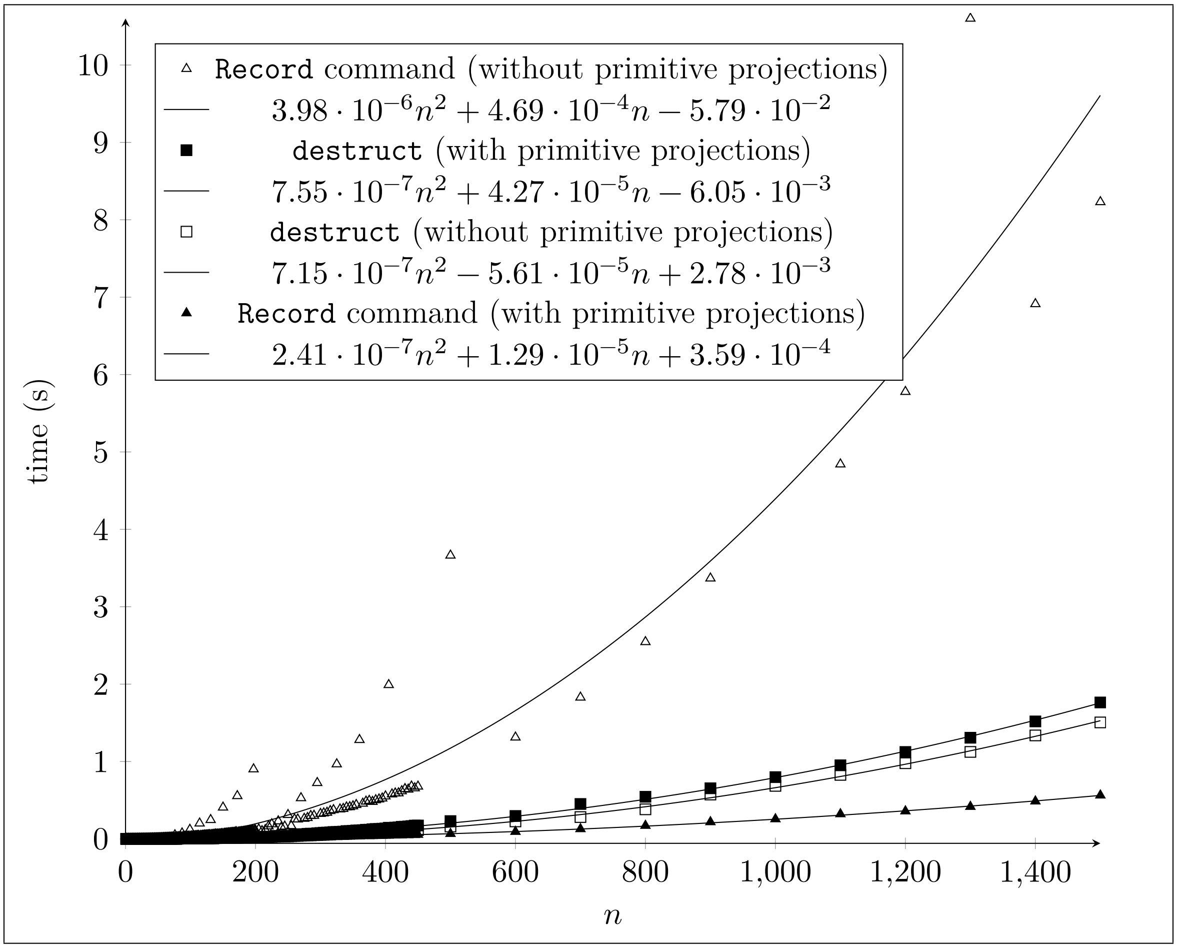



Figure 3-3: Timing of running a **`Record`** command to define a record with $n$ fields, and the time to **`destruct`** such a record. Note that building the goal involving projecting out the last field takes less than 0.001s for all numbers of fields that we tested. (Presumably for large enough numbers 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 non-monotonic timing is *reproducible*, and we have asked the Coq developers about it at #12270.

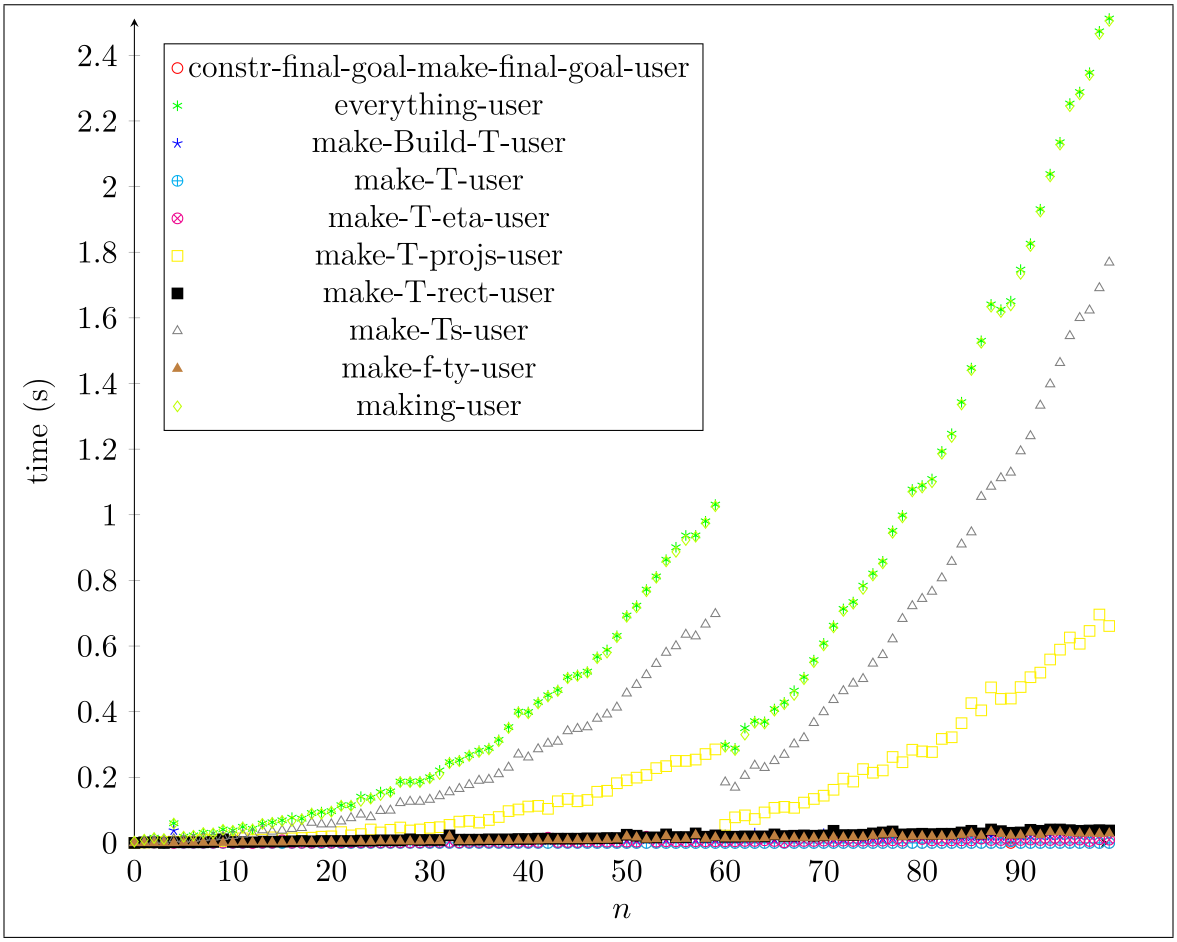



Figure 3-4: timing-performance-experiments/make-nested-prim-prod-abstraction.txt

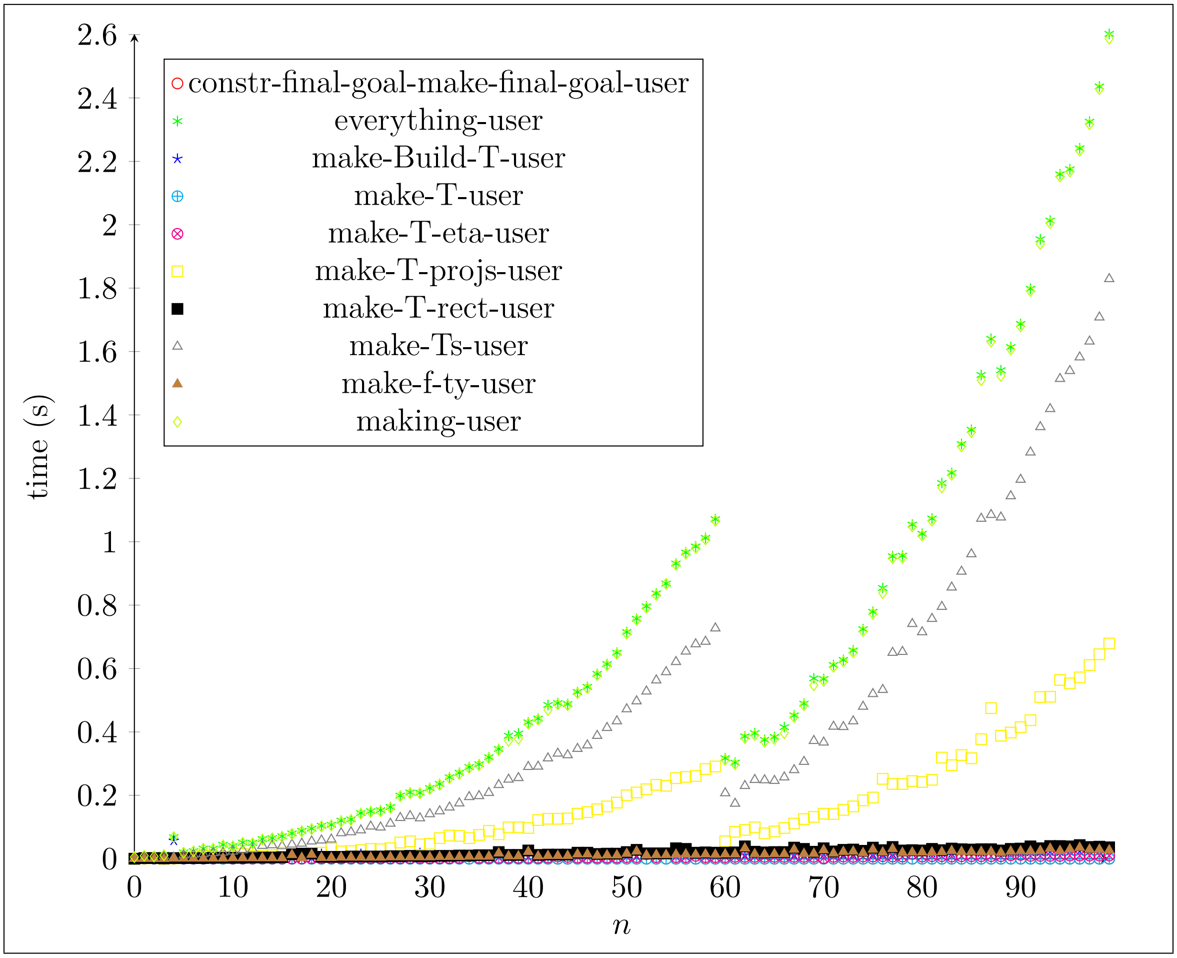


Figure 3-5: timing-performance-experiments/make-nested-prim-sig-abstraction.txt

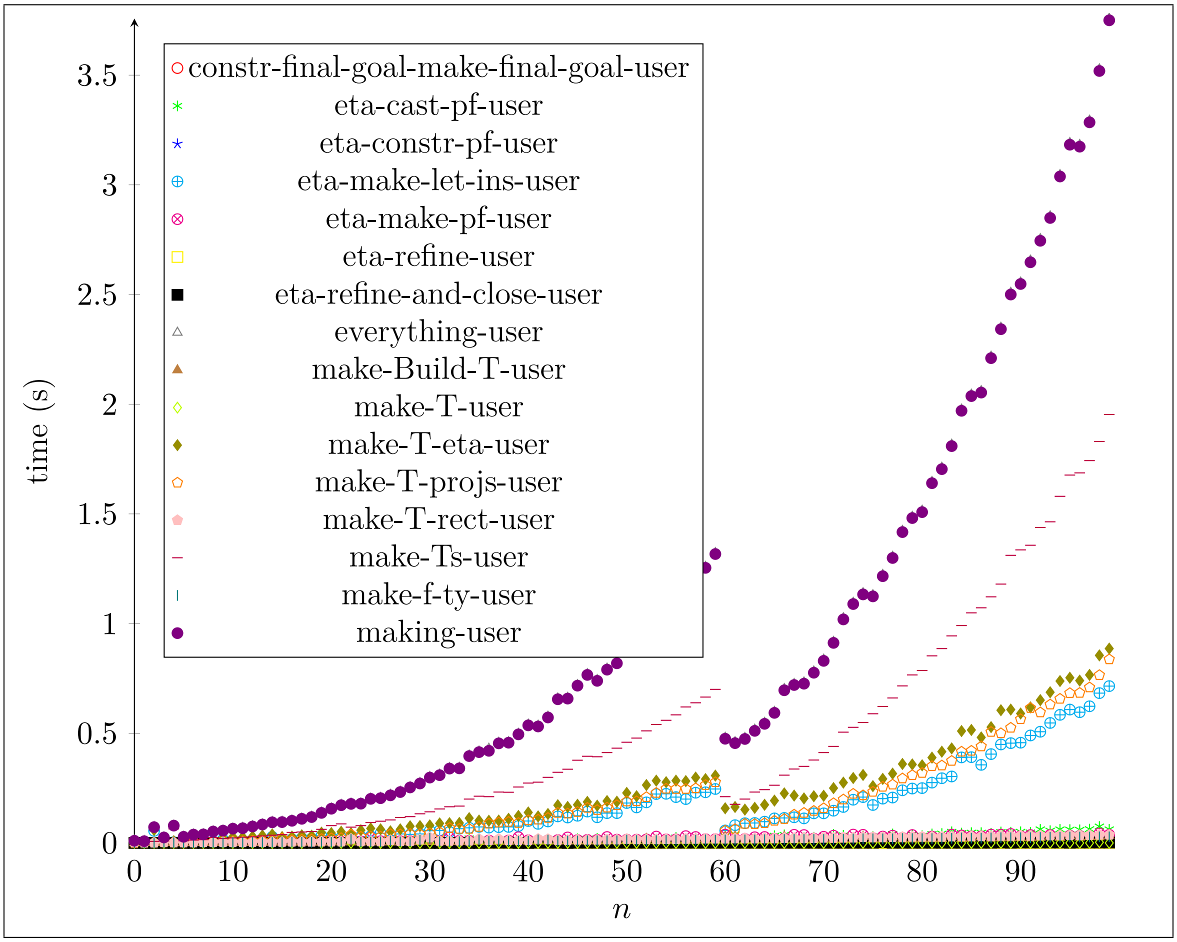


Figure 3-6: timing-performance-experiments/make-nested-prod-abstraction.txt

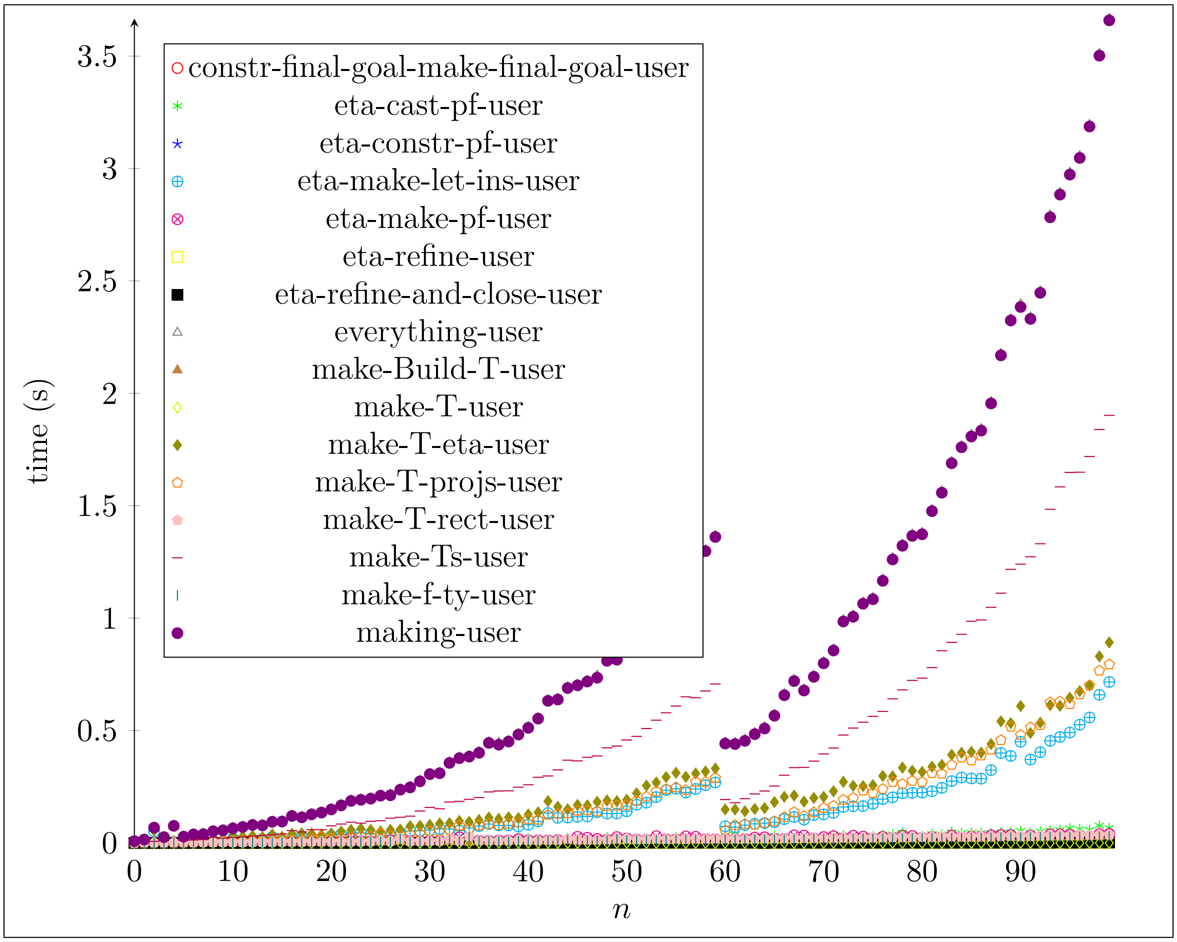



Figure 3-7: timing-performance-experiments/make-nested-sig-abstraction.txt

# Part III

# Program Transformation and Rewriting

# Chapter 4

# Reflective Program Transformation

## 4.1 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 1.2.2, 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.

The starting point for proof by reflection is *reification*: translating a "native" term of the logic into an explicit abstract syntax tree. We may then feed that tree to verified procedures or any other functional programs in the logic. The benefits listed above are particularly appealing in domains where goals are very large. For instance, consider verification of large software systems, where we might want to reify thousands of lines of source code. Popular methods turn out to be surprisingly slow, often to the point where, counter-intuitively, 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).

[**TODO:** move this paragraph elsewhere] In this paper, we show 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. Though our techniques should be broadly applicable, especially in proof assistants based on type theory, our experience is with Coq, and we review the requisite background in the remainder of this introduction. In Section 6.2, we summarize our survey into prior approaches to reification and provide high-quality implementations and documentation for them, serving a tutorial function independent of our new contributions. Experts on the subject might want to skip directly to Section 6.3, which explains our alternative technique. We benchmark our approach against 18 competitors in Section 6.4.

### 4.1.1 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. Very quickly, proofs can become long and tedious, both to write and to read, and hence Coq provides $\mathcal{L}_{tac}$, a scripting language for proofs. As theorems and proofs grow in complexity, users frequently run into performance and maintainability issues with $\mathcal{L}_{tac}$. 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_O : is_even O
| 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 naive 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.

### 4.1.2 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 4-1: Evenness Checking

A simple evenness checker can just operate on the unary encoding of natural numbers (Figure 6-1). 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_O). (* 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.

### 4.1.3 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.

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.

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

Figure 4-2: Simple Expressions

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 6.3, we get the inductive type of Figure 6-2.

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
     | 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 `check_is_even_expr`. 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] 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 6, 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.
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)).
Notation "'elet' x := v 'in' f" := (LetIn v (fun x => f)).
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.
```

Note, importantly, that `check_is_even_expr` and `denote` take `expr`s with *different* instantiations of the `var` parameter. 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.

Such a 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.

```
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).
```

Conventionally, syntax trees are parametric over the value of the `var` parameter, and we require that all instantiations give related ASTs (in the empty context), whence we call the parametric AST *well-formed*:

```
Definition Expr := ∀ var, @expr var.
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[1]

[1]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`.

```
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 the might functions become

```
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, e.g., [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 mono-typed 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 multi-typed, we end up with something like

```
Inductive context :=
| emp : context
| push : type -> context -> context.
Inductive var : context -> type -> Type :=
| VarO {t Γ} : var (push t Γ) t
| VarS {t t' Γ} : var Γ t -> var (push t' Γ) t.
Inductive expr : context -> type -> Type :=
| NatO {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 closed terms, 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 **??**.

### 4.1.4 Performance of Proving Reflective Well-Formedness of PHOAS

We saw in Section 4.1.3 that in order to prove the soundness theorem, we needed a way to relate two PHOASTs, 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 2.2.1 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 /\ forall 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 truly linear[2] proofs is to pick a more optimized representation for list membership and to convert the proposition to be an eliminator. This consists of replacing $A \wedge B$ with $\forall P, A \to B \to P$, 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.
```

[2]Actually, the size of the proof term will still have an extra logarithmic factor in the size of the syntax tree, due to the way we represent list membership proofs.

We can now prove that `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. 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`.

## 4.2 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 6, we discuss 18 different ways of implementing reification, using 6 different metaprogramming facilities in the Coq ecosystem: $\mathcal{L}_{tac}$, Ltac2, Mtac [Gon+13b; Kai+18], type classes [SO08], canonical structures [GMT16], and reification-specific OCaml plugins (quote [Coq17b], template-coq [Ana+18], ours). Figure 6-3 displays the simplest case: an Ltac 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 $\lambda$ nodes.

We have made detailed walkthroughs and source code of these implementations available[3] 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.

```
Ltac f v x := (* reify var term *)
  lazymatch x 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 4-3: Reification Without Binders in $\mathcal{L}_{tac}$

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 non-existent 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 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.

[**TODO:** what's a good way to end this chapter?]

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

# Chapter 5

# Engineering Challenges in the Rewriter

> premature optimization is the root of all evil
>
> — Donald Knuth

?? discussed in detail our framework for building verified partial evaluators, going into the context, motivation, and the 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 thesis 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 non-toy 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 to solve performance scaling issues in verified syntax transformation, itself serves as a good case study of some of the pain that arises when scaling proof-based engineering projects.

---

[1] See `https://github.com/JasonGross/fiat-crypto/blob/3b3e926e4186caa1a4003c81c65dad0a1c04b43d/src/E` for the file `src/Experiments/RewriteRulesSimpleNat.v` from the branch `experiments-small-rewrite-rule-compilation` on `JasonGross/fiat-crypto` on GitHub.

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 we've identified earlier in this thesis: issues arising from the power and (mis)use of dependent types, as introduced in Subsection 1.2.1 (Dependent Types: What? Why? How?); and issues arising arising from API mismatches, as described in Chapter 3 (Design-based fixes).

## 5.1 Pre-Reduction

The the two biggest underlying causes of engineering challenges are expression API mismatch, which we'll discuss in Section 5.2 (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.

### 5.1.1 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 the most of the decisions we seek to eliminate depend only on the *head identifier* of any application.[2] We thus augment the $\mathrm{reduce}(c)$ constant case of **??** in **??** by first

[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 $\eta$-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 things like `List.map f` to `List.map g`—we only support rewriting `List.map f xs` to `List.map g ys`.

$\eta$-expanding the identifier, before proceeding to $\eta$-expand the identifier application and perform rewriting with rewrite-head once we have an $\eta$-long form.

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

### 5.1.2 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 pain, 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)
          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
```

### 5.1.3 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 terms are indexed over types. 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 non-linear 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.

### 5.1.4 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 pre-evaluation 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 pre-reduction time.

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

We'll come back to this in Subsection 5.4.1.

## 5.2 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 $\lambda$-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 4.1.3 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.

### 5.2.1 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 decisions 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 $k^{\text{th}}$ 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 are quite painful.

Note that the type of `eval_decision_tree` would be something like `∀ {T : Type} (d : decision_tree) (ts : list type) (es : exprlist ts) (K : exprlist ts → option T), option T`.

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 two 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 an *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). It may alternatively be to much dependent typing, and the developer might be well-served by removing more dependency from 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 super-linear runtime factor in the number of rewrite rules. As noted in Section 5.1, our early reduction nets us a $2\times$ 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.

### 5.2.2 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 4.1.3 (de Bruijn Indices), converting from untyped syntax to intrinsically-typed syntax, as well as writing a denotation function, requires either that all types be non-empty, or that we carry around a proof of well-typedness to use during recursion. As discussed in Chapter 3 and specifically in Section 3.2 (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.

### 5.2.3 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 non-dependently-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 in Chapter 3, is that imperfectly opaque abstraction barriers cause headaches in a non-local 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 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 non-type-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 5.7 (Delayed Rewriting in Variable Nodes).
2. The constructors `rIdent` and `rExpr` hold "alternate" PHOAS expressions. We will discuss this in Subsection 5.4.2 (Revealing "Enough" Structure).
3. The constructor `rIdent` has an extra boolean `known`. We will discuss this in Section 5.4.1 (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 relatively little pain, 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 thesis of ways in which inadequate abstraction barriers cause pain.

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

### 5.2.4 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.

## 5.3 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 3 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 meta-level 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 pre-reduction, 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 5.2. 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 causes an enormous amount of pain, 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 5.1.3 (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.

## 5.4 Pre-evaluation 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 5.1

### 5.4.1 How Do We Know What We Can Unfold?

We can now revisit Subsection 5.1.4 in a bit more detail.

**The `known` argument** We noted in Subsection 5.2.3 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, 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 pain 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 wanting `known` to be `false`. As a toy example, suppose we are rewriting with the rules `@List.map A B f (x::xs) = f x :: List.map f xs` and `@List.map (option A) (option B) (option_map f) (List.map (@Some A) xs) = 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 fully unfold it 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.[3]

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 $\eta$-expansion of identifiers discussed in Subsection 5.1.1 (note that this is distinct from the $\eta$-expansion of identifier applications), and the identifier has no type arguments.[4] 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

[3] 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.

[4] 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.

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 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 pain induced by needless dependency at the type level.

### 5.4.2 Revealing "Enough" Structure

We noted in Subsection 5.2.3 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.

## 5.5 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 $\texttt{NbE}_\text{t}$ 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 5.1.2.

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 pain 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 pain is in our mixing of the option and CPS monads in `eval_decision_tree`. The type of `eval_decision_tree` is `∀ {T : Type} (es : list rawexpr) (d : decision_tree) (K : ℕ → 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 expressions which is appropriately equivalent to es. (See Subsection 5.7.1 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 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 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 fail, or both succeed with related outputs. Then we can conclude that the calls to `eval_decision_tree` either both fail, or both succeed with related outputs. Note, importantly, that we connect the lists of `rawexpr`s fed to the continuations with the lists `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` 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 do the same thing regardless 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 `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 5.7.2). 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.

## 5.6 Rewriting Again in the Output of a Rewrite Rule

We now come to the feature of the rewriter that caused the most pain: 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 5.7.2 (Which Equivalence Relation?).

## 5.7 Delayed Rewriting in Variable Nodes

We saw in Subsection 5.2.3 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 $\eta$-long application form by calling reflect at the base case of reduce($c$). When performing this $\eta$-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 their 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.

### 5.7.1 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 Section 4.1.3, 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 6.1.2), 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 Section 4.1.3, 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 $\mathtt{NbE}_\mathrm{t}$ values in `rValue`. Since native PHOAS expressions do not permit such thunked values, we can only relate $\mathtt{NbE}_\mathrm{t}$ values to the interpretation of such expressions. Even this is not straightforward, as we must use an extensional equivalence relation, saying that an $\mathtt{NbE}_\mathrm{t}$ value of arrow type is equivalent to an interpreted function only when equivalence between the $\mathtt{NbE}_\mathrm{t}$ value argument and the interpreted function argument implies equivalence of their outputs.

### 5.7.2 Which Equivalence Relation?

Generalizing the challenge Subsection 5.7.1, 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 5.5, we introduced without definition the four-place `wf_rawexpr` relation. This relation, a beefed up version of the PHOAS definition of `related` in Section 4.1.3, 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 $\texttt{NbE}_{\mathrm{t}}$ values match expressions. We say that an $\texttt{NbE}_{\mathrm{t}}$ value is equivalent only to the result of calling NbE's `reify` function on that value. That this definition sufficies is highly non-obvious; we refer the reader to our Coq proofs, performed with no axioms other than functional extensionality, as our justification of sufficiency. That each $\texttt{NbE}_{\mathrm{t}}$ 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 $\eta$-long identifier application as discussed briefly in Subsection 5.1.1 (What does this reduction consist of?). Namely, when forming applications of `rawexpr`s to $\texttt{NbE}_{\mathrm{t}}$ values during $\eta$-expansion, we say that the "unrevealed" structure of an $\texttt{NbE}_{\mathrm{t}}$ value `v` is `reify v`.

**`interp_maybe_do_again`** In Section 5.6, we discussed a small subset of the implications of supporting rewriting again in the output of a rewrite rule. The most easily describable pain 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[5]) 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 5.8 (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 not subject to more rewriting—or else is the $\texttt{NbE}_{\mathrm{t}}$ value type—if the rule is subject

[5] 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.

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 5.7 (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 must instead ensure that the "interpretation" of $\lambda$s 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 $(\lambda\ \texttt{f x. v}_1\texttt{(f x)})$ to $\texttt{NbE}_\text{t}$ values $\texttt{v}_2$ and $\texttt{v}_3$ is appropriately related to the interpreted integer 5, how do we materialize the interpreted functions equivalent to $(\lambda\ \texttt{f x. v}_1\texttt{(f x)})$ and $\texttt{v}_2$? The answer is "not very well", as we were unable to materialize them in a sufficiently constructive manner as to eliminate all uses of the axiom of function extensionality, despite sinking many hours into our attempt to eliminate this axiom.[6]

**Related Miscellanea** While delving into the details of all two-dozen ways of relating objects is beyond the scope of this thesis, we mention a couple of other non-obvious 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

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

The PHOAS relatedness condition of Section 4.1.3 (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 $\mathrm{NbE}_t(t_1)$, $\mathrm{NbE}_t(t_2)$, and $\mathrm{NbE}_t(t_1 \to t_2)$. Spoiler: The list for $\mathrm{NbE}_t(t_1)$ 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

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

[6]Our current thoughts are that it might be possible to prove that an interpreted function being related to any expression implies that the function respects function extensionality. We invite any brave and masochistic readers to take a stab at eliminating this axiom for us.

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.)

## 5.8 What's the Ground Truth: Patterns Or Expressions?

We mentioned in Subsection 5.7.2 (Which Equivalence Relation?) that there were two subtleties to expressing the interpretation correctness condition for intermediate representations of rewrite rules, and 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 5.6 (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 5.1.3 (Type Codes). This interpretation function is itself also severely complicated by the use of dependent types in talking about unification data.

## 5.9 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.

# Chapter 6

# Reification by Parametricity

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

[**TODO:** include the trick from https://github.com/coq/coq/issues/5996#issuecomment-670955273]


## Abstract

We present a new strategy for performing reification in Coq. That is, we show how to generate first-class abstract syntax trees from "native" terms of Coq's logic, suitable as inputs to verified compilers or procedures in the *proof-by-reflection* style. Our new strategy, based on simple generalization of subterms as variables, is straightforward, short, and fast. In its pure form, it is only complete for constants and function applications, but "let" binders, eliminators, lambdas, and quantifiers can be accommodated through lightweight coding conventions or preprocessing.

We survey the existing methods of reification across multiple Coq metaprogramming facilities, describing various design choices and tricks that can be used to speed them up, as well as various limitations. We report benchmarking results for 18 variants, in addition to our own, finding that our own reification outperforms 16 of these methods in all cases, and one additional method in some cases; writing an OCaml plugin is the only method tested to be faster. Our method is the most concise of the strategies we considered, reifying terms using only two to four lines of $\mathcal{L}_{tac}$—beyond lists of the identifiers to reify and their reified variants. Additionally, our strategy automatically provides error messages that are no less helpful than Coq's own error messages.

## 6.1 Introduction

Proof by reflection [Bou97] is an established method for employing verified proof procedures, within larger proofs. 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, those 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.

The starting point for proof by reflection is *reification*: translating a "native" term of the logic into an explicit abstract syntax tree. We may then feed that tree to verified procedures or any other functional programs in the logic. The benefits listed above are particularly appealing in domains where goals are very large. For instance, consider verification of large software systems, where we might want to reify thousands of lines of source code. Popular methods turn out to be surprisingly slow, often to the point where, counter-intuitively, 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 paper, we show 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. Though our techniques should be broadly applicable, especially in proof assistants based on type theory, our experience is with Coq, and we review the requisite background in the remainder of this introduction. In Section 6.2, we summarize our survey into prior approaches to reification and provide high-quality implementations and documentation for them, serving a tutorial function independent of our new contributions. Experts on the subject might want to skip directly to Section 6.3, which explains our alternative technique. We benchmark our approach against 18 competitors in Section 6.4.

### 6.1.1 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. Very quickly, proofs can become long and tedious, both to write and to read, and hence Coq provides $\mathcal{L}_{tac}$, a scripting language for proofs. As theorems and proofs grow in complexity, users frequently run into performance and maintainability issues with $\mathcal{L}_{tac}$. 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_O : is_even O
| 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 naive 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.

### 6.1.2 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.

A simple evenness checker can just operate on the unary encoding of natural numbers (Figure 6-1). We can use its correctness theorem to prove goals much more quickly:

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

Figure 6-1: Evenness Checking

```
Theorem soundness : forall n, check_is_even n = true -> is_even n.
Goal is_even 2000.
  Time repeat (apply even_SS || apply even_O). (* 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 `is_even (10 * 10 * 10 * 10 * 10 * 10 * 10 * 10 * 10)`. To do better, we need a formal representation of terms or expressions.

### 6.1.3 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.

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.

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

Figure 6-2: Simple Expressions

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 6.3, we get the inductive type of Figure 6-2.

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
     | 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 `check_is_even_expr`. 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 include higher-order abstract syntax (HOAS) [PE88], parametric higher-order abstract syntax (PHOAS) [Chl08], and de Bruijn indices [Bru72]. The PHOAS representation 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.
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)).
Notation "'elet' x := v 'in' f" := (LetIn v (fun x => f)).
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.
```

A full treatment of evenness checking for PHOAS would require proving well-formedness of syntactic expressions; for a more complete discussion of PHOAS, we refer the reader elsewhere [Chl08]. Using `Wf` to denote the well-formedness predicate, we could prove a theorem

```
Theorem check_is_even_expr_sound (e : ∀ var, @expr var) (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[1]

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

## 6.2 Methods of Reification

We implemented reification in 18 different ways, using 6 different metaprogramming facilities in the Coq ecosystem: Ltac, Ltac2, Mtac [Gon+13b], type classes [SO08], canonical structures [GMT16], and reification-specific OCaml plugins (quote [Coq17b], template-coq [Ana+18], ours). Figure 6-3 displays the simplest case: an Ltac 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 $\lambda$ nodes.

```
Ltac f v x := (* reify var term *)
  lazymatch x 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 6-3: Reification Without Binders in $\mathcal{L}_{tac}$

We have made detailed walkthroughs and source code of these implementations available[2] 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.

[1]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`.

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

## 6.3 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 6-4.

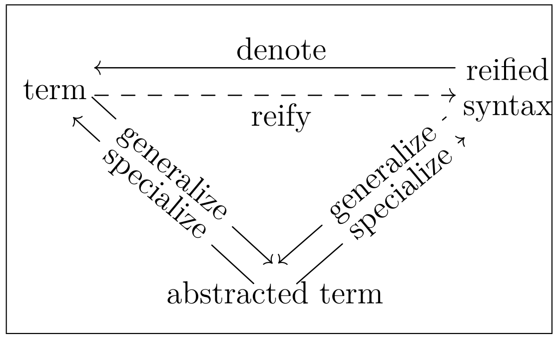



Figure 6-4: Abstraction and Reification

### 6.3.1 Case-By-Case Walkthrough

**Function Applications And Constants.**

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

To reify, we first *generalize* or *abstract* the term $2 \times 2$ 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 syntax 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:

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

We may once again specialize this term to obtain either our original term or the reified syntax. Note that to obtain reified PHOAS syntax, 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 $\textsc{LetIn}$ to obtain the PHOAS syntax tree

$$\texttt{LetIn (NatS NatO)}\ (\lambda v.\ \texttt{NatMul (Var}\ v\texttt{) (Var}\ v\texttt{))}$$

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` $\to$ `expr`. Naive reification by parametricity would turn a lambda of type $N \to N$ into a lambda of type `expr` $\to$ `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 $(\lambda n\ f.\ \texttt{SumUpTo}\ n\ (\texttt{Abs}\ (\lambda v.\ f\ (\texttt{Var}\ v))))$ 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.

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

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

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

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.

### 6.3.2 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.[3] 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

[3]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 reifying the resulting partially reduced term.

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.
```

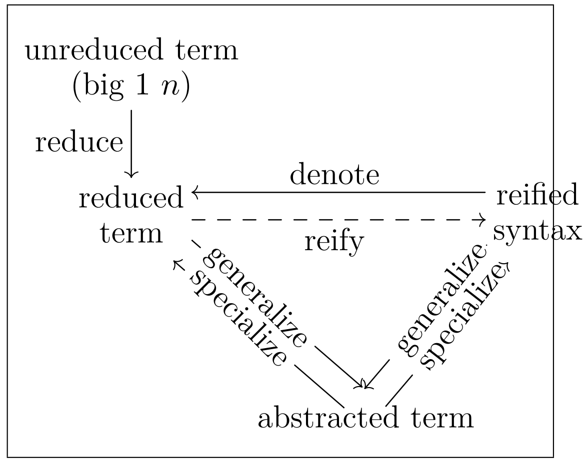
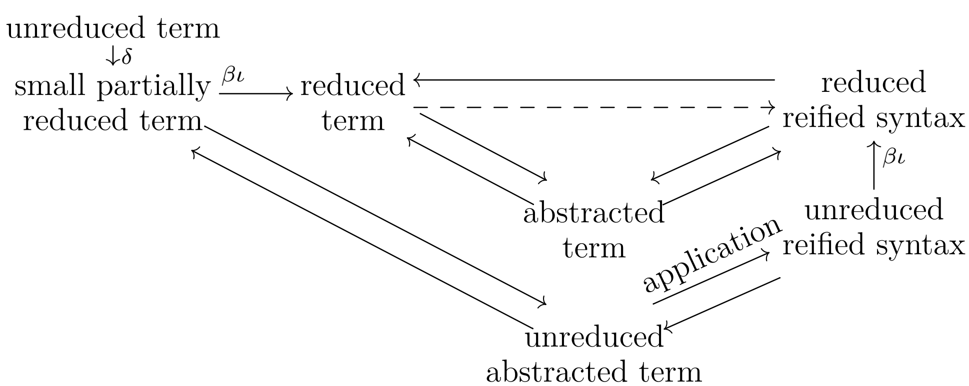



Figure 6-5: Abstraction, Reification, Reduction.

Our commutative diagram in Figure 6-4 now has an additional node, becoming Figure 6-5. 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 $\delta$ or unfolding step, followed by a $\beta\iota$ step that reduces applications of $\lambda$s and evaluates recursive calls. In specialization, there is an application step, where the $\lambda$ is applied to arguments, and a $\beta$-reduction step, where the arguments are substituted. To obtain reified syntax, we may perform generalization after $\delta$-reduction (before $\beta\iota$-reduction), and we are not required to perform the final $\beta$-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 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 $\delta$ 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 $\delta$-reduction, that is, unfolding `big`, gives us the small partially reduced term

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

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

$$\begin{aligned}
&\Lambda N.\ \lambda(\textsc{Mul} : N \to N \to N)(\mathrm{O} : N)(\mathrm{S} : N \to N)(\textsc{LetIn} : N \to (N \to N) \to N).\\
&\qquad (\lambda(x : N).\ \lambda(n : \texttt{count}).\ \texttt{count_rec}\ (N \to N)\ (\lambda x.\ x)\\
&\qquad\qquad (\lambda n'.\ \lambda \texttt{big}_{n'}.\ \lambda x.\ \textsc{LetIn}\ (\textsc{Mul}\ x\ x)\ (\lambda x'.\ \texttt{big}_{n'}\ x')))\\
&\qquad (\mathrm{S}\ \mathrm{O})\ \ 100
\end{aligned}$$

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, $(\lambda v\, f.\ \texttt{LetIn}\ v\ (\lambda x.\ f\ (\texttt{Var}\ x)))$. 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` $\to$ `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.[4]

[4] 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 mod-

### 6.3.3 Implementation in $\mathcal{L}_{tac}$

`ExampleMoreParametricity.v` in the code supplement mirrors the development of reification by parametricity in Subsection 6.3.1.

Unfortunately, Coq does not have a tactic that performs abstraction.[5] However, the `pattern` tactic suffices; it performs abstraction followed by application, making it a sort of one-sided inverse to $\beta$-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`, an additional step involving retyping the term is needed; we refer the reader to `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
```

ulus of arithmetic). A couple hundred is enough to beat the overhead.

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

```
  | ?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 `ExampleMoreParametricity.v`.

### 6.3.4 Advantages and Disadvantages

This method is faster than all but $\mathcal{L}_{tac}2$ and OCaml reification, and commuting reduction and abstraction makes this method faster even than the low-level $\mathcal{L}_{tac}2$ 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.

## 6.4 Performance Comparison

We have done a performance comparison of the various methods of reification to the PHOAS language `@expr var` from Figure 6.1.3 in Coq 8.7.1. 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

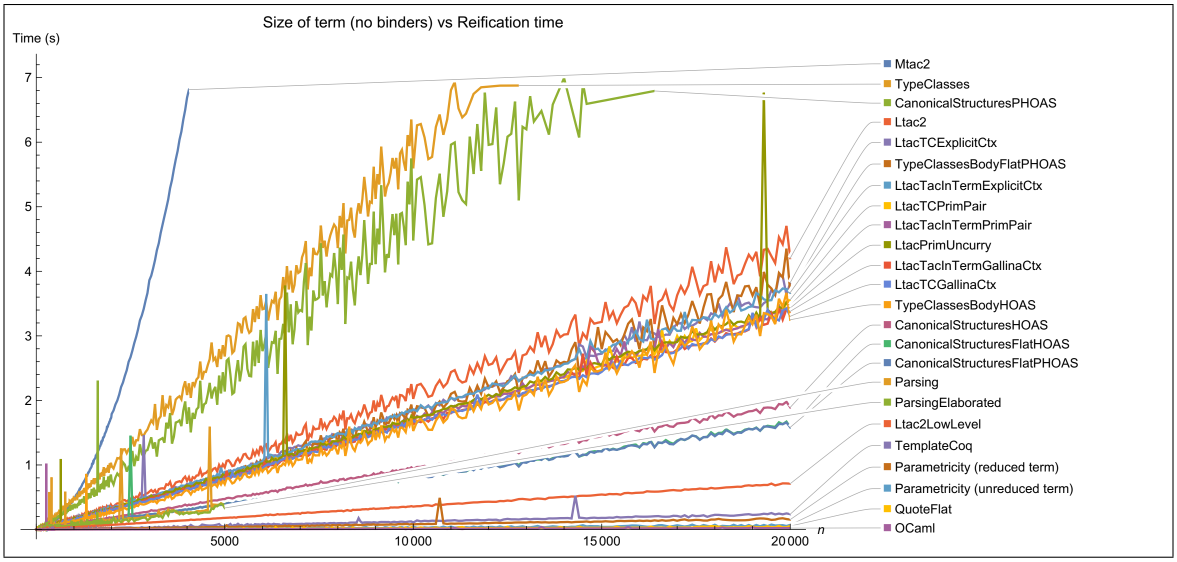


Figure 6-6: Performance of Reification without Binders

consider (1), because it provides the most fine-grained analysis of the actual reification method.

### 6.4.1 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 6-6 should be self-explanatory and are found in similarly named `.v` files in the associated code; we call out a few potentially confusing ones:

- 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 and the OCaml plugin we wrote by hand.

### 6.4.2 With Binders

We look at terms of the form `dlet` $\texttt{a}_1$ `:= 1 * 1 in dlet` $\texttt{a}_2$ `:=` $\texttt{a}_1$ `*` $\texttt{a}_1$ `in …` `dlet` $\texttt{a}_n$ `:=` $\texttt{a}_{n-1}$ `*` $\texttt{a}_{n-1}$ `in` $\texttt{a}_n$, 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.

In addition to reification benchmarks, the graph in Figure 6-7 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.

A nontrivial portion of the cost of "Parametricity (reduced term)" seems to be due to the fact that looking up the type of a binder is linear in the number of binders in the context, thus resulting in quadratic behavior of the retyping step that comes after abstraction in the `pattern` tactic. In Coq 8.8, this lookup will be $\log n$, and so reification will become even faster [Péd17a].

## 6.5 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 6.3.2. We perform $\delta$ 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

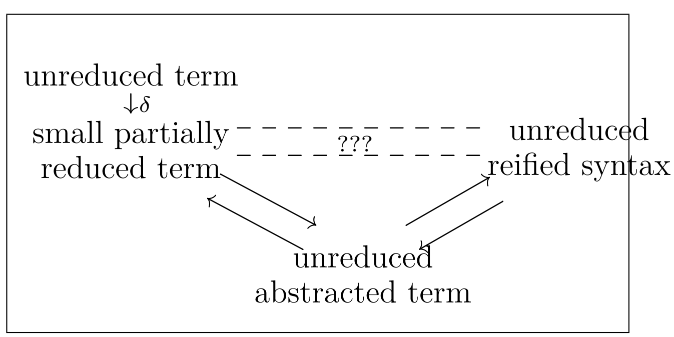



Figure 6-8: Completing the commutative triangle

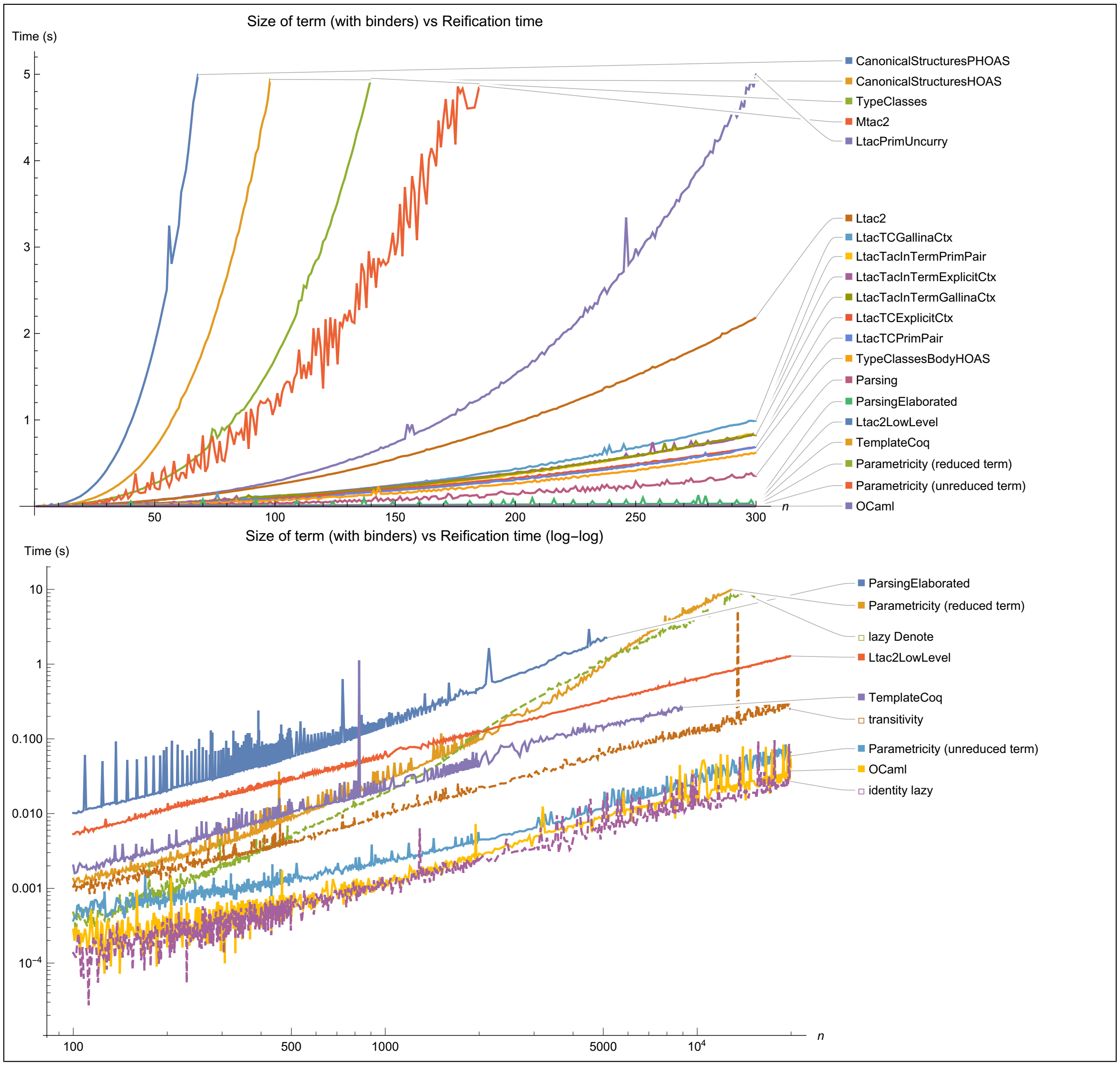


Figure 6-7: Performance of Reification with Binders

$\beta\iota$-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.

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 6-8.

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

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

We hypothesize that a form of internalized parametricity would suffice for proving this lemma. In particular, we could specialize $F$'s type argument 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:

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

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.

## 6.6 Acknowledgments and Historical Notes

We would like to thank Hugo Herbelin for sharing the trick with `type of` to propagate universe constraints[6] as well as useful conversations on Coq's bug tracker that allowed us to track down performance issues.[7] We 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. We 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. We also thank John Wiegley for feedback on the paper.

For those interested in history, our method of reification by parametricity 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; $\beta$-reduction; reification, we could move $\beta$-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.

[6] `https://github.com/coq/coq/issues/5996#issuecomment-338405694`
[7] `https://github.com/coq/coq/issues/6252`

# Part IV

# Conclusion

# Chapter 7

# A Retrospective on Performance Improvements

Throughout this thesis, 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 non-trivial ways from performance bottlenecks in non-dependently-typed languages. Part II discussed design principles to avoid performance pitfalls, and Part III took a deep dive into a particular set of performance bottlenecks and presented a tool, and, we hope, exposed the underlying design methodology, that allows eliminating asymptotic bottlenecks in one important part of proof assistant systems.

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 bottleneck 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. Coq 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 4.1.3), where closed terms use de Bruijn indices, but open terms, i.e., evars, refer to the variables in their context by name. Hence each evar carries with it a named context, which causes a great deal of trouble as described in Section 2.2.3 (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 well—replacing all instantiated evars with their instantiation. Needless to say, this was very expensive when the size of the goal was large.

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

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 []. Type inference fills in the holes in untyped `glob_constr`s to turn them into typed `constr`s, possibly with remaining existential variables []. 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 []. Finally, implicit arguments must be erased and notations must be re-sugared when externalizing `glob_constr`s into `constr_expr`s, which can be printed relatively straightforwardly [; ].

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 a game-changer 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].

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× to 5× 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

[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`

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× 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 thesis.

### 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 3.5.4 (Nested $\Sigma$ 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].

### 7.1.7 Fast Typing of Application Nodes

In Section 2.2.3 (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.

[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.

## 7.2 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 3 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 $2\times$ 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 the last subsection's 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]

### 7.2.2 Judgmental $\eta$ for Record Types

The same commit that introduced universe polymorphism in Coq 8.5 also introduced judgmental $\eta$ 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 $\eta$ in Section 3.4.1 (Dodging Judgmental $\eta$ on Records) and Section 3.4 (Internalizing Duality Arguments in Type Theory).

[6]See, for example, the commit message of a445bc3 in the HoTT/HoTT library on GitHub, where moving from Coq 8.5$\beta$2 to 8.5$\beta$3 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 #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 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.

The $\eta$ 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 $\eta$ rule equates `x` with `(fst x, snd x)`.

As discussed in Section 3.4, having records with judgmental $\eta$ conversion allows deduplicating code that would otherwise have to be duplicated.

### 7.2.3 SProp: The Definitionally Proof Irrelevant Universe

We discussed in Subsection 3.5.2 (Opacity; Linear Dependence of Speed on Term Size) how irrelevant proof arguments tended to cause performance issues just by existing as part of the goal. While there is as-yet no way to erase these arguments, Coq versions 8.10 and later have the 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 3 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 non-trivial 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 ⇒ match i with zero ⇒ true | one ⇒ false end` of type `Interval → bool` should not typecheck for this reason.

The key insight is that most types do not need any special equivalence relation, 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 5.2 (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 5 and Chapter 3 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.[7]

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+16]. 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].

[7] 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 come, at last, to the closing remarks of this thesis.

We spent Part I mapping out the landscape of the problem of performance in dependently-typed proof assistants. In Part II and Part III, we laid out more-or-less systematic principles and tools for avoiding 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 1.2 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 thesis 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 incurred by unpredictable computation at the type-level, the source of much of the power of dependent type theory, we broadly suggest in Part II 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 3.4) and proof by reflection (Chapter 4)). 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 III that users effectively *throw away the entire tactic engine* and instead code tactics reflectively.

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 3 lays out in Section 3.2 (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 thesis 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 is the performance is of the "underlying algorithm"?

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.

[**QUESTION FOR READERS:** Adam found this following paragraph hard to follow; what do other readers think?] 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 at least $\Omega(2^x)$, 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 non-reflective 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 thesis 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 perhaps-nascent field themselves.

[1]Note that if we are targeting a lazy functional language rather than a strict one, it may in fact be possible to always achieve a transformation without any overhead.

# Bibliography


[] *ACL2 User's Manual — Interesting-applications*. URL: https://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/index.html?topic=ACL2____INTERESTING-APPLICATIONS.

[] *Constrextern (coq.Constrextern)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Constrextern/index.html.

[] *Constrintern (coq.Constrintern)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Constrintern/index.html.

[] *Detyping (coq.Detyping)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Detyping/index.html.

[] *Formalizations of category theory in proof assistants*. MathOverflow. URL: http://mathoverflow.net/questions/152497/formalizations-of-category-theory-in-proof-assistants.

[] *HoTT/HoTT Categories*. URL: https://github.com/HoTT/HoTT/tree/V8.12/theories/Categories.

[] *Nqthm, the Boyer-Moore prover*. URL: https://www.cs.utexas.edu/users/boyer/ftp/nqthm/.

[] *Ppconstr (coq.Ppconstr)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Ppconstr/index.html.

[] *Pretyping (coq.Pretyping)*. URL: https://coq.github.io/doc/V8.12.0/api/coq/Pretyping/index.html.

[] *Therac-25*. URL: https://en.wikipedia.org/wiki/Therac-25.

[12a] *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.

[12b] *subobject classifier*. nLab. Sept. 2012. URL: http://ncatlab.org/nlab/show/subobject+classifier.

[20] *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.

[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*. Tech. rep. 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), pp. 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, pp. 57–68. DOI: 10.1145/1292597.1292608. URL: http://www.strictlypositive.org/obseqnow.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, pp. 500–513. DOI: `10.1007/3-540-52885-7_110`.

[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*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 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), pp. 109–139.

[Asp+11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. "The Matita Interactive Theorem Prover". In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 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 $\lambda$-calculus implementations". In: *International Conference on Rewriting Techniques and Applications*. Ed. by Jieh Hsiang. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 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*. Ed. by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 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, pp. 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*. Ed. by Mark Aagaard and John Harrison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 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), pp. 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, pp. 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*. Ed. by Alan Robinson and Andrei Voronkov. NLD: Elsevier Science Publishers B. V., 2001, pp. 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*. Ed. by Luke Ong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 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), pp. 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 ed. 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), pp. 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, pp. 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*. Ed. by Martín Abadi and Takayasu Ito. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 515–529. ISBN: 978-3-540-69530-1. DOI: `10.1007/bfb0014565`.

[BP99] Richard S. Bird and Ross Paterson. "de Bruijn notation as a nested datatype". In: *Journal of Functional Programming* 9.1 (1999), pp. 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), pp. 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*. Ed. by M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1970, pp. 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)*. Vol. 75. 5. Elsevier. 1972, pp. 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*. Ed. by R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. Vol. 133. Studies in Logic and the Foundations of Mathematics. Elsevier, 1994, pp. 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 $\lambda$–calculus". In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. July 1991, pp. 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), pp. 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, pp. 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: 76 (1988), pp. 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), pp. 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/.

[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, pp. 609–622. ISBN: 9781450333009. DOI: 10.1145/2676726.2677003. URL: http://adam.chlipala.net/papers/BedrockPOPL15/.

[CM98] Alexandra Carvalho and Paulo Mateus. *Category Theory in Coq*. Tech. rep. 1049-001 Lisboa, Portugal, 1998. URL: http://sqig.math.ist.utl.pt/pub/CarvalhoA/98-C-DiplomaThesis/maintext.ps.

[Coh+16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: (Nov. 7, 2016). 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/.

[Coq17a] Coq Development Team. "The Coq Proof Assistant Reference Manual". In: 8.7.1. INRIA, 2017. Chap. 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. Chap. 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, pp. 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, pp. 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*. Ed. by Richard J. Boulton and Paul B. Jackson. Vol. 2152. Lecture Notes in Computer Science. Springer Berlin Heidelberg, June 2001, pp. 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, pp. 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.

[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.

[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), p. 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), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[Dyc85] Roy Dyckhoff. *Category Theory as an Extension of Martin-Löf Type Theory*. Tech. rep. 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), pp. 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*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 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*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 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". In: (Sept. 12, 2016). arXiv: 1609.03543 [cs.AI].

[Gaw14] Nicole Gawron. "Infamous Software Bugs: FDIV Bug". In: (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: *ACM SIGPLAN Notices* 42.10 (2007), pp. 57–76. 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), pp. 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*. Ed. by Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 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, pp. 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*. Ed. by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 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*. Tech. rep. 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), pp. 11–25.

[Gon+11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. "How to Make Ad Hoc Proof Automation Less Ad Hoc". In: *ACM SIGPLAN Notices*. Vol. 46. ACM. 2011, pp. 163–175. DOI: 10.1145/2034574.2034798. 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, pp. 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), pp. 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), pp. 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, pp. 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, pp. 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), pp. 427–450. ISSN: 1088-9485. DOI: 10.1090/bull/1616.

[Gro15a] Jason Gross. "An Extensible Framework for Synthesizing Efficient, Verified Parsers". MA 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.

[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.

[Hal06] Thomas C. Hales. "Introduction to the Flyspeck Project". In: *Mathematics, Algorithms, Proofs*. Ed. 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*. Ed. by Joakim von Wright, Jim Grundy, and John Harrison. Vol. 1125. Lecture Notes in Computer Science. Turku, Finland: Springer-Verlag, Aug. 1996, pp. 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), pp. 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), pp. 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.

[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, pp. 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), pp. 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, pp. 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), pp. 83–111. URL: http://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/agroupoidinterpretationoftypetheroy.pdf.

[HTt14] 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.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving". In: *Computational Logic*. 2014, pp. 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*. Ed. 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). Ed. by Bertrand Meyer, pp. 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, pp. 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, pp. 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.

[KM20] 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), pp. 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), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[Kova] András Kovács. “Non-deterministic normalization-by-evaluation in Olle Fredriksson's flavor”. In: (). 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/0c7f8a676c0964cc05c24AIMprez/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, pp. 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, pp. 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, p. 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), pp. 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*. Ed. 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), pp. 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), pp. 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*. Tech. rep. 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, pp. 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*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 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*. Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 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. "Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings". In: ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. Chap. Extensible and Efficient Automation Through Reflective Tactics, pp. 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, pp. 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*. Vol. 2044. TLCA’01. Springer. Kraków, Poland: Springer-Verlag, 2001, pp. 344–359. ISBN: 3540419608. URL: http://www.pps.univ-paris-diderot.fr/~miquel/publis/tlca01.pdf.

[Moh95] Takahisa Mohri. “On formalization of category theory”. MA 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), pp. 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: *International Conference on Automated Deduction*. Ed. by Amy P. Felty and Aart Middeldorp. 9195 vols. Springer. Springer, 2015, pp. 378–388. 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*. Vol. 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: *ACM Sigplan Notices* 44.3 (2009), pp. 265–276. DOI: `10.1145/1508284.1508275`. URL: `https://users.cs.northwestern.edu/~robby/courses/322-2013-spring/mytkowicz-wrong-data.pdf`.

[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`.

[NL] 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, p. 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*. Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 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*. Vol. 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), pp. 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, pp. 748–752. ISBN: 3540556028. DOI: 10.1007/3-540-55602-8_217.

[Pau18] Lawrence C. Paulson. "Formalising Mathematics In Simple Type Theory". In: (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*. Vol. 828. Springer, 1994.

[PCG18] Karl Palmskog, Ahmet Celik, and Milos Gligoric. "Cq: Parallel regression proving for large-scale verification projects". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 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, pp. 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*. Ed. by Helmut Schwichtenberg and Ralf Steinbrüggen. Dordrecht: Springer Netherlands, 2002, pp. 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*. Tech. rep. 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*. Ed. by Mark E. Stickel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 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), pp. 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), pp. 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, pp. 92–101.

[PS99] Frank Pfenning and Carsten Schürmann. "System Description: Twelf – A Meta-Logical Framework for Deductive Systems". In: *LNCS* (1999).

[Raa20] Panu Raatikainen. "Gödel's Incompleteness Theorems". In: *The Stanford Encyclopedia of Philosophy*. Ed. 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*. Ed. 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, pp. 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), pp. 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, pp. 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), pp. 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), pp. 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*. Ed. by Mandayam Srivas and Albert Camilleri. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 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: *ACM SIGPLAN Notices* 34.1 (1998), pp. 313–323. DOI: `10.1145/291251.289460`.

[SN08] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 28–32. ISBN: 978-3-540-71067-7.

[SO08] Matthieu Sozeau and Nicolas Oury. "First-class type classes". In: *Lecture Notes in Computer Science* 5170 (2008), pp. 278–293. 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 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*. Ed. by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Vol. 44. Studies in Logic and the Foundations of Mathematics. Elsevier, 1966, pp. 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/2dbe106c09b60690b87e31e58d505b1f4e05b57f`

[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 ed. 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*. Tech. rep. 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: (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). 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), pp. 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]`.

[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), pp. 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, pp. 347–359.

[Web02] Tjark Weber. “Program Transformations in Nuprl”. MA 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), pp. 193–220. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0005-1. URL: http://www.cs.ru.nl/~freek/pubs/arrow.pdf.

[Wik20a] 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.

[Wik20b] 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*. Tech. rep. Technical report, University of Uppsala, 2005.

[Wil12] Olov Wilander. “Constructing a small category of setoids”. In: *Mathematical Structures in Computer Science* 22.1 (2012), pp. 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). Ed. by Sean W. Smith, pp. 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.

## .1 Full Example of Nested Abstraction Barrier Performance Issues

In Section 2.2.4, we discussed an example where unfolding nested abstraction barriers caused performance issues. Here we include the complete code for that example.

```
Require Import Coq.Program.Basics.
Require Import Coq.Program.Tactics.

Set Primitive Projections.
Set Implicit Arguments.
Set Universe Polymorphism.

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} _ _.
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) : NaturalTransformat
    := {| 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_morph
Arguments Build_NaturalTransformation [C D]%category_scope (F G)%functor_scope & _%na

Notation "x : A  o  f" := (fun x : A%category => f) (at level 70) : functor_object_scop
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) : func
Notation "m @ s --> d  m  f" := (fun s d m => f) (at level 70) : functor_morphism_scop
Notation "' m @ s --> d  m  f" := (fun s d 'm => f) (at level 70, m strict pattern) :
Notation "m : A  m  f" := (fun s d (m : A%category) => f) (at level 70) : functor_morp
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) :
Notation "x : A  t  f" := (fun x : A%category => f) (at level 70) : natural_transformat
Notation "' x  t  f" := (fun '(x%category) => f) (x strict pattern, at level 70) : natu
Notation "x  t  f" := (fun x => f) (at level 70) : natural_transformation_components_sc

Notation " fo ; mo " := (@Build_Functor _ _ fo mo _ _) (only parsing) : functor_scope
Notation " f " := (@Build_NaturalTransformation _ _ _ _ f _) (only parsing) : natural


Notation "'λ'  fo ; mo " := (@Build_Functor _ _ fo mo _ _) (only parsing) : functor_
Notation "'λ'  f " := (@Build_NaturalTransformation _ _ _ _ f _) (only parsing) : nat
```

```
Notation "'λo' x1 .. xn , fo ; 'λm' m1 .. mn , mo" := (@Build_Functor _ _ (fun x1 =
Notation "'λt' x1 .. xn , f" := (@Build_NaturalTransformation _ _ _ _ (fun x1 => .. (

(** [(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 752)

  ============================
  forall C1 C2 D : Category,
  {|
  object_of := fun F : (C1 -> C2 -> D)%category =>
               {|
               object_of := fun pat : (C1 * C2)%category =>
                            (F0 (fst pat))0 (snd pat);
               morphism_of := fun (pat pat0 : (C1 * C2)%category)
                                (pat1 : morphism (C1 * C2)
                                         (fst pat, snd pat)
                                         (fst pat0, snd pat0)) =>
                              (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 := fun (s d : (C1 -> C2 -> D)%category)
                   (T : morphism (C1 -> C2 -> D) s d) =>
                 {|
                 components_of := fun pat : (C1 * C2)%category =>
                                  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 := fun F : (C1 * C2 -> D)%category =>
                 {|
                 object_of := fun c1 : C1 =>
                              {|
                              object_of := fun c2 : C2 => F0 (c1, c2);
                              morphism_of := fun (s d : C2) (m : morphism C2 s d)
                                             => F1 (1, m);
                              composition_of := curry_iso_obligation_1 F c1;
                              identity_of := curry_iso_obligation_2 F c1 |};
                 morphism_of := fun (s d : C1) (m1 : morphism C1 s d) =>
                                {|
                                components_of := fun c2 : C2 => 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 := fun (s d : (C1 * C2 -> D)%category)
                     (T : morphism (C1 * C2 -> D) s d) =>
```

```
                   {|
                   components_of := fun c1 : C1 =>
                                    {|
                                    components_of := fun c2 : C2 => 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 757)

  ============================
  forall C1 C2 D : Category,
  {|
  object_of := fun x : (C1 * C2 -> D)%category =>
               {|
               object_of := fun F : (C1 -> C2 -> D)%category =>
                            {|
                            object_of := fun pat : (C1 * C2)%category =>
                                         (F0 (fst pat))0
                                         (snd pat);
                            morphism_of := fun (pat pat0 : (C1 * C2)%category)
                                             (pat1 : morphism
                                                       (C1 * C2)
                                                       (fst pat, snd pat)
                                                       (fst pat0, snd pat0)) =>
                                           (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 := fun (s d : (C1 -> C2 -> D)%category)
                                (T : morphism (C1 -> C2 -> D) s d) =>
                              {|
                              components_of := fun pat : (C1 * C2)%category =>
                                               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 := fun F : (C1 * C2 -> D)%category =>
                             {|
```

```
                              object_of := fun c₁ : C₁ =>
                                          {|
                                          object_of := fun c₂ : C₂ => F₀ (c₁, c₂);
                                          morphism_of := fun
                                                   (s d : C₂)
                                                   (m : morphism C₂ s d) =>
                                                   F₁ (1, m);
                                          composition_of := curry_iso_obligation_1
                                                   F c₁;
                                          identity_of := curry_iso_obligation_2 F
                                                   c₁ |};
                              morphism_of := fun (s d : C₁) (m₁ : morphism C₁ s d)
                                            =>
                                            {|
                                            components_of := fun c₂ : C₂ =>
                                                   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 := fun (s d : (C₁ * C₂ -> D)%category)
                                     (T : morphism (C₁ * C₂ -> D) s d) =>
                                   {|
                                   components_of := fun c₁ : C₁ =>
                                                 {|
                                                 components_of := fun c₂ : C₂ =>
                                                     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) |}₀ x);
          morphism_of := fun (s d : (C₁ * C₂ -> D)%category)
                           (m : morphism (C₁ * C₂ -> D) s d) =>
                         {|
                         object_of := fun F : (C₁ -> C₂ -> D)%category =>
                                      {|
                                      object_of := fun pat : (C₁ * C₂)%category =>
                                                  (F₀ (fst pat))₀
                                                  (snd pat);
                                      morphism_of := fun (pat pat0 : (C₁ * C₂)%category)
                                                       (pat1 :
                                                        morphism
                                                          (C₁ * C₂)
                                                          (fst pat, snd pat)
```

```
                              (fst pat0, snd pat0)) =>
                         (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 := fun (s0 d0 : (C_1 -> C_2 -> D)%category)
                   (T : morphism (C_1 -> C_2 -> D) s0 d0) =>
               {|
               components_of := fun pat : (C_1 * C_2)%category =>
                                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 := fun F : (C_1 * C_2 -> D)%category =>
              {|
              object_of := fun c_1 : C_1 =>
                           {|
                           object_of := fun c_2 : C_2 =>
                                        F_0 (c_1, c_2);
                           morphism_of := fun
                                        (s0 d0 : C_2)
                                        (m0 : morphism C_2 s0 d0) =>
                                        F_1 (1, m0);
                           composition_of := curry_iso_obligation_
                                        F c_1;
                           identity_of := curry_iso_obligation_2
                                        F c_1 |};
              morphism_of := fun (s0 d0 : C_1)
                                (m_1 : morphism C_1 s0 d0) =>
                             {|
                             components_of := fun c_2 : C_2 =>
                                        F_1 (m_1, 1);
                             commutes := curry_iso_obligation_3 F
                                        s0 d0 m_1 |};
              composition_of := curry_iso_obligation_5 F;
              identity_of := curry_iso_obligation_4 F |};
 morphism_of := fun (s0 d0 : (C_1 * C_2 -> D)%category)
                    (T : morphism (C_1 * C_2 -> D) s0 d0) =>
                {|
                components_of := fun c_1 : C_1 =>
                                 {|
                                 components_of := fun c_2 : C_2 =>
                                        T (c_1, c_2);
```

```
                                        commutes := curry_iso_obligation_
                                          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 := fun F : (C₁ -> C₂ -> D)%category =>
                                {|
                                object_of := fun pat : (C₁ * C₂)%category =>
                                        (F₀ (fst pat))₀
                                        (snd pat);
                                morphism_of := fun
                                           (pat
                                            pat0 :
                                            (C₁ * C₂)%category)
                                           (pat1 :
                                            morphism
                                            (C₁ * C₂)
                                            (fst pat, snd pat)
                                            (fst pat0, snd pat0)) =>
                                         (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 := fun (s d : (C₁ -> C₂ -> D)%category)
                                  (T : morphism (C₁ -> C₂ -> D) s d) =>
                                 {|
                                 components_of := fun pat : (C₁ * C₂)%category
                                            =>
                                            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 := fun F : (C₁ * C₂ -> D)%category =>
                                {|
                                object_of := fun c₁ : C₁ =>
                                        {|
                                        object_of := fun c₂ : C₂ =>
                                           F₀ (c₁, c₂);
                                        morphism_of := fun
                                           (s d : C₂)
                                           (m : morphism C₂ s d) =>
```

```
                                            F₁ (1, m);
                                        composition_of := curry_iso_obligat
                                            F c₁;
                                        identity_of := curry_iso_obligation
                                            F c₁ |};
                              morphism_of := fun
                                           (s d : C₁)
                                           (m₁ : morphism C₁ s d) =>
                                         {|
                                         components_of := fun c₂ : C₂ =>
                                            F₁ (m₁, 1);
                                         commutes := curry_iso_obligation_
                                            F s d m₁ |};
                              composition_of := curry_iso_obligation_5 F;
                              identity_of := curry_iso_obligation_4 F |};
                    morphism_of := fun (s d : (C₁ * C₂ -> D)%category)
                                     (T : morphism (C₁ * C₂ -> D) s d) =>
                                   {|
                                   components_of := fun c₁ : C₁ =>
                                                    {|
                                                    components_of := fun c₂ : C₂
                                                    => T (c₁, c₂);
                                                    commutes := curry_iso_obligati
                                                    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 := fun F : (C₁ -> C₂ -> D)%category =>
                            {|
                            object_of := fun pat : (C₁ * C₂)%category =>
                                         (F₀ (fst pat))₀
                                         (snd pat);
                            morphism_of := fun (pat pat0 : (C₁ * C₂)%category)
                                             (pat1 :
                                              morphism
                                                (C₁ * C₂)
                                                (fst pat, snd pat)
                                                (fst pat0, snd pat0)) =>
                                           (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 := fun (s d : (C1 -> C2 -> D)%category)
                                     (T : morphism (C1 -> C2 -> D) s d) =>
                                   {|
                                   components_of := fun pat : (C1 * C2)%category =>
                                                            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 := fun F : (C1 * C2 -> D)%category =>
                                  {|
                                  object_of := fun c1 : C1 =>
                                                 {|
                                                 object_of := fun c2 : C2 =>
                                                                F0 (c1, c2);
                                                 morphism_of := fun
                                                                (s d : C2)
                                                                (m : morphism C2 s d) =>
                                                                F1 (1, m);
                                                 composition_of := curry_iso_obligation
                                                                F c1;
                                                 identity_of := curry_iso_obligation_2
                                                                F c1 |};
                                  morphism_of := fun (s d : C1)
                                                   (m1 : morphism C1 s d) =>
                                                  {|
                                                  components_of := fun c2 : C2 =>
                                                                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 := fun (s d : (C1 * C2 -> D)%category)
                                     (T : morphism (C1 * C2 -> D) s d) =>
                                   {|
                                   components_of := fun c1 : C1 =>
                                                      {|
                                                      components_of := fun c2 : C2 =>
                                                        T (c1, c2);
                                                      commutes := curry_iso_obligation
                                                        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 1443)

  ============================
  forall C₁ C₂ D : Category,
  {|
  object_of := fun x : (C₁ * C₂ -> D)%category =>
               {|
               object_of := fun pat : (C₁ * C₂)%category => x₀ (fst pat, snd pat);
               morphism_of := fun (pat pat0 : (C₁ * C₂)%category)
                                  (pat1 : morphism (C₁ * C₂)
                                             (fst pat, snd pat)
                                             (fst pat0, snd pat0)) =>
                                x₁ (fst pat1, 1) ∘ x₁ (1, snd pat1);
               composition_of := curry_iso_obligation_7
                                   {|
                                   object_of := fun c₁ : C₁ =>
                                                {|
                                                object_of := fun c₂ : C₂ =>
                                                    x₀ (c₁, c₂);
                                                morphism_of := fun
                                                    (s d : C₂)
                                                    (m : morphism C₂ s d) =>
                                                    x₁ (1, m);
                                                composition_of := curry_iso_obligat
                                                    x c₁;
                                                identity_of := curry_iso_obligation
                                                    x c₁ |};
                                   morphism_of := fun
                                                   (s d : C₁)
                                                   (m₁ : morphism C₁ s d) =>
                                                  {|
                                                  components_of := fun c₂ : C₂ =>
                                                    x₁ (m₁, 1);
                                                  commutes := curry_iso_obligation_
                                                    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 := fun c₁ : C₁ =>
                                             {|
                                             object_of := fun c₂ : C₂ =>
```

```
                                                      x0 (c1, c2);
                                               morphism_of := fun
                                                      (s d : C2)
                                                      (m : morphism C2 s d) =>
                                                      x1 (1, m);
                                               composition_of := curry_iso_obligation_
                                                      x c1;
                                               identity_of := curry_iso_obligation_2
                                                      x c1 |};
                                    morphism_of := fun (s d : C1)
                                                    (m1 : morphism C1 s d) =>
                                                 {|
                                                 components_of := fun c2 : C2 =>
                                                      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 := fun (s d : (C1 * C2 -> D)%category)
               (m : morphism (C1 * C2 -> D) s d) =>
             {|
             components_of := fun pat : (C1 * C2)%category =>
                                m (fst pat, snd pat);
             commutes := curry_iso_obligation_9
                          {|
                          components_of := fun c1 : C1 =>
                                             {|
                                             components_of := fun c2 : C2 =>
                                                  m (c1, c2);
                                             commutes := curry_iso_obligation_6
                                                  m c1 |};
                          commutes := curry_iso_obligation_12 m |} |};
composition_of := Functor.compose_obligation_1
                 {|
                 object_of := fun F : (C1 -> C2 -> D)%category =>
                                {|
                                object_of := fun pat : (C1 * C2)%category =>
                                               (F0 (fst pat))0
                                               (snd pat);
                                morphism_of := fun
                                                   (pat
                                                    pat0 :
                                                    (C1 * C2)%category)
                                                   (pat1 :
                                                    morphism
```

```
                                    (C₁ * C₂)
                                    (fst pat, snd pat)
                                    (fst pat0, snd pat0)) =>
                                 (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 := fun (s d : (C₁ -> C₂ -> D)%category)
                     (T : morphism (C₁ -> C₂ -> D) s d) =>
                   {|
                   components_of := fun pat : (C₁ * C₂)%category
                                    =>
                                    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 := fun F : (C₁ * C₂ -> D)%category =>
                 {|
                 object_of := fun c₁ : C₁ =>
                              {|
                              object_of := fun c₂ : C₂ =>
                                   F₀ (c₁, c₂);
                              morphism_of := fun
                                   (s d : C₂)
                                   (m : morphism C₂ s d) =>
                                   F₁ (1, m);
                              composition_of := curry_iso_obligat
                                   F c₁;
                              identity_of := curry_iso_obligation
                                   F c₁ |};
                 morphism_of := fun
                                  (s d : C₁)
                                  (m₁ : morphism C₁ s d) =>
                                 {|
                                 components_of := fun c₂ : C₂ =>
                                    F₁ (m₁, 1);
                                 commutes := curry_iso_obligation_
                                    F s d m₁ |};
                 composition_of := curry_iso_obligation_5 F;
                 identity_of := curry_iso_obligation_4 F |};
morphism_of := fun (s d : (C₁ * C₂ -> D)%category)
                     (T : morphism (C₁ * C₂ -> D) s d) =>
                   {|
```

```
                                components_of := fun c1 : C1 =>
                                                 {|
                                                 components_of := fun c2 : C2
                                                 => T (c1, c2);
                                                 commutes := curry_iso_obligati
                                                 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 := fun F : (C1 -> C2 -> D)%category =>
                                   {|
                                   object_of := fun pat : (C1 * C2)%category =>
                                                (F0 (fst pat))0
                                                (snd pat);
                                   morphism_of := fun (pat pat0 : (C1 * C2)%category)
                                                    (pat1 :
                                                     morphism
                                                       (C1 * C2)
                                                       (fst pat, snd pat)
                                                       (fst pat0, snd pat0)) =>
                                                  (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 := fun (s d : (C1 -> C2 -> D)%category)
                                         (T : morphism (C1 -> C2 -> D) s d) =>
                                     {|
                                     components_of := fun pat : (C1 * C2)%category =>
                                                      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 := fun F : (C1 * C2 -> D)%category =>
                                   {|
                                   object_of := fun c1 : C1 =>
                                                {|
                                                object_of := fun c2 : C2 =>
                                                             F0 (c1, c2);
                                                morphism_of := fun
                                                             (s d : C2)
                                                             (m : morphism C2 s d) =>
```

```
                                                F₁ (1, m);
                                        composition_of := curry_iso_obligation
                                                F c₁;
                                        identity_of := curry_iso_obligation_2
                                                F c₁ |};
                            morphism_of := fun (s d : C₁)
                                              (m₁ : morphism C₁ s d) =>
                                            {|
                                            components_of := fun c₂ : C₂ =>
                                                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 := fun (s d : (C₁ * C₂ -> D)%category)
                                   (T : morphism (C₁ * C₂ -> D) s d) =>
                                {|
                                components_of := fun c₁ : C₁ =>
                                                  {|
                                                  components_of := fun c₂ : C₂ =>
                                                    T (c₁, c₂);
                                                  commutes := curry_iso_obligation
                                                    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 c
          (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 _] (H
```

```
          (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
              = ((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
              = ((G 1 m) c2 c3)),
    F = G.

Local Ltac unfold_stuff
  := intros;
     cbv [compose Cat 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.
```

[**TODO:** which notation is best?]

### .1.1 Example in the Category of Sets

We include here the code for the components defined in the category of sets.

```
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) *)
```

[**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]