[**TODO:** Run ‘make update-thesis’ before submission to update the date on the cover page]

[**TODO:** change \finalfalseto \finaltrue]

[**TODO:** Is the "department committee chairman" still Leslie A. Kolodziejski?]

Performance Engineering of Proof-Based Software Systems

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 2020

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

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

[**TODO:** More formal wording] 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.

[**TODO:** Fix runon sentence] 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.

[**TODO:** Maybe instead use the alternative from the thesis-proposal?] The proposed research is a study of performance issues that come up in engineering large-scale proof-based systems in Coq. The thesis presents lessons learned about achieving acceptable performance in Coq in the course of case-studies on formalizing category theory, developing a parser synthesizer, and constructing a verified compiler for synthesizing efficient low-level cryptographic primitives. We also present a novel method of simple and fast reification, and a prototype tool for faster rewriting and customizable reduction which does not require extending Coq's trusted code base.

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

Acknowledgments

[**TODO:** uniform style] [**TODO:** rearrange] 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. [**TODO:** Add more acknowledgments] I want to thank Andres Erbsen for pointing out to me some of the particular performance bottlenecks in Coq that I made use of in this thesis, including those of subsubsection Sharing in Section 2.2.1 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. [**TODO:** cite various grants]

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" [16].

[**TODO:** rearrange] 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: [**TODO:** how to format citations here] 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" [16].

?? is based on the introduction to Gross, Erbsen, and Chlipala [17].

[**QUESTION FOR ADAM:** Decide on whether to copyright myself or MIT; Adam, are there considerations to be aware of here?] [**TODO:** Download and fill out `https://libraries.mit.edu/wp-content/uploads/2019/08/umi-proquest-form.pdf`]

Contents

[**TODO:** decide on chapter/section capitalization and be consistent] [**TODO:** figure out if regression stats should be included, and, if so, how to calculate them]

Part I

Introduction

> The most common mistake in performance engineering is to blindly optimize without profiling; you most often spend your time optimizing parts that aren't actually bottlenecks.
>
> — Charles Leiserson (heavily paraphrased, reconstructed from memory of 6.172)

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

Chapter 1

Introduction

1.1 Introduction

Paragraph 1: Orient the reader to very broad things Software w/o bugs via formal methods [**TODO:** Initial intro paragraphs]

Paragraph 2: Success of foundational tools [**TODO:** transition to foundational tools] [**TODO:** explain how foundational tools like Coq are already useful]

Paragraph 3: performance of tools is important and non-trivial, and what we're doing here is explaining / demonstrating how and why it's non-trivial, and how to fix nontrivial performance issues [**TODO:** explain that performance, including of build-time, is important] [**TODO:** read and cite perf engineering study `https://arxiv.org/abs/2003.06458`] [**TODO:** read / look into https://dl.acm.org/doi/pdf/10.1145/1297027.1297033] [**TODO:** maybe read https://users.cs.northwestern.edu/ robby/courses/322-2013-spring/mytkowicz-wrong-data.pdf] [**TODO:** Karl Palmskog: @Jason Gross I saw your post on performance optimization/measurements for proof assistants on Coq-Club. We summarize a lot of work on parallelization/selection for proof assistants in our regression proving papers (ASE 17 & ISSTA 18): http://users.ece.utexas.edu/ gligoric/papers/CelikETAL17iCoq.pdf http://users.ece.utexas.edu/ gligoric/papers/PalmskogETAL18piCoq.pdf] [**TODO:** read and cite https://leanprover.github.io/papers/tactic.pdf] [**TODO:** read Implementing Typed Intermediate Languages] by Zhong Shao, Christopher League, and Stefan Monnier In Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98) "It's not directly related to proof assistants, but the techniques described can be applicable to proof assistants and the experience *may* be applicable to some extent." [**TODO:** read "Inductive families need not store their indices" by Edwin Brady, Connor McBride, James McKinna] [**TODO:** read Aleksey Nogin's Cornell PhD thesis from the 1990s] [**TODO:** read Grégoire and Leroy's paper from ICFP 2002 https://xavierleroy.org/bibrefs/Gregoire-Leroy-02.html] [**TODO:** read Dirk Kleeblatt's PhD thesis.] "Both of these are about using

compiled code in dependent type checkers instead of interpreters." [**TODO:** read Andras Kovacs has smalltt at https://github.com/AndrasKovacs/smalltt]", which I don't think has been written up anywhere but has nonetheless been influential, both on Idris 2 and on Olle Fredriksson's reimplementation of Sixten at https://github.com/olle-f/sixty" [**TODO:** look at https://github.com/AndrasKovacs/normalization-bench] [**TODO:** look at https://github.com/AndrasKovacs/smalltt] [**TODO:** "I haven't yet updated the smalltt repo, but there's a simplified (https://gist.github.com/AndrasKovacs/a0e0938113b193d6b9c1c0620d853784) implementation of its evaluator, which seems to have roughly the same performance but which is much simpler to implement."]

András Kovács wrote:

> The basic idea is that in elaboration there are two primary computational tasks, one is conversion checking and the other is generating solutions for metavariables. Clearly, we should use NbE/environment machines for evaluation, and implement conversion checking in the semantic domain. However, when we want to generate meta solutions, we need to compute syntactic terms, and vanilla NbE domain only supports quote/readback to normal forms. Normal forms are way too big and terrible for this purpose. Hence, we extend vanilla NbE domain with lazy non-deterministic choice between two or more evaluation strategies. In the simplest case, the point of divergence is whether we unfold some class of definitions or not. Then, the conversion checking algorithm can choose to take the full-unfolding branch, and the quoting operation can choose to take the non-unfolding branch. At the same time, we have a great deal of shared computation between the two branches; we avoid recomputing many things if we choose to look at both branches.
>
> I believe that a feature like this is absolutely necessary for robust performance. Otherwise, we choke on bad asymptotics, which is surprisingly common in dependent settings. In Agda and Coq, even something as trivial as elaborating a length-indexed vector expression has quadratic complexity in the length of the vector.
>
> It is also extremely important to stick to the spirit of Coquand's semantic checking algorithm as much as possible. In summary: core syntax should support *no* expensive computation: no substitution, shifting, renaming, or other ad-hoc term massaging. Core syntax should be viewed as immutable machine code, which supports evaluation into various semantic domains, from which sometimes we can read syntax back; this also leaves it open to swap out the representation of core syntax to efficient alternatives such as bytecode or machine code.
>
> Only after we get the above two basic points right, can we start to think about more specific and esoteric optimizations. I am skeptical of proposed solutions which do not include these. Hash consing has been

brought up many times, but it is very unsatisfying compared to non-deterministic NbE, because of its large constant costs, implementation complexity, and the failure to handle sharing loss from beta-redexes in any meaningful way (which is the most important source of sharing loss!). I am also skeptical of exotic evaluators such as interaction nets and optimal beta reducers; there is a good reason that all modern functional languages run on environment machines instead of interaction nets.

If we want to support type classes, then tabled instance resolution is also a must, otherwise we are again smothered by bad asymptotics even in modestly complex class hierarchies. This can be viewed as a specific instance of hash-consing (or rather "memoization"), so while I think ubiquitous hash-consing is bad, some focused usage can do good.

Injectivity analysis is another thing which I believe has large potential impact. By this I mean checking whether functions are injective up to definitional equality, which is decidable, and can be used to more precisely optimize unfolding in conversion checking.

I'd be very interested in your findings about proof assistant performance. This has been a topic that I've been working on on-and-off for several years. I've recently started to implement a system which I intend to be eventually "production strength" and also as fast as possible, and naturally I want to incorporate existing performance know-how.

[**TODO:** look into "So technically, the lost sharing is the second-order sharing that is preserved in "optimal reduction" of lambda calculi [Levy-1980, Lamping-1990, Asperti-Laneve-1992], while hash consing normally is directly usable only for first-order sharing."] [**TODO:** look at https://math.stackexchange.com/questions/3466976/online-reference-book-for-implementing-concepts-in-type-theory] [**TODO:** look at https://github.com/Andras zoo/blob/0c7f8a676c0964cc05c247879393e97729f59e5b/AIMprez/AIMprez.pdf or https://eu-types.cs.ru.nl/eutypes_pmwiki/uploads/Meetings/Kovacs_slides.pdf]

1.1.1 What are proof assistants?

[**TODO:** wide look at various proof assistants and successes] [**TODO:** read and cite Talia's paper `http://tlringer.github.io/pdf/QEDatLarge.pdf`?]

1.2 Basic Design Choices

[**TODO:** introduce section where we talk about what some big design decisions are and why we might make them the way we do]

1.2.1 Dependent Types: What? Why? How?

[**TODO:** explain dependent type theory, motivate using it] [**TODO:** read and cite `https://lamport.azurewebsites.net/pubs/lamport-types.pdf` and `https://arxiv.`

org/abs/1804.07860 h/t Karl Palmskog @palmskog on gitter https://gitter.im/coq/coq?at=5e5ec0ae4eefc06dcf31943f]

1.2.2 The De Bruijn Criterion

[**TODO:** explain the De Bruijn criterion] [**TODO:** look into https://arxiv.org/abs/2003.01685 Sealing Pointer-Based Optimizations Behind Pure Functions] [**TODO:** example: Ltac vs Gallina, kind-of]

1.3 Look ahead: Layout and contributions of the thesis

[**TODO:** describe layout] [**TODO:** describe main contributions]

1.A Transcript bits from talking with Adam

And my phone is now recording.

Yeah, so my son's this story is that the there'll be sort of introduction and at some point I'll have to introduce caulk and some amount of detail and I'm not sure where exactly that bit of it goes. But the main thing I want to talk about in the introduction is.

Like what makes performance in caulk and assistance especially dependently type ones different from performance and other programming languages, let me suggest here for the very beginning. I would try to write introduction that doesn't go into a lot of detail about conflict but paints a broad picture.

Say sort of like going through a breath first traversal of the material starting with the higher level motivation, so that's whoever reads it understands what you're trying to accomplish and what major it's a progress you made towards those goals and then chapter two. Presents more the details on call background, that'll be needed to understand the precise rules of the game and what's going to come later okay so then I'm thinking the chapter one in the introduction is something like performance in crew or like proof assistance or thing preface systems are important performance improve assistance is important yeah.

I'm not sure how much it feels like in order to like talk about why performance improve assistance is important. I need to say something about like what makes it different like why it's not already solved. The information at a very high level there so the the like highest level the sketch here is that in most languages the performance looks something like EU write something and it works and maybe you have to optimize it a little and like as you get bigger examples, it slowly gets slower.

Whereas in caulk the experience is that he writes something in that works and you get bigger examples and it gets a bit slower and then you make your examples just a little bit bigger and now it takes a week or a month or like unclear just I'm sure we can find examples like that in traditional software also like you you just pushed your working set beyond the cast size or.

Something like that.

I think that's true put my senses that in preface this then it like this is. This is just how the like. It seems like this is pervasive in preface systems. I think this is the wrong level abstraction for. Section one. I would first try to convey the big message of the detention between flexibility and trust in a free persistent building out alcohol methods tools, we can typically get around a lot of these issues.

I say okay, so. So is this section where I want to talk about the divide between kernel trusted code base and the rest of it you might try starting out with with just introducing the debate on criteria remind me what that does. Roughly what you just said small currently that's in terms of some sort of record of approved that can be appreciated in terms of only a small senators okay, yeah that seems good.

And you might want to give an idea for what dependent types are what's the peel they give people using them and some sort of fuzzy idea for the challenges that might emerge are.

Yeah.

I'm trying to figure out what it feels like. I have like two very different levels that I can pitch dependent types at and not sure if either one is adequate for thesis. One of them is explaining dependent types as in many languages you want to know at compile time if you try to pass an integer when you're function is expecting a string yeah in caulk you like you have this part of these system on steroids where you can do things like oh.

You need to pass element of the empty set if the turning machine does not halt and if the turning machine does halt the need to pass an element of the one element set. There was definitely skipping forward way through an explanation of what depends sorry why are we here?

Well represented about the actual way is so this is sort of the like what? What makes dependent types powerful and challenging. The the what dependent types are. I feel like my technical explanation is something like they let the home. So one version of it is that they return value for your the return type of your function gets the depend on the value of the arguments.

Yes, that does sound like a definition. I mean, it's it's not.

Hit. To only works precisely if you're either fully curried or like it doesn't capture segment types exactly or something.

Sure. Okay. I'm right. And no, and then the the turning machine bit is an explanation of like what? What dependent types let you do and how they let you encode proofs and this part of the system. I don't think it's an explanation of why it pays off to do things that way.

They're really important to have in the first section. I feel like I don't have a good explanation of why why it appears off to be a superficial systems on type theory, rather than doing something like Isabel Hall. It's gonna be a problem, but if you spend a while. Theseus explaining how to handle dependent types properly and you have an explained why that was the right size choice to put them in there in the first place.

Yeah. It feels like currently I like to have the knowledge about to have an argument for them. Okay.

Other than like lots of people do it or something. I think part of it is connected to the small trusted go-based story of how tribulation checking works. Really don't have another option but using types of things sometimes. I feel like if you base everything on the axioms of sub theory, you can have a.

TCP that's much smaller than that of call. Maybe. We get the same performance advantages ignore this issue that there's a design choice that creates a large fraction of the challenges and the thesis and not explain why it's there. If you want to start out by saying there was a time when we were confused and thought this was a good idea and now we're going to trace through the consequences.

That's better than trying to ignore it.

I think I want guidance from you on what to do here. It feels like the best I can say is like this is a choice that lots of people make and we're going to look at like assuming that you've made this choice like what follows? Maybe should try to pull people who seem like on a bashed fans of a family type programming improve assistance even today and get looked there.

There didn't justifications over. I think it's fun. Okay, it's a good start. Which might translate into what happens to match every people of intuition as well. I think that's close correlative fun here. Yeah.

I guess I can like email. Call club or something.

Okay. Okay, I will plan to email golf club. It's good. And figure out something to write. Feels like this will be one of the one of the week links in my thesis. Okay.

Okay, so there will be some chunks that's like talking about dependent types and.

You won't you want to make a similar introduction and motivation for every major feature that leaked into interesting and significant challenges in the work you presents.

Yeah, so I guess the it feels like the two main.

Like two main features that I'm going to talk about are motivated by the Deborah and Criterion and using dependent types.

I think there's something about computation for efficiency being built into certain parts of the system, that is. Fundamental here having compiled my thoughts. Yeah. I feel like that that's sort of runs through both of them or something. So we're sorry that like runs through both so I'm with my overall plan is that there'll be this introduction section, there'll be another section that like introduces call can talks about alike.

Survey of what performance issues and calls look like from a like bird's eye view. Okay, and then there'll be a section on.

1.B more transcript from Adam

I expect. In terms of where I expect to have trouble feels like I expect that. I'll have a lot of trouble making the technical details of the rewriter the targeted right level.

And I'll have trouble sort of weaving all of the different parts together coherently.

Because of what to be feels like it's sort of inherent. Between the category theory part and the the article is very important yeah. I feel like. Like I like the current way that I'm dealing with that of like we're going to look at like two completely different parts of the system that like overlapping little bit and how you solve them or like ways to solve them and like one of them is convergent.

And then the other one is or like one of them is conversion and the root of the issue is dependent types and then the other one is ah. Program transformation and rewriting in the root of the issue is this separation between the trusted code base and the parts that you freely optimize.

Okay. And like, Ah in the conversion that there's like a couple solutions one of them is that is basically all the way on the never call conversion end of things and the other one is on the like shove everything into the type level and this is where the like the amber efficient computation shows up and then in the the program transformation and rewriting section the solution that we're using is the shove everything into the piano.

Okay.

Did you bitch you reification in your little walk-through? I did not but I expect the the. Program transformation section is going to or like the program transformation chapter is going to split and will have. For like we'll have a bunch of sections and one of them will be on like.

Proof by reflection and then another one will be on like reification and some part of that will be on reification by parametricity. It's probably the case of the content from that paper as long enough that it shouldn't just be one section within a chapter. It's the reader. Feel happy about making a progress if you don't have evidence long chapters.

Length of an independent research paper as a good standard for roughly how long a chapter can get I could instead of calling them chapters call them like half parts yeah and I'm beneath that have chapters and have like perk introduction that has the like true introduction and then the like here is called and performance issues in call and then part conversion.

And then part.

I don't know what to call the next word. It's not you're writing in production. I mean, maybe it's rewriting in reduction but it's also like it also covers fruit by reflection in general, but if you're general pattern is apart of corresponds to a problem and then that treatment of each problem you introduce a solution including the background for it then makes sense to me, then you'd have reflection be introduced within there well within the first portion be conversion, it should be API design or something, okay?

Yeah, I could do that. Oh so API design is a part and then in part that's on rewriting and reduction.

With chapters on. The like introduce the problem the talk about proof by reflection talk about reification and reification by parametricity.

And then talk about the rewriter and then your like talk about the performance of the rewriter and the like broad strokes of it and then talk about the technical details and challenges and implementing the rewriter and then there's like a part conclusion.

That talks about the like let's look back on on performance over the past decade and talk and see where we've like made strides and what this can say about future proof assistance and then also look forward and be like here's the like next challenged tackle and performance of previous systems like call.

Yeah. Sounds good to me. Cool was that the level of detail you were looking for here home.

I think I. To the part that I feel most fuzzy on still is the introduction, oh.

I could go into more detail and tell you it feels like I've given a good level of detail of the outline, okay, oh.

Except it feels like I still don't know how to do the introduction or split it up. Okay, so you want to drill down I think I want to drill down more on the introduction unless you think it's better to save the introduction part for later. Do we have anything higher priority by later?

I meant after I read the other parts of the thesis or something, okay? You try to productively use the remaining time in this meeting and if you don't have a better idea of what to do, we should talk about. The only other idea of that I have for what to do is to tell you more bits of the story for from the other parts.

When it sounds like you're much more confident that you have that story inside you. Yeah, it feels like I'm I'm like still a little bit fuzzy, but I like every time I tell that it gets more clear and I feel confident that this will continue to be the pattern.

Whereas for the introduction, I feel like Like every time I tell it it's completely different and. It's not getting any more clear. You know, most people these things don't take shape until they're actually writing. You didn't necessarily expect to reach it fixed point by speaking it over and over again.

Yeah that way I feel like I'm I'm trying out a methodology of this like recording and using transcripts. Yeah and.

Then like taking the transcripts and putting them into tech and polishing them and soon it might be the case that's speaking it. Is closer to writing. Okay the way I'm trying to approach this.

1.C more Adam transcript

So tell me the story of land called Intro. Okay, so. One cold in true has. Two three chapters.

It feels like I I know how to end the introduction more than I know how to start it. How do you end it? Oh so I ended with a the like final chapter in the introduction is a sort of a painting of where map of like. This is what performance and caulk looks like.

1.D more Adam transcript

Ah. Okay, so things that need to go in the earlier parts. What are what is proof assistant and what is cock? To proud criterion.

Whatever dependent types. Maybe like what is what is conversion what is or I don't know if I introduce conversion on the here if I save it for later. Should you included what is a proof assistant but you didn't include why do we care about them, okay, maybe you all said mind.

I mean, I do have it in mind but I didn't have it in mind here idea why do we care about purpose systems, what are they already been used for was the basis of confidence that this is a painful tool? So this is like like look at all the prior work things that.

Shouldn't shouldn't less literally yeah.

And then why do we care about performance and purpose systems?

I do worry that there is prior work that I'm not going to find on performance improvement systems. I feel like I'm not currently aware of much other than maybe now there's. I think there's like making. I feel like there's some things that like touch on performance, that's like. Canonical structures for less ad hoc automation and.

The maybe some of burglaries stuff on reflection. If yep stars not raise and so forth native stuff.

Yeah.

I can't explain to you don't exhausted literature search there myself, okay. I shouldn't it's done some time it's returned alert literature yeah. I'm spending a few hours trying to find some other things out there. I remember what I did that with a much smaller time parameter than a few hours right before the poll deadline.

I found this this paper from the the Isabel crowd doing allows him with things and, Wasn't the ideal time to realize that yeah for the the rewriter paper yeah, yeah. Apparently we just never done a web search before for. Something like rewriting normalization by calculation proof assistance came up pretty quickly, yeah.

I feel like we ran into a similar issue with pressures and that I dove into implementing part series without having read any of literature on pursuers.

So what's these? Push down literature search. Early in the writing process this time even if it's too late to be early in the research process, you know. I will in to do that, okay?

Happy set enough about chapter one. I don't think so. I think we've like thrown a bunch of things out but I like and like I know how to say a little bit about each of them, okay, but I don't I don't know how to say enough about or like I don't know what is enough about each of them some of them.

I don't know how to say enough about them and I definitely don't know how to weave them into a story. While you're trying to introduce main performance element aspects of performance metal and aspects of. Particular. Design philosophy using caulk in some related systems, you're trying to explain why they were introduced originally.

And something of the challenges for the user that they introduced.

Were trying we need to keep our focus on not having this come across as here's a system that someone threw at us and we figured out how to use it well. That could be the nature of the experiment we ran to answer larger design questions, but we have we want to keep relating back to the larger questions.

I feel like when I when I try to imagine the intricate trend I keep running into the problem where. I can like. Say a lot of things and eventually I'll get to where I want to go, but. The it feels like I'm going to leave the reader not knowing what we're doing for the chapter or two.

That's good it's good it's it it's good relative to what's possible. What we're doing is so technical that we need to introduce a lot of background before anyone can appreciate it.

I feel like it would be good to give them a sense and maybe even like part of me wants to be like we should give them a sense in the first paragraph that we're like, I don't know dealing with performance issues and verified in life. Making. Systems not have bugs or something okay, yeah if we're just literally putting that phrase into the first paragraphs.

Sounds plausible. So that would evangelize more detail. I've been something that's like roughly it feels like sort of what I'm want to do is I want to orient the reader. I want to be like here here's what here's the broad thing of what we're going to look at and I want that and I don't know the first sentence the first paragraph.

That worries the like first sentence of the second paragraph. And then I want to like introduce some amount of context and then be like okay now that you have this context. I can orient you better like here's a better version of what we're going to be doing and then that paves the way for more context and then I can be like, okay now that you have this more context.

Like here's an even better version of what we're going to be doing and either that will be the last iteration or one more iteration and be like okay now that you have

all of the context now I can actually start talking about what we're doing you just need to give yourself permission to write low-polity texts ready to revise it later, ah feels like a skill that I've never learned when I'm very handy one.

It feels like so the skill that I do have is like talk to someone with a low quality explanation, okay, and then as they express confusion revise on that. And it feels like that's a suddenly different skill yeah. It's just really hard to get oh you work your way up to complex information if you're just speaking it.

There's a reason we use written explanations, what do you mean? Working memories not sufficient to. Receive a complex idea just by listening to someone talk with no other visual aids up. So it may be that you're. By forcing yourself to use the conversational medium, you're so eliminating you're the set of what you could possibly convey you for you restricted your attention to such easy things and you feel like you're making progress, but oh.

It feels like the things that I can convey to the conversational medium or enough to get me to the point where I'm comfortable writing details or something, okay? Like it feels like like we have the rewriting paper and like, Even if I throw out the introduction bits of it.

I feel like I should be able to get to the point where I should be able to get up to the meat of it with the conversational medium and then just take the written made of it. I feel like you might need to point to code examples to do that.

I could believe that.

It feels like I'm floundering much earlier in the process.

Like I'm floundering that the orient the reader step. You know.

Tempted to just try again with the. Telling you that to know that I have this picture about what I'm trying to do with the introduction which I did not have before okay attempted to just try again to give you the story of the introduction, okay for minutes give you as much as I can give you in three minutes, let's do it, okay, so.

Story is that Jesus is going to look at. How.

We're looking at verified or at getting systems that don't have bugs in them and how to. Be performant when doing this what's hard about being performant and like how to. Succeed you haven't mentioned a aspect of foundational tools that a small trusted basis. I think that is central to this.

Depends on how you interpret it says it's not having buck you might be worried about bugs in the fall methods tools. Which case perhaps this is the unified. Oh people

could use the nudge with a more explicit framing. So, I feel like I want that to be in the.

Non leave like super initial contacts but in the in the like background after the first contact setting that's like, okay the way that we're going to the like tools that we're using for not forgetting systems without bugs are proof assistance and these are foundational tools and here's this large body of work that's about how this has been useful and like why this is a reasonable way to get systems without bugs make sense as a buildup principle for the introduction.

I think throughout most of the paper you want the top-level frame of the problem people's minds to be fun. Damentally about foundational tools. Yeah, I think I'm going to.

Like. I think by paragraph three. I want to stop talking about or will. Yeah, I feel like by something around paragraph three of the introduction. I want to like like we're not. We're not talking about other ways of getting systems without bugs, we're talking about proofs. And purposes those sorts of foundational tools, okay?

And.

I feel like now I need to say something about performance and I don't know what to say.

Can't use the tools. I don't get what's that's kind of the maybe I just say that oh and then.

So I'm tempted here to be like okay and like this is this is what makes performance in pure physicists different but I feel like. You're suggesting from earlier was don't do that here. Like like my story wasn't compelling enough about why performance is bad and purpose. I think talking about this dope the proof system is this organism we found in the jungle we're going to tell you what's why isn't how you deal with it isn't near this convincing as talking about a fundamental trade-off between flexibility and trust.

Okay, so it goes here is where I. Want to introduce the grind criteria.

And I feel like then I want to introduce dependent types and maybe this is very.

Um and that. I'll need to like pull cockclub or something. And then I feel like now I want to reorient it's also three o'clock. Okay, oh and then I want to like reorient where. Like what we're going to be doing or something. Like spiral back to the.

So we're looking at. Performance improvement systems and these two issues you're going to generate. Like broad swaths of the. System where performance issues occur

and I'll be talking about. What the performance issues look like and also ways to solve them. And then after that I get more context on.

In cock here is this palette of performance issues and like what they look like okay, so that seemed like a good introduction sketch. That level of abstraction you have my. Cult thanks sure. I'm assuming that you don't want to have any input into this strategy.

1.E Transcript bits from Talking with Rajee

High level story:

Coq and proof assistance are important. Performance in them is important, especially at scale. Performance engineering in proof assistants has some unique challenges that don't show up in other programming languages.

And I want to paint a picture of what the unique challenges are. There are two main areas in the existing system that I want to call attention to, in regards to performance bottlenecks. I will describe them, and describe the performance issues, and propose some reasons about why there might be performance bottlenecks, and describe solutions for them. And maybe also there'll be another section that has some miscellaneous other performance bottlenecks.

[**TODO:** where does this description of Coq's design go?]

1.F Coq's design

Coq is split into two parts.

There's the part of the system that is called the kernel or the trusted code base. Once you get a proof this part will be like "yup, I believe the proof" or like "nope your proof is bad." And then there's the other part that is like "here's magic and it will make proof for you." You're like "I have an arithmetic expression please prove that it's true" and there's a bit in this other part that's like "I know how to prove arithmetic expressions" and it gets the arithmetic expression and then it generates a certificate or proof that this other trusted part checks. If the part generating the arithmetic proof is wrong then the users come complaining to you that you have a bug. If the part checking the proofs is wrong, then you don't see the bug and now suddenly your users can prove anything they want. And the system is no longer trustworthy right and so for that bit of it you need to be very careful with any changes you make.

[**TODO:** this chapter]

Chapter 2

The Performance Landscape in Type-Theoretic Proof Assistants

2.1 The Story

[**TODO:** come up with a better name for this section] 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-of** 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 [**bogosort**], 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 [**bogosort-name**], 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. [**TODO:** cite https://github.com/coq/coq/issues/12200]

In fact, this issue already arises for compilers of mainstream programming languages. [**TODO:** literature search on perf of compiletime in mainstream compilers?] 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.[**TODO:** cite?] 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[**TODO:** cite fiat-crypto] 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

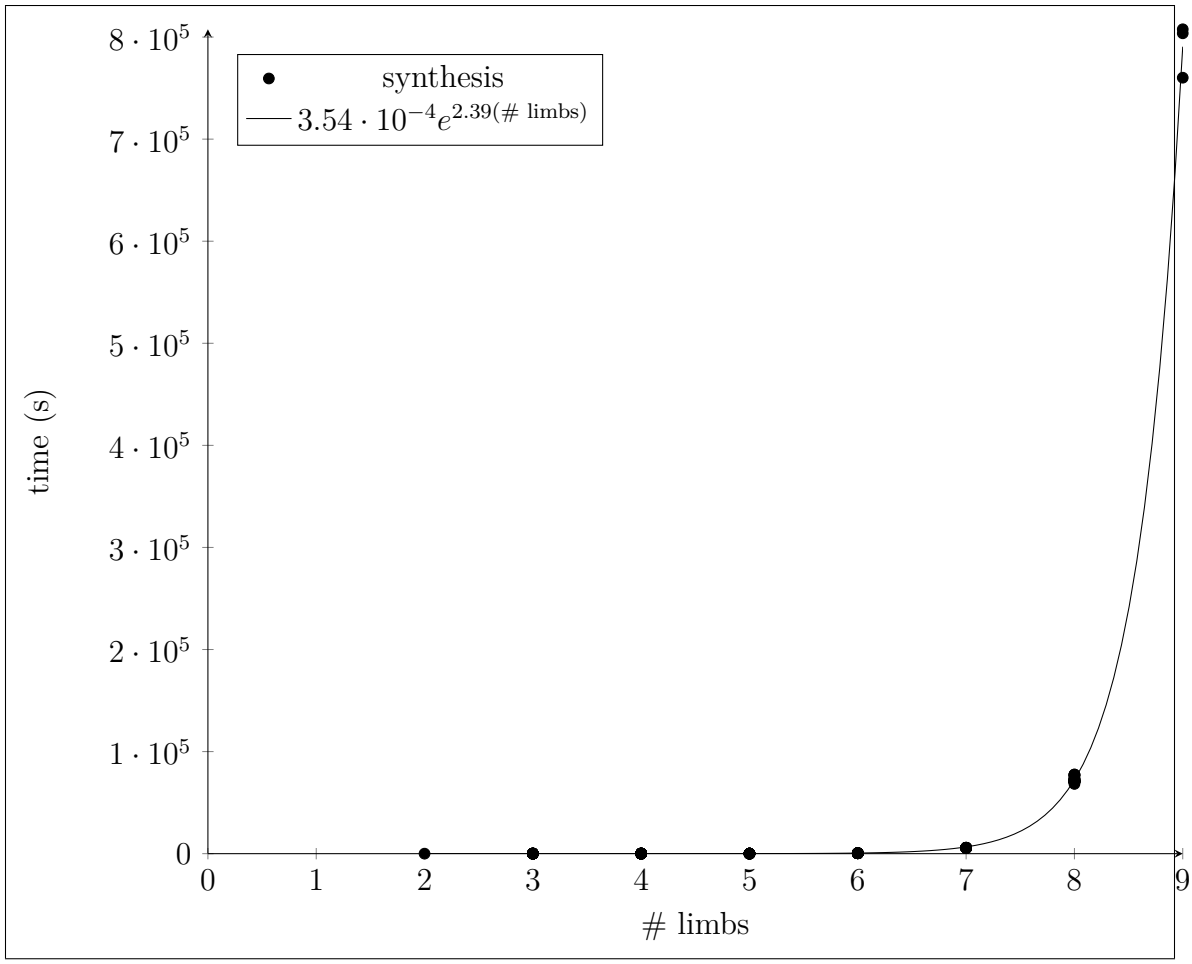


Figure 2-1: Timing of synthesizing subtraction

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 example—17 machine words—would have taken over four thousand *millenia*! 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

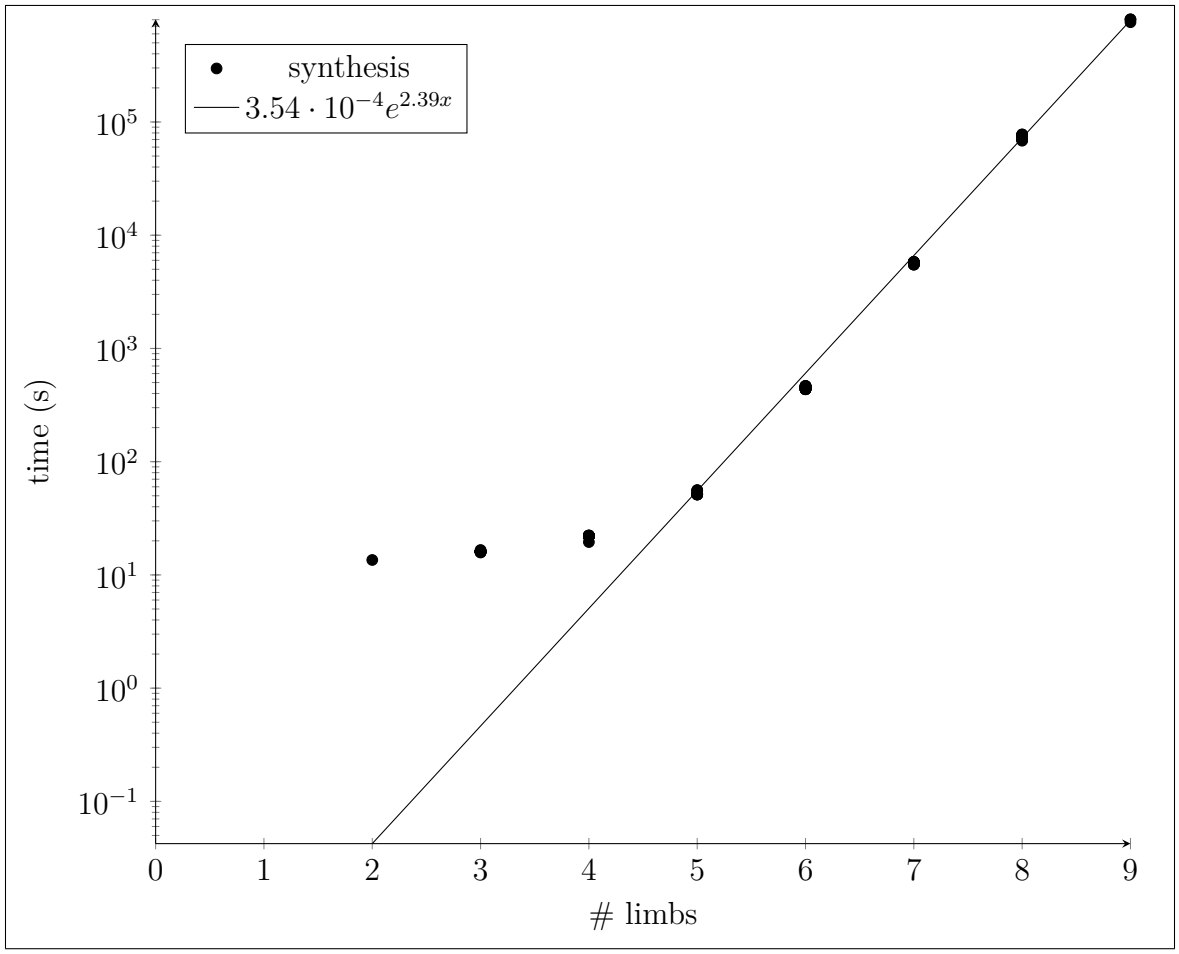


Figure 2-2: Timing of synthesizing subtraction (log-scale)

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 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 **??**). 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

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

[3] It's 200, for those who are curious.[**TODO:** cite https://gcc.gnu.org/onlinedocs/gcc-7.5.0/cpp/Implementation-limits.html]

code and reading the error messages it gives you. These are details that should not need to be exposed to the user, but because dependent type theory is so complicated—complicated enough that it's generally assumed that users will get *line-by-line interactive feedback from the compiler* while developing, the numerous design decisions and seemingly 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. [**TODO:** incorporate Andres' suggestions:] you running the typechecker in your head is essentially the statement that if the entire implementation is part of the spec, it is possible to engineer better, and something close to this has been necessary in practice. the research direction you are advocating is finding a simpler performance-aware spec (perhaps by moving around interfaces or etc; my thought is that maybe we just want to get rid of the kernel and trust the proof engine).

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

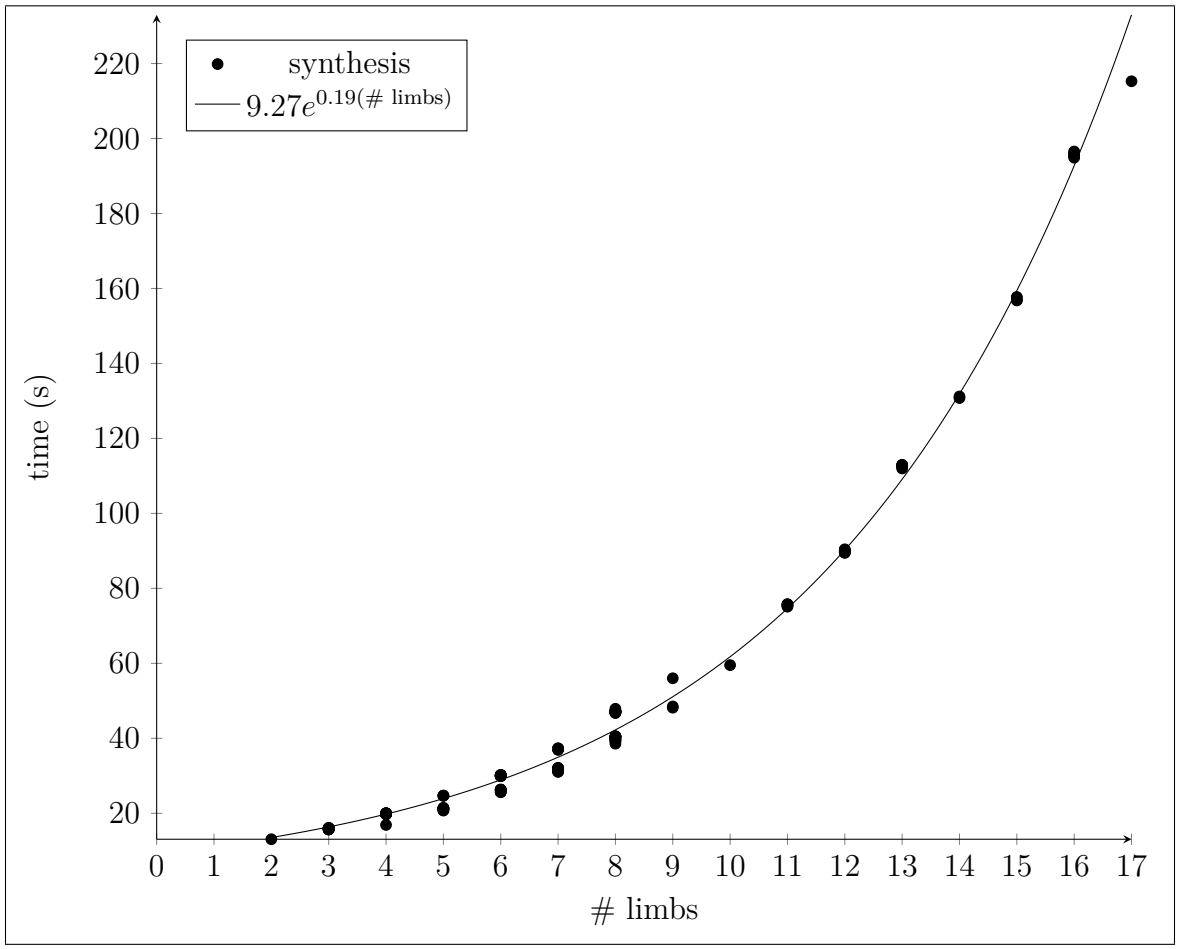


Figure 2-3: Timing of synthesizing subtraction after fixing the bottleneck

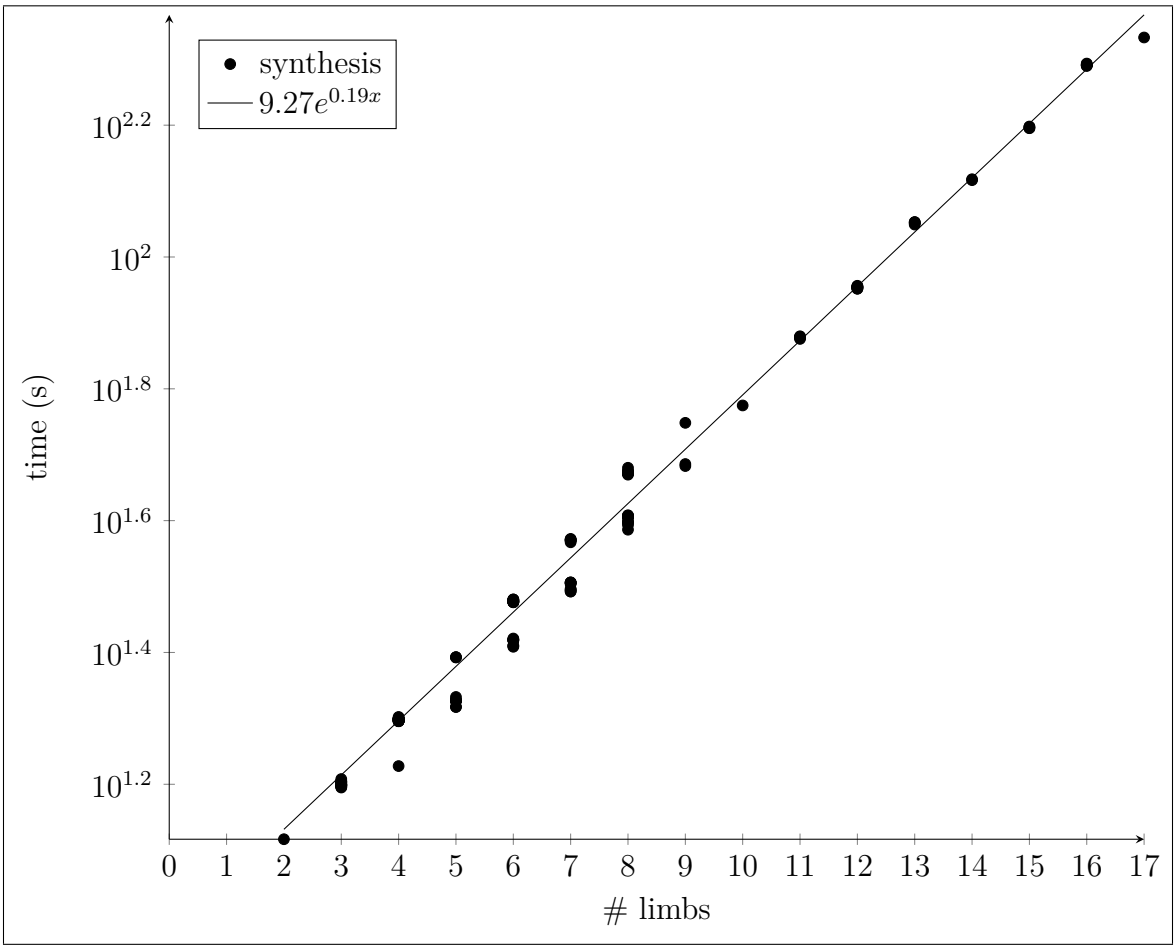


Figure 2-4: Timing of synthesizing subtraction after fixing the bottleneck (log-scale)

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 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:** incorparate 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. [**TODO:** introduce this section better]

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

Recall from Subsection 1.2.2 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.

[**TODO:** Mention possibility of not building proof terms at all somewhere] Andrew Appel said via private correspondence on May 7, 2020, 7:39 PM:

> There's some work on typechecking LF, in the Twelf system, where there can be performance bottlenecks if you're not careful.

> The most glaringly obvious performance bottleneck in Coq is that it builds proof terms, when one should really use the futuristic technique of using data abstraction, in the type system, to distinguish "proposition" from "theorem"; as done in that state-of-the-art system, Edinburgh LCF. And presumably HOL, HOL light, Isabelle/HOL, etc.

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. [**TODO:** improve data collection on perf test] [**TODO:** make a note about more complicated types causing scaling factors to be worse, and not just impacting the leaves]

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 [**structural-typing**] rather than nominal typing [**nominal-typing**]. [**TODO:** Find a place for this note:] 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. [**TODO:** maybe look into TAPL] [**TODO:** explain structural and nominal typing more] Morally, the reason for this is that if we have an inductive record type [**TODO:** have we explained inductive types yet?] [**TODO:** have we explained records yet?] whose fields do not constrain the parameters of the inductive type family [**TODO:** have we explained parameters vs indices and inductive type families yet?], 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[**TODO:** mention where the name "phantom" comes from?]

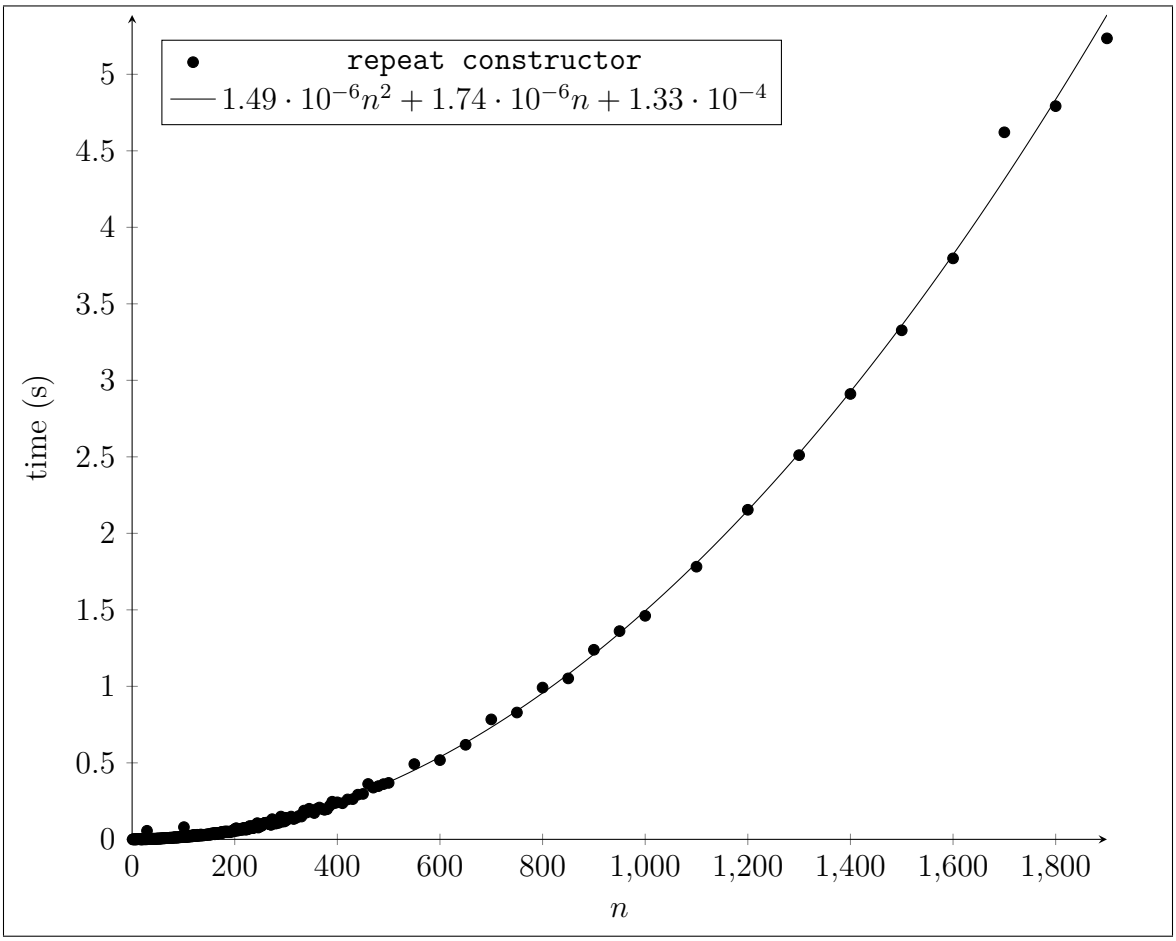


Figure 2-5: Timing of **`repeat constructor`** to prove a conjunction of n `True`s

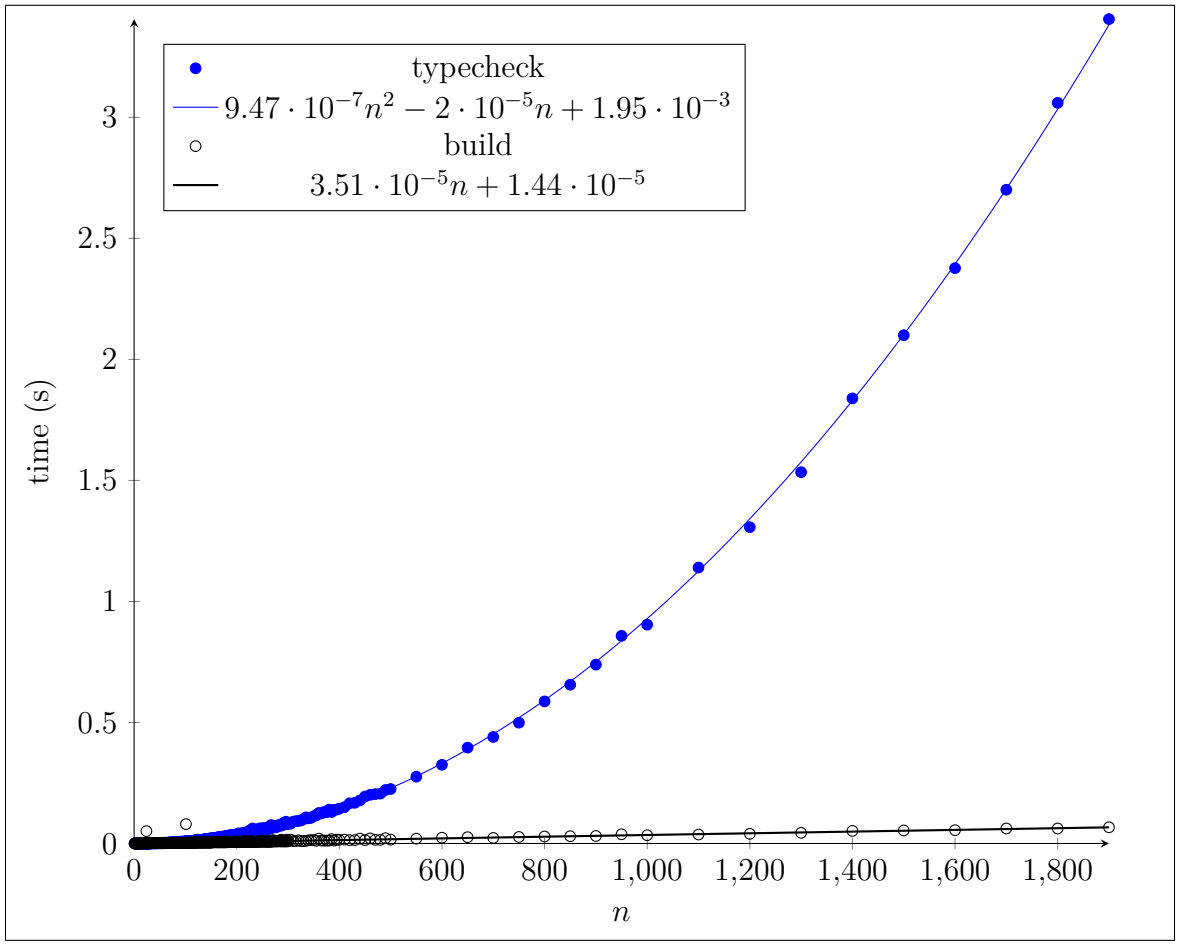


Figure 2-6: Timing of manually building and typechecking a certificate to prove a conjunction of n `True`s using Ltac2

```
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. [**TODO:** Come up with better justification for having nominal typing available?]

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 [**TODO:** explain this better].

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[4] 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
```

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

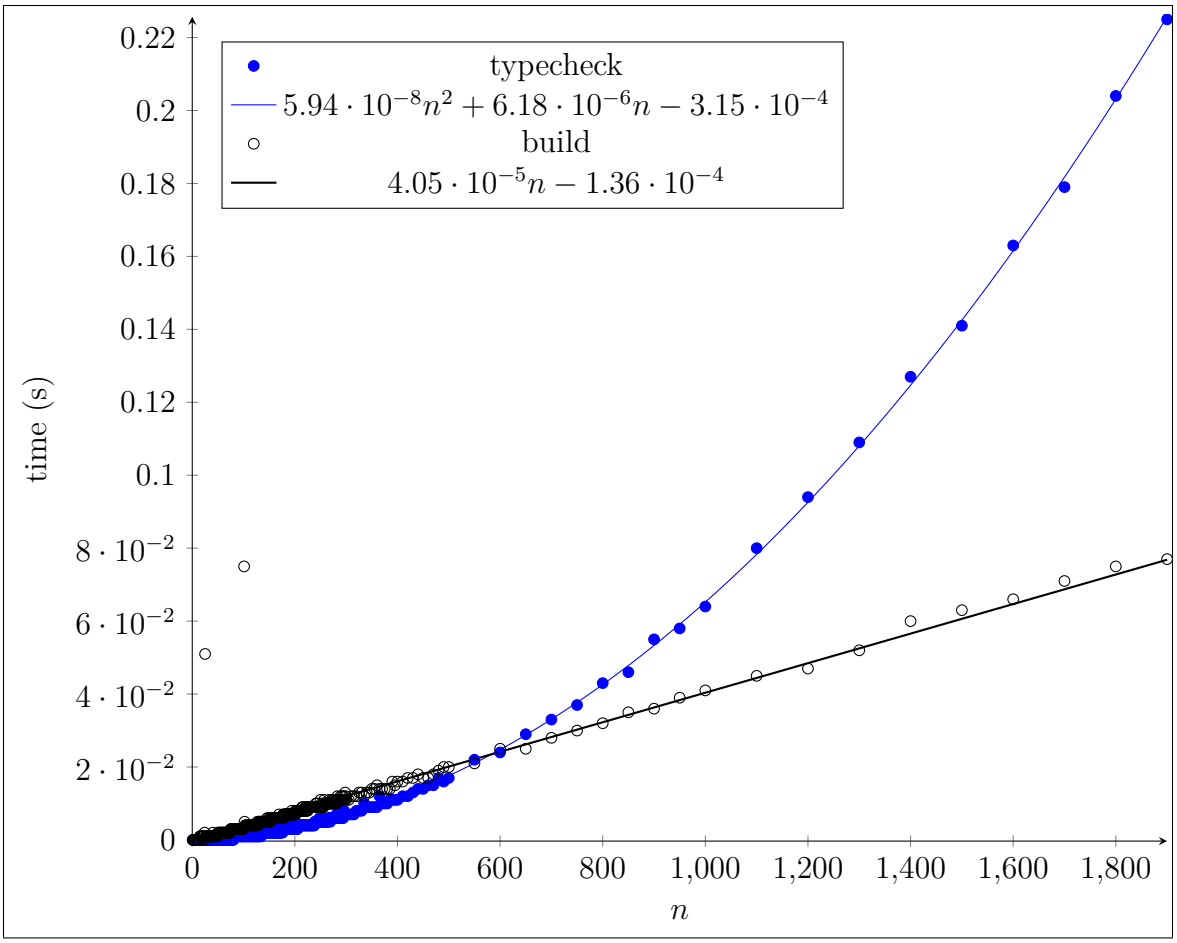


Figure 2-7: Timing of manually building and typechecking a certificate to prove a conjunction of n `Trues` using `let`-binders using Ltac2[**TODO:** find pdf of manual for styling]

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. [**TODO:** improve data collection on perf test]

Recall that the typing rules for λ and `let` are as follows:[**TODO:** cite appendix with typing rules of Coq?] [**TODO:** maybe look in https://github.com/achlipala/frap and/or TAPL by Benjamin C. Pierce for how to render typing rules] [**QUESTION FOR ADAM:** What's the suggested way of pretty-printing typing rules?] [**QUESTION FOR ADAM:** Which way do substitution brackets go?] [**QUESTION FOR ADAM:** What convention should we use for typing rules with regard to things being types? Maybe just copy the HoTT book?]

```
Γ ⊢ A type      Γ, x:A ⊢ B type
Γ, x:A ⊢ f:B
------------------------------
Γ ⊢ (λ (x:A), f) : ∀ x:A, B
```

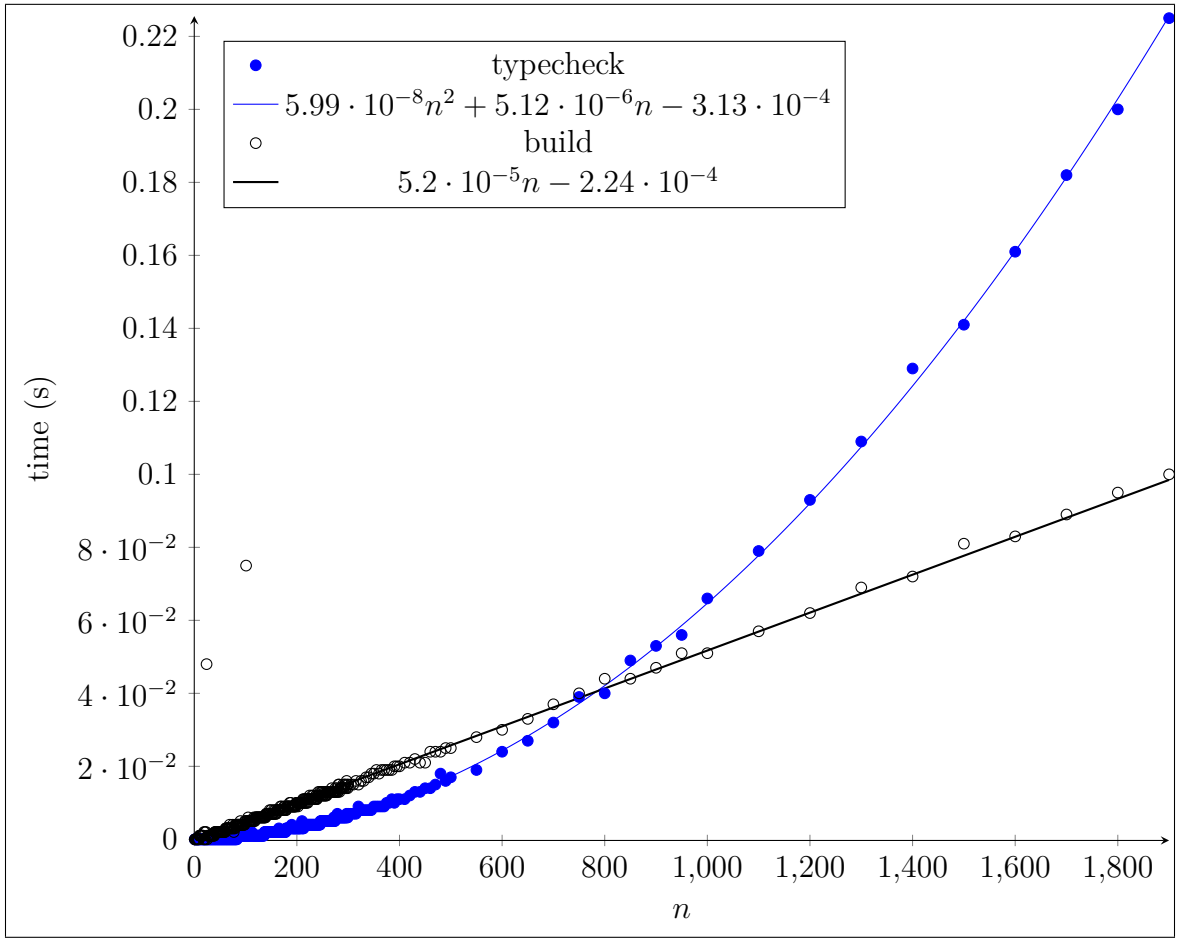


Figure 2-8: Timing of manually building and typechecking a certificate to prove a conjunction of n `True`s using abstraction and application using Ltac2

```
Γ ⊢ A type      Γ, x:A ⊢ B type
Γ ⊢ f : ∀ x:A, B
Γ ⊢ y : A
------------------------------
Γ ⊢ f y : B[x/y]

Γ ⊢ A type
Γ ⊢ y : A
Γ, x:A:=y ⊢ B type
Γ, x:A:=y ⊢ f : B
------------------------------
Γ ⊢ (let x : A := y in f) : B[x/y]
```

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

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

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

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

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

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

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

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

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

- We substitute `T1 := and True T0` into this type to get the type `and True (and True (and True` 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` 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. [**TODO:** cite https://github.com/coq/coq/issues/8232 maybe?]

[**TODO:** Should we run though typechecking in more detail here?]

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.

[**TODO:** maybe cite https://github.com/coq/coq/issues/11838?]

[**TODO:** reference quadratic cbv here, which had a similar issue?]

[**TODO:** some sort of section division marker here?]

The proof assistant can also try to heuristically share subterms for us. Many proof assistants do some version of this, called *hashconsing*. [**TODO:** explain and cite hashconsing?]

However, hashconsing 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. [**TODO:** cite hashconsing needing to be full to get perf benefit] Furthermore, it's an open problem how to efficiently persist full hashconsing to disk in a way that allows for diamond dependencies. [**TODO:** explain this more, find citation for hashconsing being hard with disk] [**TODO:** flesh out hashconsing section more] [**TODO:** maybe cite https://github.com/coq/coq/issues/9028#issuecomment-600013284 about hashconsing being slow]

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.

[**TODO:** maybe move some text form the sharing section to here?]

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.

[**TODO:** discussion of conversion checking, and conversion modulo delta-beta]

Consider the problem of typechecking `@eq_refl nat (fact 100) : @id nat (fact 100) = fact 10` where `fact` is the factorial function on natural numbers and `id` is the polymorphic identity function. [**TODO:** should we define polymorphic identity function somewhere?] 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 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

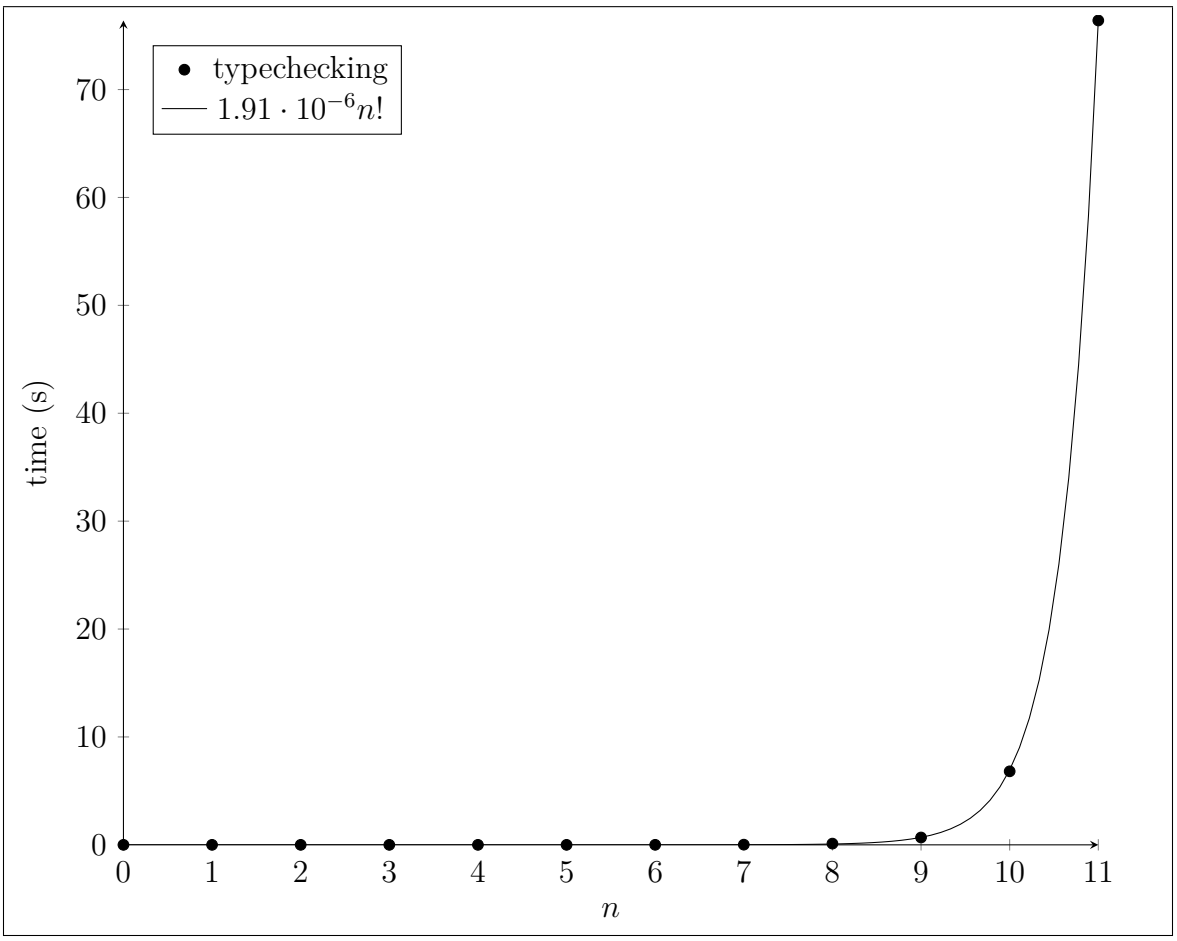


Figure 2-9: Timing of typechecking `@eq_refl nat (fact n) : @id nat (fact n) = fact n`

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 [**TODO:** reference above example about fiat-crypto], in fiat-crypto, we got exponential slowdown due to this issue, with an estimated overhead of over four millenia of extra typechecking time in the worst examples we were trying to handle.

[**TODO:** maybe forward reference to the number of abstraction barriers] [**TODO:** maybe include more about real-world fiat-crypto example here?]

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.

[**TODO:** make sure we've explained proof engine and Ltac by here]

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.

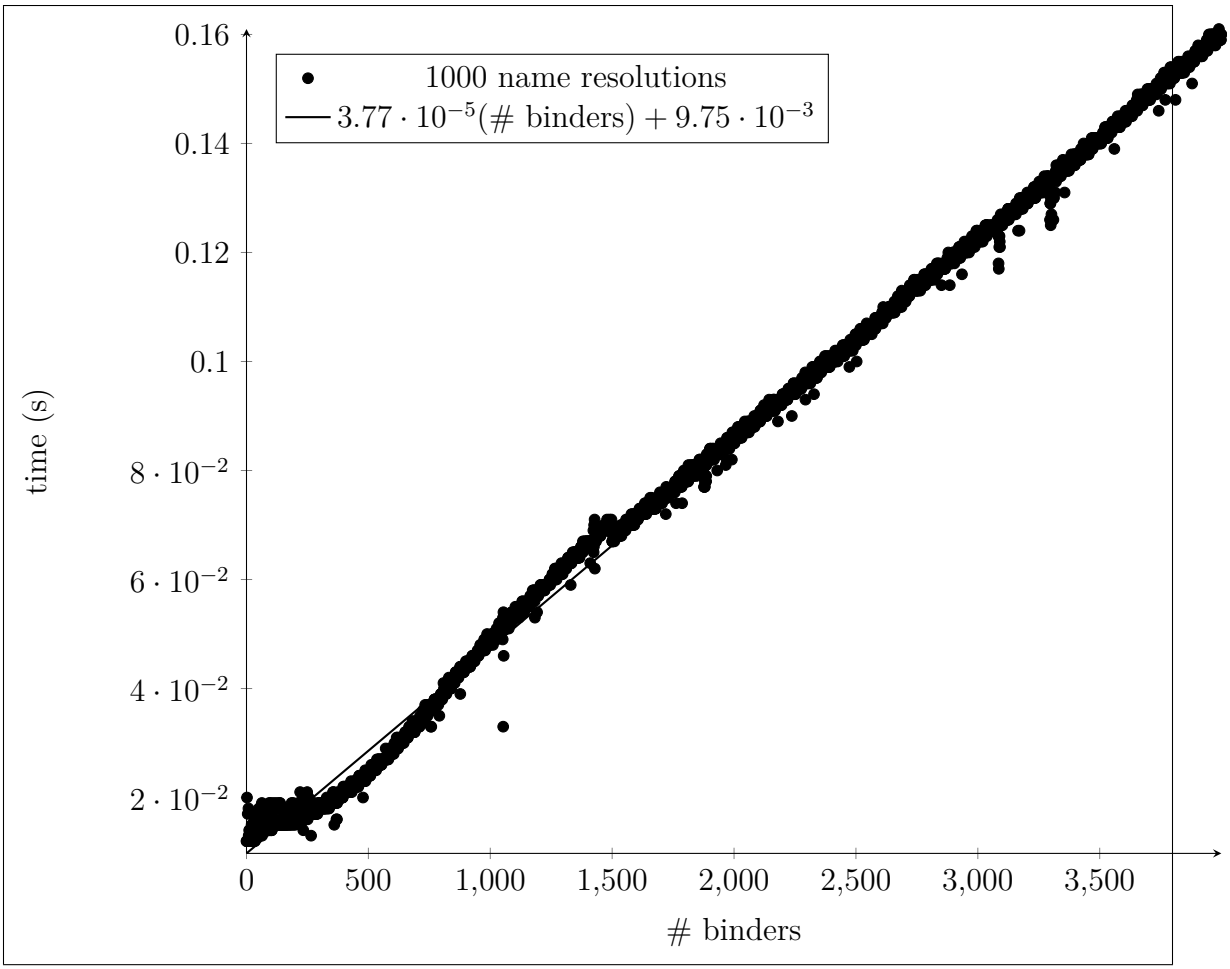


Figure 2-10: Timing of internalizing a name 1000 times under n binders

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.

[**TODO:** mention Coq version automatically, mention why we're using a different Coq version]

[**TODO:** Is this called internalization or is it called elaboration]

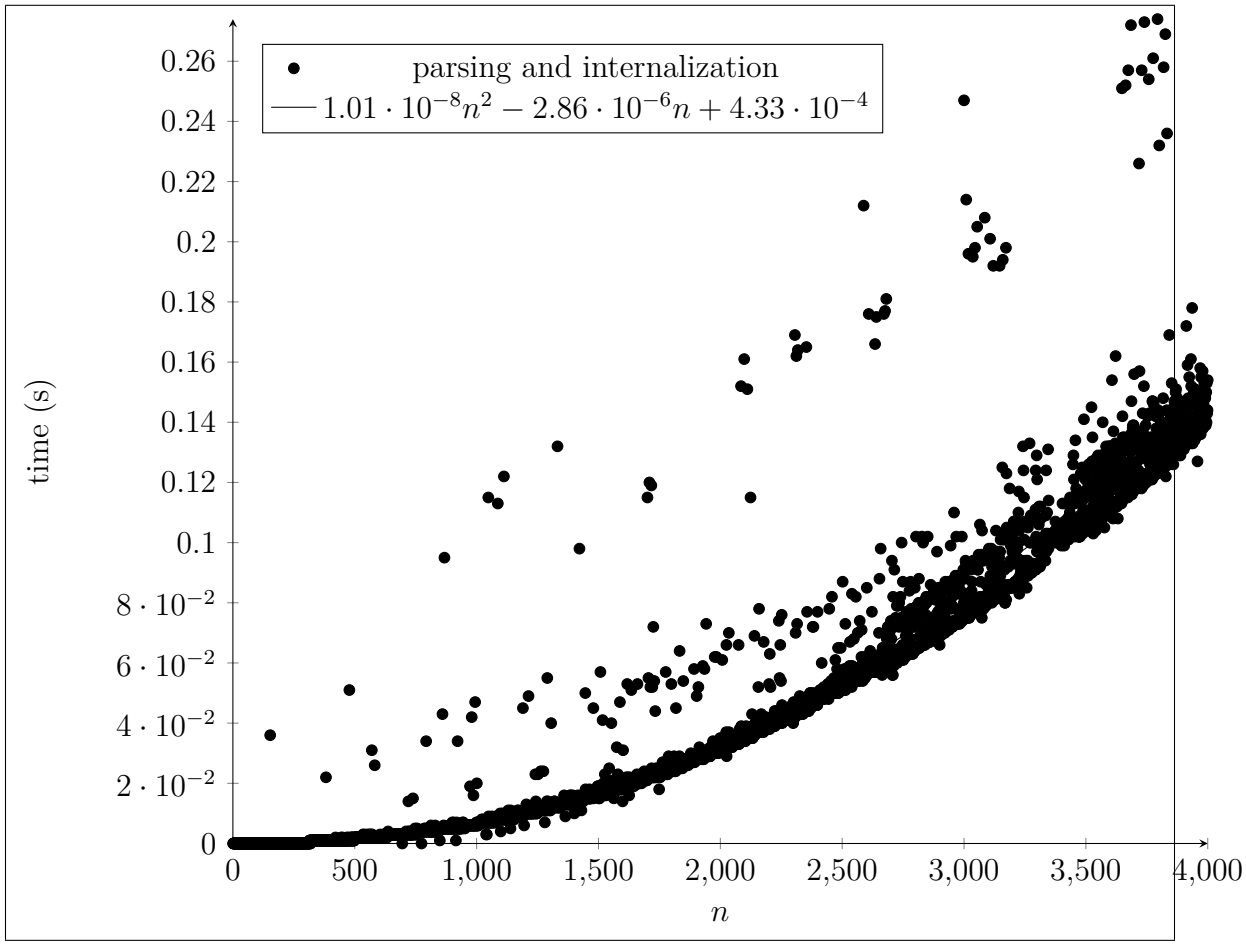


Figure 2-11: Timing of internalizing a function with n differently-named arguments of type `True`

Capture-Avoiding Substitution

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

[**TODO:** ADD PLOT: try to come up with a graph for renaming stuff] [**TODO:** ADD PLOT: check if confounders come up]

Quadratic Creation of Substitutions for Existential Variables

Recall [**TODO:** make sure that this is mentioned previously, and that we're not rehashing things too much] 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. [**TODO:** cite original idea for evars? (what is it?)] 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 β-reduce `(λ x, ?e₁) (S y)`, then the result is the evar `?e₁` 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. [**TODO:** define identity and lifting substitutions]

One popular representation is the locally nameless representation. [**locally-nameless**] [**TODO:** Justify it? Discuss other representations? Say why and how it's convenient?] However, if we use a locally nameless term representation, then finding a compact representation for identity and lifting substitutions is quite tricky. If the substitution representation takes $\mathcal{O}(n)$ time to create in a context of size n, then having a λ with n

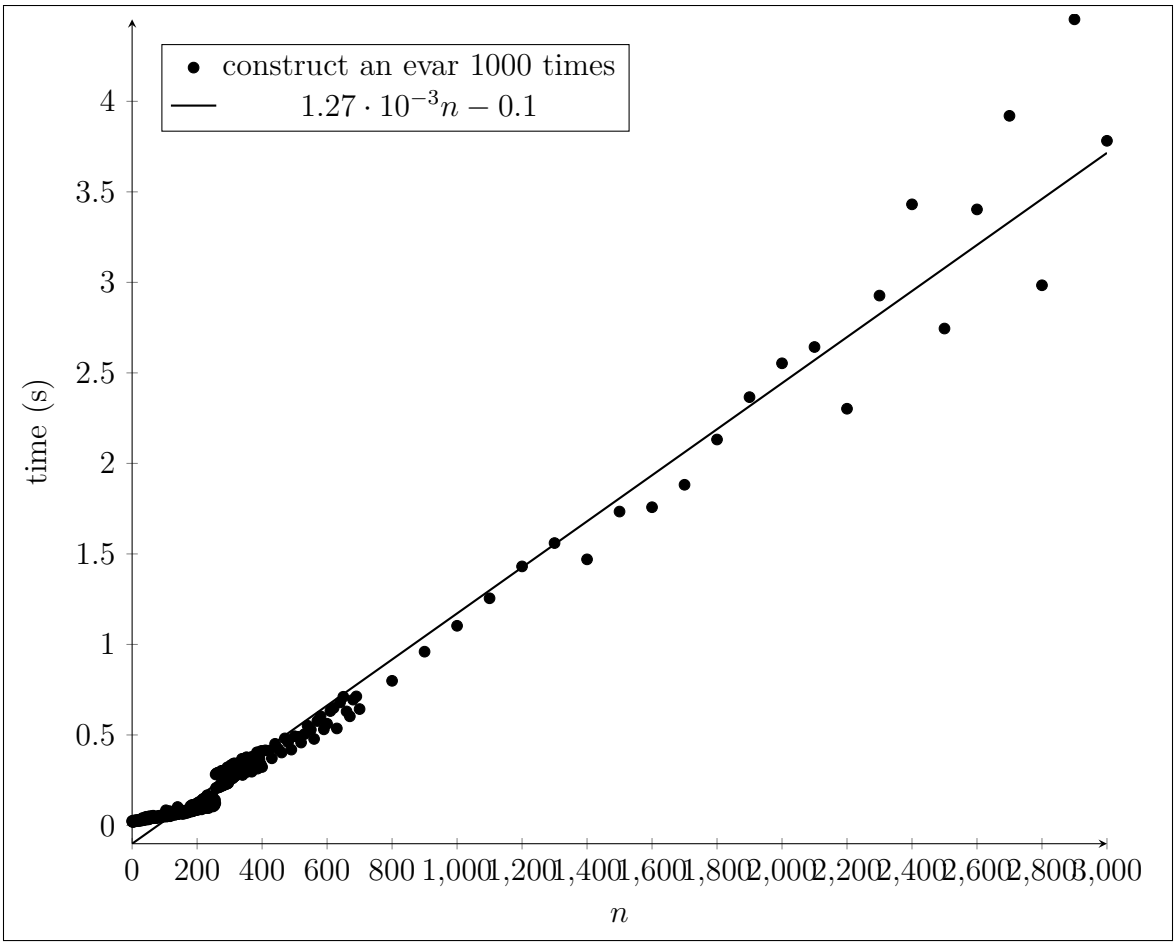


Figure 2-12: Timing of generating 1000 evars in a context of size n

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, [**TODO:** cite the rules / reference an appendix], 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.

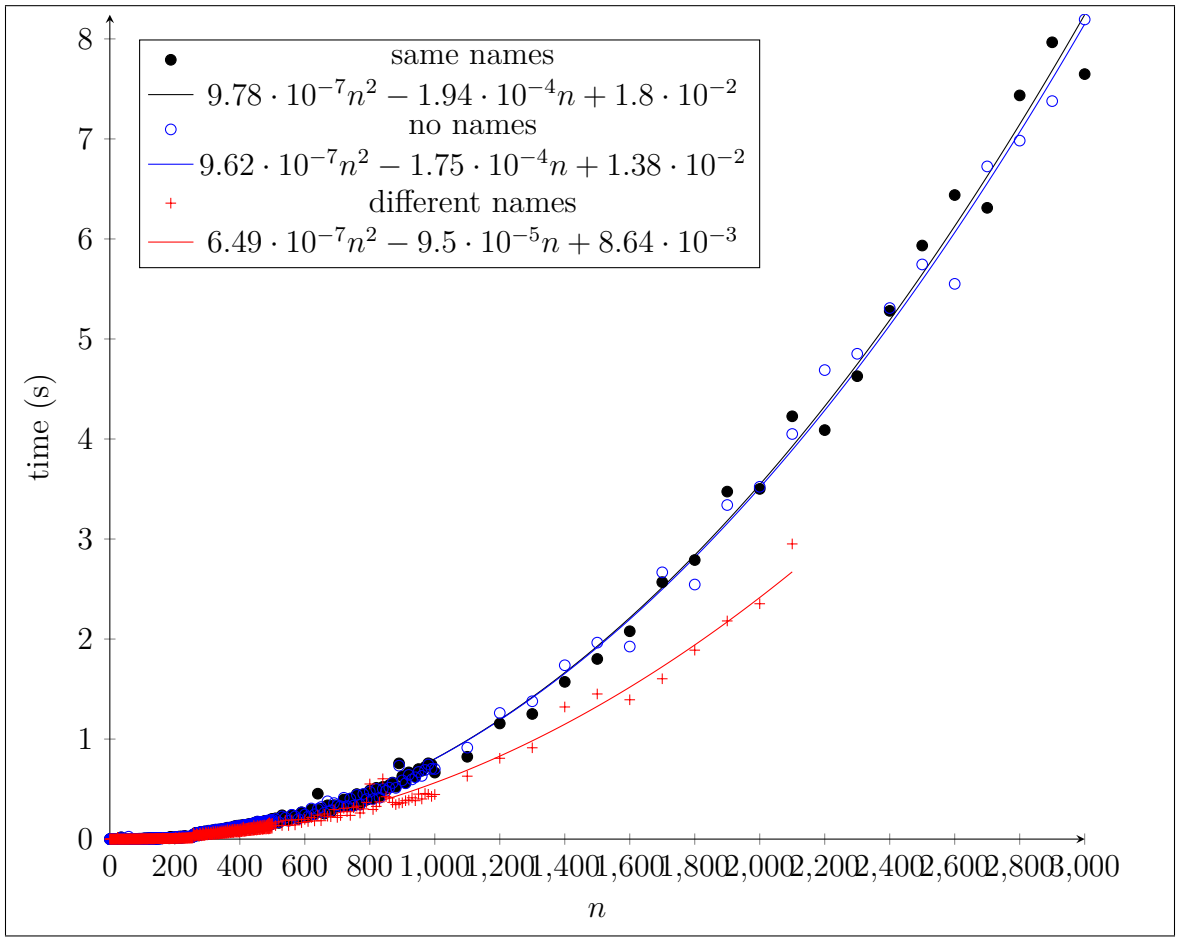


Figure 2-13: Timing of generating a λ with n binders of unknown/evar type, all of which have either no name, the same name, or different names

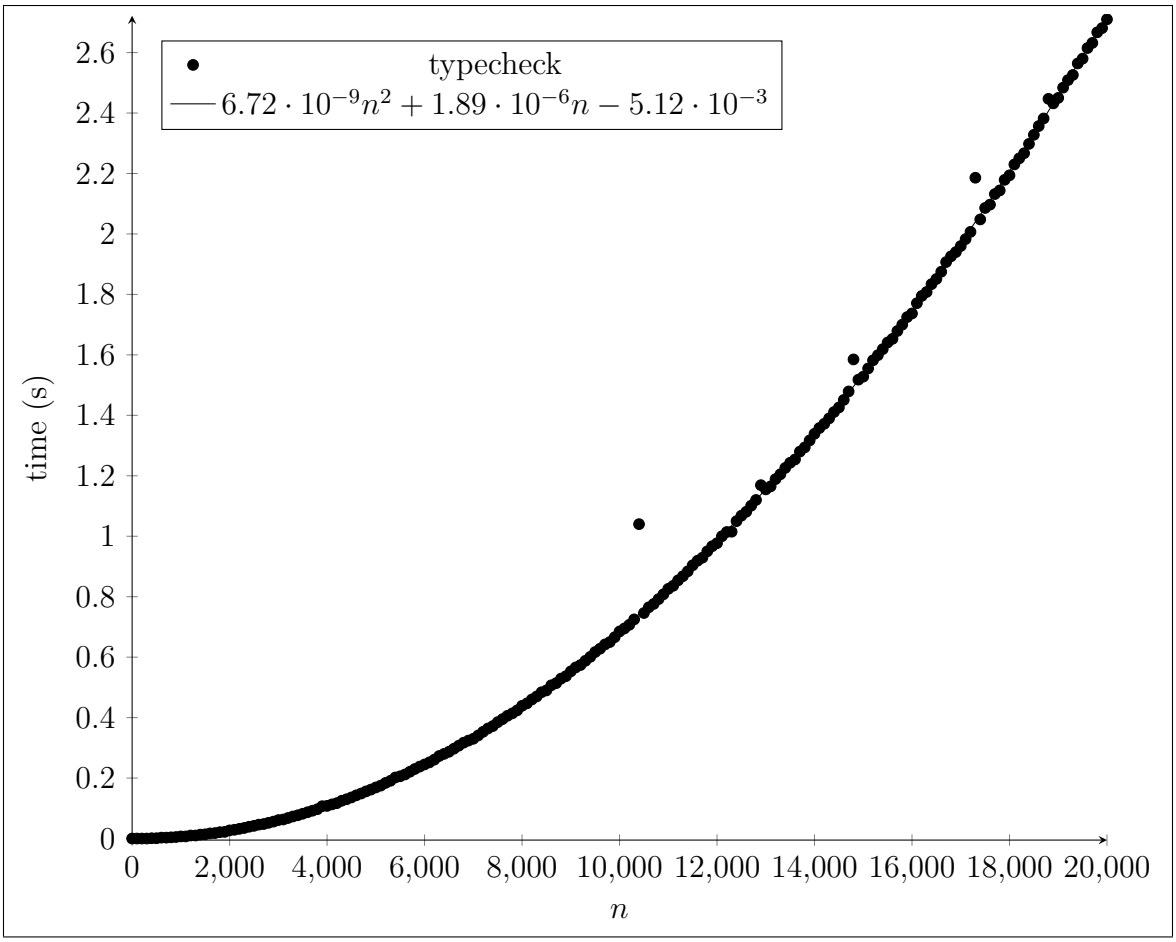


Figure 2-14: Timing of typechecking a function applied to n arguments

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. [**TODO:** explain or forward-reference NbE] [**TODO:** Read stuff in

https://github.com/HoTT/book/issues/995#issuecomment-418825844 for NbE stuff] [**TODO:** Say more about it than that it's "nifty"] 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 [**TODO:** reference explanation of PHOAS] 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. [**TODO:** maybe give a better introductory justification to the vm and native compiler than "it's important"?] 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. [**TODO:** cite https://flint.cs.yale.edu/shao/papers/escc.html or its references for closure compilation] 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 [**TODO:** cite something for flat closure compilation] 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

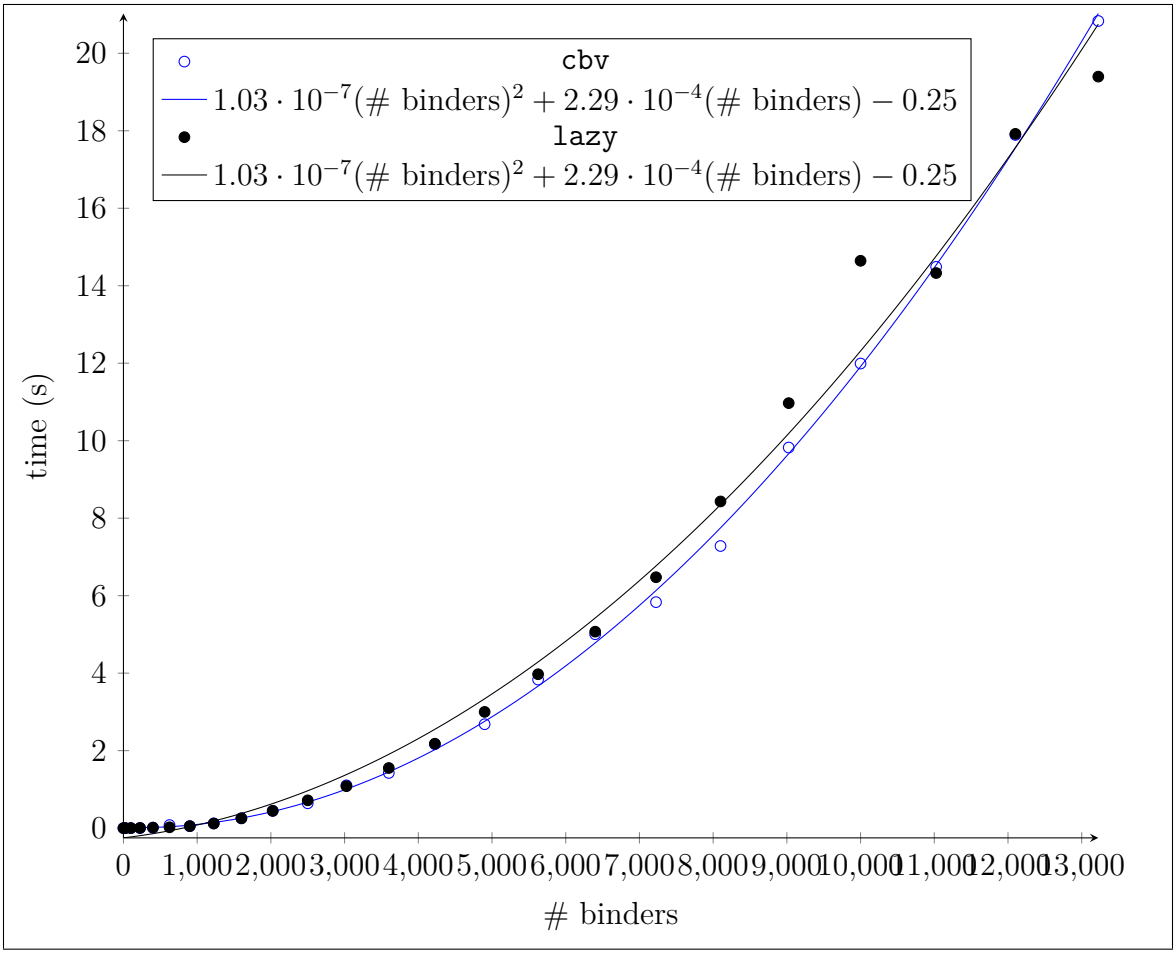


Figure 2-15: Timing of running **`cbv`** and **`lazy`** reduction on interpreting a PHOAS expression as a function of the number of binders

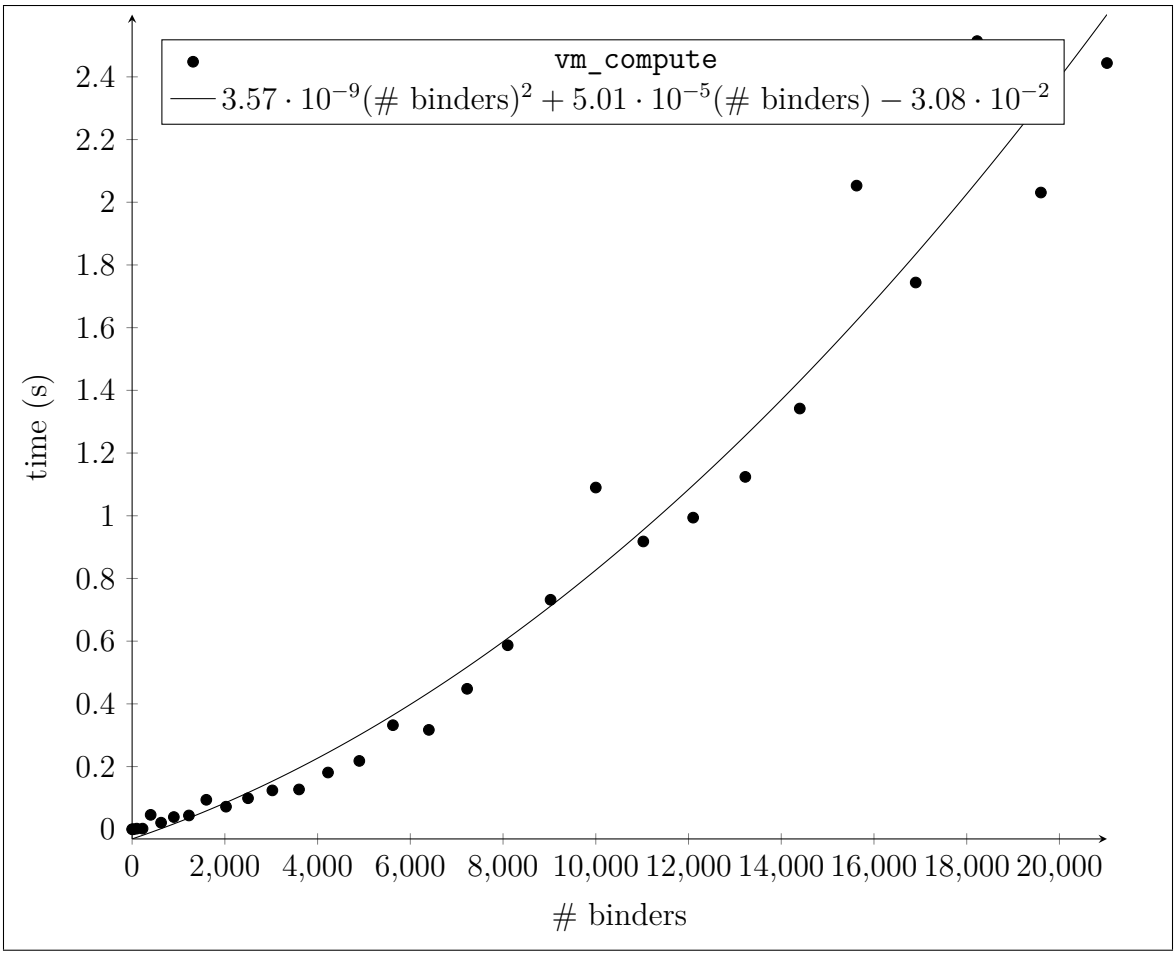


Figure 2-16: Timing of running **`vm_compute`** reduction on interpreting a PHOAS expression as a function of the number of binders

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. [**TODO:** reorganize this paragraph, improve it]

See also Figure 2-16 and Figure 2-17.

[**TODO:** maybe more examples here?]

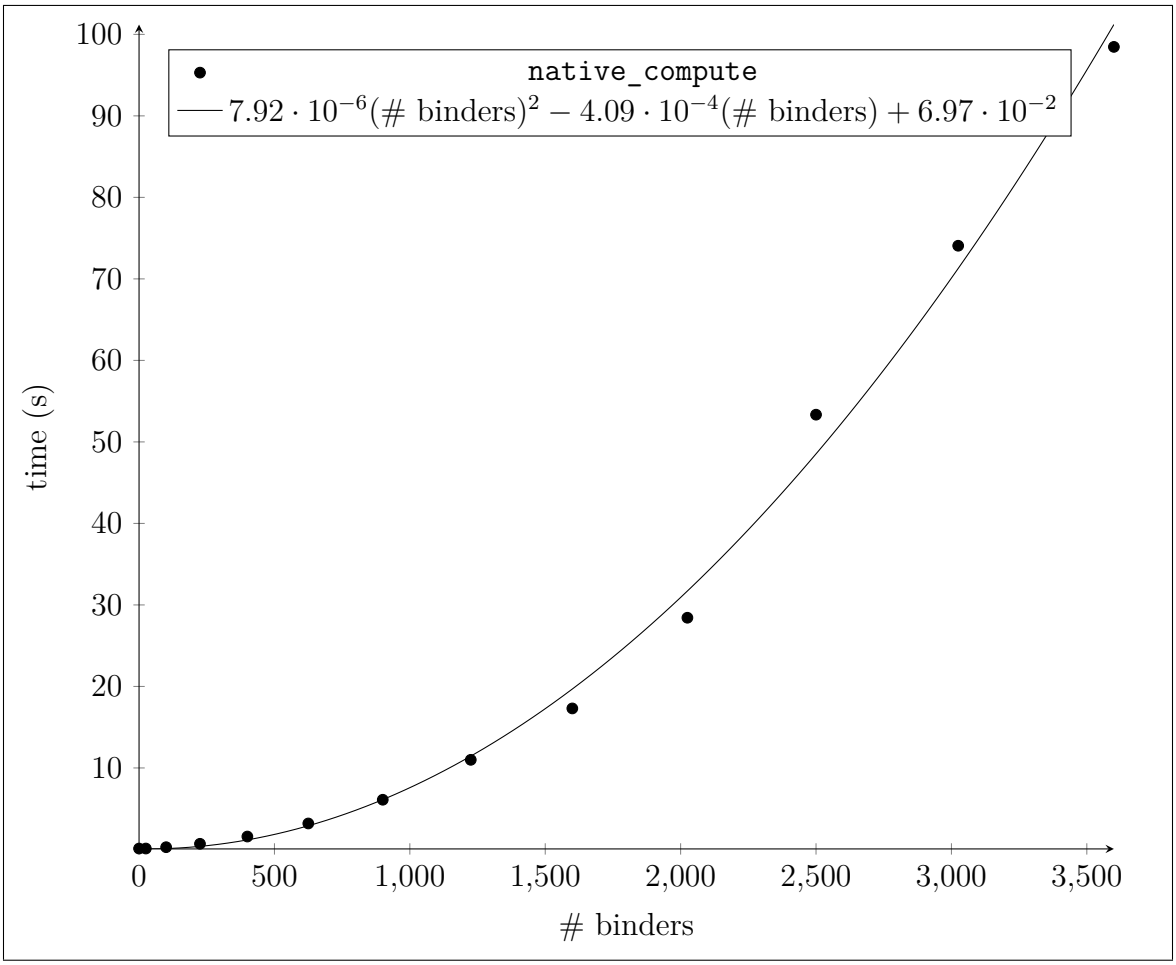


Figure 2-17: Timing of running **`native_compute`** reduction on interpreting a PHOAS expression as a function of the number of binders

2.2.4 The Number of Nested Abstraction Barriers

[**TODO:** ADD PLOT: This section requires digging into the historical performance issues around this to find convincing stand-alone examples so we can have graphs here.]

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. [**TODO:** should I spell this example out more?] 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. [**TODO:** add example?] 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.

[**TODO:** maybe be more concrete here?]

Type Size Blowup: Packed vs. Unpacked Records

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

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

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

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

4. (mostly unpacked) A *monoid on a carrier type* A *under the binary operation* $\cdot : A \to A \to A$ *with identity element* e consists of a proof that e is a left- and right-identity and a proof of associativity.

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.

[**TODO:** cite math-classes and HoTT for past design work here]

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.

[**TODO:** maybe figure out some examples and include perf data?] [**TODO:** does this need more exposition?]

2.3 Conclusion of this Chapter

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

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. [**TODO:** adjust the tone of what comes after the colon in the previous sentence to fit with this paper] 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 [**TODO:** cite math classes, etc], 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 [16], 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.

[**TODO:** flesh out this section] [**TODO:** More examples from the rewriter?]

3.3 A Brief Introduction To Our Category Theory Library

3.3.1 Introduction

Category theory [23] 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" [19], 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). Working around slow projections of Σ types (Subsection 3.5.4) and being more careful about code reuse each gave us back half of that lost time.

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 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:

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

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. [**TODO:** this is duplicative with text above]

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

It is generally straightforward to see that there is an isomorphism between a theorem and its dual, and the technique of dualization is well-known to category theorists,

among others. We discovered that, by being careful about how we defined 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 C^op y x H' H.
```

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

In contrast to the simplicity of witnessing the isomorphism, it takes a significant amount of care in defining concepts, often to get around deficiencies of Coq, to achieve *judgmental* duality. Even now, we were unable to achieve this ideal for some theorems. For example, category theorists typically identify the functor category $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$ (whose objects are functors $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$ and whose morphisms are natural transformations) with $(\mathcal{C} \to \mathcal{D})^{\mathrm{op}}$ (whose objects are functors $\mathcal{C} \to \mathcal{D}$ and whose morphisms are flipped natural transformations). These categories are canonically isomorphic (by the dualizing natural transformations), and, with the univalence axiom [34], 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 $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}} = \mathcal{C}$. In order to internalize proof by duality via judgmental equality, we sometimes need this equality to be judgmental. Although it is impossible in general in Coq 8.4 (see dodging judgmental η on records below), the latest version of Coq available when we were creating this library, we want at least to have it be true for any explicit category (that is, any category specified by giving its objects, morphisms, etc., rather than referred to via a local variable).

Removing Symmetry

Taking the dual of a category, one constructs a proof that $f \circ (g \circ h) = (f \circ g) \circ h$ from a proof that $(f \circ g) \circ h = f \circ (g \circ h)$. The standard approach is to apply symmetry. However, because applying symmetry twice results in a judgmentally different proof, we decided instead to extend the definition of `Category` to require both a proof of $f \circ (g \circ h) = (f \circ g) \circ h$ and a proof of $(f \circ g) \circ h = f \circ (g \circ h)$. Then our dualizing operation simply swaps the proofs. We added a convenience constructor for categories that asks only for one of the proofs, and applies symmetry to get the other one. Because we formalized 0-truncated category theory, where the type of morphisms is required to have unique identity proofs, asking for this other proof does not result in any coherence issues.

Dualizing the Terminal Category

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

Which Side Does the Identity Go On?

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

Dodging Judgmental η on Records

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

In particular, we wanted to show that any functor out of the terminal category is the opposite of some other functor; namely, any $F : 1 \to \mathcal{C}$ should be equal to $(F^{\mathrm{op}})^{\mathrm{op}} : 1 \to (\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$. However, without the judgmental η rule for records, a local variable $\mathcal{C}$ cannot be judgmentally equal to $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$, which reduces to an application of the constructor for a category, unless the η rule is built into the proof assistant. To get around the problem, we made two variants of dual functors: given $F : \mathcal{C} \to \mathcal{D}$, we have $F^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, and given $F : C^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, we have $F^{\mathrm{op}'} : \mathcal{C} \to \mathcal{D}$. There are two other flavors of dual functors, corresponding to the other two pairings of $^{\mathrm{op}}$ with domain and codomain, but we have been glad to avoid defining them so far. As it was, we ended up having four variants of dual natural transformation, and are very glad that we did not need sixteen. When Coq 8.5 was released, we no longer needed to pull this trick, as we could simply enable the η rule for records judgmentally.

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 η rules. The η rule for records is explained above. The η rule for equality says that the identity function is judgmentally equal to the function $f : \forall x\, y, x = y \to x = y$ defined by pattern matching on the first proof of equality; this rule is necessary to have any hope that applying symmetry twice is judgmentally the identity transformation.

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.[2]

3.5 A Sampling of Abstraction Barriers

[**TODO:** maybe this section should come first?]

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

[2] See `https://coq.inria.fr/bugs/show_bug.cgi?id=3179` and `https://coq.inria.fr/bugs/show_bug.cgi?id=3119`.

Therefore, we will describe a sampling of abstraction barriers and the design choices that went into them, drawn from real examples [**QUESTION FOR ADAM:** how much should I talk about the category theory library itself in my thesis?], 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[3], we have found that simple tactics like `apply f_equal` take around 1 second to execute, which makes interactive theorem proving very frustrating.[4] Even more frustrating is the fact that the largest contribution to this size is often arguments to irrelevant functions, i.e., functions that are provably equal to all other functions of the same type. (These are proofs related to algebraic laws like associativity, carried inside many constructions.)

Opacification helps by preventing the type checker from unfolding some definitions, but it is not enough: the type checker still has to deal with all of the large arguments

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

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

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 [4, 28].

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![5]

3.5.4 Nested Σ Types

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

The first is that while a theorem statement may abstract over all possible Σ types, it may not abstract over all record types, which somehow have a less first-class status.

[5] See `https://github.com/HoTT/HoTT/commit/eb0099005171` for the exact change.

Such a limitation is inconvenient and leads to code duplication.

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

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

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

If we want to project out the final field, we must write `projT2 (projT2 x)` which, when implicit arguments are included, expands to [**TODO:** figure out why spacing is strange above this code example]

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

This term grows quadratically in the number of projections because the type of the n^{th} field is repeated approximately $2n$ times. This is even more of a problem when we need to `destruct x` to prove something about the projections, as we need to `destruct` it as many times as 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 **??** 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 hashconsing the projections; see #12270.

[**TODO:** remove use of "you"] If your proof assistant does not support records out-of-the-box, or you want to avoid using them for whatever reason[6], 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.
```

[6]Note that the UniMath library [**TODO:** cite unimath] does this. [**TODO:** explain reasoning]

```
Definition _T2_proj2 (x : _T2) : _T1 := projT2 x.
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.[7] 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 η-conversion for primitive records, whence we can prove `T_eta` with the proof term `@eq_refl T x`. See **??** for performance details.

[**TODO:** make this timing graph]

[**TODO:** Find more examples to talk about here?]

[7] 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 η proof, we have two different types for the η-principle. One of them uses the globally-defined projections out of `T`, while the other uses the projections of `x` defined in the local context. We need to convert between these two types in linear time. Converting between two differently defined projections takes time linear in the number of under-the-hood projections, i.e., linear in the number of fields. Doing this once for each projection thus takes quadratic time. Using a reflective representation of nested Σ types, and thus being able to prove the η principle once and for all in constant time, would not help here, because it takes quadratic time to convert between the type of the η principle in reflective-land and the type that we want. One thing that might help would be to have a version of conversion checking that was both memoized and could perform in-place reduction; see [**TODO:** cite https://github.com/coq/coq/issues/12269].

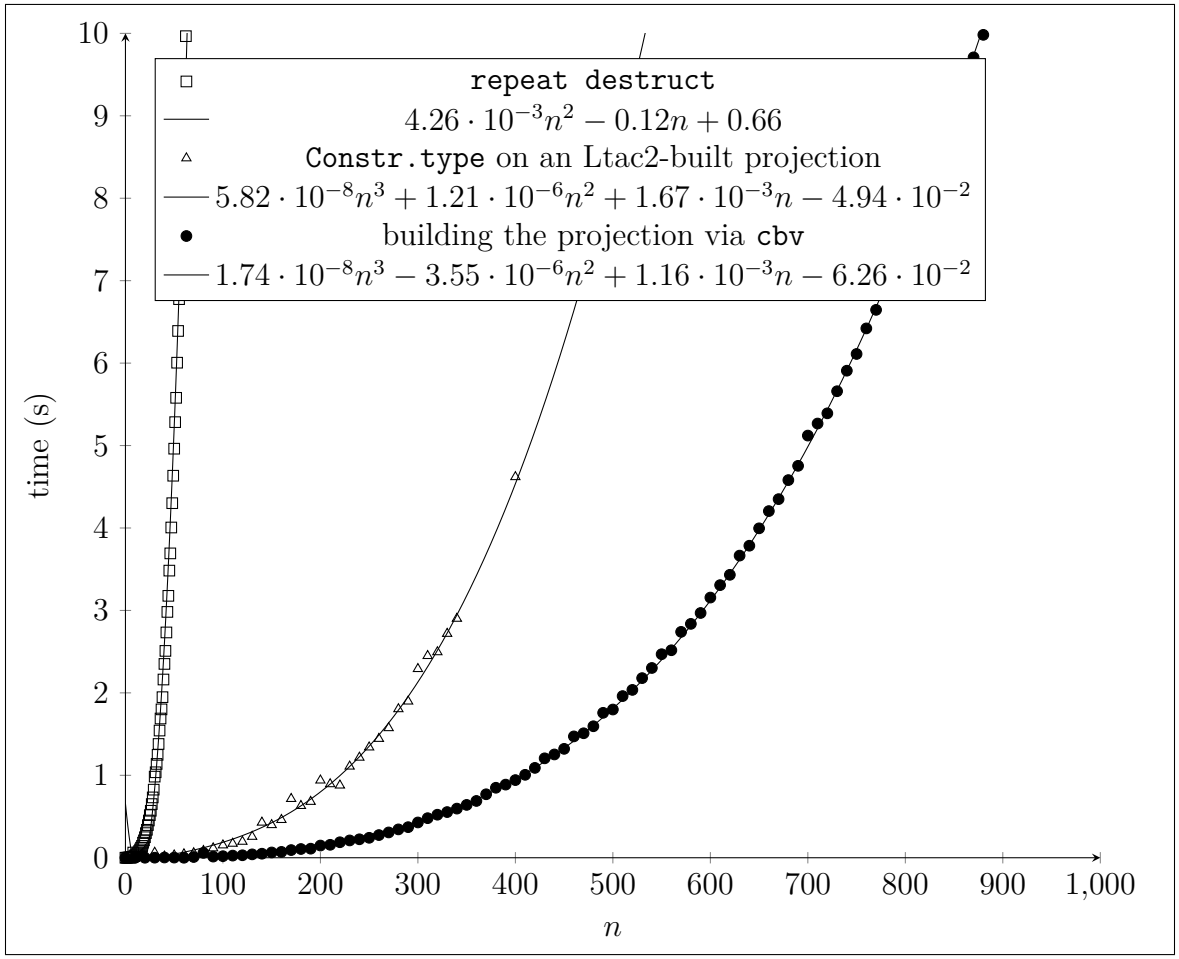


Figure 3-1: There are two ways we look at the performance of building a term like `projT1 (projT2 ... (projT2 x))` with n `projT2`s: we can define a recursive function that computes this term and then use **`cbv`** to reduce away the recursion, and time how long this takes; or we can build the term using 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 Σ type with n fields.

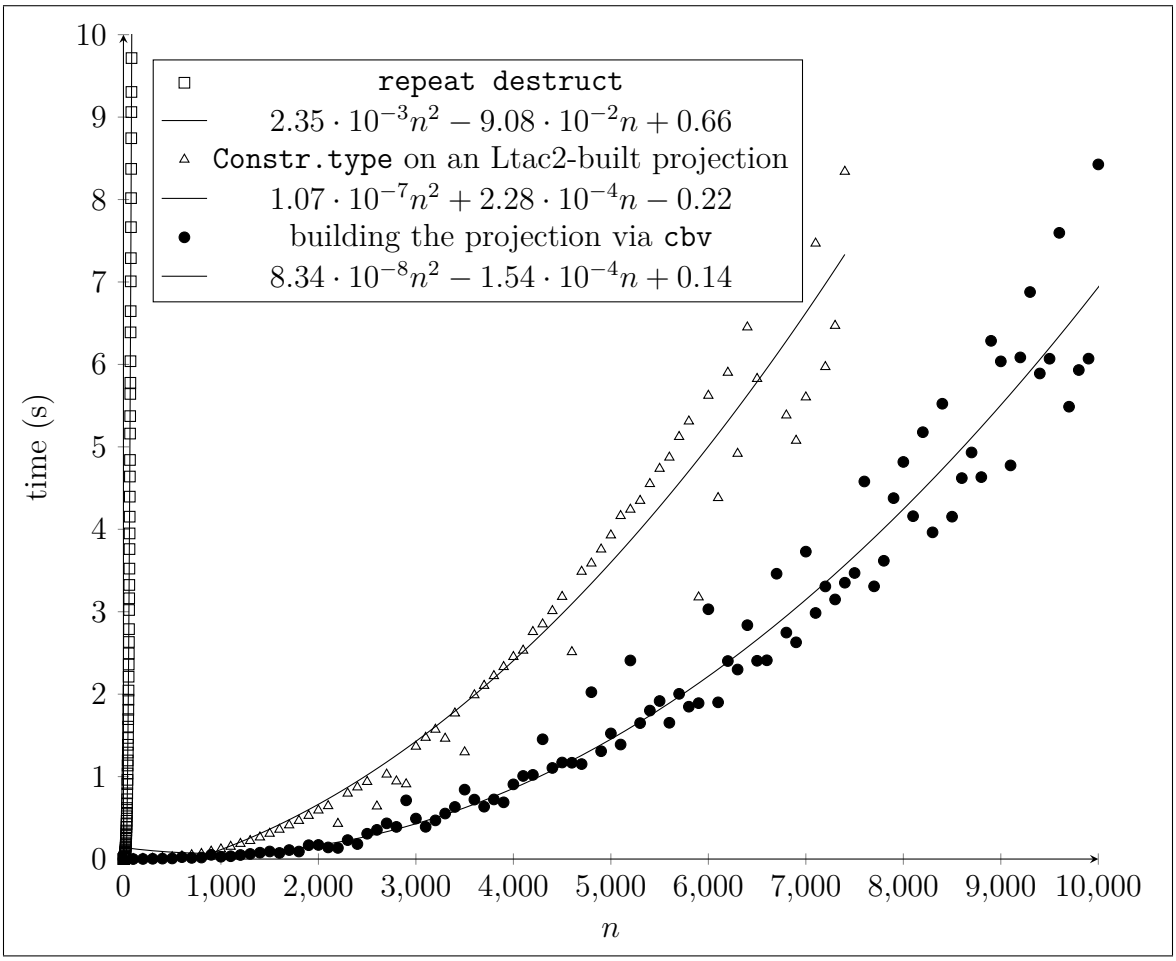


Figure 3-2: The same graph as Figure 3-1, but with primitive projections turned on. Note that the x-axis is $10\times$ larger on this plot.

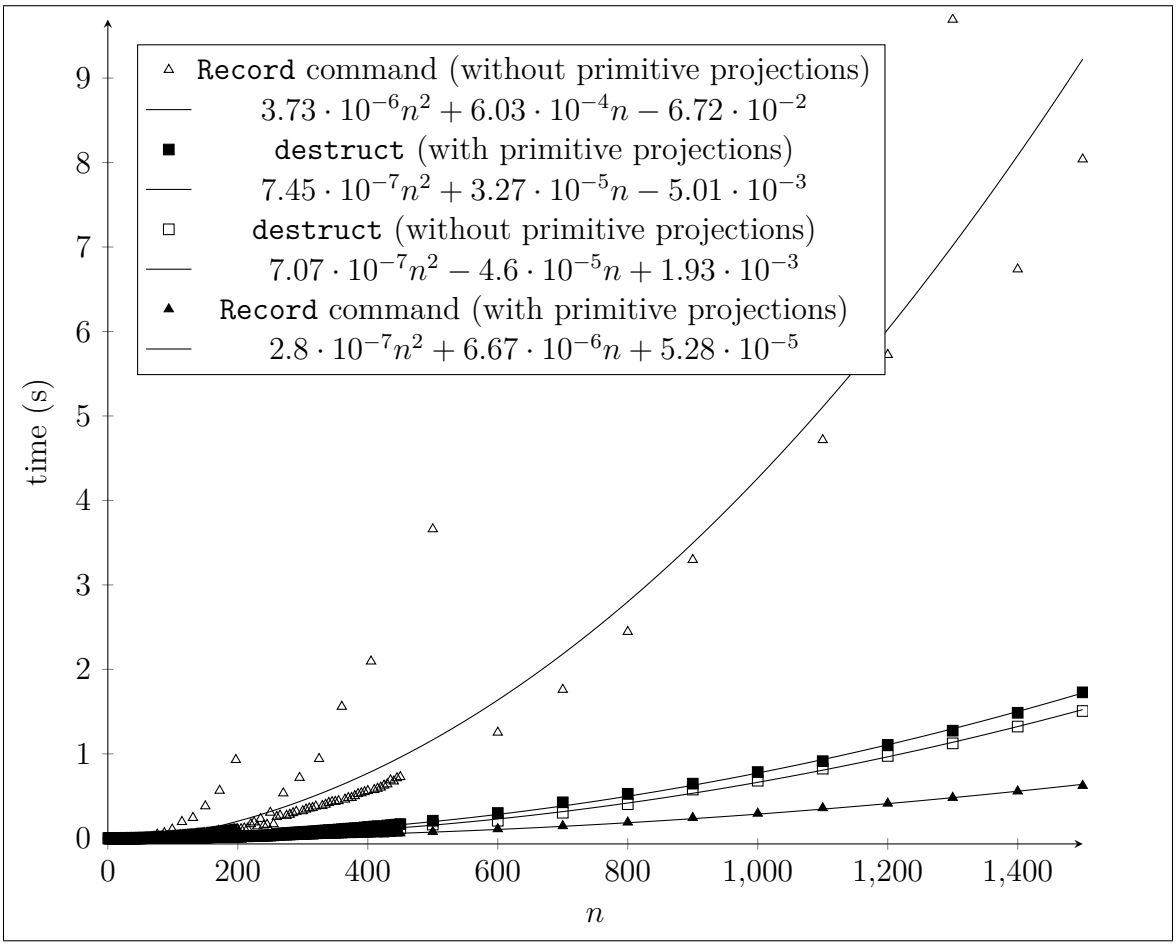


Figure 3-3: Timing of running a **`Record`** command to define a record with n fields, and the time to **`destruct`** such a record. Note that building the goal involving projecting out the last field takes less than 0.001s for all numbers of fields that we tested. (Presumably for large enough numbers 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.

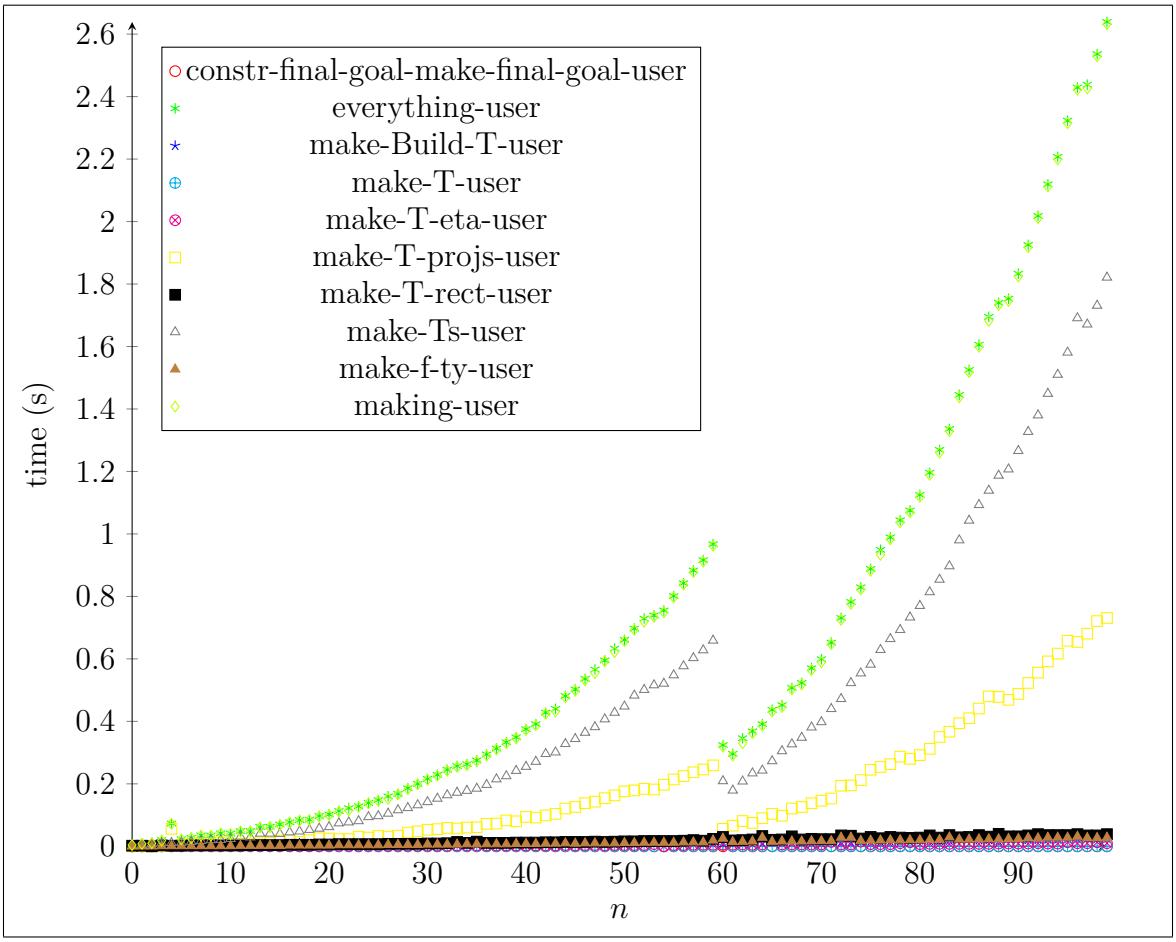


Figure 3-4: timing-performance-experiments/make-nested-prim-prod-abstraction.txt

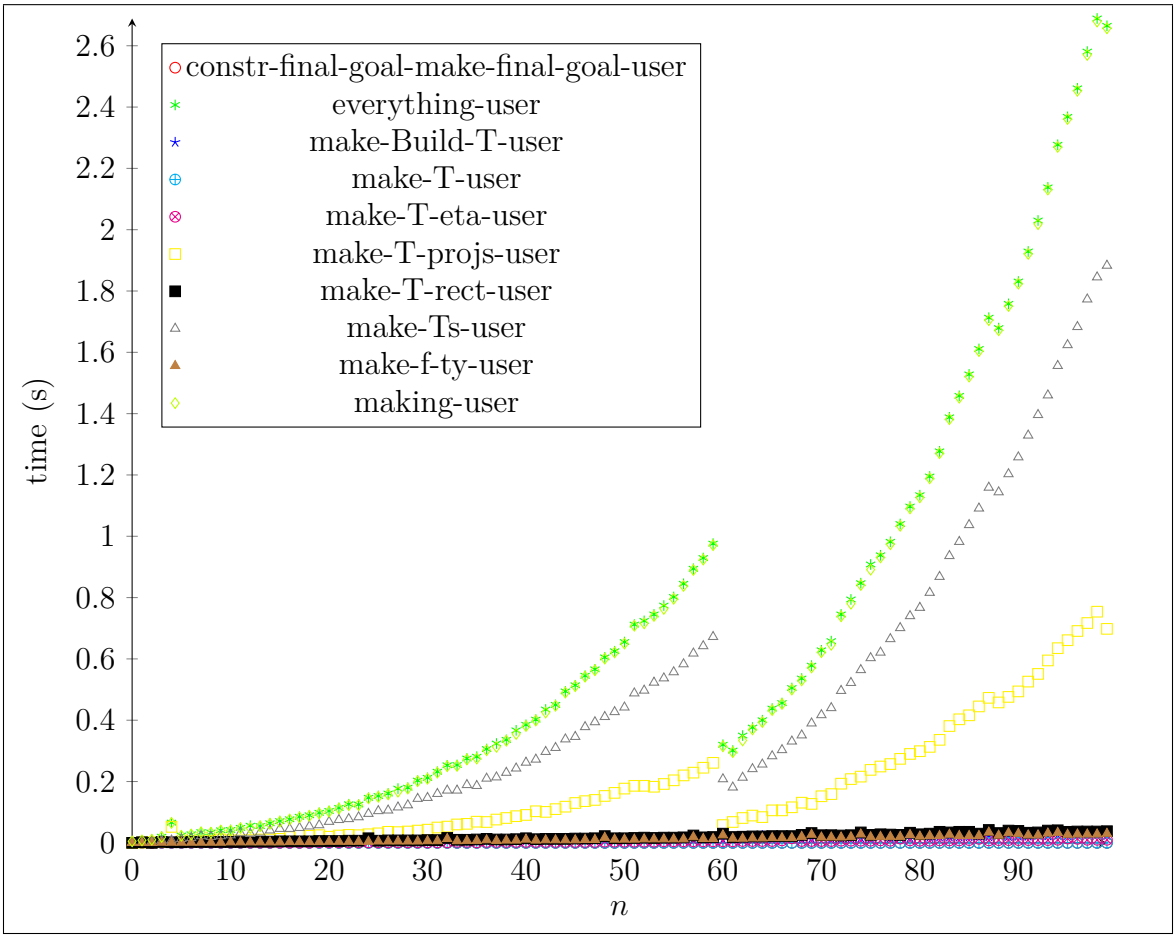


Figure 3-5: timing-performance-experiments/make-nested-prim-sig-abstraction.txt

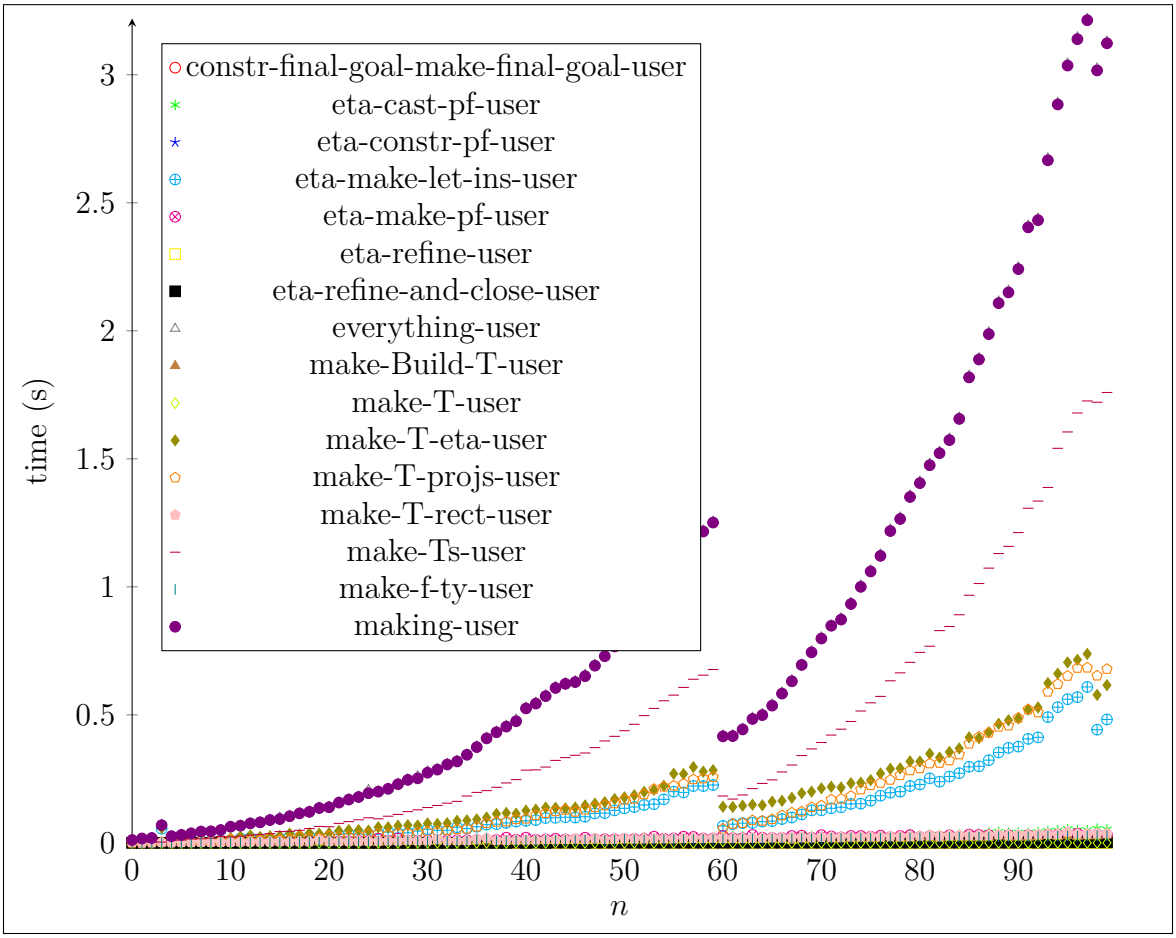


Figure 3-6: timing-performance-experiments/make-nested-prod-abstraction.txt

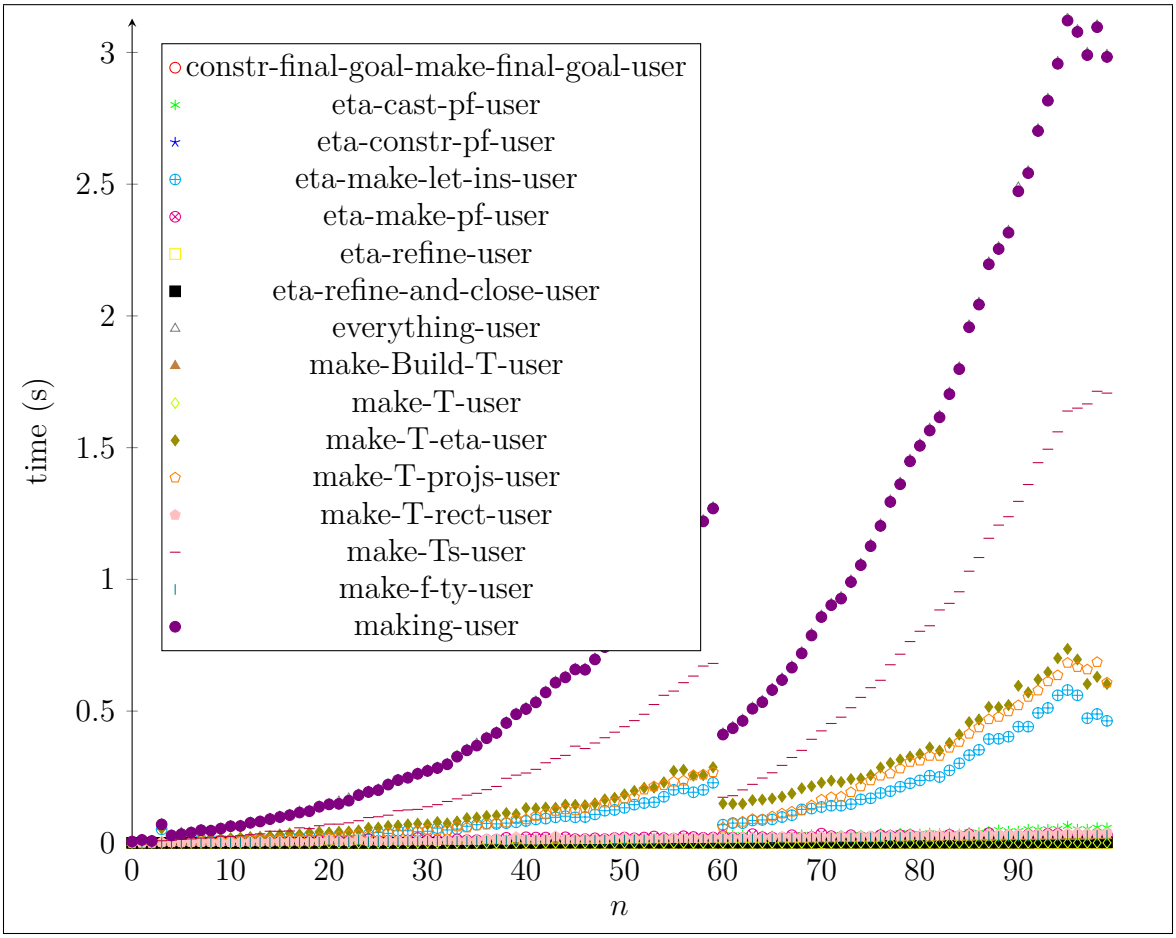


Figure 3-7: timing-performance-experiments/make-nested-sig-abstraction.txt

Part III

Program Transformation and Rewriting

Chapter 4

Reflective Program Transformation

4.1 Introduction

[**TODO:** talk about de Bruijn criterion again, how often building proof certificates is expensive]

[**TODO:** reformat this for flow, right now it's from reification-by-parametricity] [**TODO:** reformat code to match rest of paper] Proof by reflection [7] 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).

[**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 7.2, 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 7.3, which explains our alternative technique. We benchmark our approach against 18 competitors in Section 7.4.

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_0 : is_even 0
| even_SS : forall x, is_even x -> is_even (S (S x)).
Goal is_even (let x := 100 * 100 * 100 * 100 in
              let y := x * x * x * x in
              y * y * y * y).
```

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

Even on smaller terms, though, proof size can quickly become an issue. If we give a 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 7-1). We can use its correctness theorem to prove goals much more quickly:

```
Theorem soundness : forall n, check_is_even n = true -> is_even n.
Goal is_even 2000.
  Time repeat (apply even_SS || apply even_0). (* 1.8 s *)
  Undo.
  Time apply soundness; vm_compute; reflexivity. (* 0.004 s *)
```

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

However, even this procedure takes a bit over three minutes to prove `is_even (10 * 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 7.3, we get the inductive type of Figure 7-2.

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) [30], parametric higher-order abstract syntax (PHOAS) [9], and de Bruijn indices [8]. 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 [9]. [**TODO:** give a full treatment of well-formedness] 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.
```

4.2 Reflective Program Tranformation

[**TODO:** prep the reader for the upcoming chapter on the rewriter]

[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`.

4.3 Reflective Proofs of Well-Formedness

[**TODO:** figure out where this section belongs] [**TODO:** talk about tricks for avoiding quadratic overhead in well-formedness proofs: minimizing the certificate size reflectively, and going via de Bruijn]

4.4 related work (incl. RTac)

[**TODO:** talk about related work] [**TODO:** figure out where this section belongs]

[**TODO:** should I talk about parsers at all?]

Chapter 5

A Framework for Building Verified Partial Evaluators

Abstract

Partial evaluation is a classic technique for generating lean, customized code from libraries that start with more bells and whistles. It is also an attractive approach to creation of *formally verified* systems, where theorems can be proved about libraries, yielding correctness of all specializations "for free." However, it can be challenging to make library specialization both performant and trustworthy. We present a new approach, prototyped in the Coq proof assistant, which supports specialization at the speed of native-code execution, without adding to the trusted code base. Our extensible engine, which combines the traditional concepts of tailored term reduction and automatic rewriting from hint databases, is also of interest to replace these ingredients in proof assistants' proof checkers and tactic engines, at the same time as it supports extraction to standalone compilers from library parameters to specialized code.

5.1 Introduction

Mechanized proof is gaining in importance for development of critical software infrastructure. Oft-cited examples include the CompCert verified C compiler [22] and the seL4 verified operating-system microkernel [21]. Here we have very flexible systems that are ready to adapt to varieties of workloads, be they C source programs for CompCert or application binaries for seL4. For a verified operating system, such adaptation takes place at *runtime*, when we launch the application. However, some important bits of software infrastructure commonly do adaptation at *compile time*, such that the fully general infrastructure software is not even installed in a deployed system.

Of course, compilers are a natural example of that pattern, as we would not expect CompCert itself to be installed on an embedded system whose application code was compiled with it. The problem is that writing a compiler is rather labor-intensive, with its crafting of syntax-tree types for source, target, and intermediate languages, its fine-tuning of code for transformation passes that manipulate syntax trees explicitly, and so on. An appealing alternative is *partial evaluation* [20], which relies on reusable compiler facilities to specialize library code to parameters, with no need to write that library code in terms of syntax-tree manipulations. Cutting-edge tools in this tradition even make it possible to use high-level functional languages to generate performance-competitive low-level code, as in Scala's Lightweight Modular Staging [31].

It is natural to try to port this approach to construction of systems with mechanized proofs. On one hand, the typed functional languages in popular proof assistants' logics make excellent hosts for flexible libraries, which can often be specialized through means as simple as partial application of curried functions. Term-reduction systems built into the proof assistants can then generate the lean residual programs. On the other hand, it is surprisingly difficult to realize the last sentence with good performance. The challenge is that we are not just implementing algorithms; we also want a proof to be checked by a small proof checker, and there is tension in designing such a checker, as fancier reduction strategies grow the trusted code base. It would seem like an abandonment of the spirit of proof assistants to bake in a reduction strategy per library, yet effective partial evaluation tends to be rather fine-tuned in this way. Performance tuning matters when generated code is thousands of lines long.

In this paper, we present an approach to verified partial evaluation in proof assistants, which requires no changes to proof checkers. To make the relevance concrete, we use the example of Fiat Cryptography [12], a Coq library that generates code for big-integer modular arithmetic at the heart of elliptic-curve cryptography algorithms. This domain-specific compiler has been adopted, for instance, in the Chrome Web browser, such that about half of all HTTPS connections from browsers are now initiated using code generated (with proof) by Fiat Cryptography. However, Fiat Cryptography was only used successfully to build C code for the two most widely used curves (P-256 and Curve25519). Their method of partial evaluation timed out trying to compile code for the third most widely used curve (P-384). Additionally, to achieve acceptable reduction performance, the library code had to be written manually in continuation-passing style. We will demonstrate a new Coq library that corrects both weaknesses, while maintaining the generality afforded by allowing rewrite rules to be mixed with partial evaluation.

5.1.1 A Motivating Example

We are interested in partial-evaluation examples that mix higher-order functions, inductive datatypes, and arithmetic simplification. For instance, consider the following Coq code.

```
Definition prefixSums (ls:list nat) : list nat :=
  let ls' := combine ls (seq 0 (length ls)) in
  let ls'' := map (λ p, fst p * snd p) ls' in
  let '(_, ls''') := fold_left (λ acc_ls''' n,
    let '(acc, ls''') := acc_ls''' in
    let acc' := acc + n in
    (acc', acc' :: ls''')) ls'' (0, []) in
  ls'''.
```

This function first computes list `ls'` that pairs each element of input list `ls` with its position, so, for instance, list $[a; b; c]$ becomes $[(a, 0); (b, 1); (c, 2)]$. Then we map over the list of pairs, multiplying the components at each position. Finally, we traverse that list, building up a list of all prefix sums.

We would like to specialize this function to particular list lengths. That is, we know in advance how many list elements we will pass in, but we do not know the values of those elements. For a given length, we can construct a schematic list with one free variable per element. For example, to specialize to length four, we can apply the function to list `[a; b; c; d]`, and we expect this output:

```
let acc := b + c * 2 in
let acc' := acc + d * 3 in
  [acc'; acc; b; 0]
```

Notice how subterm sharing via `let`s is important. As list length grows, we avoid quadratic blowup in term size through sharing. Also notice how we simplified the first two multiplications with $a \cdot 0 = 0$ and $b \cdot 1 = b$ (each of which requires explicit proof in Coq), using other arithmetic identities to avoid introducing new variables for the first two prefix sums of `ls''`, as they are themselves constants or variables, after simplification.

To set up our partial evaluator, we prove the algebraic laws that it should use for simplification, starting with basic arithmetic identities.

```
Lemma zero_plus  : forall n, 0 + n = n.
Lemma plus_zero  : forall n, n + 0 = n.
Lemma times_zero : forall n, n * 0 = 0.
Lemma times_one  : forall n, n * 1 = n.
```

Next, we prove a law for each list-related function, connecting it to the primitive-recursion combinator for some inductive type (natural numbers or lists, as appropriate). We use a special apostrophe marker to indicate a quantified variable that may only match with *compile-time constants*. We also use a further marker `ident.eagerly`

to ask the reducer to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree.

```
Lemma eval_map A B (f : A -> B) l
: map f l = ident.eagerly list_rect _ _ []
    (λ x _ l', f x :: l') l.
Lemma eval_fold_left A B (f : A -> B -> A) l a
: fold_left f l a = ident.eagerly list_rect
    _ _ (λ a, a)
    (λ x _ r a, r (f a x)) l a.
Lemma eval_combine A B (la : list A) (lb : list B)
: combine la lb = list_rect _ (λ _, [])
    (λ x _ r lb, list_case (λ _, _) []
      (λ y ys, (x, y) :: r ys) lb) la lb.
Lemma eval_length A (ls : list A)
: length ls = list_rect _ 0 (λ _ _ n, S n) ls.
```

With all the lemmas available, we can package them up into a rewriter, which triggers generation of a specialized rewrite procedure and its soundness proof. Our Coq plugin introduces a new command `Make` for building rewriters

```
Make rewriter := Rewriter For (zero_plus, plus_zero,
  times_zero, times_one, eval_map, eval_fold_left,
  do_again eval_length, do_again eval_combine,
  eval_rect nat, eval_rect list, eval_rect prod)
  (with delta) (with extra idents (seq)).
```

Most inputs to `Rewriter For` list quantified equalities to use for left-to-right rewriting. However, we also use options `do_again`, to request that some rules trigger an extra bottom-up pass after being used for rewriting; `eval_rect`, to queue up eager evaluation of a call to a primitive-recursion combinator on a known recursive argument; **`with delta`**, to request evaluation of all monomorphic operations on concrete inputs; and **`with extra idents`**, to inform the engine of further permitted identifiers that do not appear directly in any of the rewrite rules.

Our plugin also provides new tactics like `Rewrite_rhs_for`, which applies a rewriter to the righthand side of an equality goal. That last tactic is just what we need to synthesize a specialized `prefixSums` for list length four, along with a proof of its equivalence to the original function.

```
Definition prefixSums4 :
 {f : nat -> nat -> nat -> nat -> list nat
 | forall a b c d, f a b c d = prefixSums [a;b;c;d]} :=
  ltac:(eexists; Rewrite_rhs_for rewriter; reflexivity).
```

5.1.2 Concerns of Trusted-Code-Base Size

Crafting a reduction strategy is challenging enough in a standalone tool. A large part of the difficulty in a proof assistant is reducing in a way that leaves a proof trail that can be checked efficiently by a small kernel. Most proof assistants present user-friendly surface tactic languages that generate proof traces in terms of more elementary tactic steps. The trusted proof checker only needs to know about the elementary steps, and there is pressure to be sure that these steps are indeed elementary, not requiring excessive amounts of kernel code. However, hardcoding a new reduction strategy in the kernel can bring dramatic performance improvements. Generating thousands of lines of code with partial evaluation would be intractable if we were outputting sequences of primitive rewrite steps justifying every little term manipulation, so we must take advantage of the time-honored feature of type-theoretic proof assistants that reductions included in the definitional equality need not be requested explicitly.

Which kernel-level reductions *does* Coq support today? Currently, the trusted code base knows about four different kinds of reduction: left-to-right conversion, right-to-left conversion, a virtual machine (VM) written in C based on the OCaml compiler, and a compiler to native code. Furthermore, the first two are parameterized on an arbitrary user-specified ordering of which constants to unfold when, in addition to internal heuristics about what to do when the user has not specified an unfolding order for given constants. Recently, native support for 63-bit integers has been added to the VM and native machines. A recent pull request proposes adding support for native IEEE 754-2008 binary64 floats [27], and support for native arrays is in the works [11].

To summarize, there has been quite a lot of "complexity creep" in the Coq trusted base, to support efficient reduction, and yet realistic partial evaluation has *still* been rather challenging. Even the additional three reduction mechanisms outside Coq's kernel (`cbn`, `simpl`, `cbv`) are not at first glance sufficient for verified partial evaluation.

5.1.3 Our Solution

Aehlig, Haftmann, and Nipkow [1] presented a very relevant solution to a related problem, using *normalization by evaluation (NbE)* [5] to bootstrap reduction of open terms on top of full reduction, as built into a proof assistant. However, it was simultaneously true that they expanded the proof-assistant trusted code base in ways specific to their technique, and that they did not report any experiments actually using the tool for partial evaluation (just traditional full reduction), potentially hiding performance-scaling challenges or other practical issues. We have adapted their approach in a new Coq library embodying **the first partial-evaluation approach to satisfy the following criteria**.

- It integrates with a general-purpose, foundational proof assistant, **without growing the trusted base**.

- For a wide variety of initial functional programs, it provides **fast** partial evaluation with reasonable memory use.
- It allows reduction that **mixes** *rules of the definitional equality* with *equalities proven explicitly as theorems*.
- It **preserves sharing** of common subterms.
- It also allows **extraction of standalone partial evaluators**.

Our contributions include answers to a number of challenges that arise in scaling NbE-based partial evaluation in a proof assistant. First, we rework the approach of Aehlig, Haftmann, and Nipkow [1] to function *without extending a proof assistant's trusted code base*, which, among other challenges, requires us to prove termination of reduction and encode pattern matching explicitly (leading us to adopt the performance-tuned approach of Maranget [26]).

Second, using partial evaluation to generate residual terms thousands of lines long raises *new scaling challenges*:

- Output terms may contain so *many nested variable binders* that we expect it to be performance-prohibitive to perform bookkeeping operations on first-order-encoded terms (e.g., with de Bruijn indices, as is done in $\mathcal{R}_{tac}$ by Malecha and Bengtson [24]). For instance, while the reported performance experiments of Aehlig, Haftmann, and Nipkow [1] generate only closed terms with no binders, Fiat Cryptography may generate a single routine (e.g., multiplication for curve P-384) with nearly a thousand nested binders.
- Naive representation of terms without proper *sharing of common subterms* can lead to fatal term-size blow-up. Fiat Cryptography's arithmetic routines rely on significant sharing of this kind.
- Unconditional rewrite rules are in general insufficient, and we need *rules with side conditions*. For instance, in Fiat Cryptography, some rules for simplifying modular arithmetic depend on proofs that operations in subterms do not overflow.
- However, it is also not reasonable to expect a general engine to discharge all side conditions on the spot. We need integration with *abstract interpretation* that can analyze whole programs to support reduction.

Briefly, our respective solutions to these problems are the *parametric higher-order abstract syntax (PHOAS)* [9] term encoding, a *let-lifting* transformation threaded throughout reduction, extension of rewrite rules with executable Boolean side conditions, and a design pattern that uses decorator function calls to include analysis results in a program.

Finally, we carry out the *first large-scale performance-scaling evaluation* of partial evaluation in a proof assistant, covering all elliptic curves from the published Fiat Cryptography experiments, along with microbenchmarks.

This paper proceeds through explanations of the trust stories behind our approach and earlier ones (Section 5.2), the core structure of our engine (Section 5.3), the additional scaling challenges we faced (Section 5.4), performance experiments (Section 5.5), and related work (Section 5.6) and conclusions. Our implementation is included as an anonymous supplement.

5.2 Trust, Reduction, and Rewriting

Since much of the narrative behind our design process depends on tradeoffs between performance and trustworthiness, we start by reviewing the general situation in proof assistants.

Across a variety of proof assistants, simplification of functional programs is a workhorse operation. Proof assistants like Coq that are based on type theory typically build in *definitional equality* relations, identifying terms up to reductions like β-reduction and unfolding of named identifiers. What looks like a single "obvious" step in an on-paper equational proof may require many of these reductions, so it is handy to have built-in support for checking a claimed reduction. 5-1a diagrams how such steps work in a system like Coq, where the system implementation is divided between a trusted *kernel*, for checking *proof terms* in a minimal language, and additional untrusted support, like a *tactic* engine evaluating a language of higher-level proof steps, in the process generating proof terms out of simpler building blocks. It is standard to include a primitive proof step that validates any reduction compatible with the definitional equality, as the latter is decidable. The figure shows a tactic that simplifies a goal using that facility.

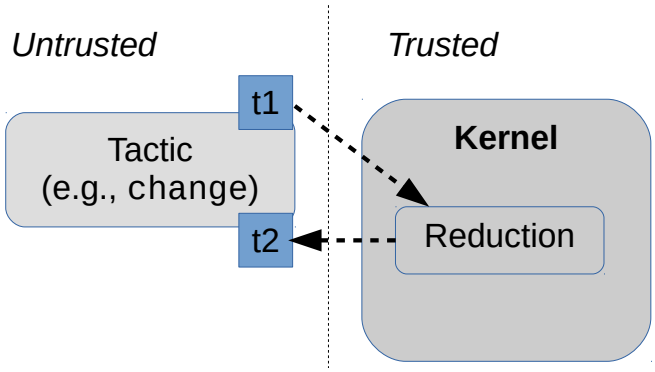
In proof goals containing free variables, executing subterms can get stuck before reaching normal forms. However, we can often achieve further simplification by using equational rules that we prove explicitly, rather than just relying on the rules built into the definitional equality and its decidable equivalence checker. Coq's `autorewrite` tactic, as diagrammed in 5-1b, is a good example: it takes in a database of quantified equalities and applies them repeatedly to rewrite in a goal. It is important that Coq's kernel does not trust the `autorewrite` tactic. Instead, the tactic must output a proof term that, in some sense, is the moral equivalent of a line-by-line equational proof. It can be challenging to keep these proof terms small enough, as naive rewrite-by-rewrite versions repeatedly copy large parts of proof goals, justifying a rewrite like $C[e_1] = C[e_2]$ for some context C given a proof of $e_1 = e_2$, with the full value of C replicated in the proof term for that single rewrite. Overcoming these challenges while retaining decidability of proof checking is tricky, since we may use `autorewrite` with rule sets that do not always lead to terminating reduction. Coq includes more

experimental alternatives like `rewrite_strat`, which use bottom-up construction of multi-rewrite proofs, with sharing of common contexts. Still, as Section 5.5 will show, these methods that generate substantial proof terms are at significant performance disadvantages.

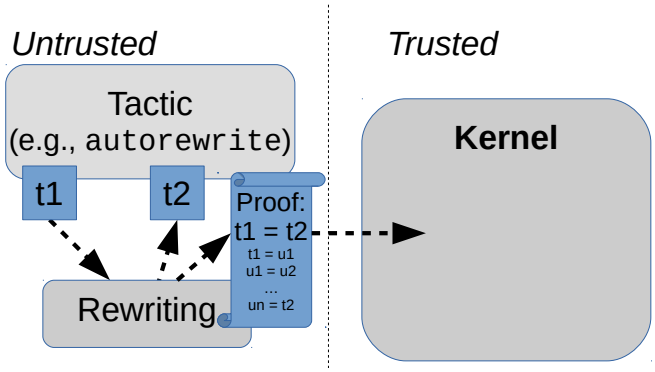
Now we summarize how Aehlig, Haftmann, and Nipkow [1] provide flexible and fast interleaving of standard λ-calculus reduction and use of proved equalities (the next section will go into more detail). 5-1c demonstrates a workflow based on *a deep embedding of a core ML-like language*. That is, within the logic of the proof assistant (Isabelle/HOL, in their case), a type of syntax trees for ML programs is defined, with an associated operational semantics. The basic strategy is, for a particular set of rewrite rules and a particular term to simplify, to *generate a (deeply embedded) ML program that, if it terminates, produces a syntax tree for the simplified term*. Their tactic uses *reification* to created ML versions of rule sets and terms. They also wrote a reduction function in ML and proved it sound once and for all, against the ML operational semantics. Combining that proof with proofs generated by reification, we conclude that an application of the reduction function to the reified rules and term is indeed an ML term that generates correct answers. The tactic then "throws the ML term over the wall," using a general code-generation framework for Isabelle/HOL [18]. Trusted code compiles the ML code into the concrete syntax of a mainstream ML language, Standard ML in their case, and compiles it with an off-the-shelf compiler. The output of that compiled program is then passed back over to the tactic, in terms of an axiomatic assertion that the ML semantics really yields that answer.

As Aehlig, Haftmann, and Nipkow [1] argue, their use of external compilation and evaluation of ML code adds no real complexity on top of that required by the proof assistant – after all, the proof assistant itself must be compiled and executed somehow. However, the perceived increase of trusted code base is not spurious: it is one thing to trust that the toolchain and execution environment used by the proof assistant and the partial evaluator are well-behaved, and another to rely on two descriptions of ML (one deeply embedded in the proof assistant and another implied by the compiler) to agree on every detail of the semantics. Furthermore, there still is new trusted code to translate from the deeply embedded ML subset into the concrete syntax of the full-scale ML language. The vast majority of proof-assistant developments today rely on no such embeddings with associated mechanized semantics, so need we really add one to a proof-checking kernel to support efficient partial evaluation?

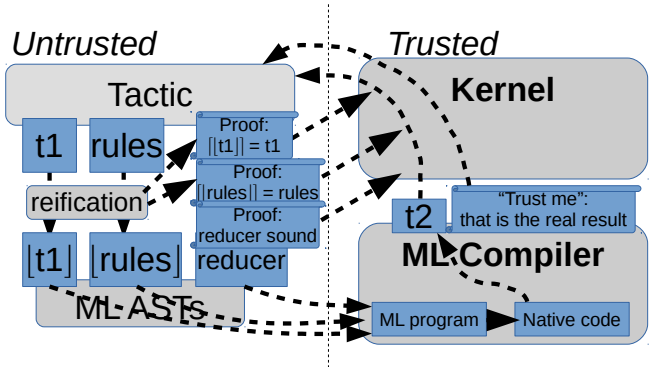
Our answer, diagrammed in 5-1d, shows a different way. We still reify terms and rules into a deeply embedded language. However, *the reduction engine is implemented directly in the logic*, rather than as a deeply embedded syntax tree of an ML program. As a result, the kernel's own reduction engine is prepared to execute our reduction engine for us – using an operation that would be included in a type-theoretic proof assistant in any case, with no special support for a language deep embedding. We also stage the process for performance reasons. First, the `Make` command creates a rewriter out of a list of rewrite rules, by specializing a generic partial-evaluation



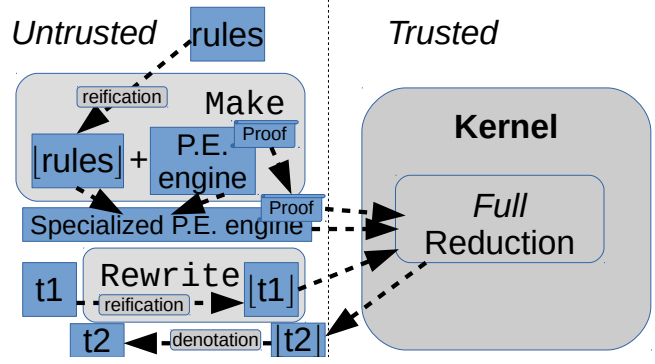
(a) Reduction via the definitional equality



(b) Rewriting via proved rules



(c) Approach of Aehlig, Haftmann, and Nipkow [1]



(d) Our approach

Figure 5-1: Different approaches to reduction and rewriting

engine, which has a generic proof that applies to any set of proved rewrite rules. We perform partial evaluation on the specialized partial evaluator, using Coq's normal reduction mechanisms, under the theory that we can afford to pay performance costs at this stage because we only need to create new rewriters relatively infrequently. Then individual rewritings involve reifying terms, asking the kernel to execute the specialized evaluator on them, and simplifying an application of an interpretation function to the result (this last step must be done using Coq's normal reduction, and it is the bottleneck for outputs with enormous numbers of nested binders as discussed in section 5.5.1).

5.2.1 Our Approach in Nine Steps

Here is a bit more detail on the steps that go into applying our Coq plugin, many of which we expand on in the following sections. In order to build a precomputed rewriter with the `Make` command, the following actions are performed:

1. The given lemma statements are scraped for which named functions and types the rewriter package will support.
2. Inductive types enumerating all available primitive types and functions are emitted.
3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. Definitions include operations like Boolean equality on type codes and lemmas like "all representable primitive types have decidable equality."
4. The statements of rewrite rules are reified, and we prove soundness and syntactic-well-formedness lemmas about each of them. Each instance of the former involves wrapping the user-provided proof with the right adapter to apply to the reified version.
5. The definitions needed to perform reification and rewriting and the lemmas needed to prove correctness are assembled into a single package that can be passed by name to the rewriting tactic.

When we want to rewrite with a rewriter package in a goal, the following steps are performed:

1. We rearrange the goal into a single logical formula: all free-variable quantification in the proof context is replaced by changing the equality goal into an equality between two functions (taking the free variables as inputs).
2. We reify the side of the goal we want to simplify, using the inductive codes in the specified package. That side of the goal is then replaced with a call to a denotation function on the reified version.

3. We use a theorem stating that rewriting preserves denotations of well-formed terms to replace the denotation subterm with the denotation of the rewriter applied to the same reified term. We use Coq's built-in full reduction (`vm_compute`) to reduce the application of the rewriter to the reified term.

4. Finally, we run `cbv` (a standard call-by-value reducer) to simplify away the invocation of the denotation function on the concrete syntax tree from rewriting.

5.3 The Structure of a Rewriter

We now simultaneously review the approach of Aehlig, Haftmann, and Nipkow [1] and introduce some notable differences in our own approach, noting similarities to the reflective rewriter of Malecha and Bengtson [24] where applicable.

First, let us describe the language of terms we support rewriting in. Note that, while we support rewriting in full-scale Coq proofs, where the metalanguage is dependently typed, the object language of our rewriter is nearly simply typed, with limited support for calling polymorphic functions. However, we still support identifiers whose definitions use dependent types, since our reducer does not need to look into definitions.

$$\begin{aligned} e ::= {} & \texttt{App}\ e_1\ e_2 \mid \texttt{Let}\ v := e_1\ \texttt{In}\ e_2 \\ & \mid \texttt{Abs}\ (\lambda v.\, e) \mid \texttt{Var}\ v \mid \texttt{Ident}\ i \end{aligned}$$

The `Ident` case is for identifiers, which are described by an enumeration specific to a use of our library. For example, the identifiers might be codes for $+$, $\cdot$, and literal constants. We write $[\![e]\!]$ for a standard denotational semantics.

5.3.1 Pattern-Matching Compilation and Evaluation

Aehlig, Haftmann, and Nipkow [1] feed a specific set of user-provided rewrite rules to their engine by generating code for an ML function, which takes in deeply embedded term syntax (actually *doubly* deeply embedded, within the syntax of the deeply embedded ML!) and uses ML pattern matching to decide which rule to apply at the top level. Thus, they delegate efficient implementation of pattern matching to the underlying ML implementation. As we instead build our rewriter in Coq's logic, we have no such option to defer to ML. Indeed, Coq's logic only includes primitive pattern-matching constructs to match one constructor at a time.

We could follow a naive strategy of repeatedly matching each subterm against a pattern for every rewrite rule, as in the rewriter of Malecha and Bengtson [24], but in that case we do a lot of duplicate work when rewrite rules use overlapping function symbols. Instead, we adopted the approach of Maranget [26], who describes compilation of pattern matches in OCaml to decision trees that eliminate needless repeated work (for example, decomposing an expression into $x + y + z$ only once even if two

different rules match on that pattern). We have not yet implemented any of the optimizations described therein for finding *minimal* decision trees.

There are three steps to turn a set of rewrite rules into a functional program that takes in an expression and reduces according to the rules. The first step is pattern-matching compilation: we must compile the lefthand 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. Because the decision tree is merely a decomposition hint, we require no proofs about it to ensure soundness of our rewriter. The second step is decision-tree evaluation, during which we decompose the expression as per the decision tree, selecting which rewrite rules to attempt. The only correctness lemma needed for this stage is that any result it returns is equivalent to picking some rewrite rule and rewriting with it. The third and final step is to actually rewrite with the chosen rule. Here the correctness condition is that we must not change the semantics of the expression. Said another way, any rewrite-rule replacement expression must match the semantics of the rewrite-rule pattern.

While pattern matching begins with comparing one pattern against one expression, Maranget's approach works with intermediate goals that check multiple patterns against multiple expressions. A decision tree describes how to match a vector (or list) of patterns against a vector of expressions. It is built from these constructors:

- `TryLeaf k onfailure`: Try the k^{th} rewrite rule; if it fails, keep going with `onfailure`.
- `Failure`: Abort; nothing left to try.
- `Switch icases app_case default`: With the first element of the vector, match on its kind; if it is an identifier matching something in `icases`, 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`.

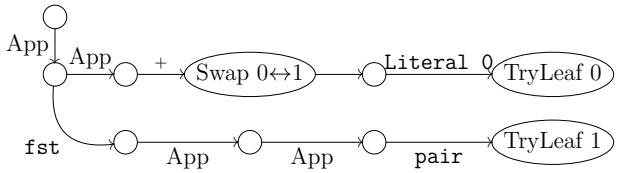
Consider the encoding of two simple example rewrite rules, where we follow Coq's $\mathcal{L}_{tac}$ language in prefacing pattern variables with question marks.

$$?n + 0 \to n$$
$$\texttt{fst}_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$$

We embed them in an AST type for patterns, which largely follows our ASTs for expressions.

```
0. App (App (Ident +) Wildcard) (Ident (Literal 0))
1. App (Ident fst) (App (App (Ident pair) Wildcard)
                    Wildcard)
```

The decision tree produced is



where every non-swap node implicitly has a "default" case arrow to `Failure`.

We implement, in Coq's logic, an evaluator for these trees against terms. Note that we use Coq's normal partial evaluation to turn our general decision-tree evaluator into a specialized matcher to get reasonable efficiency. Although this partial evaluation of our partial evaluator is subject to the same performance challenges we highlighted in the introduction, it only has to be done once for each set of rewrite rules, and we are targeting cases where the time of per-goal reduction dominates this time of meta-compilation.

For our running example of two rules, specializing gives us this match expression.

```
match e with
| App f y => match f with
  | Ident fst => match y with
    | App (App (Ident pair) x) y => x
    | _ => e end
  | App (Ident +) x => match y with
    | Ident (Literal 0) => x | _ => e end
  | _ => e end | _ => e end.
```

5.3.2 Adding Higher-Order Features

Fast rewriting at the top level of a term is the key ingredient for supporting customized algebraic simplification. However, not only do we want to rewrite throughout the structure of a term, but we also want to integrate with simplification of higher-order terms, in a way where we can prove to Coq that our syntax-simplification function always terminates. Normalization by evaluation (NbE) [5] is an elegant technique for adding the latter aspect, in a way where we avoid needing to implement our own λ-term reducer or prove it terminating.

To orient expectations: we would like to enable the following reduction

$$(\lambda f\ x\ y.\ f\ x\ y)\ (+)\ z\ 0 \rightsquigarrow z$$

using the rewrite rule

$$?n + 0 \to n$$

Aehlig, Haftmann, and Nipkow [1] also use NbE, and we begin by reviewing its most classic variant, for performing full β-reduction in a simply typed term in a guaranteed-terminating way. The simply typed λ-calculus syntax we use is:

$$t ::= t \to t \mid b \qquad\qquad e ::= \lambda v.\ e \mid e\ e \mid v \mid c$$

with v for variables, c for constants, and b for base types.

We can now define normalization by evaluation. First, we choose a "semantic" representation for each syntactic type, which serves as the result type of an intermediate interpreter.

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

Function types are handled as in a simple denotational semantics, while base types receive the perhaps-counterintuitive treatment that the result of "executing" one is a syntactic expression of the same type. We write $\texttt{expr}(b)$ for the metalanguage type of object-language syntax trees of type b, relying on a dependent type family `expr`.

Now the core of NbE, shown in Figure 5-2, is a pair of dual functions reify and reflect, for converting back and forth between syntax and semantics of the object language, defined by primitive recursion on type syntax. We split out analysis of term syntax in a separate function reduce, defined by primitive recursion on term syntax, when usually this functionality would be mixed in with reflect. The reason for this choice will become clear when we extend NbE to handle our full problem domain.

We write v for object-language variables and x for metalanguage (Coq) variables, and we overload λ notation using the metavariable kind to signal whether we are building a host λ or a λ syntax tree for the embedded language. The crucial first clause for reduce replaces object-language variable v with fresh metalanguage variable x, and then we are somehow tracking that all free variables in an argument to reduce must have been replaced with metalanguage variables by the time we reach them. We reveal in Subsection 5.4.1 the encoding decisions that make all the above legitimate, but first let us see how to integrate use of the rewriting operation from the previous section. To fuse NbE with rewriting, we only modify the constant case of `reduce`. First, we bind our specialized decision-tree engine under the name rewrite-head. Recall that

$$
\begin{aligned}
\text{reify}_t &: \text{NbE}_t(t) \to \text{expr}(t) \\
\text{reify}_{t_1 \to t_2}(f) &\coloneqq \lambda v.\, \text{reify}_{t_2}(f(\text{reflect}_{t_1}(v))) \\
\text{reify}_b(f) &\coloneqq f \\
\\
\text{reflect}_t &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reflect}_{t_1 \to t_2}(e) &\coloneqq \lambda x.\, \text{reflect}_{t_2}(e(\text{reify}_{t_1}(x)) \\
\text{reflect}_b(e) &\coloneqq e \\
\\
\text{reduce} &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reduce}(\lambda v.\ e) &\coloneqq \lambda x.\ \text{reduce}([x/v]e) \\
\text{reduce}(e_1\ e_2) &\coloneqq (\text{reduce}(e_1))\,(\text{reduce}(e_2)) \\
\text{reduce}(x) &\coloneqq x \\
\text{reduce}(c) &\coloneqq \text{reflect}(c) \\
\\
\text{NbE} &: \text{expr}(t) \to \text{expr}(t) \\
\text{NbE}(e) &\coloneqq \text{reify}(\text{reduce}(e))
\end{aligned}
$$

Figure 5-2: Implementation of normalization by evaluation

this function only tries to apply rewrite rules at the top level of its input.

In the constant case, we still reflect the constant, but underneath the binders introduced by full η-expansion, we perform one instance of rewriting. In other words, we change this one function-definition clause:

$$\text{reflect}_b(e) \coloneqq \text{rewrite-head}(e)$$

It is important to note that a constant of function type will be η-expanded only once for each syntactic occurrence in the starting term, though the expanded function is effectively a thunk, waiting to perform rewriting again each time it is called. From first principles, it is not clear why such a strategy terminates on all possible input terms, though we work up to convincing Coq of that fact.

The details so far are essentially the same as in the approach of Aehlig, Haftmann, and Nipkow [1]. Recall that their rewriter was implemented in a deeply embedded ML, while ours is implemented in Coq's logic, which enforces termination of all functions. Aehlig et al. did not prove termination, which indeed does not hold for their rewriter in general, which works with untyped terms, not to mention the possibility of rule-specific ML functions that diverge themselves. In contrast, we need to convince Coq up-front that our interleaved λ-term normalization and algebraic simplification always terminate. Additionally, we need to prove that our rewriter preserves denotations

of terms, which can easily devolve into tedious binder bookkeeping, depending on encoding.

The next section introduces the techniques we use to avoid explicit termination proof or binder bookkeeping, in the context of a more general analysis of scaling challenges.

5.4 Scaling Challenges

Aehlig, Haftmann, and Nipkow [1] only evaluated their implementation against closed programs. What happens when we try to apply the approach to partial-evaluation problems that should generate thousands of lines of low-level code?

5.4.1 Variable Environments Will Be Large

We should think carefully about representation of ASTs, since many primitive operations on variables will run in the course of a single partial evaluation. For instance, Aehlig, Haftmann, and Nipkow [1] reported a significant performance improvement changing variable nodes from using strings to using de Bruijn indices [8]. However, de Bruijn indices and other first-order representations remain painful to work with. We often need to fix up indices in a term being substituted in a new context. Even looking up a variable in an environment tends to incur linear time overhead, thanks to traversal of a list. Perhaps we can do better with some kind of balanced-tree data structure, but there is a fundamental performance gap versus the arrays that can be used in imperative implementations. Unfortunately, it is difficult to integrate arrays soundly in a logic. Also, even ignoring performance overheads, tedious binder bookkeeping complicates proofs.

Our strategy is to use a variable encoding that pushes all first-order bookkeeping off on Coq's kernel, which is itself performance-tuned with some crucial pieces of imperative code. Parametric higher-order abstract syntax (PHOAS) [9] is a dependently typed encoding of syntax where binders are managed by the enclosing type system. It allows for relatively easy implementation and proof for NbE, so we adopted it for our framework.

Here is the actual inductive definition of term syntax for our object language, PHOAS-style. The characteristic oddity is that the core syntax type `expr` is parameterized on a dependent type family for representing variables. However, the final representation type `Expr` uses first-class polymorphism over choices of variable type, bootstrapping on the metalanguage's parametricity to ensure that a syntax tree is agnostic to variable type.

```
Inductive type := arrow (s d : type)
| base (b : base_type).
Infix "->" := arrow.
```

```
  Inductive expr (var : type -> Type)
    : type -> Type :=
  | Var {t} (v : var t) : expr var t
  | Abs {s d} (f : var s -> expr var d)
    : expr var (s -> d)
  | App {s d} (f : expr var (s -> d))
    (x : expr var s) : expr var d
  | Const {t} (c : const t) : expr var t
  Definition Expr (t : type) : Type :=
    forall var, expr var t.
```

A good example of encoding adequacy is assigning a simple denotational semantics. First, a simple recursive function assigns meanings to types.

```
Fixpoint denoteT (t : type) : Type
  := match t with
     | arrow s d => denoteT s -> denoteT d
     | base b    => denote_base_type b
     end.
```

Next we see the convenience of being able to *use* an expression by choosing how it should represent variables. Specifically, it is natural to choose *the type-denotation function itself* as the variable representation. Especially note how this choice makes rigorous the convention we followed in the prior section, where a recursive function enforces that values have always been substituted for variables early enough.

```
Fixpoint denoteE {t} (e : expr denoteT t) : denoteT t
  := match e with
     | Var v     => v
     | Abs f     => λ x, denoteE (f x)
     | App f x   => (denoteE f) (denoteE x)
     | Ident c   => denoteI c
     end.
Definition DenoteE {t} (E : Expr t) : denoteT t
  := denoteE (E denoteT).
```

It is now easy to follow the same script in making our rewriting-enabled NbE fully formal. Note especially the first clause of `reduce`, where we avoid variable substitution precisely because we have chosen to represent variables with normalized semantic values. The subtlety there is that base-type semantic values are themselves expression syntax trees, which depend on a nested choice of variable representation, which we retain as a parameter throughout these recursive functions. The final definition λ-quantifies over that choice.

```
Fixpoint nbeT var (t : type) : Type
  := match t with
     | arrow s d => nbeT var s -> nbeT var d
     | base b    => expr var b
     end.
Fixpoint reify {var t} : nbeT var t -> expr var t
  := match t with
     | arrow s d => λ f,
       Abs (λ x, reify (f (reflect (Var x))))
     | base b    => λ e, e
     end
with reflect {var t} : expr var t -> nbeT var t
  := match t with
     | arrow s d => λ e,
       λ x, reflect (App e (reify x))
     | base b    => rewrite_head
     end.
Fixpoint reduce {var t}
  (e : expr (nbeT var) t) : nbeT var t
  := match e with
     | Abs e     => λ x, reduce (e (Var x))
     | App e1 e2 => (reduce e1) (reduce e2)
     | Var x     => x
     | Ident c   => reflect (Ident c)
     end.
Definition Rewrite {t} (E : Expr t) : Expr t
  := λ var, reify (reduce (E (nbeT var t))).
```

One subtlety hidden above in implicit arguments is in the final clause of `reduce`, where the two applications of the `Ident` constructor use different variable representations. With all those details hashed out, we can prove a pleasingly simple correctness theorem, with a lemma for each main definition, with inductive structure mirroring recursive structure of the definition, also appealing to correctness of last section's pattern-compilation operations.

$$\forall t, E : \texttt{Expr t}.\ [\![\textbf{Rewrite}(E)]\!] = [\![E]\!]$$

Even before getting to the correctness theorem, we needed to convince Coq that the function terminates. While for Aehlig, Haftmann, and Nipkow [1], a termination proof would have been a whole separate enterprise, it turns out that PHOAS and NbE line up so well that Coq accepts the above code with no additional termination proof. As a result, the Coq kernel is ready to run our **Rewrite** procedure during checking.

To understand how we now apply the soundness theorem in a tactic, it is important

to note that the Coq kernel's built-in reduction strategies have, to an extent, been tuned to work well to show equivalence between a simple denotational-semantics application and the semantic value it produces, while it is rather difficult to code up one reduction strategy that works well for all partial-evaluation tasks. Therefore, we should restrict ourselves to (1) running full reduction in the style of functional-language interpreters and (2) running normal reduction on "known-good" goals like correctness of evaluation of a denotational semantics on a concrete input.

Operationally, then, we apply our tactic in a goal containing a term e that we want to partially evaluate. In standard proof-by-reflection style, we *reify* e into some E where $[\![E]\!] = e$, replacing e accordingly, asking Coq's kernel to validate the equivalence via standard reduction. Now we use the `Rewrite` correctness theorem to replace $[\![E]\!]$ with $[\![\texttt{Rewrite}(E)]\!]$. Next we may ask the Coq kernel to simplify $\texttt{Rewrite}(E)$ by *full reduction via compilation to native code*, since we carefully designed $\texttt{Rewrite}(E)$ and its dependencies to produce closed syntax trees. Finally, where E' is the result of that reduction, we simplify $[\![E']\!]$ with standard reduction, producing a normal-looking Coq term.

5.4.2 Subterm Sharing is Crucial

For some large-scale partial-evaluation problems, it is important to represent output programs with sharing of common subterms. Redundantly inlining shared subterms can lead to exponential increase in space requirements. Consider the Fiat Cryptography [12] example of generating a 64-bit implementation of field arithmetic for the P-256 elliptic curve. The library has been converted manually to continuation-passing style, allowing proper generation of `let` binders, whose variables are often mentioned multiple times. We ran their code generator (actually just a subset of its functionality, but optimized by us a bit further, as explained in Subsection 5.5.2) on the P-256 example and found it took about 15 seconds to finish. Then we modified reduction to inline `let` binders instead of preserving them, at which point the reduction job terminated with an out-of-memory error, on a machine with 64 GB of RAM. (The successful run uses under 2 GB.)

We see a tension here between performance and niceness of library implementation. The Fiat Cryptography authors found it necessary to CPS-convert their code to coax Coq into adequate reduction performance. Then all of their correctness theorems were complicated by reasoning about continuations. It feels like a slippery slope on the path to implementing a domain-specific compiler, rather than taking advantage of the pleasing simplicity of partial evaluation on natural functional programs. Our reduction engine takes shared-subterm preservation seriously while applying to libraries in direct style.

Our approach is `let`-lifting: we lift `let`s to top level, so that applications of functions

to `let`s are available for rewriting. For example, we can perform the rewriting

$$\begin{aligned} & \texttt{map}\ (\lambda x.\, y + x)\ (\texttt{let}\ z := e\ \texttt{in}\ [0; 1; 2; z; z + 1]) \\ \rightsquigarrow\ & \texttt{let}\ z := e\ \texttt{in}\ [y; y + 1; y + 2; y + z; y + (z + 1)] \end{aligned}$$

using the rules

$$\begin{aligned} \texttt{map}\ ?f\ [] &\to [] & ?n + 0 &\to n \\ \texttt{map}\ ?f\ (?x :: ?xs) &\to f\ x :: \texttt{map}\ f\ xs \end{aligned}$$

Our approach is to define a telescope-style type family called `UnderLets`:

```
Inductive UnderLets {var} (T : Type) :=
| Base (v : T)
| UnderLet {A}(e : @expr var A)(f : var A -> UnderLets T).
```

A value of type `UnderLets T` is a series of `let` binders (where each expression `e` may mention earlier-bound variables) ending in a value of type `T`. It is easy to build various "smart constructors" working with this type, for instance to construct a function application by lifting the `let`s of both function and argument to a common top level.

Such constructors are used to implement an NbE strategy that outputs `UnderLets` telescopes. Recall that the NbE type interpretation mapped base types to expression syntax trees. We now parameterize that type interpretation by a Boolean declaring whether we want to introduce telescopes.

```
Fixpoint nbeT' {var} (with_lets : bool) (t : type)
  := match t with
     | base t => if with_lets
                 then @UnderLets var (@expr var t)
                 else @expr var t
     | arrow s d => nbeT' false s -> nbeT' true d
     end.
Definition nbeT := nbeT' false.
Definition nbeT_with_lets := nbeT' true.
```

There are cases where naive preservation of `let` binders leads to suboptimal performance, so we include some heuristics. For instance, when the expression being bound is a constant, we always inline. When the expression being bound is a series of list "cons" operations, we introduce a name for each individual list element, since such a list might be traversed multiple times in different ways.

5.4.3 Rules Need Side Conditions

Many useful algebraic simplifications require side conditions. One simple case is supporting *nonlinear* patterns, where a pattern variable appears multiple times. We can encode nonlinearity on top of linear patterns via side conditions.

$$?n_1 + ?m - ?n_2 \to m \text{ if } n_1 = n_2$$

The trouble is how to support predictable solving of side conditions during partial evaluation, where we may be rewriting in open terms. We decided to sidestep this problem by allowing side conditions only as executable Boolean functions, to be applied only to variables that are confirmed as *compile-time constants*, unlike Malecha and Bengtson [24] who support general unification variables. We added a variant of pattern variable that only matches constants. Semantically, this variable style has no additional meaning, and in fact we implement it as a special identity function that should be called in the right places within Coq lemma statements. Rather, use of this identity function triggers the right behavior in our tactic code that reifies lemma statements. We introduce a notation where a prefixed apostrophe signals a call to the "constants only" function.

Our reification inspects the hypotheses of lemma statements, using type classes to find decidable realizations of the predicates that are used, synthesizing one Boolean expression of our deeply embedded term language, standing for a decision procedure for the hypotheses. The `Make` command fails if any such expression contains pattern variables not marked as constants. Therefore, matching of rules can safely run side conditions, knowing that Coq's full-reduction engine can determine their truth efficiently.

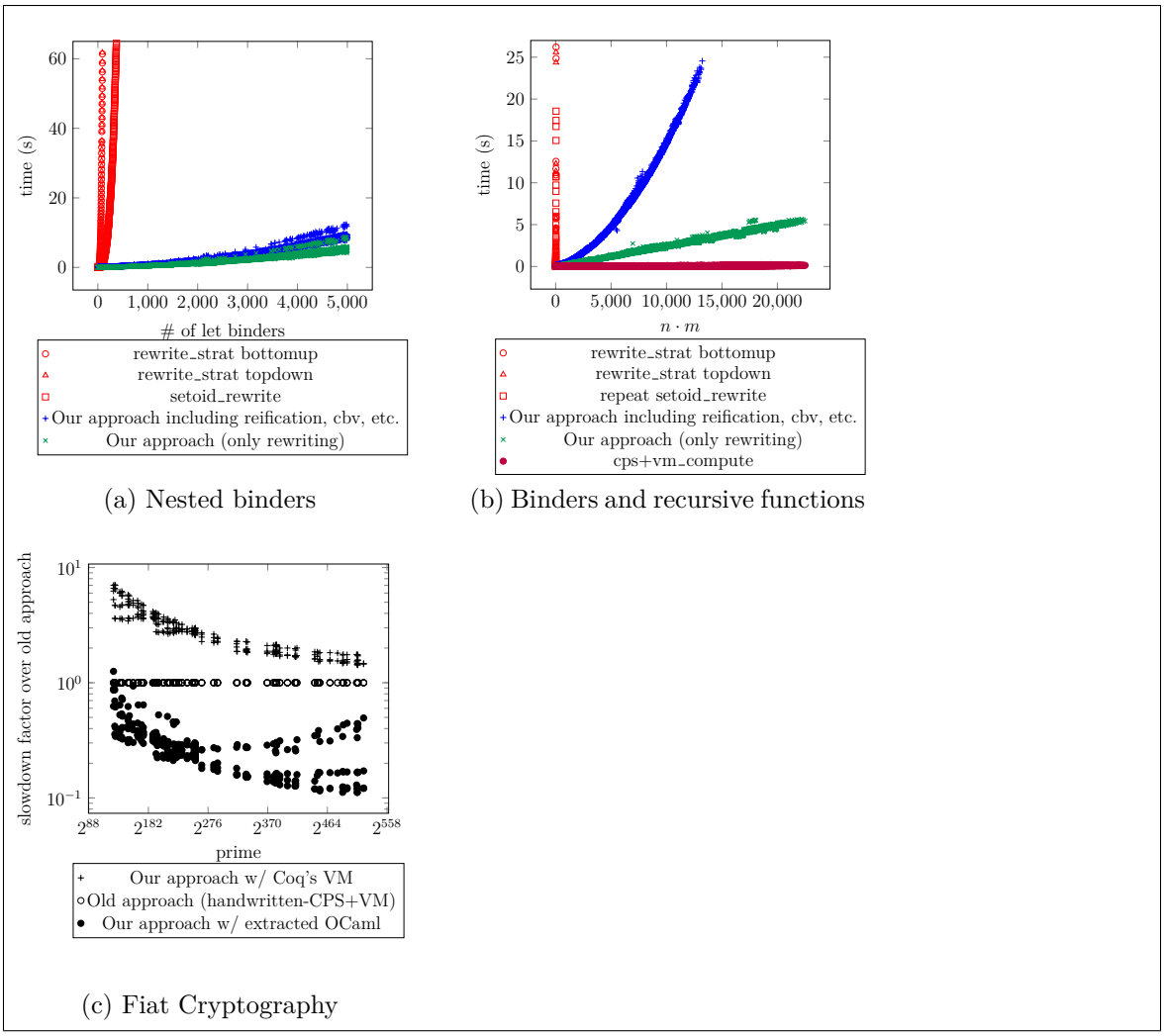
5.4.4 Side Conditions Need Abstract Interpretation

With our limitation that side conditions are decided by executable Boolean procedures, we cannot yet handle directly some of the rewrites needed for realistic partial evaluation. For instance, Fiat Cryptography reduces high-level functional to low-level code that only uses integer types available on the target hardware. The starting library code works with infinite-precision integers, while the generated low-level code should be careful to avoid unintended integer overflow. As a result, the setup may be too naive for our running example rule $?n + 0 \to n$. When we get to reducing fixed-precision-integer terms, we must be legalistic:

$$\texttt{add\_with\_carry}_{64}(?n, 0) \to (0, n) \text{ if } 0 \leq n < 2^{64}$$

We developed a design pattern to handle this kind of rule.

First, we introduce a family of functions $\texttt{clip}_{l,u}$, each of which forces its integer argu-



(a) Nested binders

(b) Binders and recursive functions

(c) Fiat Cryptography

Figure 5-3: Timing of different partial-evaluation implementations

ment to respect lower bound l and upper bound u. Partial evaluation is proved with respect to unknown realizations of these functions, only requiring that $\texttt{clip}_{l,u}(n) = n$ when $l \leq n < u$. Now, before we begin partial evaluation, we can run a verified abstract interpreter to find conservative bounds for each program variable. When bounds l and u are found for variable x, it is sound to replace x with $\texttt{clip}_{l,u}(x)$. Therefore, at the end of this phase, we assume all variable occurrences have been rewritten in this manner to record their proved bounds.

Second, we proceed with our example rule refactored:

$$\texttt{add\_with\_carry}_{64}(\texttt{clip}_{\prime?l,\prime?u}(?n), 0) \to (0, \texttt{clip}_{l,u}(n)) \quad \text{if } u < 2^{64}$$

If the abstract interpreter did its job, then all lower and upper bounds are constants, and we can execute side conditions straightforwardly during pattern matching.

5.5 Evaluation

Our implementation, attached to this submission as an anonymized supplement with a roadmap in Section 5.D, includes a mix of Coq code for the proved core of rewriting, tactic code for setting up proper use of that core, and OCaml plugin code for the manipulations beyond the current capabilities of the tactic language. We report here on experiments to isolate performance benefits for rewriting under binders and reducing higher-order structure.

5.5.1 Microbenchmarks

We start with microbenchmarks focusing attention on particular aspects of reduction and rewriting, with Section 5.A going into more detail.

Rewriting Under Binders

Consider

$$\begin{aligned}&\texttt{let } v_1 := v_0 + v_0 + 0 \texttt{ in}\\ &\vdots\\ &\texttt{let } v_n := v_{n-1} + v_{n-1} + 0 \texttt{ in}\\ &v_n + v_n + 0\end{aligned}$$

We want to remove all of the $+\,0$s. We can start from this expression directly, in which case reification alone takes as much time as `setoid_rewrite`. As the reification method was not especially optimized, and there exist fast reification methods [17], we instead start from a call to a recursive function that generates such a sequence of `let` bindings.

5-3a shows the results. The comparison points are Coq's `setoid_rewrite` and `rewrite_strat`. The former performs one rewrite at a time, taking minimal advantage of commonalities across them and thus generating quite large, redundant proof terms. The latter makes top-down or bottom-up passes with combined generation of proof terms. For our own approach, we list both the total time and the time taken for core execution of a verified rewrite engine, without counting reification (converting goals to ASTs) or its inverse (interpreting results back to normal-looking goals).

The comparison here is very favorable for our approach. The competing tactics spike upward toward timeouts at just a few hundred generated binders, while our engine is only taking about 10 seconds for examples with 5,000 nested binders.

As detailed in Subsection 5.A.2, we ran a variant of this experiment with inlining of `let`s, forcing terms to grow quite large. Specifically, we generate n nested `let`s, each repeatedly adding a designated free variable into a sum, m times. Holding m fixed at a small value and letting n scale, we continue dominating the methods described above, though Coq's `rewrite!` tactic (to rewrite with one lemma many times) does

better for $m < 2$. Holding n fixed and letting m scale, all other approaches quickly spike upward to timeouts, while ours holds steady even for $m = 1000$.

Binders and Recursive Functions

The next experiment uses the following example.

$$\text{map\_dbl}(\ell) \coloneqq \begin{cases} [] & \text{if } \ell = [] \\ \texttt{let } y := h + h \texttt{ in } y :: \text{map\_dbl}(t) & \text{if } \ell = h :: t \end{cases}$$

$$\text{make}(n, m, v) \coloneqq \begin{cases} [\underbrace{v, \ldots, v}_{n}] & \text{if } m = 0 \\ \text{map\_dbl}(\text{make}(n, m-1, v)) & \text{if } m > 0 \end{cases}$$

$$\text{example}_{n,m} \coloneqq \forall v,\ \text{make}(n, m, v) = []$$

Note that the `let ⋯ in ⋯` binding blocks further reduction of map_dbl, which we iterate m times, and so we need to take care to preserve sharing when reducing here.

5-3b compares performance between our approach, `repeat setoid_rewrite`, and two variants of `rewrite_strat`. Additionally, we consider another option, which was adopted by Fiat Cryptography at a larger scale: rewrite our functions to improve reduction behavior. Specifically, both functions are rewritten in continuation-passing style, which makes them harder to read and reason about but allows standard VM-based reduction to achieve good performance. The figure shows that `rewrite_strat` variants are essentially unusable for this example, with `setoid_rewrite` performing only marginally better, while our approach applied to the original, more readable definitions loses ground steadily to VM-based reduction on CPSed code. On the largest terms ($n \cdot m > 20,000$), the gap is 6s vs. 0.1s of compilation time, which should often be acceptable in return for simplified coding and proofs, plus the ability to mix proved rewrite rules with built-in reductions. See Subsection 5.A.3 for more on this microbenchmark and Subsection 5.A.4 for an even more extreme example of full reduction with a Sieve of Eratosthenes as in the experiments of Aehlig, Haftmann, and Nipkow [1] (ours 10s, VM 0.3s).

5.5.2 Macrobenchmark: Fiat Cryptography

Finally, we consider an experiment (described in more detail in Section 5.B) replicating the generation of performance-competitive finite-field-arithmetic code for all popular elliptic curves by Erbsen et al. [12]. In all cases, we generate essentially the same code as they did, so we only measure performance of the code-generation process. We stage partial evaluation with three different reduction engines (i.e., three `Make` invocations), respectively applying 85, 56, and 44 rewrite rules (with only 2 rules shared across engines), taking total time of about 5 minutes to generate all three engines. These engines support 95 distinct function symbols.

5-3c graphs running time of three different partial-evaluation methods for Fiat Cryptography, as the prime modulus of arithmetic scales up. Times are normalized to the performance of the original method, which relied entirely on standard Coq reduction. Actually, in the course of running this experiment, we found a way to improve the old approach for a fairer comparison. It had relied on Coq's configurable `cbv` tactic to perform reduction with selected rules of the definitional equality, which the Fiat Cryptography developers had applied to blacklist identifiers that should be left for compile-time execution. By instead hiding those identifiers behind opaque module-signature ascription, we were able to run Coq's more-optimized virtual-machine-based reducer.

As the figure shows, our approach running partial evaluation inside Coq's kernel begins with about a 10× performance disadvantage vs. the original method. With log scale on both axes, we see that this disadvantage narrows to become nearly negligible for the largest primes, of around 500 bits. (We used the same set of prime moduli as in the experiments run by Erbsen et al. [12], which were chosen based on searching the archives of an elliptic-curves mailing list for all prime numbers.) It makes sense that execution inside Coq leaves our new approach at a disadvantage, as we are essentially running an interpreter (our normalizer) within an interpreter (Coq's kernel), while the old approach ran just the latter directly. Also recall that the old approach required rewriting Fiat Cryptography's library of arithmetic functions in continuation-passing style, enduring this complexity in library correctness proofs, while our new approach applies to a direct-style library. Finally, the old approach included a custom reflection-based arithmetic simplifier for term syntax, run after traditional reduction, whereas now we are able to apply a generic engine that combines both, without requiring more than proving traditional rewrite rules.

The figure also confirms clear performance advantage of running reduction in code extracted to OCaml, which is possible because our plugin produces verified code in Coq's functional language. By the time we reach middle-of-the-pack prime size around 300 bits, the extracted version is running about 10× as quickly as the baseline.

5.6 Related Work

We have already discussed the work of Aehlig, Haftmann, and Nipkow [1], which introduced the basic structure that our engine shares, but which required a substantially larger trusted code base, did not tackle certain challenges in scaling to large partial-evaluation problems, and did not report any performance experiments in partial evaluation.

We have also mentioned $\mathcal{R}_{tac}$ [24], which implements an experimental reflective version of `rewrite_strat` supporting arbitrary setoid relations, unification variables, and arbitrary semi-decidable side conditions solvable by other reflective tactics, using de Bruijn indexing to manage binders. We were unfortunately unable to get the rewriter

to work with Coq 8.10 and were also not able to determine from the paper how to repurpose the rewriter to handle our benchmarks.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [15] or compilation to native code [6]. Especially the latter is similar in adopting an NbE style for full reduction, simplifying even under λs, on top of a more traditional implementation of OCaml that never executes preemptively under λs. Neither approach unifies support for rewriting with proved rules, and partial evaluation only applies in very limited cases, where functions that should not be evaluated at compile time must have properly opaque definitions that the evaluator will not consult. Neither implementation involved a machine-checked proof suitable to bootstrap on top of reduction support in a kernel providing simpler reduction.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [31] in Scala as one of the best-known current examples. A kind of type-based overloading for staging annotations is used to smooth the rough edges in writing code that manipulates syntax trees. The LMS-Verify system [2] can be used for formal verification of generated code after-the-fact. Typically LMS-Verify has been used with relatively shallow properties (though potentially applied to larger and more sophisticated code bases than we tackle), not scaling to the kinds of functional-correctness properties that concern us here, justifying investment in verified partial evaluators.

5.7 Future Work

There are a number of natural extensions to our engine. For instance, we do not yet allow pattern variables marked as "constants only" to apply to container datatypes; we limit the mixing of higher-order and polymorphic types, as well as limiting use of first-class polymorphism; we do not support proving equalities on functions; we only support decidable predicates as rule side conditions, and the predicates may only mention pattern variables restricted to matching constants; we have hardcoded support for a small set of container types and their eliminators; we support rewriting with equality and no other relations (e.g., subset inclusion); and we require decidable equality for all types mentioned in rules. It may be helpful to design an engine that lifts some or all of these limitations, building on the basic structure that we present here.

5.A Additional Information on Microbenchmarks

We performed all benchmarks on a 3.5 GHz Core i7 running Linux and Coq 8.10.0. We name the subsections here with the names that show up in the code supplement.

5.A.1 UnderLetsPlus0

We provide more detail on the "nested binders" microbenchmark of Section 5.5.1 displayed in 5-3a.

Recall that we are removing all of the $+0$s from

$$\begin{aligned}
&\texttt{let } v_1 := v_0 + v_0 + 0 \texttt{ in} \\
&\vdots \\
&\texttt{let } v_n := v_{n-1} + v_{n-1} + 0 \texttt{ in} \\
&v_n + v_n + 0
\end{aligned}$$

The code used to define this microbenchmark is

```
Definition make_lets_def (n:nat) (v acc : Z) :=
 @nat_rect
   (fun _ => Z * Z -> Z)
   (fun '(v, acc) => acc + acc + v)
   (fun _ rec '(v, acc) =>
     dlet acc := acc + acc + v in rec (v, acc))
   n
   (v, acc).
```

We note some details of the rewriting framework that were glossed over in the main body of the paper, which are useful for using the code: Although the rewriting framework does not support dependently typed constants, we can automatically preprocess uses of eliminators like `nat_rect` and `list_rect` into non-dependent versions. The tactic that does this preprocessing is extensible via $\mathcal{L}_{tac}$'s reassignment feature. Since pattern-matching compilation mixed with NbE requires knowing how many arguments a constant can be applied to, we must internally use a version of the recursion principle whose type arguments do not contain arrows; current preprocessing can handle recursion principles with either no arrows or one arrow in the motive. Even though we will eventually plug in 0 for v, we jump through some extra hoops to ensure that our rewriter cannot cheat by rewriting away the $+0$ before reducing the recursion on n.

We can reduce this expression in three ways.

Our Rewriter

One lemma is required for rewriting with our rewriter:

```
Lemma Z.add_0_r : forall z, z + 0 = z.
```

Creating the rewriter takes about 12 seconds on the machine we used for running the performance experiments:

```
Make myrew := Rewriter For
 (Z.add_0_r, eval_rect nat, eval_rect prod).
```

Recall from Subsection 5.1.1 that `eval_rect` is a definition provided by our framework for eagerly evaluating recursion associated with certain types. It functions by triggering typeclass resolution for the lemmas reducing the recursion principle associated to the given type. We provide instances for `nat`, `prod`, `list`, `option`, and `bool`. Users may add more instances if they desire.

`setoid_rewrite` and `rewrite_strat`

To give as many advantages as we can to the preexisting work on rewriting, we prereduce the recursion on `nat`s using `cbv` before performing `setoid_rewrite`. (Note that `setoid_rewrite` cannot itself perform reduction without generating large proof terms, and `rewrite_strat` is not currently capable of sequencing reduction with rewriting internally due to bugs such as #10923.) Rewriting itself is easy; we may use any of `repeat setoid_rewrite Z.add_0_r`, `rewrite_strat topdown Z.add_0_r`, or `rewrite_strat bottomup Z.add_0_r`.

5.A.2 Plus0Tree

This is a version of Subsection 5.A.1 without any let binders, discussed in Section 5.5.1 but not displayed in Figure 5-3.

We use two definitions for this microbenchmark:

```
Definition iter (m : nat) (acc v : Z) :=
  @nat_rect
    (fun _ => Z -> Z)
    (fun acc => acc)
    (fun _ rec acc => rec (acc + v))
    m
    acc.
```

```
Definition make_tree (n m : nat) (v acc : Z) :=
 Eval cbv [iter] in
  @nat_rect
    (fun _ => Z * Z -> Z)
    (fun '(v, acc) => iter m (acc + acc) v)
    (fun _ rec '(v, acc) =>
      iter m (rec (v, acc) + rec (v, acc)) v)
    n
    (v, acc).
```

We can see from the graphs in Figure 5-4 and Figure 5-5 that (a) we incur constant overhead over most of the other methods which dominates on small examples; (b) when the term is quite large and there are few opportunities for rewriting relative to the term-size (i.e., $m \leq 2$), we are worse than `rewrite !Z.add_0_r`, but still better than the other methods; and (c) when there are many opportunities for rewriting relative to the term-size ($m > 2$), we thoroughly dominate the other methods.

5.A.3 LiftLetsMap

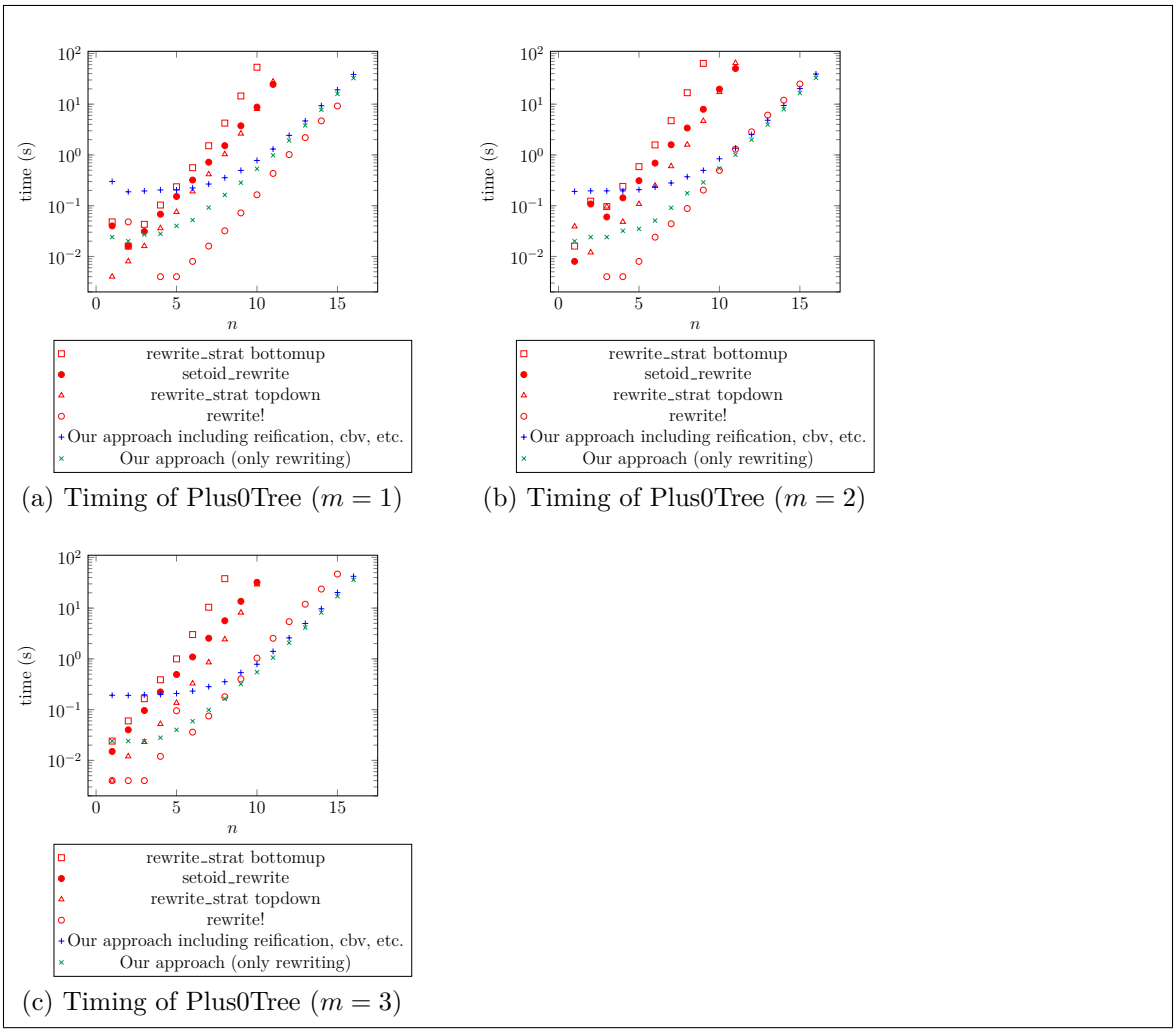
We now discuss in more detail the "binders and recursive functions" example from Section 5.5.1.

The expression we want to get out at the end looks like:

$$\begin{aligned}
&\texttt{let } v_{1,1} := v + v \texttt{ in} \\
&\vdots \\
&\texttt{let } v_{1,n} := v + v \texttt{ in} \\
&\texttt{let } v_{2,1} := v_{1,1} + v_{1,1} \texttt{ in} \\
&\vdots \\
&\texttt{let } v_{2,n} := v_{1,n} + v_{1,n} \texttt{ in} \\
&\vdots \\
&[v_{m,1}, \ldots, v_{m,n}]
\end{aligned}$$

Recall that we make this example with the code

```
Definition map_double (ls : list Z) :=
  list_rect
    _
    []
    (λ x xs rec, let y := x + x in y :: rec)
    ls.
```

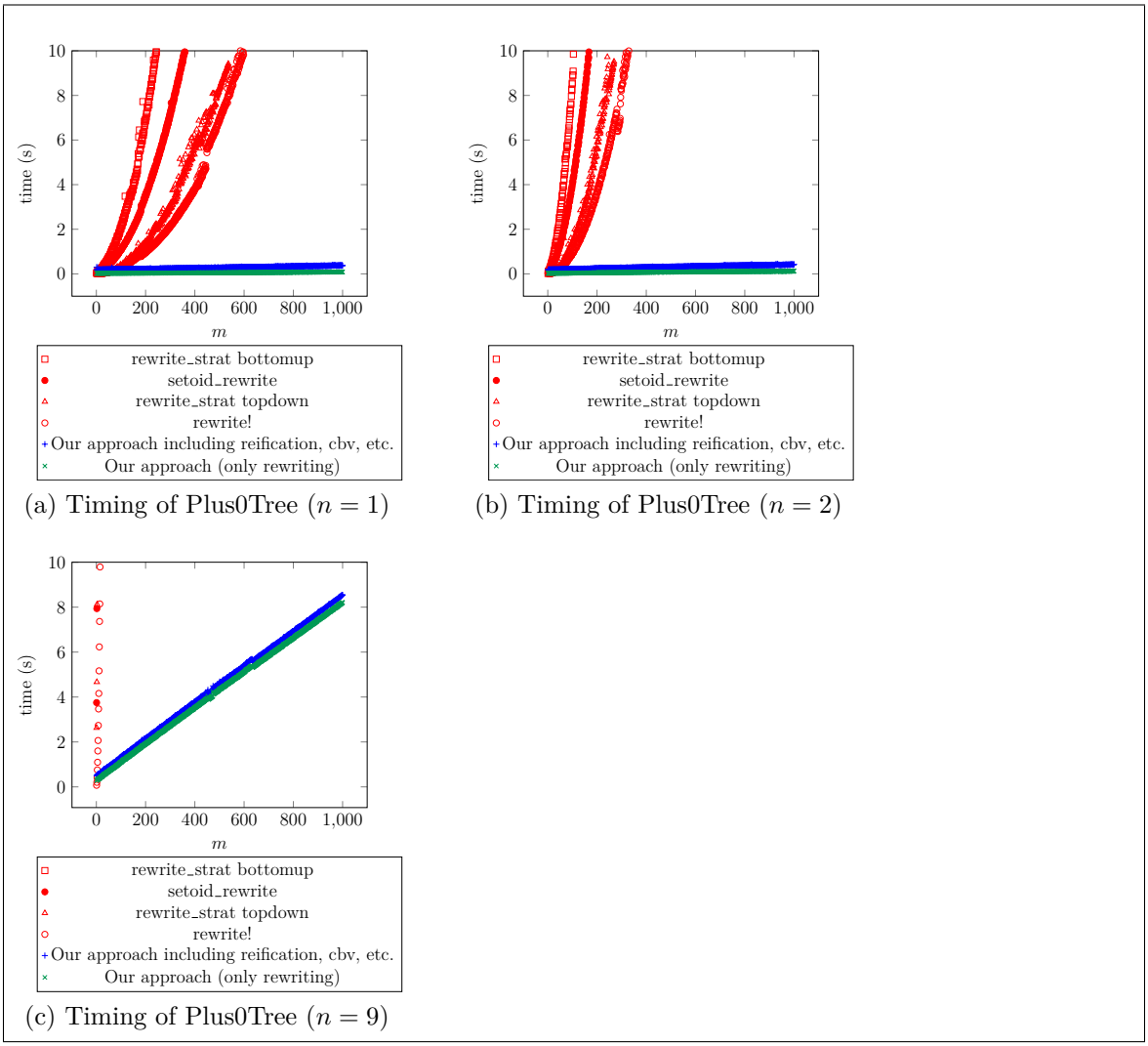


(a) Timing of Plus0Tree ($m = 1$)

(b) Timing of Plus0Tree ($m = 2$)

(c) Timing of Plus0Tree ($m = 3$)

Figure 5-4: Timing of different partial-evaluation implementations for Plus0Tree for fixed m. Note that we have a logarithmic time scale, because term size is proportional to 2^n.



(a) Timing of Plus0Tree ($n = 1$)

(b) Timing of Plus0Tree ($n = 2$)

(c) Timing of Plus0Tree ($n = 9$)

Figure 5-5: Timing of different partial-evaluation implementations for Plus0Tree for fixed n (1, 2, and then we jump to 9)

```
Definition make (n : nat) (m : nat) (v : Z) :=
  nat_rect
    _
    (List.repeat v n)
    (λ _ rec, map_double rec)
    m.
```

We can perform this rewriting in four ways; see 5-3b.

Note that `rewrite_strat` grows quite quickly, hitting a minute when the total number of rewrites ($n \cdot m$) is in the mid-40s. Our method performs much better, but the fact that we have to perform `cbv` at the end costs us; about 99% of the difference between the full time of our method and just the rewriting is spent in the final `cbv` at the end. This is due to the unfortunate fact that reduction in Coq is quadratic in the number of nested binders present; see Coq bug #11151. Finally, and unsurprisingly, `vm_compute` outperforms us.

Our Rewriter

One lemma is required for rewriting with our rewriter:

```
Lemma eval_repeat A x n :
  @List.repeat A x ('n)
  = ident.eagerly nat_rect _
      []
      (λ k repeat_k, x :: repeat_k)
      ('n).
```

Recall that the apostrophe marker (`'`) is explained in Subsection 5.1.1. Recall again from Subsection 5.1.1 that we use `ident.eagerly` to ask the reducer to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree. Our current version only allows a limited, hard-coded set of eliminators with `ident.eagerly` (`nat_rect` on return types with either zero or one arrows, `list_rect` on return types with either zero or one arrows, and `List.nth_default`), but nothing in principle prevents automatic generation of the necessary code.

We construct our rewriter with

```
Make myrew := Rewriter For
  (eval_repeat, eval_rect list, eval_rect nat)
  (with extra idents (Z.add)).
```

On the machine we used for running all our performance experiments, this command takes about 13 seconds to run. Note that all identifiers which appear in any goal to be rewritten must either appear in the type of one of the rewrite rules or in the tuple passed to `with extra idents`.

Rewriting is relatively simple, now. Simply invoke the tactic `Rewrite_for myrew`. We support rewriting on only the left-hand-side and on only the right-hand-side using either the tactic `Rewrite_lhs_for myrew` or else the tactic `Rewrite_rhs_for myrew`, respectively.

`rewrite_strat`

To reduce adequately using `rewrite_strat`, we need the following two lemmas:

```
Lemma lift_let_list_rect T A P N C (v : A) fls
: @list_rect T P N C (Let_In v fls)
  = Let_In v (fun v => @list_rect T P N C (fls v)).
Lemma lift_let_cons T A x (v : A) f
: @cons T x (Let_In v f)
  = Let_In v (fun v => @cons T x (f v)).
```

Note that `Let_In` is the constant we use for writing `let ⋯ in ⋯` expressions that do not reduce under ζ. Throughout most of this paper, anywhere that `let ⋯ in ⋯` appears, we have actually used `Let_In` in the code. It would alternatively be possible to extend the reification preprocessor to automatically convert `let ⋯ in ⋯` to `Let_In`, but this may cause problems when converting the interpretation of the reified term with the pre-reified term, as Coq's conversion does not allow fine-tuning of when to inline or unfold `let`s.

To rewrite, we start with `cbv [example make map_dbl]` to expose the underlying term to rewriting. One would hope that one could just add these two hints to a database `db` and then write `rewrite_strat (repeat (eval cbn [list_rect]; try bottomup hints db))`, but unfortunately this does not work due to a number of bugs in Coq: #10934, #10923, #4175, #10955, and the potential to hit #10972. Instead, we must put the two lemmas in separate databases, and then write `repeat (cbn [list_rect]; (rewrite_strat (try repeat bottomup hints db1)); (rewrite_strat (try repeat bottomup hints db2)))`. Note that the rewriting with `lift_let_cons` can be done either top-down or bottom-up, but `rewrite_strat` breaks if the rewriting with `lift_let_list_rect` is done top-down.

CPS and the VM

If we want to use Coq's built-in VM reduction without our rewriter, to achieve the prior state-of-the-art performance, we can do so on this example, because it only involves partial reduction and not equational rewriting. However, we must (a) module-

opacify the constants which are not to be unfolded, and (b) rewrite all of our code in CPS.

Then we are looking at

$$\text{map\_dbl\_cps}(\ell, k) \coloneqq \begin{cases} k([]) & \text{if } \ell = [] \\ \texttt{let } y := h +_{\text{ax}} h \texttt{ in} & \text{if } \ell = h :: t \\ \text{map\_dbl\_cps}(t, & \\ \quad (\lambda ys, k(y :: ys))) & \end{cases}$$

$$\text{make\_cps}(n, m, v, k) \coloneqq \begin{cases} k(\underbrace{[v, \ldots, v]}_{n}) & \text{if } m = 0 \\ \text{make\_cps}(n, m-1, v, & \text{if } m > 0 \\ \quad (\lambda \ell, \text{map\_dbl\_cps}(\ell, k)) & \end{cases}$$

$$\text{example\_cps}_{n,m} \coloneqq \forall v,\ \text{make\_cps}(n, m, v, \lambda x.\, x) = []$$

Then we can just run `vm_compute`. Note that this strategy, while quite fast, results in a stack overflow when $n \cdot m$ is larger than approximately $2.5 \cdot 10^4$. This is unsurprising, as we are generating quite large terms. Our framework can handle terms of this size but stack-overflows on only slightly larger terms.

Takeaway

From this example, we conclude that `rewrite_strat` is unsuitable for computations involving large terms with many binders, especially in cases where reduction and rewriting need to be interwoven, and that the many bugs in `rewrite_strat` result in confusing gymnastics required for success. The prior state of the art—writing code in CPS—suitably tweaked by using module pacity to allow `vm_compute`, remains the best performer here, though the cost of rewriting everything is CPS may be prohibitive. Our method soundly beats `rewrite_strat`. We are additionally bottlenecked on `cbv`, which is used to unfold the goal post-rewriting and costs about a minute on the largest of terms; see Coq bug #11151 for a discussion on what is wrong with Coq's reduction here.

5.A.4 SieveOfEratosthenes

To benchmark how much overhead we add when we are reducing fully, we compute the Sieve of Eratosthenes, taking inspiration on benchmark choice from Aehlig, Haftmann, and Nipkow [1]. We find in Figure 5-6 that we are slower than `vm_compute`, **`native_compute`**, and `cbv`, but faster than **`lazy`**, and of course much faster than `simpl` and `cbn`, which are quite slow.

We define the sieve using `PositiveMap.t` and **`list Z`**:

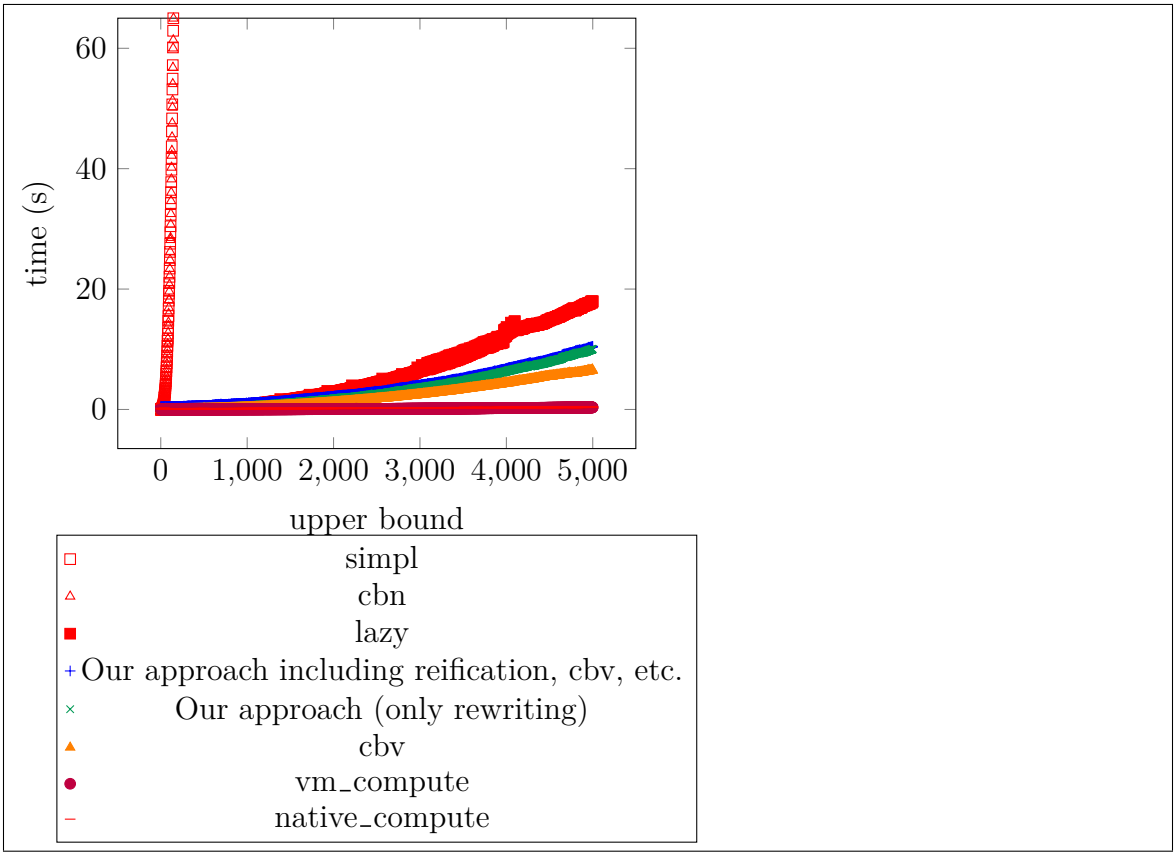


Figure 5-6: Timing of different full-evaluation implementations for SieveOfEratosthenes

```
Definition sieve' (fuel : nat) (max : Z) :=
 List.rev
  (fst
   (@nat_rect
    (λ _, list Z (* primes *) *
     PositiveSet.t (* composites *) *
     positive (* np (next_prime) *) ->
     list Z (* primes *) *
     PositiveSet.t (* composites *))
    (λ '(primes, composites, next_prime),
     (primes, composites))
    (λ _ rec '(primes, composites, np),
      rec
       (if (PositiveSet.mem np composites ||
            (Z.pos np >? max))%bool%Z
        then
         (primes, composites, Pos.succ np)
        else
         (Z.pos np :: primes,
          List.fold_right
           PositiveSet.add
           composites
           (List.map
            (λ n, Pos.mul (Pos.of_nat (S n)) np)
            (List.seq 0 (Z.to_nat(max/Z.pos np)))),
          Pos.succ np)))
    fuel
    (nil, PositiveSet.empty, 2%positive))).

Definition sieve (n : Z)
  := Eval cbv [sieve'] in sieve' (Z.to_nat n) n.
```

We need four lemmas and an additional instance to create the rewriter:

```
Lemma eval_fold_right A B f x ls :
@List.fold_right A B f x ls
= ident.eagerly list_rect _ _
    x
    (λ l ls fold_right_ls, f l fold_right_ls)
    ls.

Lemma eval_app A xs ys :
xs ++ ys
= ident.eagerly list_rect A _
    ys
    (λ x xs app_xs_ys, x :: app_xs_ys)
```

```
    xs.

Lemma eval_map A B f ls :
@List.map A B f ls
= ident.eagerly list_rect _ _
    []
    (λ l ls map_ls, f l :: map_ls)
    ls.

Lemma eval_rev A xs :
@List.rev A xs
= (@list_rect _ (fun _ => _))
    []
    (λ x xs rev_xs, rev_xs ++ [x])%list
    xs.

Scheme Equality for PositiveSet.tree.

Definition PositiveSet_t_beq
   : PositiveSet.t -> PositiveSet.t -> bool
  := tree_beq.

Global Instance PositiveSet_reflect_eqb
 : reflect_rel (@eq PositiveSet.t) PositiveSet_t_beq
 := reflect_of_brel
      internal_tree_dec_bl internal_tree_dec_lb.
```

We then create the rewriter with

```
Make myrew := Rewriter For
  (eval_rect nat, eval_rect prod, eval_fold_right,
   eval_map, do_again eval_rev, eval_rect bool,
   @fst_pair, eval_rect list, eval_app)
   (with extra idents (Z.eqb, orb, Z.gtb,
    PositiveSet.elements, @fst, @snd,
    PositiveSet.mem, Pos.succ, PositiveSet.add,
    List.fold_right, List.map, List.seq, Pos.mul,
    S, Pos.of_nat, Z.to_nat, Z.div, Z.pos, O,
    PositiveSet.empty))
  (with delta).
```

To get `cbn` and `simpl` to unfold our term fully, we emit

```
Global Arguments Pos.to_nat !_ / .
```

5.B Additional Information on Fiat Cryptography Benchmarks

It may also be useful to see performance results with absolute times, rather than normalized execution ratios vs. the original Fiat Cryptography implementation. Furthermore, the benchmarks fit into four quite different groupings: elements of the cross product of two algorithms (unsaturated Solinas and word-by-word Montgomery) and bitwidths of target architectures (32-bit or 64-bit). Here we provide absolute-time graphs by grouping in Figure 5-7.

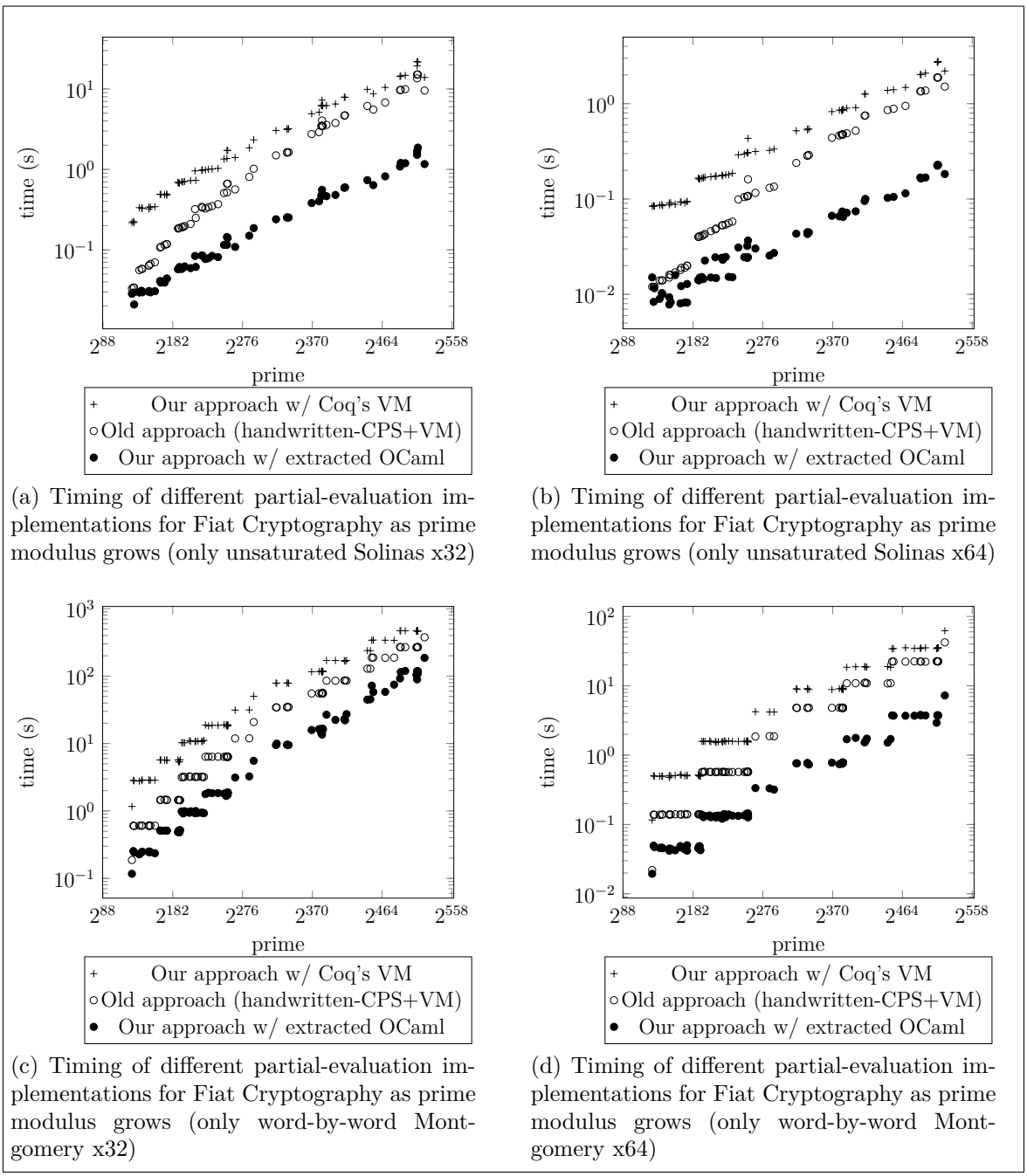
5.C Experience vs. Lean and `setoid_rewrite`

Although all of our toy examples work with `setoid_rewrite` or `rewrite_strat` (until the terms get too big), even the smallest of examples in Fiat Cryptography fell over using these tactics. When attempting to use `rewrite_strat` for partial evaluation and rewriting on unsaturated Solinas with 1 limb on small primes (such as 29), we were able to get `rewrite_strat` to finish after about 90 seconds. The bugs in `rewrite_strat` made finding the right magic invocation quite painful, nonetheless; the invocation we settled on involved *sixteen* consecutive calls to `rewrite_strat` with varying arguments and strategies. Trying to synthesize code for two limbs on slightly larger primes (such as 113, which needs two limbs on a 64-bit machine) took about three hours. The widely used primes tend to have around five to ten limbs; we leave extrapolating this slowdown to the reader.

We have attached this experiment using `rewrite_strat` as `fiat_crypto_via_rewrite_strat.v`, which is meant to be run in emacs/PG from inside the `fiat-crypto` directory, or in `coqc` by setting `COQPATH` to the value emitted by `make printenv` in `fiat-crypto` and then invoking the command `coqc -q -R /path/to/fiat-crypto/src Crypto /path/to/fiat_crypto_via_rewrite_strat.v`. To test with the two-limb prime 113, change `of_string "2^5-3" 8` in the definition of `p` to `of_string "2^7-15" 64`.

We also tried Lean, in the hopes that rewriting in Lean, specifically optimized for performance, would be up to the challenge. Although Lean performed about 30% better than Coq on the 1-limb example, taking a bit under a minute, it did not complete on the two-limb example even after four hours (after which we stopped trying), and a five-limb example was still going after 40 hours.

We have attached our experiments with running `rewrite` in Lean on the Fiat Cryptography code as a supplement as well. We used Lean version 3.4.2, commit cbd2b6686ddb, Release. Run `make` in `fiat-crypto-lean` to run the one-limb example; change `open ex` to `open ex2` to try the two-limb example, or to `open ex5` to try the five-limb example.



(a) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x32)

(b) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x64)

(c) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only word-by-word Montgomery x32)

(d) Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only word-by-word Montgomery x64)

Figure 5-7: Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows

5.D Reading the Code Supplement

We have attached both the code for implementing the rewriter, as well as a copy of Fiat Cryptography adapted to use the rewriting framework. Both code supplements build with Coq 8.9 and Coq 8.10, and they require that whichever OCaml was used to build Coq be installed on the system to permit building plugins. (If Coq was installed via opam, then the correct version of OCaml will automatically be available.) Both code bases can be built by running `make` in the top-level directory.

The performance data for both repositories are included at the top level as `.txt` and `.csv` files.

The performance data for the microbenchmarks can be rebuilt using `make perf-SuperFast perf-Fast perf-Medium` followed by `make perf-csv` to get the `.txt` and `.csv` files. The microbenchmarks should run in about 24 hours when run with `-j5` on a 3.5 GHz machine. There also exist targets `perf-Slow` and `perf-VerySlow`, but these take significantly longer.

The performance data for the macrobenchmark can be rebuilt from the Fiat Cryptography copy included by running `make perf -k`. We ran this with `PERF_MAX_TIME=3600` to allow each benchmark to run for up to an hour; the default is 10 minutes per benchmark. Expect the benchmarks to take over a week of time with an hour timeout and five cores. Some tests are expected to fail, making `-k` a necessary flag. Again, the `perf-csv` target will aggregate the logs and turn them into `.txt` and `.csv` files.

The entry point for the rewriter is the Coq source file `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v`.

The rewrite rules used in Fiat Cryptography are defined in `fiat-crypto/src/Rewriter/Rules.v` and proven in `fiat-crypto/src/Rewriter/RulesProofs.v`. Note that the Fiat Cryptography copy uses `COQPATH` for dependency management, and `.dir-locals.el` to set `COQPATH` in emacs/PG; you must accept the setting when opening a file in the directory for interactive compilation to work. Thus interactive editing either requires ProofGeneral or manual setting of `COQPATH`. The correct value of `COQPATH` can be found by running `make printenv`.

We will now go through this paper and describe where to find each reference in the code base.

5.D.1 Code from Section 5.1, Introduction
Code from Subsection 5.1.1, A Motivating Example

The `prefixSums` example appears in the Coq source file `rewriter/src/Rewriter/Rewriter/Examples/PrefixSums.v`. Note that we use `dlet` rather than `let` in binding `acc'` so that we can preserve the `let` binder even under ι reduction, which much of

Coq's infrastructure performs eagerly. Because we attempt to isolate the dependency on the axiom of functional extensionality as much as possible, we also in practice require `Proper` instances for each higher-order identifier saying that each constant respects function extensionality. We hope to remove the dependency on function extensionality altogether in the future. Although we glossed over this detail in the body of this paper, we also prove

```
Global Instance: forall A B,
 Proper ((eq ==> eq ==> eq) ==> eq ==> eq ==> eq)
        (@fold_left A B).
```

The `Make` command is exposed in the file `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v` and defined in the OCaml file `rewriter/src/Rewriter/Util/plugins/rewriter_build_plugin.mlg`. Note that one must run `make` to create this latter file; it is copied over from a version-specific file at the beginning of the build.

The `do_again`, `eval_rect`, and `ident.eagerly` constants are defined at the bottom of module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`.

Code from Subsection 5.1.2, Concerns of Trusted-Code-Base Size

There is no code mentioned in this section.

Code from Subsection 5.1.3, Our Solution

We claimed that our solution meets five criteria. We briefly justify each criterion with a sentence or a pointer to code:

- We claimed that we **did not grow the trusted base** (excepting the axiom of functional extensionality). In any example file (of which a couple can be found in `rewriter/src/Rewriter/Rewriter/Examples/`), the `Make` command creates a rewriter package. Running `Print Assumptions` on this new constant (often named `rewriter` or `myrew`) should demonstrate a lack of axioms other than functional extensionality. `Print Assumptions` may also be run on the proof that results from using the rewriter.
- We claimed **fast** partial evaluation with reasonable memory use; we assume that the performance graphs stand on their own to support this claim. Note that memory usage can be observed by making the benchmarks while passing `TIMED=1` to `make`.
- We claimed to allow reduction that **mixes** *rules of the definitional equality* with *equalities proven explicitly as theorems*; the "rules of the definitional equality" are, for example, β reduction, and we assert that it should be self-evident that our rewriter supports this.

- We claimed common-subterm **sharing preservation**. This is implemented by supporting the use of the `dlet` notation which is defined in `rewriter/src/Rewriter/Util/LetIn.v` via the `Let_In` constant. We will come back to the infrastructure that supports this.

- We claimed **extraction of standalone partial evaluators**. The extraction is performed in the Coq source file `perf_unsaturated_solinas.v`, in the source file `perf_word_by_word_montgomery.v`, and in the source files `saturated_solinas.v`, `unsaturated_solinas.v`, and `word_by_word_montgomery.v`, all in the directory `fiat-crypto/src/ExtractionOCaml/`. The OCaml code can be extracted and built using the target `make standalone-ocaml` (or `make perf-standalone` for the `perf_` binaries). There may be some issues with building these binaries on Windows as some versions of `ocamlopt` on Windows seem not to support outputting binaries without the `.exe` extension.

The P-384 curve is mentioned. This is the curve with prime modulus $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$, and the benchmarks for this curve can be found in the files matching the glob `fiat-crypto/src/Rewriter/PerfTesting/Specific/generated/p2384m2128m296p232m1__*__word` While the `.log` files are included in the tarball, the `.v` and `.sh` files are automatically generated in the course of running `make perf -k`.

We mention integration with abstract interpretation; the abstract-interpretation pass is implemented in `fiat-crypto/src/AbstractInterpretation/`.

5.D.2 Code from Section 5.2, Trust, Reduction, and Rewriting

The individual rewritings mentioned are implemented via the `Rewrite_*` tactics exported at the top of `rewriter/src/Rewriter/Util/plugins/RewriterBuild.v`. These tactics bottom out in tactics defined at the bottom of `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

Code from Subsection 5.2.1, Our Approach in Nine Steps

We match the nine steps with functions from the source code:

1. The given lemma statements are scraped for which named functions and types the rewriter package will support. This is performed by `rewriter_scrape_data` in the file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` which invokes the tactic named `make_scrape_data` in a submodule in `rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v` on a goal headed by the constant we provide under the name `Pre.ScrapedData.t_with_args` in `rewriter/src/Rewriter/Language/PreCommon.v`.

2. Inductive types enumerating all available primitive types and functions are emitted. This step is performed by `rewriter_emit_inductives` in file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` invoking tactics, like `make_base_elim` in `rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v`, on goals headed by constants from `rewriter/src/Rewriter/Language/IdentifiersBasicLibrary.v`, including `base_elim_with_args` for example, to turn scraped data into eliminators for the inductives. The actual emitting of inductives is performed by code in the file `rewriter/src/Rewriter/Util/plugins/inductive_from_elim.ml`.

3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. This step is performed by `make_rewriter_of_scraped_and_ind` in the source file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml` which invokes `make_rewriter_all` defined in the file `rewriter/src/Rewriter/Rewriter/AllTactics.v` on a goal headed by the provided constant `VerifiedRewriter_with_ind_args` defined in `rewriter/src/Rewriter/Rewriter/ProofsCommon.v`. The definitions emitted can be found by looking at the tactic `Build_Rewriter` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`, the tactics `build_package` in the source file `rewriter/src/Rewriter/Language/IdentifiersBasicGenerate.v` and also in the Coq source file found in `rewriter/src/Rewriter/Language/IdentifiersGenerate.v` (there is a different tactic named `build_package` in each of these files), and the tactic `prove_package_proofs_via` which can be found in the Coq source file `rewriter/src/Rewriter/Language/IdentifiersGenerateProofs.v`.

4. The statements of rewrite rules are reified, and we prove soundness and syntactic-well-formedness lemmas about each of them. This step is performed as part of the previous step, when the tactic `make_rewriter_all` transitively calls `Build_Rewriter` from `rewriter/src/Rewriter/Rewriter/AllTactics.v`. Reification is handled by the tactic `Build_RewriterT` in `rewriter/src/Rewriter/Rewriter/Reify.v`, while soundness and syntactic-well-formedness are handled by the tactics `prove_interp_good` and `prove_good` respectively, both in the source file `rewriter/src/Rewriter/Rewriter/ProofsCommonTactics.v`.

5. The definitions needed to perform reification and rewriting and the lemmas needed to prove correctness are assembled into a single package that can be passed by name to the rewriting tactic. This step is also performed by `make_rewriter_of_scrape` in the source file `rewriter/src/Rewriter/Util/plugins/rewriter_build.ml`.

When we want to rewrite with a rewriter package in a goal, the following steps are performed, with code in the following places:

1. We rearrange the goal into a single logical formula: all free-variable quantification in the proof context is replaced by changing the equality goal into an equality between two functions (taking the free variables as inputs). Note that

it is not actually an equality between two functions but rather an `equiv` between two functions, where `equiv` is a custom relation we define indexed over type codes that is equality up to function extensionality. This step is performed by the tactic `generalize_hyps_for_rewriting` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

2. We reify the side of the goal we want to simplify, using the inductive codes in the specified package. That side of the goal is then replaced with a call to a denotation function on the reified version. This step is performed by the tactic `do_reify_rhs_with` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

3. We use a theorem stating that rewriting preserves denotations of well-formed terms to replace the denotation subterm with the denotation of the rewriter applied to the same reified term. We use Coq's built-in full reduction (`vm_compute`) to reduce the application of the rewriter to the reified term. This step is performed by the tactic `do_rewrite_with` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

4. Finally, we run `cbv` (a standard call-by-value reducer) to simplify away the invocation of the denotation function on the concrete syntax tree from rewriting. This step is performed by the tactic `do_final_cbv` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

These steps are put together in the tactic `Rewrite_for_gen` in `rewriter/src/Rewriter/Rewriter/AllTactics.v`.

Our Approach in More Than Nine Steps

As the nine steps of Subsection 5.2.1 do not exactly match the code, we describe here a more accurate version of what is going on. For ease of readability, we do not clutter this description with references to the code supplement, instead allowing the reader to match up the steps here with the more coarse-grained ones in Subsection 5.2.1 or Section 5.D.2.

In order to allow easy invocation of our rewriter, a great deal of code (about 6500 lines) needed to be written. Some of this code is about reifying rewrite rules into a form that the rewriter can deal with them in. Other code is about proving that the reified rewrite rules preserve interpretation and are well-formed. We wrote some plugin code to automatically generate the inductive type of base-type codes and identifier codes, as well as the two variants of the identifier-code inductive used internally in the rewriter. One interesting bit of code that resulted was a plugin that can emit an inductive declaration given the Church encoding (or eliminator) of the inductive type to be defined. We wrote a great deal of tactic code to prove basic properties about these inductive types, from the fact that one can unify two identifier codes and extract constraints on their type variables from this unification, to the fact that type codes

have decidable equality. Additional plugin code was written to invoke the tactics that construct these definitions and prove these properties, so that we could generate an entire rewriter from a single command, rather than having the user separately invoke multiple commands in sequence.

In order to build the precomputed rewriter, the following actions are performed:

1. The terms and types to be supported by the rewriter package are scraped from the given lemmas.
2. An inductive type of codes for the types is emitted, and then three different versions of inductive codes for the identifiers are emitted (one with type arguments, one with type arguments supporting pattern type variables, and one without any type arguments, to be used internally in pattern-matching compilation).
3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. Definitions cover categories like "Boolean equality on type codes" and "how to extract the pattern type variables from a given identifier code," and lemma categories include "type codes have decidable equality" and "the types being coded for have decidable equality" and "the identifiers all respect function extensionality."
4. The rewrite rules are reified, and we prove interpretation-correctness and well-formedness lemmas about each of them.
5. The definitions needed to perform reification and rewriting and the lemmas needed to prove correctness are assembled into a single package that can be passed by name to the rewriting tactic.
6. The denotation functions for type and identifier codes are marked for early expansion in the kernel via the `Strategy` command; this is necessary for conversion at **`Qed`**-time to perform reasonably on enormous goals.

When we want to rewrite with a rewriter package in a goal, the following steps are performed:

1. We use **`etransitivity`** to allow rewriting separately on the left- and right-hand-sides of an equality. Note that we do not currently support rewriting in non-equality goals, but this is easily worked around using `let v := open_constr:(_) in replace <some term> with v` and then rewriting in the second goal.
2. We revert all hypotheses mentioned in the goal, and change the form of the goal from a universally quantified statement about equality into a statement that two functions are extensionally equal. Note that this step will fail if any hypotheses are functions not known to respect function extensionality via typeclass search.

3. We reify the side of the goal that is not an existential variable using the inductive codes in the specified package; the resulting goal equates the denotation of the newly reified term with the original evar.

4. We use a lemma stating that rewriting preserves denotations of well-formed terms to replace the goal with the rewriter applied to our reified term. We use `vm_compute` to prove the well-formedness side condition reflectively. We use `vm_compute` again to reduce the application of the rewriter to the reified term.

5. Finally, we run `cbv` to unfold the denotation function, and we instantiate the evar with the resulting rewritten term.

There are a couple of steps that contribute to the trusted base. We must trust that the rewriter package we generate from the rewrite rules in fact matches the rewrite rules we want to rewrite with. This involves partially trusting the scraper, the reifier, and the glue code. We must also trust the VM we use for reduction at various points in rewriting. Otherwise, everything is checked by Coq. We do, however, depend on the axiom of function extensionality in one place in the rewriter proof; after spending a couple of hours trying to remove this axiom, we temporarily gave up.

5.D.3 Code from Section 5.3, The Structure of a Rewriter

The expression language e corresponds to the inductive `expr` type defined in module `Compilers.expr` in `rewriter/src/Rewriter/Language/Language.v`.

Code from Subsection 5.3.1, Pattern-Matching Compilation and Evaluation

The pattern -atching compilation step is done by the tactic `CompileRewrites` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`, which just invokes the Gallina definition named `compile_rewrites` with ever-increasing amounts of fuel until it succeeds. (It should never fail for reasons other than insufficient fuel, unless there is a bug in the code.) The workhorse function of this code is `compile_rewrites_step`.

The decision-tree evaluation step is done by the definition `eval_rewrite_rules`, also in the file `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The correctness lemmas are `eval_rewrite_rules_correct` in the file `rewriter/src/Rewriter/Rewriter/InterpProofs.v` and the theorem `wf_eval_rewrite_rules` in `rewriter/src/Rewriter/Rewriter/Wf.v`. Note that the second of these lemmas, not mentioned in the paper, is effectively saying that for two related syntax trees, `eval_rewrite_rules` picks the same rewrite rule for both. (We actually prove a slightly weaker lemma, which is a bit harder to state in English.)

The third step of rewriting with a given rule is performed by the definition `rewrite_with_rule` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The correctness proof is `interp_rewrite_with_r`

in `rewriter/src/Rewriter/Rewriter/InterpProofs.v`. Note that the well-formedness-preservation proof for this definition in inlined into the proof `wf_eval_rewrite_rules` mentioned above.

The inductive description of decision trees is `decision_tree` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`.

The pattern language is defined as the inductive `pattern` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. Note that we have a `Raw` version and a typed version; the pattern-matching compilation and decision-tree evaluation of Aehlig, Haftmann, and Nipkow [1] is an algorithm on untyped patterns and untyped terms. We found that trying to maintain typing constraints led to headaches with dependent types. Therefore when doing the actual decision-tree evaluation, we wrap all of our expressions in the dynamically typed `rawexpr` type and all of our patterns in the dynamically typed `Raw.pattern` type. We also emit separate inductives of identifier codes for each of the `expr`, `pattern`, and `Raw.pattern` type families.

We partially evaluate the partial evaluator defined by `eval_rewrite_rules` in the tactic `make_rewrite_head` in `rewriter/src/Rewriter/Rewriter/Reify.v`.

Code from Subsection 5.3.2, Adding Higher-Order Features

The type NbE_t mentioned in this paper is not actually used in the code; the version we have is described in Subsection 5.4.2 as the definition `value'` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`.

The functions `reify` and `reflect` are defined in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and share names with the functions in the paper. The function `reduce` is named `rewrite_bottomup` in the code, and the closest match to NbE is `rewrite`.

5.D.4 Code from Section 5.4, Scaling Challenges

Code from Subsection 5.4.1, Variable Environments Will Be Large

The inductives `type`, `base_type` (actually the inductive type `base.type.type` in the supplemental code), and `expr`, as well as the definition `Expr`, are all defined in `rewriter/src/Rewriter/Language/Language.v`. The definition `denoteT` is the fixpoint `type.interp` (the fixpoint `interp` in the module `type`) in `rewriter/src/Rewriter/Language/Language.v`. The definition `denoteE` is `expr.interp`, and `DenoteE` is the fixpoint `expr.Interp`.

As mentioned above, `nbeT` does not actually exist as stated but is close to `value'` in `rewriter/src/Rewriter/Rewriter/Rewriter.v`. The functions `reify` and `reflect` are defined in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and share names with the functions in the paper. The actual code is somewhat more complicated than the version presented in the paper, due to needing to deal with converting well-typed-by-construction expressions to dynamically typed expressions for use in decision-tree

evaluation and also due to the need to support early partial evaluation against a concrete decision tree. Thus the version of `reflect` that actually invokes rewriting at base types is a separate definition `assemble_identifier_rewriters`, while `reify` invokes a version of `reflect` (named `reflect`) that does not call rewriting. The function named `reduce` is what we call `rewrite_bottomup` in the code; the name `Rewrite` is shared between this paper and the code. Note that we eventually instantiate the argument `rewrite_head` of `rewrite_bottomup` with a partially evaluated version of the definition named `assemble_identifier_rewriters`. Note also that we use fuel to support `do_again`, and this is used in the definition `repeat_rewrite` that calls `rewrite_bottomup`.

The correctness theorems are `InterpRewrite` in `rewriter/src/Rewriter/Rewriter/InterpProofs.v` and `Wf_Rewrite` in `rewriter/src/Rewriter/Rewriter/Wf.v`.

Packages containing rewriters and their correctness theorems are in the record `VerifiedRewriter` in `rewriter/src/Rewriter/Rewriter/ProofsCommon.v`; a package of this type is then passed to the tactic `Rewrite_for_gen` from `rewriter/src/Rewriter/Rewriter/AllTactics.v` to perform the actual rewriting. The correspondence of the code to the various steps in rewriting is described in the second list of Section 5.D.2.

Code from Subsection 5.4.2, Subterm Sharing is Crucial

To run the P-256 example in the copy of Fiat Cryptography attached as a code supplement, after building the library, run the code

```
Require Import Crypto.Rewriter.PerfTesting.Core.
Require Import Crypto.Util.Option.

Import WordByWordMontgomery.
Import Core.RuntimeDefinitions.

Definition p : params
  := Eval compute in invert_Some
       (of_string "2^256-2^224+2^192+2^96-1" 64).

Goal True.
  (* Successful run: *)
  Time let v := (eval cbv
    -[Let_In
      runtime_nth_default
      runtime_add
      runtime_sub
      runtime_mul
      runtime_opp
      runtime_div
      runtime_modulo
```

```
      RT_Z.add_get_carry_full
      RT_Z.add_with_get_carry_full
      RT_Z.mul_split]
    in (GallinaDefOf p)) in
    idtac.
  (* Unsuccessful OOM run: *)
  Time let v := (eval cbv
    -[(*Let_In*)
      runtime_nth_default
      runtime_add
      runtime_sub
      runtime_mul
      runtime_opp
      runtime_div
      runtime_modulo
      RT_Z.add_get_carry_full
      RT_Z.add_with_get_carry_full
      RT_Z.mul_split]
    in (GallinaDefOf p)) in
    idtac.
Abort.
```

The `UnderLets` monad is defined in the file `rewriter/src/Rewriter/Language/UnderLets.v`.

The definitions `nbeT'`, `nbeT`, and `nbeT_with_lets` are in `rewriter/src/Rewriter/Rewriter/Rewriter.v` and are named `value'`, `value`, and `value_with_lets`, respectively.

Code from Subsection 5.4.3, Rules Need Side Conditions

The "variant of pattern variable that only matches constants" is actually special support for the reification of `ident.literal` (defined in the module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`) threaded throughout the rewriter. The apostrophe notation `'` is also introduced in the module `RewriteRuleNotations` in `rewriter/src/Rewriter/Language/Pre.v`. The support for side conditions is handled by permitting rewrite-rule-replacement expressions to return `option expr` instead of `expr`, allowing the function `expr_to_pattern_and_replacement` in the file `rewriter/src/Rewriter/Rewriter/Reify.v` to fold the side conditions into a choice of whether to return `Some` or `None`.

Code from Subsection 5.4.4, Side Conditions Need Abstract Interpretation

The abstract-interpretation pass is defined in `fiat-crypto/src/AbstractInterpretation/`, and the rewrite rules handling abstract-interpretation results are the Gallina definitions `arith_with_casts_rewrite_rulesT`, in addition to `strip_literal_casts_rewrite_rulesT`,

in addition to `fancy_with_casts_rewrite_rulesT`, and finally in addition to `mul_split_rewrite_rul` all defined in `fiat-crypto/src/Rewriter/Rules.v`.

The `clip` function is the definition `ident.cast` in `fiat-crypto/src/Language/PreExtra.v`.

5.D.5 Code from Section 5.5, Evaluation

Code from Subsection 5.5.1, Microbenchmarks

This code is found in the files in `rewriter/src/Rewriter/Rewriter/Examples/`. We ran the microbenchmarks using the code in `rewriter/src/Rewriter/Rewriter/Examples/PerfTesting/Harness.v` together with some `Makefile` cleverness. The file names correspond to the section titles in Section 5.A.

Code from Subsection 5.5.2, Macrobenchmark: Fiat Cryptography

The rewrite rules are defined in `fiat-crypto/src/Rewriter/Rules.v` and proven in the file `fiat-crypto/src/Rewriter/RulesProofs.v`. They are turned into rewriters in the various files in `fiat-crypto/src/Rewriter/Passes/`. The shared inductives and definitions are defined in the Coq source files `fiat-crypto/src/Language/IdentifiersBasicGENERATED.v`, `fiat-crypto/src/Language/IdentifiersGENERATED.v`, and `fiat-crypto/src/Language/IdentifiersGENERATEDProofs.v`. Note that we invoke the subtactics of the `Make` command manually to increase parallelism in the build and to allow a shared language across multiple rewriter packages.

Chapter 6

Extended Description of the Rewriter

[**TODO:** this chapter] [**TODO:** mention frowned-upon Perl scripts previously in BoringSSL(?) OpenSSL?; (ask Andres for reference?)] Perl scripts were complicated, a number of steps removed from actual running code, hard to maintain and verify. [**TODO:** Refer back to representation changes (good abstraction barriers / equivalences) being important in fiat-crypto, and being cheap only because we have a rewriter]

6.1 Rewriter

6.1.1 Introduction

- The goal of the rewriter is to take an abstract syntax tree and perform reduction or rewriting.
- There are three things that happen in rewriting: beta reduction, let-lifting, and replacement of rewrite patterns with their substitutions
- Beta reduction is replacing `(λ x. F) y` with `F[x ⇒ y]`. We do this with a normalization-by-evaluation strategy.
- Let-lifting involves replacing `f (let x := y in z)` with `let x := y in f x`. Note that for higher-order functions, we push lets under lambads, rather than lifting them; we replace `f (let x := y in (λ z. w))` with `f (λ z. let x := y in w)`. This is done for the convenience of not having to track the let-binding-structure at every level of arrow.
- Replacing rewriting patterns with substitutions involves, for example, replacing `x + 0` with `x`.
- There's actually a fourth thing, which happens during let-lifting: some let binders get inlined: In particular, any let-bound value which is a combination of variables, literals, and the identifiers `nil`, `cons`, `pair`, `fst`, `snd`, `Z.opp`, `Z.cast`,

and `Z.cast2` gets inlined. If the let-bound variable contains any lambdas, lets, or applications of identifiers other than the above, then it is not inlined.

6.1.2 Beta-Reduction and Let-Lifting

- We use the following data-type:

```
Fixpoint value' (with_lets : bool) (t : type)
ă := match t with
ă ă ă| type.base t
ă ă ă ă=> if with_lets then UnderLets (expr t) else expr t
ă ă ă| type.arrow s d
ă ă ă ă=> value' false s -> value' true d
ă ă ăend.
Definition value := value' false.
Definition value_with_lets := value' true.
```

- Here are some examples:
  - `value Z := UnderLets (expr Z)`
  - `value (Z -> Z) := expr Z -> UnderLets (expr Z)`
  - `value (Z -> Z -> Z) := expr Z -> expr Z -> UnderLets (expr Z)`
  - `value ((Z -> Z) -> Z) := (expr Z -> UnderLets (expr Z)) -> UnderLets (expr Z)`
- By converting expressions to values and using normalization-by-evaluation, we get beta reduction in the standard way.
- We use a couple of splicing combinators to perform let-lifting:
  - `Fixpoint splice {A B} (x : UnderLets A) (e : A -> UnderLets B) : UnderLets B`
  - `Fixpoint splice_list {A B} (ls : list (UnderLets A)) (e : list A -> UnderLets B) : UnderLets B`
  - `Fixpoint splice_under_lets_with_value {T t} (x : UnderLets T) : (T -> value_with_lets t) -> value_with_lets t`
  - `Definition splice_value_with_lets {t t'} : value_with_lets t -> (value t -> value_with_lets t') -> value_with_lets t'`
- There's one additional building block, which is responsible for deciding which lets to inline:
  - `Fixpoint reify_and_let_binds_base_cps {t : base.type} : expr t -> forall T, (expr t -> UnderLets T) -> UnderLets T`
- As is typical for NBE, we make use of a reify-reflect pair of functions: `coq Fixpoint reify {with_lets} {t} : value' with_lets t -> expr t  with reflect {with_lets} {t} : expr t -> value' with_lets t`

- The NBE part of the rewriter, responsible for beta reduction and let-lifting, is now expressible:

```
Local Notation "e <---- e' ; f" := (splice_value_with_lets e' (fun e => f%under_
Local Notation "e <----- e' ; f" := (splice_under_lets_with_value e' (fun e => f

Fixpoint rewrite_bottomup {t} (e : @expr value t) : value_with_lets t
ă := match e with
ă ă ă| expr.Ident t idc
ă ă ă ă=> rewrite_head _ idc
ă ă ă| expr.App s d f x => let f : value s -> value_with_lets d := @rewrite_bott
ă ă ă| expr.LetIn A B x f => x <---- @rewrite_bottomup A x;
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăxv <----- reify_and_let_binds_cps x _ UnderLets.B
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă@rewrite_bottomup B (f (reflect xv))
ă ă ă| expr.Var t v => Base_value v
ă ă ă| expr.Abs s d f => fun x : value s => @rewrite_bottomup d (f x)
ă ă ăend%under_lets.
```

6.1.3 Rewriting

There are three parts and one additional detail to rewriting:

- Pattern matching compilation

- Decision tree evaluation

- Rewriting with a particular rewrite rule

- Rewriting again in the output of a rewrite rule ### Overview

- Rewrite rules are patterns (things like `?? + #?` meaning "any variable added to any literal") paired with dependently typed replacement values indexed over the pattern. The replacement value takes in types for each type variable, `value`s for each variable (`??`), and interpreted values for each literal wildcard. Additionally, any identifier that takes extra parameters will result in the parameters being passed into the rewrite rule. The return type for replacement values is an option UnderLets expr of the correct type.

- A list of rewrite rules is compiled into (a) a decision tree, and (b) a rewriter that functions by evaluating that decision tree. ### The small extra detail: Rewriting again in the output of a rewrite rule

- We tie the entire rewriter together with a fueled repeat_rewrite; the fuel is set to the length of the list of rewrite rules. This means that as long as the intended

rewrite sequences form a DAG, then the rewriter will find all occurrences. “‘coq Notation nbe := (@rewrite_bottomup (fun t idc => reflect (expr.Ident idc))).

Fixpoint repeat_rewrite (rewrite_head : forall (do_again : forall t : base.type, @expr value (type.base t) -> UnderLets (@expr var (type.base t))) t (idc : ident t), value_with_lets t) (fuel : nat) {t} e : value_with_lets t := @rewrite_bottomup (rewrite_head (fun t' e' => match fuel with | Datatypes.O => nbe e' | Datatypes.S fuel' => @repeat_rewrite rewrite_head fuel' (type.base t') e' end%under_lets)) t e. - This feature is used to rewrite again with the literal-list append rule (appending two lists of cons cells results in a single list of cons cells) in the output of the ‘flat_map‘ rule (‘flat_map‘ on a literal list of cons cells maps the function over the list and joins the resulting lists with ‘List.app‘). ### Pattern matching compilation - This part of the rewriter does not need to be verified, because the rewriter-compiler is proven correct independent of the decision tree used. Note that we could avoid this stage all together, and simply try each rewrite rule in sequence. We include this for efficiency. TODO: perf comparison of this method. - We follow *Compiling Pattern Matching to Good Decision Trees* by Luc Maranget (http://moscova.i which describes compilation of pattern matches in OCaml to a decision tree that eliminates needless repeated work (for example, decomposing an expression into ‘x + y + z‘ only once even if two different rules match on that pattern). - We do *not* bother implementing the optimizations that they describe for finding minimal decision trees. TODO: Mention something about future work? Perf testing? - The type of decision trees - A ‘decision_tree‘ describes how to match a vector (or list) of patterns against a vector of expressions. The cases of a ‘decision_tree‘ are: - ‘TryLeaf k onfailure‘: Try the kth 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‘, remove the first element of the vector 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 its applied to; otherwise, don't modify the vectors, and use the ‘default‘ decision tree. - ‘Swap i cont‘: Swap the first element of the vector with the ith element, and keep going with ‘cont‘ - The inductive type:coq Inductive decision_tree := | TryLeaf (k : nat) (onfailure : decision_tree) | Failure | Switch (icases : list (raw_pident * decision_tree)) (app_case : option decision_tree) (default : decision_tree) | Swap (i : nat) (cont : decision_tree). - Raw identifiers - Note that the type of ‘icases‘, the list of identifier cases in the ‘Switch‘ constructor above, maps what we call a ‘raw_pident‘ ("p" for "pattern") to a decision tree. The rewriter is parameterized over a type of ‘raw_pident‘s, which is instantiated with a python-generated inductive type which names all of

the identifiers we care about, except without any arguments. We call them "raw" because they are not type-indexed. - An example where this is important: We want to be able to express a decision tree for the pattern ‘fst (x, y)‘. This involves an application of the identifier ‘fst‘ to a pair. We want to be able to talk about "‘fst‘, of any type" in the decision tree, without needing to list out all of the possible type arguments to ‘fst‘. - Swap vs indices - One design decision we copy from *Compiling Pattern Matching to Good Decision Trees* is to have a ‘Swap‘ case. We could instead augment each ‘Switch‘ with the index in the vector being examined. If we did this, we'd need to talk about splicing a new list into the middle of an existing list, which is harder than talking about swapping two indices of a list. - Note that swapping is *significantly* more painful over typed patterns and terms than over untyped ones. If we index our vectors over a list of types, then we need to swap the types, and later swap them back (when reconstructing the term for evaluation), and then we need to unswap the terms in a way that has unswap (swap ls) on the term level *judgmentally* indexed on the type level over the same index-list as ls. This is painful, and is an example of pain caused by picking the wrong abstraction, in a way that causes exponential blow-up with each extra layer of dependency added. - The type of patterns - Patterns describe the LHS of rewrite rules, or the LHS of cases of a match statement. We index patterns over their type:coq Inductive pattern.base.type := var (p : positive) | type_base (t : Compilers.base.type.base) | prod (A B : type) | list (A : type). - The type of a pattern is either an arrow or a pattern.base.type, and a pattern.base.type is either a positive-indexed type-variable (written '1, '2, ...), or a product, a list, or a standard base.type (with no type-variables) - A pattern is either a wildcard, an identifier, or an application of patterns. Note that our rewriter only handles fully applied patterns, i.e., only things of type ‘pattern (type.base t)‘, not things of type ‘pattern t‘. (This is not actually true. The rewriter can kind-of handle non-fully-applied patterns, but the Gallina won't reduce in the right places, so we restrict ourselves to fully-applied patterns.)coq Inductive pattern {ident : type -> Type} : type -> Type := | Wildcard (t : type) : pattern t | Ident {t} (idc : ident t) : pattern t | App {s d} (f : pattern (s -> d)) (x : pattern s) : pattern d. - Pattern matching *compilation* to decision trees actually uses a more raw version of patterns, which come from these patterns:coq Module Raw. Inductive pattern {ident : Type} := | Wildcard | Ident (idc : ident) | App (f x : pattern). End Raw. - This is because the pattern matching compilation algorithm is morally done over untyped patterns and terms. - The definitions - TODO: How much detail should I include about intermediate things? - Pattern matching compilation at the top-level, takes in a list of patterns, and spits out a decision tree. Note that each ‘TryLeaf‘ node in the decision tree has an index ‘k‘, which denotes the index in this initial list of patterns of the chosen rewrite rule. - The workhorse of pattern matching compilation is ‘Fixpoint compile_rewrites' (fuel : nat) (pattern_matrix : list (nat * list rawpattern)) : option decision_tree‘. This takes the

list rows of the matrix of patterns, each one containing a list (vector in the original source paper) of patterns to match against, and the original index of the rewrite rule that this list of patterns came from. Note that all of these lists are in fact the same length, but we do not track this invariant anywhere, because it would add additional overhead for little-to-no gain. - The ‘compile_rewrites'‘ procedure operates as follows: - If we are out of fuel, then we fail (return ‘None‘) - If the ‘pattern_matrix‘ is empty, we indicate ‘Failure‘ to match - If the first row is made up entirely of wildcards, we indicate to ‘TryLeaf‘ with the rewrite rule corresponding to the first row, and then to continue on with the decision tree corresponding to the rest of the rows. - If the first element of the first row is a wildcard, we ‘Swap‘ the first element with the index ‘i‘ of the first non-wildcard pattern in the first row. We then swap the first element with the ‘i‘th element in every row of the matrix, and continue on with the result of compiling that matrix. - If the first element of the first row is not a wildcard, we issue a ‘Switch‘. We first split the pattern matrix by finding the first row where the first element in that row is a wildcard, and aggregating that row and all rows after it into the ‘default_pattern_matrix‘. We partition the rows before that row into the ones where the first element is an application node and the ones where the first element is an identifier node. The application nodes get split into the pattern for the function, and the pattern for the argument, and these two are prepended to the row. We group the rows that start with identifier patterns into groups according to the pattern identifier at the beginning of the row, and then take the tail of each of these rows. We then compile all of these decision trees to make up the Switch case. - In code, this looks like:coq Definition compile_rewrites_step (compile_rewrites : list (nat * list rawpattern) -> option decision_tree) (pattern_matrix : list (nat * list rawpattern)) : option decision_tree := match pattern_matrix with | nil => Some Failure | (n1, p1) :: ps => match get_index_of_first_non_wildcard p1 with | None (* p1 is all wildcards *) => (*onfailure <- compile_rewrites ps; Some (TryLeaf n1 onfailure)) | Some Datatypes.O => let ‘(pattern_matrix, default_pattern_matrix) := split_at_first_pattern_wildcard pattern_matrix in default_case <- compile_rewrites default_pattern_matrix; app_case <- (if contains_pattern_app pattern_matrix then option_map Some (compile_rewrites (Option.List.map filter_pattern_app pattern_matrix)) else Some None); let pidcs := get_unique_pattern_ident pattern_matrix in let icases := Option.List.map (fun pidc => option_map (pair pidc) (compile_rewrites (Option.List.map (filter_pattern_pident pidc) pattern_matrix))) pidcs in Some (Switch icases app_case default_case) | Some i => let pattern_matrix' := List.map (fun ‘(n, ps) => (n, match swap_list 0 i ps with | Some ps' => ps' | None => nil (should be impossible *) end)) pat-

tern_matrix in d <- compile_rewrites pattern_matrix'; Some (Swap i d) end end%option. - We wrap ‘compile_rewrites''‘ in a definition ‘compile_rewrites‘ which extracts the well-typed patterns from a list of rewrite rules, associates them to indices, and strips the typing information off of the patterns to create raw (untyped) patterns. ### Decision Tree Evaluation - The next step in rewriting is to evaluate the decision tree to construct Gallina procedure that takes in an unknown (at rewrite-rule-compile-time AST and performs the rewrite. This is broken up into two steps. The first step is to create the ‘match‘ structure that exposes all of the relevant information in the AST, and picks which rewrite rules to try in which order, and glues together failure and success of rewriting. The second step is to actually try to rewrite with a given rule, under the assumption that enough structure has been exposed. - Because we have multiple phases of compilation, we need to track which information we have (and therefore can perform reduction on) when we know only the patterns but not the AST being rewritten in, and which reductions have to wait until we know the AST. The way we track this information is by creating a wrapper type for ASTs. Note that the wrapper type is not indexed over type codes, because pattern matching compilation naturally operates over untyped terms, and adjusting it to work when indexed over a vector of types is painful. - The wrapper type, and revealing structure - Because different rewrite rules require different amounts of structure, we want to feed in only as much structure as is required for a given rewrite rule. For example, if we have one rewrite rule that is looking at ‘(? + ?) + ¿, and another that is looking at ‘? + 0‘, we want to feed in the argument to the top-level ‘+‘ into the second rewrite rule, not a reassembled version of all of the different things an expression might be after checking for ‘+‘-like structure of the first argument. If we did not do this, every rewrite rule replacement pattern would end up being as complicated as the deepest rewrite rule being considered, and we expect this would incur performance overhead. TODO: Perf testing? - Because we want our rewrite-rule-compilation to happen by reduction in Coq, we define many operations in CPS-style, so that we can carefully manage the exact judgmental structures of the discriminees of ‘match‘ statements. - An ‘Inductive rawexpr : Type‘ is one of the following things: coq Inductive raw-expr : 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 : value t). - ‘rIdent known t idc t' alt‘ - an identifier ‘idc : ident t‘, whose unrevealed structure is ‘alt : expr t'‘. The boolean ‘known‘ indicates if the identifier is known to be simple enough that we can fully reduce matching on its type arguments during rewrite-rule-compilation-time. For example, if we know an identifier to be ‘Z.add‘ (perhaps because we have matched on it already), we can reduce equality tests against the type. However, if an identifier is ‘nil T‘, we are not guaranteed to know the type of the list judgmentally, and so we do not want to reduce type-equality tests against the list. Note that type-equality tests and type-transports are introduced

as the least-evil thing we could find to cross the broken abstraction barrier between the untyped terms of pattern matching compilation, and the typed PHOASTs that we are operating on. - ‘rApp f x t alt‘ is the application of the ‘rawexpr‘ ‘f‘ to the ‘rawexpr‘ ‘x‘, whose unrevealed structure is ‘alt : expr t‘. - ‘rExpr t e‘ is a not-as-yet revealed expression ‘e : expr t‘. - ‘rValue t e‘ is an unrevealed value ‘e : value t‘. Such NBE-style values may contain thunked computation, such as deferred rewriting opportunities. This is essential for fully evaluating rewriting in expressions such as ‘map (fun x => x + x + 0) ls‘, where you want to rewrite away the ‘map‘ (when ‘ls‘ is a concrete list of cons cells), the ‘+ 0‘ (always), and the ‘x + x‘ whenever ‘x‘ is a literal (which you do not know until you have distributed the function over the list). Allowing thunked computation in the ASTs allows us to do all of this rewriting in a single pass. - Revealing structure: ‘Definition reveal_rawexpr_cps (e : rawexpr) : ~> rawexpr‘ - For the sake of proofs, we actually define a slightly more general version of revealing structure, which allows us to specify whether or not we have already matched against the putative identifier at the top-level of the ‘rawexpr‘. - The code:coq Definition reveal_rawexpr_cps_gen (assume_known : option bool) (e : rawexpr) : ~> rawexpr := fun T k => match e, assume_known with | rExpr _e as r, / *rValue (type.base*) e as r, *=> match e with | expr.Ident t idc => k (rIdent (match assume_known with Some known => known /* => false end) idc e) | expr.App s d f x => k (rApp (rExpr f) (rExpr x) e) | _=> k r end | rIdent _t idc t' alt, Some known => k (rIdent known idc alt) | e', _=> k e' end. - To reveal a ‘rawexpr‘, CPS-style, we first match on the ‘rawexpr‘. - If it is an ‘rExpr‘, or a ‘rValue‘ at a base type (and thus just an expression), we then match on the resulting expression. - If it is an identifier or an application node, we encode that, and then invoke the continuation - Otherwise, we invoke the continuation with the existing ‘rExpr‘ or ‘rValue‘, because there was no more accessible structure to reveal; we do not allow matching on lambdas syntactically. - If it is an identifier and we are hard-coding the ‘known‘ status about if matches on the type of the identifier can be reduced, then we re-assemble the ‘rIdent‘ node with the new ‘known‘ status and invoke the continuation. - Otherwise, we just invoke the continuation on the reassembled ‘rawexpr‘. - Correctness conditions - There are a couple of properties that must hold of ‘reveal_rawexpr_cps‘. - The first is a ‘cps_id‘ rule, which says that applying ‘reveal_rawexpr_cps‘ to any continuation is the same as invoking the continuation with ‘reveal_rawexpr_cps‘ applied to the identity continuation. - The next rule talks about a property we call ‘rawexpr_types_ok‘ To say that this property holds is to say that the ‘rawexper‘s are well-typed in accordance with the unrevealed expressions stored in the tree. - Code:coq Fixpoint rawexpr_types_ok (r : @rawexpr var) (t : type) : Prop := match r with | rExpr t' / *rValue t'* => t' = t | rIdent _t1 _t2 *=> t1 = t / t2 = t | rApp f x t' alt => t' = t / match alt*

with | *expr.App s d* => *rawexpr_types_ok f (type.arrow s d)* / *rawexpr_types_ok x s* | => False end end. - We must then have that a ‘rawexpr‘ is ‘rawexpr_types_ok‘ if and only if the result of revealing one layer of structure via ‘reveal_rawexpr_cps‘ is ‘rawexpr_types_ok‘. - We also define a relation ‘rawexpr_equiv‘ which says that two ‘rawexpr‘s represent the same expression, up to different amounts of revealed structure. - Code:coq Local Notation e1 === e2 := (existT expr _e1 = existT expr _e2) : type_scope.

```
Fixpoint rawexpr_equiv_expr {t0} (e1 : expr t0) (r2 : rawexpr) {struct r2} : Pr
ă := match r2 with
ă ă ă| rIdent _ t idc t' alt
ă ă ă ă=> alt === e1 /\ expr.Ident idc === e1
ă ă ă| rApp f x t alt
ă ă ă ă=> alt === e1
ă ă ă ă ă /\ match e1 with
ă ă ă ă ă ă ă| expr.App _ _ f' x'
ă ă ă ă ă ă ă ă=> rawexpr_equiv_expr f' f /\ rawexpr_equiv_expr x' x
ă ă ă ă ă ă ă| _ => False
ă ă ă ă ă ă ăend
ă ă ă| rExpr t e
ă ă ă| rValue (type.base t) e
ă ă ă ă=> e === e1
ă ă ă| rValue t e => False
ă ă ăend.

Fixpoint rawexpr_equiv (r1 r2 : rawexpr) : Prop
ă := match r1, r2 with
ă ă ă| rExpr t e, r
ă ă ă| r, rExpr t e
ă ă ă| rValue (type.base t) e, r
ă ă ă| r, rValue (type.base t) e
ă ă ă ă=> rawexpr_equiv_expr e r
ă ă ă| rValue t1 e1, rValue t2 e2
ă ă ă ă=> existT _ t1 e1 = existT _ t2 e2
ă ă ă| rIdent _ t1 idc1 t'1 alt1, rIdent _ t2 idc2 t'2 alt2
ă ă ă ă=> alt1 === alt2
ă ă ă ă ă /\ (existT ident _ idc1 = existT ident _ idc2)
ă ă ă| rApp f1 x1 t1 alt1, rApp f2 x2 t2 alt2
ă ă ă ă=> alt1 === alt2
ă ă ă ă ă /\ rawexpr_equiv f1 f2
ă ă ă ă ă /\ rawexpr_equiv x1 x2
ă ă ă| rValue _ _, _
ă ă ă| rIdent _ _ _ _ _, _
ă ă ă| rApp _ _ _ _, _
```

```
      ă ă ă ă=> False
      ă ă ăend.
      ```
    - The relation `rawexpr_equiv` is effectively the recursive closure of `reveal_ra
  <!---
  - Finally, we define a notation of `wf` for `rawexpr`s called `wf_rawexpr`, and we
related, then the results of calling `reveal_rawexpr` on both of them are `wf_rawexpr
related.
    - The definition of `wf_rawexpr` is:
      ```coq
      Inductive wf_rawexpr : list { t : type & var1 t * var2 t }%type -> forall {t},
      | Wf_rIdent {t} G known (idc : ident t) : wf_rawexpr G (rIdent known idc (expr.
      | Wf_rApp {s d} G
      ă ă ă ă ă f1 (f1e : @expr var1 (s -> d)) x1 (x1e : @expr var1 s)
      ă ă ă ă ă f2 (f2e : @expr var2 (s -> d)) x2 (x2e : @expr var2 s)
      ă : wf_rawexpr G f1 f1e f2 f2e
      ă ă -> wf_rawexpr G x1 x1e x2 x2e
      ă ă -> wf_rawexpr G
      ă ă ă ă ă ă ă ă ă (rApp f1 x1 (expr.App f1e x1e)) (expr.App f1e x1e)
      ă ă ă ă ă ă ă ă ă (rApp f2 x2 (expr.App f2e x2e)) (expr.App f2e x2e)
      | Wf_rExpr {t} G (e1 e2 : expr t)
      ă : expr.wf G e1 e2 -> wf_rawexpr G (rExpr e1) e1 (rExpr e2) e2
      | Wf_rValue {t} G (v1 v2 : value t)
      ă : wf_value G v1 v2
      ă ă -> wf_rawexpr G (rValue v1) (reify v1) (rValue v2) (reify v2).
      ``` --->
```

- Evaluating the decision tree
- Decision tree evaluation is performed by a single monolithic recursive function:

```
Fixpoint eval_decision_tree {T} (ctx : list rawexpr) (d : decision_tree)
(cont : nat -> list rawexpr -> option T) {struct d} : option T

Fixpoint eval_decision_tree {T} (ctx : list rawexpr) (d : decision_tree) (cont :
ă := match d with
ă ă ă| TryLeaf k onfailure
ă ă ă ă=> let res := cont k ctx in
ă ă ă ă ă match onfailure with
ă ă ă ă ă | Failure => res
ă ă ă ă ă | _ => res ;; (@eval_decision_tree T ctx onfailure cont)
ă ă ă ă ă end
ă ă ă| Failure => None
ă ă ă| Switch icases app_case default_case
ă ă ă ă=> match ctx with
ă ă ă ă ă | nil => None
```
```

```
ă ă ă ă ă ă | ctx0 :: ctx'
ă ă ă ă ă ă ă => let res
ă ă ă ă ă ă ă ă ă ă ă:= reveal_rawexpr_cps
ă ă ă ă ă ă ă ă ă ă ă ă ă ctx0 _
ă ă ă ă ă ă ă ă ă ă ă ă ă (fun ctx0'
ă ă ă ă ă ă ă ă ă ă ă ă ă ă=> match ctx0' with
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă | rIdent known t idc t' alt
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă => fold_right
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă(fun '(pidc, icase) rest
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă => let res
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă:= if known
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă then
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (args <- invert_bind_args _ idc
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă@eval_decision_tree
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăT ctx' icase
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă(fun k ctx''
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă => cont k (rIdent
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă(raw_pident_i
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă(raw_pident_t
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă else
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă @eval_decision_tree
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă T ctx' icase
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (fun k ctx''
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă=> option_bind'
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (invert_bind_args_unknow
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (fun args
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă=> cont k (rIdent
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (raw_piden
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (raw_piden
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăin
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ămatch rest with
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă| None => Some res
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă| Some rest => Some (res ;; rest)
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăend)
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăNone
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăicases;;;
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăNone
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă | rApp f x t alt
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă => match app_case with
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă| Some app_case
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă=> @eval_decision_tree
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă T (f :: x :: ctx') app_case
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă (fun k ctx''
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă=> match ctx'' with
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă | f' :: x' :: ctx'''
```

```
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă => cont k (rApp f' x' alt :: ctx''
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă | _ => None
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă end)
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă| None => None
ă ă ă ă ă ă ă ă ă ă ă ă ă ă ă ăend
ă ă ă ă ă ă ă ă ă ă ă ă ă | rExpr t e
ă ă ă ă ă ă ă ă ă ă ă ă ă | rValue t e
ă ă ă ă ă ă ă ă ă ă ă ă ă => None
ă ă ă ă ă ă ă ă ă ă ă ă ă end) in
ă ă ă ă ă ă ă ămatch default_case with
ă ă ă ă ă ă ă ă| Failure => res
ă ă ă ă ă ă ă ă| _ => res ;; (@eval_decision_tree T ctx default_case cont)
ă ă ă ă ă ă ă ăend
ă ă ă ă ă end
ă ă ă| Swap i d'
ă ă ă ă=> match swap_list 0 i ctx with
ă ă ă ă ă | Some ctx'
ă ă ă ă ă ă => @eval_decision_tree
ă ă ă ă ă ă ă ă ăT ctx' d'
ă ă ă ă ă ă ă ă ă(fun k ctx''
ă ă ă ă ă ă ă ă ă => match swap_list 0 i ctx'' with
ă ă ă ă ă ă ă ă ă ă| Some ctx''' => cont k ctx'''
ă ă ă ă ă ă ă ă ă ă| None => None
ă ă ă ă ă ă ă ă ă ă ăend)
ă ă ă ă ă | None => None
ă ă ă ă ă end
ă ă ăend%option.
```

- This function takes a list (vector in the original source paper) `ctx` of `rawexpr`s to match against, a `decision_tree d` to evaluate, and a "continuation" `cont` which tries a given rewrite rule based on the index of the rewrite rule (in the original list of rewrite rules) and the list of `rawexpr`s to feed into the rewrite rule. This continuation is threaded through the decision tree evaluation procedure, and each time we split up the structure of the pattern matrix (virtually, in the decision tree) and the list of `rawexpr`s (concretely, as an argument), we add a bit to the continuation that "undoes" the splitting. In the end, the top-level "continuation" gets fed a singleton list containing a `rawexpr` with enough structure for the rewrite rule it is trying. TODO: Figure out how to be more clear here; I anticipate this is unclear, and I'm not sure how to fix it except by randomly throwing more sentences in to try to explain it in various different ways.
- Correctness conditions
  - There are two correctness conditions for `eval_decision_tree`: one for `wf`, and the other for `Interp`.

  - The interpretation-correctness rule says that either `eval_decision_tree` returns `None`, or it returns the result of calling the continuation on some index and with some list of `rawexpr`s which is element-wise `rawexpr_equiv` to the input list. In other words, `eval_decision_tree` does nothing more than revealing some structure, and then eventually calling the continuation (which is to be filled in with "rewrite with this rule") on the revealed `rawexpr`.
  - The `wf` correctness condition is significantly more verbose to state, but it boils down to saying that as long as the continuation behaves "the same" (for some parameterized notion of "the same") on `wf_rawexpr`-related `rawexpr`s, then `eval_decision_tree` will similarly behave "the same" on element-wise `wf_rawexpr`-related lists of `rawexpr`s.

- Definition

  - The `eval_decision_tree` procedure proceeds recursively on the structure of the `decision_tree`.
  - If the decision tree is a `TryLeaf k onfailure`, then we try the continuation on the `k`th rewrite rule. If it fails (by returning `None`), we proceed with `onfailure`. In the code, there is a bit of extra care taken to simplify the resulting output code when `onfailure` is just `Failure`, i.e., no remaining matches to try. This probably does not impact performance, but it makes the output of the rewrite-rule-compilation procedure slightly easier to read and debug.
  - If the decision tree is `Failure` return `None`, i.e., we did not succeed in rewriting.
  - If the decision tree starts with `Swap i d'`, we swap the first element with the `i`th element in the list of `rawexpr`s we are matching on (to mirror the swapping in the pattern matrix that happened when compiling the decision tree), and then continue on evaluating `d'`. We augment the continuation by reversing the swap in the list of `rawexpr`s passed in at the beginning, to cancel out the swap we did "on the outside" before continuing with evaluation of the decision tree. Note that here we are jumping through some extra hoops to get the right reduction behavior at rewrite-rule-compilation time.
  - If none of the above match, the decision tree must begin with `Switch icases app_case default_case`. In this case, we start by revealing the structure of the first element of the list of `rawexpr`s. (If there is no first element, which should never happen, we indicate failure by returning `None`.) In the continuation of `reveal_rawexpr_cps`, we check which sort of `rawexpr` we have. Note that in all cases of failure below, we try again with the `default_case`.
    * If we have no accessible structure (`rExpr` or `rValue`), then we fail with `None`.
    * If we have an application, we take the two arguments of `rApp`, the

function and its argument, and prepend them to the tail of the list of `rawexpr`s. We then continue evaluation with `app_case` (if it is non-`None`), and, in the continuation, we reassemble the `rawexpr` by taking the first two elements of the passed-in-list, and combining them in a new `rApp` node. We keep the unrevealed structure in `alt` the same as it was in the `rApp` that we started with.

  * If we have an identifier, then we look at `icases`. We fold through the list of identifiers, looking to see if any of them match the identifier that we have. If the identifier is `known`, then we perform the match before evaluating the corresponding decision tree, because we want to avoid revealing useless structure. If the identifier is not `known`, then first we reveal all of the necessary structure for this identifier by continuing decision tree evaluation, and only then in the continuation do we try to match against the identifier.
  * In both cases, we drop the first element of the list of `rawexpr`s being matched against when continuing evaluation, to mirror the dropping that happens in compilation of the decision tree. We then prepend a re-built identifier onto the head of the list inside the continuation. We have a table of which pattern identifiers have `known` types, and we have conversion functions between pattern identifiers and PHOAST identifiers (autogenerated in Python) which allow us to extract the arguments from the PHOAST identifier and re-insert them into the pattern identifier. For example, this will extract the list type from `nil` (because the pattern-identifier version does not specify what the type of the list is—we will say more about this in the next section), or the literal value from the `Literal` identifier, and allow recreating the fully-typed identifier from the pattern-identifier with these arguments. This allows more rewriter-compile-time reduction opportunities which allows us to deduplicate matches against the same identifier. Note that we have two different constants that we use for binding these arguments; they do the same thing, but one is reduced away completely at rewrite-rule-compilation time, and the other is preserved. ### Rewriting with a particular rewrite rule

- The final big piece of the rewriter is to rewrite with a particular rule, given a `rawexpr` with enough revealed structure, a `pattern` against which we bind arguments, and a replacement rule which is a function indexed over the `pattern`. We saw above the inductive type of patterns. Let us now discuss the structure of the replacement rules.

- Replacement rule types

- The data for a replacement rule is indexed over a pattern-type `t` and a `p : pattern t`. It has three options, in addition to the actual replacement rule:

```
Record rewrite_rule_data {t} {p : pattern t} :=
```

```
ă { rew_should_do_again : bool;
ă ă rew_with_opt : bool;
ă ă rew_under_lets : bool;
ă ă rew_replacement : @with_unif_rewrite_ruleTP_gen value t p rew_should_do_agai
```

  - `rew_should_do_again` determines whether or not to rewrite again in the output of this rewrite rule. For example, the rewrite rule for `flat_map` on a concrete list of cons cells maps the function over the list, and joins the resulting list of lists with append. We want to rewrite again with the rule for `List.app` in the output of this replacement.
  - `rew_with_opt` determines whether or not the rewrite rule might fail. For example, rewrite rules like `x + 0 ~> x` are encoded by talking about the pattern of a wildcard added to a literal, and say that the rewrite only succeeds if the literal is 0. Additionally, as another example, all rewrite rules involving casts fail if bounds on the input do not line up; in the pattern `Z.cast @ ((Z.cast @ ??) + (Z.cast @ ??))` the cast node in front of an addition must be loose enough to hold the sum of the ranges taken from the two cast nodes in front of each of the wildcards.
  - `rew_under_lets` determines whether or not the replacement rule returns an explicit `UnderLets` structure. This can be used for let-binding a part of the replacement value.

- The rewrite rule replacement itself is a function. It takes in first all type variables which are mentioned in the pattern, and then, in an in-order traversal of the pattern syntax tree, the non-type arguments for each identifier (e.g., interpreted values of literals, ranges of cast nodes) and a `value (pattern.type.subst_default t evm)` for each wildcard of type `t` (that is, we plug in the known type variables into the pattern-type, and use `unit` for any unknown type variables). It may return a thing in the `option` and/or `UnderLets` monads, depending on `rew_with_opt` and `rew_under_lets`. Underneath these possible monads, it returns an `expr` of the correct type (we substitute the type variables we take in into the type of the pattern), whose `var` type is either `@value var` (if `rew_should_do_again`) or `var` (if not `rew_should_do_again`). The different `var` types are primarily to make the type of the output of the rewrite rule line up with the expression type that is fed into the rewriter as a whole. We have a number of definitions that describe this in a dependently typed mess:

  - We aggregate the type variables into a `PositiveSet.t` with `Fixpoint pattern.base.collect_vars (t : base.type) : PositiveSet.t` and `Fixpoint pattern.type.collect_vars (t : type) : PositiveSet.t`:
    ```
    coq   Module base.     Fixpoint collect_vars (t : type) : PositiveSet.t
    := match t with          | type.var p => PositiveSet.add p PositiveSet.empty
    | type.type_base t => PositiveSet.empty          | type.prod A
    B => PositiveSet.union (collect_vars A) (collect_vars B)          |
    type.list A => collect_vars A          end.   End base.   Module
    type.       Fixpoint collect_vars (t : type) : PositiveSet.t         :=
    ```

```
match t with | type.base t => base.collect_vars t | type.arrow s d => PositiveSet.union (collect_vars s) (collect_vars d) end. End type.
```

– We quantify over type variables for each of the numbers in the `PositiveSet.t` and aggregate the bound types into a `PositiveMap.t` with `pattern.type.forall_vars`. Note that we use the possibly ill-chosen abbreviation `EvarMap` for `PositiveMap.t Compilers.base.type`. “‘coq Local Notation forall_vars_body K LS EVM0 := (fold_right (fun i k evm => forall t : Compilers.base.type, k (PositiveMap.add i t evm)) K LS EVM0).

Definition forall_vars (p : PositiveSet.t) (k : EvarMap -> Type) := forall_vars_body k (List.rev (PositiveSet.elements p)) (PositiveMap.empty _). `- We take in the context variable ‘pident_arg_types : forall t, pident t -> list Type‘ which describes the arguments bound for a given pattern identifier. - We then quantify over identifier arguments and wildcard values with ‘with_unification_resultT‘:`coq Local Notation type_of_list_cps := (fold_right (fun a K => a -> K)).

Fixpoint with_unification_resultT' {var} {t} (p : pattern t) (evm : EvarMap) (K : Type) : Type := match p return Type with | pattern.Wildcard t => var (pattern.type.subst_default t evm) -> K | pattern.Ident t idc => type_of_list_cps K (pident_arg_types t idc) | pattern.App s d f x => @with_unification_resultT‘var _f evm (@with_unification_resultT' var _x evm K) end%type.

Definition with_unification_resultT {var t} (p : pattern t) (K : type -> Type) : Type := pattern.type.forall_vars (@pattern.collect_vars _t p) (fun evm => @with_unification_resultT‘var t p evm (K (pattern.type.subst_default t evm))). `- Finally, we can define the type of rewrite replacement rules:`coq Local Notation deep_rewrite_ruleTP_gen' should_do_again with_opt under_lets t := (match (@expr.expr base.type ident (if should_do_again then value else var) t) with | x0 => match (if under_lets then UnderLets x0 else x0) with | x1 => if with_opt then option x1 else x1 end end).

Definition deep_rewrite_ruleTP_gen (should_do_again : bool) (with_opt : bool) (under_lets : bool) t := deep_rewrite_ruleTP_gen' should_do_again with_opt under_lets t.

Definition with_unif_rewrite_ruleTP_gen {var t} (p : pattern t) (should_do_again : bool) (with_opt : bool) (under_lets : bool) := @with_unification_resultT var t p (fun t => deep_rewrite_ruleTP_gen' should_do_again with_opt under_lets t). `Whence we have`coq rew_replacement : @with_unif_rewrite_ruleTP_gen value t p rew_should_do_again rew_with_opt rew_under_lets “‘

- There are two steps to rewriting with a rule, both conceptually simple but in practice complicated by dependent types. We must unify a pattern with an expression, gathering binding data for the replacement rule as we go; and we must

apply the replacement rule to the binding data (which is non-trivial because the rewrite rules are expressed as curried dependently-typed towers indexed over the rewrite rule pattern). In order to state the correctness conditions for gathering binding data, we must first talk about applying replacement rules to binding data.

- Applying binding data
- The general strategy for applying binding data is to define an uncurried package (sigma type, or dependent pair) holding all of the arguments, and to define an application function that applies the replacement rule (at various stages of construction) to the binding data package.
- The uncurried package types
  - To turn a list of Types into a Type, we define `Local Notation type_of_list := (fold_right (fun a b => prod a b) unit).`
  - The type `unification_resultT'` describes the binding data for a pattern, given a map of pattern type variables to types: `coq Fixpoint unification_resultT' {var} {t} (p : pattern t) (evm : EvarMap) : Type := match p return Type with | pattern.Wildcard t => var (pattern.type.subst_default t evm) | pattern.Ident t idc => type_of_list (pident_arg_types t idc) | pattern.App s d f x => @unification_resultT' var _ f evm * @unification_resultT var _ x evm end%type.`
  - A `unification_resultT` packages up the type variable replacement map with the bound values: `coq Definition unification_resultT {var t} (p : pattern t) : Type := { evm : EvarMap & @unification_resultT' var t p evm }.`
- The application functions
  - The definition `app_type_of_list` applies a CPS-type `type_of_list_cps` function to uncurried arguments: `coq Definition app_type_of_list {K} {ls : list Type} (f : type_of_list_cps K ls) (args : type_of_list ls) : K := list_rect (fun ls => type_of_list_cps K ls -> type_of_list ls -> K) (fun v _ => v) (fun T Ts rec f x => rec (f (fst x)) (snd x)) ls f args.`
  - Given two different maps of type variables (another instance of abstraction-barrier-breaking), we can apply a `with_unification_resultT'` to a `unification_resultT` by inserting casts in the appropriate places: `coq (** TODO: Maybe have a fancier version of this that doesn't actually need to insert casts, by doing a fixpoint on the list of elements / the evar map *) Fixpoint app_transport_with_unification_resultT'_cps {var t p evm1 evm2 K} {struct p} : @with_unification_resultT' var t p evm1 K -> @unification_resultT' var t p evm2 -> forall`

```
T, (K -> option T) -> option T     := fun f x T k        => match
p return with_unification_resultT' p evm1 K -> unification_resultT'
p evm2 -> option T with           | pattern.Wildcard t             =>
fun f x                => (tr <- type.try_make_transport_cps base.try_make_t
var _ _;                        (tr <- tr;                          k
(f (tr x)))%option)%cps         | pattern.Ident t idc => fun f x
=> k (app_type_of_list f x)        | pattern.App s d f x          =>
fun F (xy : unification_resultT' f _ * unification_resultT' x _)
=> @app_transport_with_unification_resultT'_cps                    _
_ f _ _ _ F (fst xy) T                   (fun F'                    =>
@app_transport_with_unification_resultT'_cps                         _
_ x _ _ _ F' (snd xy) T                          (fun x' => k x'))
end%option f x.
```

– We can apply a `forall_vars` tower over the type variables in a pattern to a particular mapping of type variables to types, with a headache of dependently typed code: “‘coq Fixpoint app_forall_vars_gen {k : EvarMap -> Type} (evm : EvarMap) (ls : list PositiveMap.key) : forall evm0, forall_vars_body k ls evm0 -> option (k (fold_right (fun i k evm' => k (match PositiveMap.find i evm with Some v => PositiveMap.add i v evm' | None => evm' end)) (fun evm => evm) ls evm0)) := match ls return forall evm0, forall_vars_body k ls evm0 -> option (k (fold_right (fun i k evm' => k (match PositiveMap.find i evm with Some v => PositiveMap.add i v evm' | None => evm' end)) (fun evm => evm) ls evm0)) with | nil => fun evm0 val => Some val | cons x xs => match PositiveMap.find x evm as xt return (forall evm0, (forall t, fold_right _k xs (PositiveMap.add x t evm0)) -> option (k (fold_right __xs match xt with | Some v => PositiveMap.add x v evm0 | None => evm0 end))) with | Some v => fun evm0 val => @app_forall_vars_gen k evm xs _(val v) | None => fun evm0 val => None end end.

Definition app_forall_vars {p : PositiveSet.t} {k : EvarMap -> Type} (f : forall_vars p k) (evm : EvarMap) : option (k (fold_right (fun i k evm' => k (match PositiveMap.find i evm with Some v => PositiveMap.add i v evm' | None => evm' end)) (fun evm => evm) (List.rev (PositiveSet.elements p)) (PositiveMap.empty))) := *@app_forall_vars_gen k evm (List.rev (PositiveSet.elements p)) (PositiveMap.empty)*

f. - Finally, we can apply a ‘with_unification_resultT‘ to a ‘unification_resultT‘ package in the obvious way, inserting casts as

needed:coq Definition app_with_unification_resultT_cps {var t p K} : @with_unification_resultT var t p K -> @unification_resultT var t p -> forall T, ({ evm' : _& K (pattern.type.subst_default t evm') } -> option T) -> option T := fun f x T k => (f' <- pattern.type.app_forall_vars f (projT1 x); app_transport_with_unification_resultT'*cp f' (projT2 x)* (fun fx => k (existT __fx)))%option. "'

- Unifying patterns with expressions
- First, we unify the types, in continuation-passing-style, returning an optional `PositiveMap.t` from type variable indices to types.
  - This is actually done in two steps, so that rewrite-rule-compilation can reduce away all occurrences of patterns. First, we check that the expression has the right structure, and extract all of the relevant types both from the pattern and from the expression. Then we connect the types with `type.arrow` (used simply for convenience, so we don't have to unify lists of types, only individual types), and we unify the two resulting types, extracting a `PositiveMap.t` describing the assignments resulting from the unification problem.
  - We first define a few helper definitions that should be self-explanatory:
    * The function `type_of_rawexpr` gets the type of a `rawexpr`: `coq Definition type_of_rawexpr (e : rawexpr) : type := match e with | rIdent _ t idc t' alt => t' | rApp f x t alt => t | rExpr t e => t | rValue t e => t end.`
    * The functions `pattern.base.relax` and `pattern.type.relax` take a PHOAST type and turn it into a pattern type, which just happens to have no pattern type variables. `Module base. Fixpoint relax (t : Compilers.base.type) : type := match t with | Compilers.base.type.type_base t => type.type_base t | Compilers.base.type.prod A B => type.prod (relax A) (relax B) | Compilers.base.type.list A => type.list (relax A) end. End base. Module type. Fixpoint relax (t : type.type Compilers.base.type) : type := match t with | type.base t => type.base (base.relax t) | type.arrow s d => type.arrow (relax s) (relax d) end. End type.`
  - The function responsible for checking the structure of patterns and extracting the types to be unified is `preunify_types {t} (e : rawexpr) (p : pattern t) : option (option (ptype * type))`. It will return `None` if the structure does not match, `Some None` if the type of an identifier of known type in the `rawexpr` does not match the type of the identifier in the pattern (which is guaranteed to always be known, and thus this comparison is safe to perform at rewriter-rule-compilation time), and will

return `Some (Some (t1, t2))` if the structures match, where `t1` and `t2` are the types to be unified.

```
Fixpoint preunify_types {t} (e : rawexpr) (p : pattern t) {struct p}
ă : option (option (ptype * type))
ă := match p, e with
ă ă ă| pattern.Wildcard t, _
ă ă ă ă=> Some (Some (t, type_of_rawexpr e))
ă ă ă| pattern.Ident pt pidc, rIdent known t idc _ _
ă ă ă ă=> if andb known (type.type_beq _ pattern.base.type.type_beq pt (patt
ă ă ă ă ă then Some None
ă ă ă ă ă else Some (Some (pt, t))
ă ă ă| pattern.App s d pf px, rApp f x _ _
ă ă ă ă=> (resa <- @preunify_types _ f pf;
ă ă ă ă ă ă ăresb <- @preunify_types _ x px;
ă ă ă ă ă ă ăSome match resa, resb with
ă ă ă ă ă ă ă ă ă | None, None => None
ă ă ă ă ă ă ă ă ă | None, Some t
ă ă ă ă ă ă ă ă ă | Some t, None => Some t
ă ă ă ă ă ă ă ă ă | Some (a, a'), Some (b, b')
ă ă ă ă ă ă ă ă ă ă => Some (type.arrow a b, type.arrow a' b')
ă ă ă ă ă ă ă ă ă end)
ă ă ă| pattern.Ident _ _, _
ă ă ă| pattern.App _ _ _ _, _
ă ă ă ă=> None
ă ă ăend%option.
```

– We have two correctness conditions on `preunify_types`.
  * The `wf` correctness condition says that if two `rawexpr`s are `wf_rawexpr`-related, then the result of pre-unifying one of them with a pattern `p` is the same as the result of pre-unifying the other with the same pattern `p`.
  * Second, for interpretation-correctness, we define a recursive proposition encoding the well-matching of patterns with `rawexpr`s under a given map of pattern type variables to types: `coq Fixpoint types_match_with (evm : EvarMap) {t} (e : rawexpr) (p : pattern t) {struct p} : Prop := match p, e with | pattern.Wildcard t, e => pattern.type.subst t evm = Some (type_of_rawexpr e) | pattern.Ident t idc, rIdent known t' _ _ _ => pattern.type.subst t evm = Some t' | pattern.App s d f x, rApp f' x' _ _ => @types_match_with evm _ f' f /\ @types_match_with evm _ x' x | pattern.Ident _ _, _ | pattern.App _ _ _ _, _ => False end.`
  * Then we prove that for any map `evm` of pattern type variables to types, if `preunify_types re p` returns `Some (Some (pt, t'))`, and the result of substituting into `pt` the pattern type variables in the

given map is `t'`, then `types_match_with evm re p` holds. Symbolically, this is `coq Lemma preunify_types_to_match_with {t re p evm} : match @preunify_types ident var pident t re p with | Some None => True | Some (Some (pt, t')) => pattern.type.subst pt evm = Some t' | None => False end -> types_match_with evm re p.`

– In a possibly-gratuitous use of dependent typing to ensure that no uses of `PositiveMap.t` remain after rewrite-rule-compilation, we define a dependently typed data structure indexed over the pattern type which holds the mapping of each pattern type variable to a corresponding type. This step cannot be fully reduced at rewrite-rule-compilation time, because we may not know enough type structure in the `rawexpr`. We then collect these variables into a `PositiveMap.t`; this step *can* be fully reduced at rewrite-rule-compilation time, because the pattern always has a well-defined type structure, and so we know *which* type variables will have assignments in the `PositiveMap.t`, even if we don't necessarily know concretely (at rewrite-rule-compilation time) *what* those type variables will be assigned to. We must also add a final check that substituting into the pattern type according the resulting `PositiveMap.t` actually does give the expected type; we do not want `'1 -> '1` and `nat -> bool` to unify. We could check at each addition to the `PositiveMap.t` that we are not replacing one type with a different type. However, the proofs are much simpler if we simply do a wholesale check at the very end. We eventually perform this check in `unify_types`.

  * We thus define the dependently typed structures: "‘ Module base. Fixpoint var_types_of (t : type) : Set := match t with | type.var _=> Compilers.base.type | type.type_base _=> unit | type.prod A B => var_types_of A * var_types_of B | type.list A => var_types_of A end%type.

Fixpoint add_var_types_cps {t : type} (v : var_types_of t) (evm : EvarMap) : ~> EvarMap := fun T k => match t return var_types_of t -> T with | type.var p => fun t => k (PositiveMap.add p t evm) | type.prod A B => fun '(a, b) => @add_var_types_cps A a evm _(fun evm => @add_var_types_cps B b evm _k) | type.list A => fun t => @add_var_types_cps A t evm _k | type.type_base _=> fun _=> k evm end v. End base. Module type. Fixpoint var_types_of (t : type) : Set := match t with | type.base t => base.var_types_of t | type.arrow s d => var_types_of s * var_types_of d end%type.

Fixpoint add_var_types_cps {t : type} (v : var_types_of t) (evm : EvarMap) : ~> EvarMap := fun T k => match t return var_types_of t -> T with | type.base t => fun v => @base.add_var_types_cps t v evm _k | type.arrow A B => fun ‘(a, b) => @add_var_types_cps A a evm _(fun evm => @add_var_types_cps B b

evm _k) end v. End type. - We can now write down the unifier that produces ‘var_types_of‘ from a unification problem; it is straightforward: Module base. Fixpoint unify_extracted (ptype : type) (etype : Compilers.base.type) : option (var_types_of ptype) := match ptype, etype return option (var_types_of ptype) with | type.var p, _=> Some etype | type.type_base t, Compilers.base.type.type_base t' => if base.type.base_beq t t' then Some tt else None | type.prod A B, Compilers.base.type.prod A' B' => a <- unify_extracted A A'; b <- unify_extracted B B'; Some (a, b) | type.list A, Compilers.base.type.list A' => unify_extracted A A' | type.type_base , | type.prod _, | type.list , => None end%option. End base. Module type. Fixpoint unify_extracted (ptype : type) (etype : type.type Compilers.base.type) : option (var_types_of ptype) := match ptype, etype return option (var_types_of ptype) with | type.base t, type.base t' => base.unify_extracted t t' | type.arrow A B, type.arrow A' B' => a <- unify_extracted A A'; b <- unify_extracted B B'; Some (a, b) | type.base , | type.arrow _, => None end%option. End type. “‘

– Finally, we can write down the type-unifier for patterns and `rawexpr`s. Note that the final equality check, described and motivated above, is performed in this function.

```
(* for unfolding help *)
Definition option_type_type_beq := option_beq (type.type_beq _ base.type.typ

Definition unify_types {t} (e : rawexpr) (p : pattern t) : ~> option EvarMap
ă := fun T k
ă ă ă=> match preunify_types e p with
ă ă ă ă | Some (Some (pt, t))
ă ă ă ă ă => match pattern.type.unify_extracted pt t with
ă ă ă ă ă ă ă| Some vars
ă ă ă ă ă ă ă ă=> pattern.type.add_var_types_cps
ă ă ă ă ă ă ă ă ă ă vars (PositiveMap.empty _) _
ă ă ă ă ă ă ă ă ă ă (fun evm
ă ă ă ă ă ă ă ă ă ă ă=> (* there might be multiple type variables which map
ă ă ă ă ă ă ă ă ă ă ă ăif option_type_type_beq (pattern.type.subst pt evm) (
ă ă ă ă ă ă ă ă ă ă ă ăthen k (Some evm)
ă ă ă ă ă ă ă ă ă ă ă ăelse k None)
ă ă ă ă ă ă ă| None => k None
ă ă ă ă ă ă ăend
ă ă ă ă | Some None
ă ă ă ă ă => k (Some (PositiveMap.empty _))
ă ă ă ă | None => k None
ă ă ă ă end.
```

- Now that we have unified the types and gotten a `PositiveMap.t` of pattern type variables to types, we are ready to unify the patterns, and extract the identifier arguments and `value`s from the `rawexpr`. Because it would be entirely too painful to track at the type-level that the type unifier guarantees a match on structure and types, we instead sprinkle type transports all over this definition to get the types to line up. Here we pay the price of an imperfect abstraction barrier (that we have types lying around, and we rely in some places on types lining up, but do not track everywhere that types line up). Most of the other complications in this function come from (a) working in continuation-passing-style (for getting the right reduction behavior) or (b) tracking the differences between things we can reduce at rewrite-rule-compilation time, and things we can't.

  - We first describe some helper definitions and context variables.
  - The context variable `pident_arg_types : forall t, pident t -> list Type` describes for each pattern identifier what arguments should be bound for it.
  - The context variables `(pident_unify pident_unify_unknown : forall t t' (idc : pident t) (idc' : ident t'), option (type_of_list (pident_arg_ty t idc)))` are the to-be-unfolded and not-to-be-unfolded versions of unifying a pattern identifier with a PHOAST identifier.
  - We can convert a `rawexpr` into a `value` or an `expr`:

    ```
    Definition expr_of_rawexpr (e : rawexpr) : expr (type_of_rawexpr e)
    ă := match e with
    ă ă ă| rIdent _ t idc t' alt => alt
    ă ă ă| rApp f x t alt => alt
    ă ă ă| rExpr t e => e
    ă ă ă| rValue t e => reify e
    ă ă ăend.
    Definition value_of_rawexpr (e : rawexpr) : value (type_of_rawexpr e)
    ă := Eval cbv `expr_of_rawexpr` in
    ă ă ă match e with
    ă ă ă | rValue t e => e
    ă ă ă | e => reflect (expr_of_rawexpr e)
    ă ă ă end.
    ```

  - We can now write down the pattern-expression-unifier: "'coq Definition option_bind' {A B} := @Option.bind A B. (* for help with unfolding *)

  Fixpoint unify_pattern' {t} (e : rawexpr) (p : pattern t) (evm : EvarMap) {struct p} : forall T, (unification_resultT' p evm -> option T) -> option T := match p, e return forall T, (unification_resultT' p evm -> option T) -> option T with | pattern.Wildcard t', *=> fun T k => (tro <- type.try_make_transport_cps (@base.try_make_transport_cps) value*

(type_of_rawexpr e) (pattern.type.subst_default t' evm); (tr <- tro; <- pattern.type.subst t' evm; (* ensure that we did not fall into the default case *) (k (tr (value_of_rawexpr e))))%option)%cps | pattern.Ident t pidc, rIdent known _idc _ => *fun T k => (if known then Option.bind (pident_unify* _pidc idc) else option_bind' (pident_unify_unknown __pidc idc)) k | pattern.App s d pf px, rApp f x _ => *fun T k => @unify_pattern' _f pf evm T (fun fv =>* *@unify_pattern'* x px evm T (fun xv => k (fv, xv))) | pattern.Ident _, | pattern.App ___, => fun _k => None end%option. `- We have three correctness conditions on 'unify_pattern'': - It must be the case that if we invoke 'unify_pattern'' with any continuation, the result is the same as invoking it with the continuation 'Some', binding the result in the option monad, and then invoking the continuation on the bound value. - There is the 'wf' correctness condition, which says that if two 'rawexpr's are 'wf_rawexpr'-related, then invoking 'unify_pattern'' with the continuation 'Some' either results in 'None' on both of them, or it results in two 'wf_unification_r related results. We define 'wf_unification_resultT'' as`coq Fixpoint wf_value' {with_lets : bool} G {t : type} : value'1 with_lets t -> value'2 with_lets t -> Prop := match t, with_lets with | type.base t, true => UnderLets.wf (fun G' => expr.wf G') G | type.base t, false => expr.wf G | type.arrow s d, _ => fun f1 f2 => (forall seg G' v1 v2, G' = (seg ++ G)%list -> @wf_value'false seg s v1 v2 -> @wf_value' true G' d (f1 v1) (f2 v2)) end.

```
Definition wf_value G {t} : value1 t -> value2 t -> Prop := @wf_value' false G
Definition wf_value_with_lets G {t} : value_with_lets1 t -> value_with_lets2 t

Fixpoint related_unification_resultT' {var1 var2} (R : forall t, var1 t -> var
ă : @unification_resultT' var1 t p evm -> @unification_resultT' var2 t p evm -
ă := match p in pattern.pattern t return @unification_resultT' var1 t p evm ->
ă ă ă| pattern.Wildcard t => R _
ă ă ă| pattern.Ident t idc => eq
ă ă ă| pattern.App s d f x
ă ă ă ă=> fun (v1 : unification_resultT' f evm * unification_resultT' x evm)
ă ă ă ă ă ă ă (v2 : unification_resultT' f evm * unification_resultT' x evm)
ă ă ă ă ă => @related_unification_resultT' _ _ R _ _ _ (fst v1) (fst v2)
ă ă ă ă ă ă ă/\ @related_unification_resultT' _ _ R _ _ _ (snd v1) (snd v2)
ă ă ăend.

Definition wf_unification_resultT' (G : list {t1 : type & (var1 t1 * var2 t1)%
ă : @unification_resultT' value t p evm -> @unification_resultT' value t p evm
ă := @related_unification_resultT' _ _ (fun _ => wf_value G) t p evm.
```

```
- The interp-correctness condition is (a bit more than) a bit of a mouthful, and
  - It is a bit hard to say what makes an expression interp-related to an interp
related to a interpreted value if and only if the interpretation of the expressi
    ```coq
    Section with_interp.
    ă Context {base_type : Type}
    ă ă ă ă ă {ident : type base_type -> Type}
    ă ă ă ă ă {interp_base_type : base_type -> Type}
    ă ă ă ă ă (interp_ident : forall t, ident t -> type.interp interp_base_type

    ă Fixpoint interp_related_gen
    ă ă ă ă ă ă{var : type base_type -> Type}
    ă ă ă ă ă ă(R : forall t, var t -> type.interp interp_base_type t -> Prop)
    ă ă ă ă ă ă{t} (e : @expr base_type ident var t)
    ă ă : type.interp interp_base_type t -> Prop
    ă ă := match e in expr t return type.interp interp_base_type t -> Prop with
    ă ă ă ă| expr.Var t v1 => R t v1
    ă ă ă ă| expr.App s d f x
    ă ă ă ă ă=> fun v2
    ă ă ă ă ă ă => exists fv xv,
    ă ă ă ă ă ă ă ă @interp_related_gen var R _ f fv
    ă ă ă ă ă ă ă ă /\ @interp_related_gen var R _ x xv
    ă ă ă ă ă ă ă ă /\ fv xv = v2
    ă ă ă ă| expr.Ident t idc
    ă ă ă ă ă=> fun v2 => interp_ident _ idc == v2
    ă ă ă ă| expr.Abs s d f1
    ă ă ă ă ă=> fun f2
    ă ă ă ă ă ă => forall x1 x2,
    ă ă ă ă ă ă ă ă R _ x1 x2
    ă ă ă ă ă ă ă ă -> @interp_related_gen var R d (f1 x1) (f2 x2)
    ă ă ă ă| expr.LetIn s d x f (* combine the App rule with the Abs rule *)
    ă ă ă ă ă=> fun v2
    ă ă ă ă ă ă => exists fv xv,
    ă ă ă ă ă ă ă ă @interp_related_gen var R _ x xv
    ă ă ă ă ă ă ă ă /\ (forall x1 x2,
    ă ă ă ă ă ă ă ă ă ă ă ăR _ x1 x2
    ă ă ă ă ă ă ă ă ă ă ă ă-> @interp_related_gen var R d (f x1) (fv x2))
    ă ă ă ă ă ă ă ă /\ fv xv = v2
    ă ă ă ăend.

    ă Definition interp_related {t} (e : @expr base_type ident (type.interp inte
    ă ă := @interp_related_gen (type.interp interp_base_type) (@type.eqv) t e.
    End with_interp.
    ```
  - A term in the `UnderLets` monad is `UnderLets.interp_related` to an interpre
```

let-binders with `expr`-let-binders) results in an expression that is `expr.inte
  - A `value` is `value_interp_related` to an interpreted value `v` whenever it
    ```coq
    Fixpoint value_interp_related {t with_lets} : @value' var with_lets t -> typ
    ă := match t, with_lets with
    ă ă ă| type.base _, true => UnderLets_interp_related
    ă ă ă| type.base _, false => expr_interp_related
    ă ă ă| type.arrow s d, _
    ă ă ă ă=> fun (f1 : @value' _ _ s -> @value' _ _ d) (f2 : type.interp _ s ->
    ă ă ă ă ă => forall x1 x2,
    ă ă ă ă ă ă ă @value_interp_related s _ x1 x2
    ă ă ă ă ă ă ă -> @value_interp_related d _ (f1 x1) (f2 x2)
    ă ă ăend.
    ```
  - A `rawexpr` is `rawexpr_interp_related` to an interpreted value `v` if both
    ```coq
    Fixpoint rawexpr_interp_related (r1 : rawexpr) : type.interp base.interp (ty
    ă := match r1 return type.interp base.interp (type_of_rawexpr r1) -> Prop wi
    ă ă ă| rExpr _ e1
    ă ă ă| rValue (type.base _) e1
    ă ă ă ă=> expr_interp_related e1
    ă ă ă| rValue t1 v1
    ă ă ă ă=> value_interp_related v1
    ă ă ă| rIdent _ t1 idc1 t'1 alt1
    ă ă ă ă=> fun v2
    ă ă ă ă ă ă => expr.interp ident_interp alt1 == v2
    ă ă ă ă ă ă ă/\ existT expr t1 (expr.Ident idc1) = existT expr t'1 alt1
    ă ă ă| rApp f1 x1 t1 alt1
    ă ă ă ă=> match alt1 in expr.expr t return type.interp base.interp t -> Prop
    ă ă ă ă ă | expr.App s d af ax
    ă ă ă ă ă ă => fun v2
    ă ă ă ă ă ă ă ă=> exists fv xv (pff : type.arrow s d = type_of_rawexpr f1) (
    ă ă ă ă ă ă ă ă ă ă@expr_interp_related _ af fv
    ă ă ă ă ă ă ă ă ă ă/\ @expr_interp_related _ ax xv
    ă ă ă ă ă ă ă ă ă ă/\ @rawexpr_interp_related f1 (rew pff in fv)
    ă ă ă ă ă ă ă ă ă ă/\ @rawexpr_interp_related x1 (rew pfx in xv)
    ă ă ă ă ă ă ă ă ă ă/\ fv xv = v2
    ă ă ă ă ă | _ => fun _ => False
    ă ă ă ă ă end
    ă ă ăend.
    ```
  - We can say when a `unification_resultT'` returning an `expr` whose `var` typ
related to a `unification_resultT'` returning an `expr` whose `var` type is `typ
obvious way:
    ```coq
    ```

```
Local Notation var := (type.interp base.interp) (only parsing).

Definition unification_resultT'_interp_related {t p evm}
ă : @unification_resultT' (@value var) t p evm -> @unification_resultT' var
ă := related_unification_resultT' (fun t => value_interp_related).
```

- We say that a `rawexpr`'s types are ok if the revealed and unrevealed struct
```coq
Fixpoint rawexpr_types_ok (r : @rawexpr var) (t : type) : Prop
ă := match r with
ă ă ă| rExpr t' _
ă ă ă| rValue t' _
ă ă ă ă=> t' = t
ă ă ă| rIdent _ t1 _ t2 _
ă ă ă ă=> t1 = t /\ t2 = t
ă ă ă| rApp f x t' alt
ă ă ă ă=> t' = t
ă ă ă ă ă /\ match alt with
ă ă ă ă ă ă ă| expr.App s d _ _
ă ă ă ă ă ă ă ă=> rawexpr_types_ok f (type.arrow s d)
ă ă ă ă ă ă ă ă ă /\ rawexpr_types_ok x s
ă ă ă ă ă ă ă| _ => False
ă ă ă ă ă ă ăend
ă ă ăend.
```

- We can define a transformation that takes in a `PositiveMap.t` of pattern ty
creates a new `PositiveMap.t` in accordance with the `PositiveSet.t`. This is r
```coq
Local Notation mk_new_evm0 evm ls
ă := (fold_right
ă ă ă ă (fun i k evm'
ă ă ă ă ă=> k match PositiveMap.find i evm with
ă ă ă ă ă ă ă | Some v => PositiveMap.add i v evm'
ă ă ă ă ă ă ă | None => evm'
ă ă ă ă ă ă ă end) (fun evm' => evm')
ă ă ă ă (List.rev ls)) (only parsing).

Local Notation mk_new_evm' evm ps
ă := (mk_new_evm0
ă ă ă ă evm
ă ă ă ă (PositiveSet.elements ps)) (only parsing).

Local Notation mk_new_evm evm ps
ă := (mk_new_evm' evm ps (PositiveMap.empty _)) (only parsing).
```

```
  - Given a proof of `@types_match_with evm t re p` that the types of `re : rawe
  - The final and perhaps most important auxiliary component is the notation of
    - This definition itself needs a few auxiliary definitions and context varia
    - We have a context variable `(pident_to_typed : forall t (idc : pident t) (
instantiations of type variables of pattern identifiers be valid; this means tha
    - We define `lam_type_of_list` to convert between the `cps` and non-
cps versions of type lists:
      ```coq
      Local Notation type_of_list
      ă := (fold_right (fun a b => prod a b) unit).
      Local Notation type_of_list_cps
      ă := (fold_right (fun a K => a -> K)).

      Definition lam_type_of_list {ls K} : (type_of_list ls -> K) -> type_of_lis
      ă := list_rect
      ă ă ă ă(fun ls => (type_of_list ls -> K) -> type_of_list_cps K ls)
      ă ă ă ă(fun f => f tt)
      ă ă ă ă(fun T Ts rec k t => rec (fun ts => k (t, ts)))
      ă ă ă ăls.
      ```
    - We may now define the default interpretation:
      ```coq
      Fixpoint pattern_default_interp' {K t} (p : pattern t) evm {struct p} : (v
      ă := match p in pattern.pattern t return (var (pattern.type.subst_default
      ă ă ă| pattern.Wildcard t => fun k v => k v
      ă ă ă| pattern.Ident t idc
      ă ă ă ă=> fun k
      ă ă ă ă ă => lam_type_of_list (fun args => k (ident_interp _(pident_to_typ
      ă ă ă| pattern.App s d f x
      ă ă ă ă=> fun k
      ă ă ă ă ă => @pattern_default_interp'
      ă ă ă ă ă ă ă ă_ _ f evm
      ă ă ă ă ă ă ă ă(fun ef
      ă ă ă ă ă ă ă ă => @pattern_default_interp'
      ă ă ă ă ă ă ă ă ă ă ă_ _ x evm
      ă ă ă ă ă ă ă ă ă ă ă(fun ex
      ă ă ă ă ă ă ă ă ă ă ă => k (ef ex)))
      ă ă ăend.
      ```
    - To define the unprimed version, which also accounts for the type variables
      ```coq
      Fixpoint lam_forall_vars_gen {k : EvarMap -> Type}
      ă ă ă ă ă(f : forall evm, k evm)
      ă ă ă ă ă(ls : list PositiveMap.key)
```
```

```
ă : forall evm0, forall_vars_body k ls evm0
ă := match ls return forall evm0, forall_vars_body k ls evm0 with
ă ă ă| nil => f
ă ă ă| cons x xs => fun evm t => @lam_forall_vars_gen k f xs _
ă ă ăend.

Definition lam_forall_vars {p : PositiveSet.t} {k : EvarMap -> Type}
ă ă ă ă ă ă(f : forall evm, k evm)
ă : forall_vars p k
ă := @lam_forall_vars_gen k f _ _.
```

- Now we can define the default interpretation as a `with_unification_result

```coq
Definition pattern_default_interp {t} (p : pattern t)
ă : @with_unification_resultT var t p var
ă := pattern.type.lam_forall_vars
ă ă ă ă(fun evm
ă ă ă ă => pattern_default_interp' p evm id).
```

- Now, finally, we may state the interp-correctness condition of the pattern u

```coq
Lemma interp_unify_pattern' {t re p evm res v}
ă ă ă (Hre : rawexpr_interp_related re v)
ă ă ă (H : @unify_pattern' t re p evm _ (@Some _) = Some res)
ă ă ă (Ht : @types_match_with evm t re p)
ă ă ă (Ht' : rawexpr_types_ok re (type_of_rawexpr re))
ă ă ă (evm' := mk_new_evm evm (pattern_collect_vars p))
ă ă ă (Hty : type_of_rawexpr re = pattern.type.subst_default t evm'
ă ă ă ă:= eq_type_of_rawexpr_of_types_match_with' Ht Ht')
ă : exists resv : _,
ă ă ă unification_resultT'_interp_related res resv
ă ă ă /\ app_transport_with_unification_resultT'_cps
ă ă ă ă ă ă(pattern_default_interp' p evm' id) resv _ (@Some _)
ă ă ă ă ă= Some (rew Hty in v).
```

– We can now glue the type pattern-unifier with the expression pattern-unifier in a straightforward way. Note that this pattern unifier also has three correctness conditions. `coq Definition unify_pattern {t} (e : rawexpr) (p : pattern t) : forall T, (unification_resultT p -> option T) -> option T := fun T cont => unify_types e p _ (fun evm => evm <- evm; uni e p evm T (fun v => cont (existT _ _ v)))%option.`
– The first correctness condition is again the cps-identity rule: if you invoke `unify_pattern` with any continuation, that must be the same as invoking

it with `Some`, binding the value in the option monad, and then invoking the continuation on the bound value.

– The `wf` correctness condition requires us to define a notion of `wf` for `unification_resultT`.

  * We say that two `unification_resultT`s are `wf`-related if their type-variable-maps are equal, and their identifier-arguments and wildcard binding values are appropriately `wf`-related: "'coq Definition related_sigT_by_eq {A P1 P2} (R : forall x : A, P1 x -> P2 x -> Prop) (x : @sigT A P1) (y : @sigT A P2) : Prop := { pf : projT1 x = projT1 y | R _(rew pf in projT2 x) (projT2 y) }.

Definition related_unification_resultT {var1 var2} (R : forall t, var1 t -> var2 t -> Prop) {t p} : @unification_resultT _t p -> @unification_resultT _t p -> Prop := related_sigT_by_eq (@related_unification_resultT' __R t p).

Definition wf_unification_resultT (G : list {t1 : type & (var1 t1 * var2 t1)%type}) {t p} : @unification_resultT (@value var1) t p -> @unification_resultT (@value var2) t p -> Prop := @related_unification_resultT __(fun _=> wf_value G) t p. `- The 'wf' correctness condition is then that if we have two 'wf_rawexpr'-related 'rawexpr's, invoking 'unify_pattern' on each 'rawexpr' to unify it with a singular pattern 'p', with continuation 'Some', results either in 'None' in both cases, or in two 'unification_resultT's which are 'wf_unification_resultT'-related. - The interpretation correctness condition is a bit of a mouthful. - We say that two 'unification_resultT's are interp-related if their mappings of type variables to types are equal, and their packages of non-type binding data are appropriately interp-related.`coq Local Notation var := (type.interp base.interp) (only parsing).

```
Definition unification_resultT_interp_related {t p}
ă : @unification_resultT (@value var) t p -> @unification_resultT var t p ->
ă := related_unification_resultT (fun t => value_interp_related).
```

  * We can now state the interpretation correctness condition, which is a bit hard for me to meaningfully talk about in English words except by saying "it does the right thing for a good notion of 'right' ":

```
Lemma interp_unify_pattern {t re p v res}
ă ă ă (Hre : rawexpr_interp_related re v)
ă ă ă (Ht' : rawexpr_types_ok re (type_of_rawexpr re))
ă ă ă (H : @unify_pattern t re p _ (@Some _) = Some res)
ă ă ă (evm' := mk_new_evm (projT1 res) (pattern_collect_vars p))
ă : exists resv,
ă ă unification_resultT_interp_related res resv
ă ă /\ exists Hty, (app_with_unification_resultT_cps (@pattern_default_i
```

- Plugging in the arguments to a rewrite rule: Take 2
- There is one more definition before we put all of the rewrite replacement rule pieces together: we describe a way to handle the fact that we are underneath zero, one, or two monads. The way we handle this is by just assuming that we are underneath two monads, and issuing monad-return statements as necessary to correct:

```
Definition normalize_deep_rewrite_rule {should_do_again with_opt under_lets t}
ă : deep_rewrite_ruleTP_gen should_do_again with_opt under_lets t
ă ă -> deep_rewrite_ruleTP_gen should_do_again true true t
ă := match with_opt, under_lets with
ă ă ă| true , trueă => fun x => x
ă ă ă| false, trueă => fun x => Some x
ă ă ă| true , false => fun x => (x <- x; Some (UnderLets.Base x))%option
ă ă ă| false, false => fun x => Some (UnderLets.Base x)
ă ă ăend%cps.
```

  - The `wf` correctness condition, unsurprisingly, just says that if two rewrite replacement rules are appropriately `wf`-related, then their normalizations are too. This is quite verbose to state, though, because it requires traversing multiple layers of monads and pesky dependent types. TODO: should this code actually be included? "'coq Definition wf_maybe_do_again_expr {t} {rew_should_do_again1 rew_should_do_again2 : bool} (G : list {t : _& (var1 t * var2 t)%type}) : expr (var:=if rew_should_do_again1 then @value var1 else var1) t -> expr (var:=if rew_should_do_again2 then @value var2 else var2) t -> Prop := match rew_should_do_again1, rew_should_do_again2 return expr (var:=if rew_should_do_again1 then @value var1 else var1) t -> expr (var:=if rew_should_do_again2 then @value var2 else var2) t -> Prop with | true, true => fun e1 e2 => exists G', (forall t' v1' v2', List.In (existT _t' (v1', v2')) G' -> wf_value G v1' v2') / expr.wf G' e1 e2 | false, false => expr.wf G | , => fun __=> False end.

Definition wf_maybe_under_lets_expr {T1 T2} (P : list {t : _& (var1 t * var2 t)%type} -> T1 -> T2 -> Prop) (G : list {t : _& (var1 t * var2 t)%type}) {rew_under_lets1 rew_under_lets2 : bool} : (if rew_under_lets1 then UnderLets var1 T1 else T1) -> (if rew_under_lets2 then UnderLets var2 T2 else T2) -> Prop := match rew_under_lets1, rew_under_lets2 return (if rew_under_lets1 then UnderLets var1 T1 else T1) -> (if rew_under_lets2 then UnderLets var2 T2 else T2) -> Prop with | true, true => UnderLets.wf P G | false, false => P G | , => fun __=> False end.

Definition maybe_option_eq {A B} {opt1 opt2 : bool} (R : A -> B -> Prop) : (if opt1 then option A else A) -> (if opt2 then option B else B) -> Prop :=

match opt1, opt2 with | true, true => option_eq R | false, false => R | , => fun __=> False end.

Definition wf_deep_rewrite_ruleTP_gen (G : list {t : _& (var1 t * var2 t)%type}) {t} {rew_should_do_again1 rew_with_opt1 rew_under_lets1 : bool} {rew_should_do_again2 rew_with_opt2 rew_under_lets2 : bool} : deep_rewrite_ruleTP_gen1 rew_should_do_again1 rew_with_opt1 rew_under_lets1 t -> deep_rewrite_ruleTP_gen2 rew_should_do_again2 rew_with_opt2 rew_under_lets2 t -> Prop := maybe_option_eq (wf_maybe_under_lets_exp wf_maybe_do_again_expr G).

Lemma wf_normalize_deep_rewrite_rule {G} {t} {should_do_again1 with_opt1 under_lets1} {should_do_again2 with_opt2 under_lets2} {r1 r2} (Hwf : @wf_deep_rewrite_ruleTP_gen G t should_do_again1 with_opt1 under_lets1 should_do_again2 with_opt2 under_lets2 r1 r2) : option_eq (UnderLets.wf (fun G' => wf_maybe_do_again_expr G') G) (normalize_deep_rewrite_rule r1) (normalize_deep_rewrite_rule r2). "- We do not require any interp-correctness condition onnormalize_deep_rewrite_rule. Instead, we bakenormalize_deep_rewrite_rule' into the per-rewrite-rule correctness conditions that a user must prove of every individual rewrite rule.

- Actually, I lied. We need to define the type of a rewrite rule before we can say what it means for one to be correct.

  - An `anypattern` is a dynamically-typed pattern. This is used so that we can talk about `list`s of rewrite rules. `coq Record > anypattern {ident : type -> Type} := { type_of_anypattern : type; pattern_ :> @pattern ident type_of_anypattern }.`
  - A `rewrite_ruleT` is just a sigma of a pattern of any type, with `rewrite_rule_data` over that pattern: `coq Definition rewrite_ruleTP := (fun p : anypattern => @rewrite_rule_data _ (pattern.pattern_of_anypattern p)). Definition rewrite_ruleT := sigT rewrite_ruleTP. Definition rewrite_rulesT := (list rewrite_ruleT).`

- We now define a helper definition to support rewriting again in the output of a rewrite rule. This is a separate definition mostly to make dependent types slightly less painful.

```
Definition maybe_do_againT (should_do_again : bool) (t : base.type)
ă := ((@expr.expr base.type ident (if should_do_again then value else var) t) ->
Definition maybe_do_again
ă ă ă ă ă ă(do_again : forall t : base.type, @expr.expr base.type ident value t
ă ă ă ă ă ă(should_do_again : bool) (t : base.type)
ă := if should_do_again return maybe_do_againT should_do_again t
ă ă ăthen do_again t
ă ă ăelse UnderLets.Base.
```

  - You might think that the correctness condition for this is trivial. And, indeed, the `wf` correctness condition is straightforward. In fact, we have already seen it above in `wf_maybe_do_again_expr`, as there is no proof, only a definition of what it means for things to be related depending on whether or not we are rewriting again.
  - The interpretation correctness rule, on the other hand, is surprisingly subtle. You may have noticed above that `expr.interp_related` is parameterized on an arbitrary `var` type, and an arbitrary relation between the `var` type and `type.interp base.interp`. I said that it is equivalent to equality of interpretation under the assumption of function extensionality, but that is only the case if `var` is instantiated to `type.interp` and the relation is equality or pointwise/extensional equivalence. Here, we must instantiate the `var` type with `@value var`, and the relation with `value_interp_related`. We then prove that for any "good" notion of rewriting again, if our input value is interp-related to an interpreted value, the result of maybe rewriting again is also interp-related to that interpreted value. `coq Lemma interp_maybe_do_again (do_again : forall t : base.type, @expr.expr base.type ident value t -> UnderLets (expr t)) (Hdo_again : forall t e v, expr.interp_related ident_interp (fun t => value_interp_related) e v -> UnderLets_interp_related (do_again t e) v) {should_do_again : bool} {t e v} (He : (if should_do_again return @expr.expr _ _ (if should_do_again then _ else _) _ -> _ then expr.interp_related_gen ident_interp (fun t => value_interp_related) else expr_interp_related) e v) : UnderLets_interp_related (@maybe_do_aga _ _ do_again should_do_again t e) v.`

- For the purposes of ensuring that reduction does not get blocked where it should not, we only allow rewrite rules to match on fully applied patterns, and to return base-typed expressions. We patch this broken abstraction barrier with

```
Local Notation base_type_of t
ã := (match t with type.base t' => t' | type.arrow _ __ => base.type.unit end).
```

- Finally, we can define what it means to rewrite with a particular rewrite rule. It is messy primarily due to continuation passing style, optional values, and type casts. Note that we use `<-` to mean "bind in whatever monad is the top-most scope". Other than these complications, it just unifies the pattern with the `rawexpr` to get binding data, applies the rewrite replacement rule to the binding data, normalizes the applied rewrite replacement rule, calls the rewriter again on the output if it should, and returns the result. `coq Definition rewrite_with_rule {t} e' (pf : rewrite_ruleT) : option (UnderLets (expr t)) := let 'existT p f := pf in let should_do_again := rew_should_do_again f in unify_pattern e' (pattern.pattern_of_anypat p) _ (fun x => app_with_unification_resultT_cps (rew`

```
f) x _              (fun f'               => (tr <- type.try_make_transport_cps
(@base.try_make_transport_cps) _ _ _;                    (tr <- tr;
(tr' <- type.try_make_transport_cps (@base.try_make_transport_cps)
_ _ _;                            (tr' <- tr';                                optio
(normalize_deep_rewrite_rule (projT2 f'))                                  (fun
fv                                  => Some (fv <-- fv;
<-- maybe_do_again should_do_again (base_type_of (type_of_rawexpr e'))
(tr fv);                                                UnderLets.Base
(tr' fv))%under_lets))%option)%cps)%option)%cps)%cps).
```

– We once again do not have any `wf` correctness condition for `rewrite_with_rule`; we merely unfold it as needed.

– To write down the correctness condition for `rewrite_with_rule`, we must first define what it means for `rewrite_rule_data` to be "good".

– Here is where we use `normalize_deep_rewrite_rule`. Replacement rule data is good with respect to an interpretation value if normalizing it gives an appropriately interp-related thing to that interpretation value:

```
Local Notation var := (type.interp base.interp) (only parsing).

Definition deep_rewrite_ruleTP_gen_good_relation
ă ă ă ă ă ă{should_do_again with_opt under_lets : bool} {t}
ă ă ă ă ă ă(v1 : @deep_rewrite_ruleTP_gen should_do_again with_opt under_let
ă ă ă ă ă ă(v2 : var t)
ă : Prop
ă := let v1 := normalize_deep_rewrite_rule v1 in
ă ă ămatch v1 with
ă ă ă| None => True
ă ă ă| Some v1 => UnderLets.interp_related
ă ă ă ă ă ă ă ă ă ă ident_interp
ă ă ă ă ă ă ă ă ă ă (if should_do_again
ă ă ă ă ă ă ă ă ă ă ă ă return (@expr.expr base.type ident (if should_do_aga
ă ă ă ă ă ă ă ă ă ă ăthen expr.interp_related_gen ident_interp (fun t => val
ă ă ă ă ă ă ă ă ă ă ăelse expr_interp_related)
ă ă ă ă ă ă ă ă ă ă v1
ă ă ă ă ă ă ă ă ă ă v2
ă ă ăend.
```

– Rewrite rule data is good if, for any interp-related binding data, the replacement function applied to the value-binding-data is interp-related to the default interpretation of the pattern applied to the interpreted-value-binding-data:

```
Definition rewrite_rule_data_interp_goodT
ă ă ă ă ă ă{t} {p : pattern t} (r : @rewrite_rule_data t p)
ă : Prop
```

```
ă := forall x y,
ă ă related_unification_resultT (fun t => value_interp_related) x y
ă ă -> option_eq
ă ă ă ă ă(fun fx gy
ă ă ă ă ă => related_sigT_by_eq
ă ă ă ă ă ă ă ă(fun evm
ă ă ă ă ă ă ă ă => @deep_rewrite_ruleTP_gen_good_relation
ă ă ă ă ă ă ă ă ă ă ă(rew_should_do_again r) (rew_with_opt r) (rew_under_let
ă ă ă ă ă ă ă ăfx gy)
ă ă ă ă ă(app_with_unification_resultT_cps (rew_replacement r) x _ (@Some _)
ă ă ă ă ă(app_with_unification_resultT_cps (pattern_default_interp p) y _ (@
```

– The interpretation correctness condition then says that if the rewrite rule is good, the `rawexpr re` has ok types, the "rewrite again" function is good, and `rewrite_with_rule` succeeds and outputs an expression `v1`, then `v1` is interp-related to any interpreted value which `re` is interp-related to:

```
Lemma interp_rewrite_with_rule
ă ă ă (do_again : forall t : base.type, @expr.expr base.type ident value t -
ă ă ă (Hdo_again : forall t e v,
ă ă ă ă ă expr.interp_related_gen ident_interp (fun t => value_interp_relate
ă ă ă ă ă -> UnderLets_interp_related (do_again t e) v)
ă ă ă (rewr : rewrite_ruleT)
ă ă ă (Hrewr : rewrite_rule_data_interp_goodT (projT2 rewr))
ă ă ă t e re v1 v2
ă ă ă (Ht : t = type_of_rawexpr re)
ă ă ă (Ht' : rawexpr_types_ok re (type_of_rawexpr re))
ă : @rewrite_with_rule do_again t re rewr = Some v1
ă ă -> rawexpr_interp_related re (rew Ht in v2)
ă ă -> UnderLets_interp_related v1 v2.
```

Tying it all together

- We can now say what it means to rewrite with a decision tree in a given `rawexpr re`. We evaluate the decision tree, and whenever we are asked to try the `k`th rewrite rule, we look for it in our list of rewrite rules, and invoke `rewrite_with_rule`. By default, if rewriting fails, we will eventually return `expr_of_rawexpr re`. coq Definition eval_rewrite_rules (d : decision_tree) (rews : rewrite_rulesT) (e : rawexpr) : UnderLets (expr (type_of_rawexpr e)) := let defaulte := expr_of_rawexpr e in (eval_decision_tree (e::nil) d (fun k ctx => match ctx return option (UnderLets (expr (type_of_rawexpr e))) with | e'::nil => (pf <- nth_error rews k; rewrite_with_rule e' pf)%option | _ => None end);;; (UnderLets.Base defaulte))%option.

- To define the correctness conditions, we must first define what it means for lists of rewrite rules to be good.

  - For `wf`, we need to catch up a bit before getting to lists of rewrite rules. These say the obvious things: “‘coq Definition wf_with_unif_rewrite_ruleTP_gen (G : list {t : _& (var1 t * var2 t)%type}) {t} {p : pattern t} {rew_should_do_again1 rew_with_opt1 rew_under_lets1} {rew_should_do_again2 rew_with_opt2 rew_under_lets2} : with_unif_rewrite_ruleTP_gen1 p rew_should_do_again1 rew_with_opt1 rew_under_lets1 -> with_unif_rewrite_ruleTP_gen2 p rew_should_do_again2 rew_with_opt2 rew_under_lets2 -> Prop := fun f g => forall x y, wf_unification_resultT G x y -> option_eq (fun (fx : { evm : _& deep_rewrite_ruleTP_gen1 rew_should_do_again1 rew_with_opt1 rew_under_lets1 _}) (gy : { evm : _& deep_rewrite_ruleTP_gen2 rew_should_do_again2 rew_with_opt2 rew_under_lets2 _}) => related_sigT_by_eq (fun _=> wf_deep_rewrite_ruleTP_gen G) fx gy) (app_with_unification_resultT f x _(@Some *)) (app_with_unification_resultT_cps g y (@Some _)).

    Definition wf_rewrite_rule_data (G : list {t : _& (var1 t * var2 t)%type}) {t} {p : pattern t} (r1 : @rewrite_rule_data1 t p) (r2 : @rewrite_rule_data2 t p) : Prop := wf_with_unif_rewrite_ruleTP_gen G (rew_replacement r1) (rew_replacement r2).

    `- Two lists of rewrite rules are ‘wf‘ related if they have the same length, and if any pair of rules in their zipper (‘List.combine‘) have equal patterns and ‘wf‘-related data:`coq Definition rewrite_rules_goodT (rew1 : rewrite_rulesT1) (rew2 : rewrite_rulesT2) : Prop := length rew1 = length rew2 /\ (forall p1 r1 p2 r2, List.In (existT _p1 r1, existT _p2 r2) (combine rew1 rew2) -> { pf : p1 = p2 | forall G, wf_rewrite_rule_data G (rew [fun tp => @rewrite_rule_data1 _(pattern.pattern_of_anypattern tp)] pf in r1) r2 }). `- A list of rewrite rules is good for interpretation if every rewrite rule in that list is good for interpretation:`coq Definition rewrite_rules_interp_goodT (rews : rewrite_rulesT) : Prop := forall p r, List.In (existT _p r) rews -> rewrite_rule_data_interp_goodT r. `- The ‘wf‘-correctness condition for ‘eval_rewrite_rules‘ says the obvious thing: for ‘wf‘-related "rewrite again" functions, ‘wf‘-related lists of rewrite rules, and ‘wf‘-related ‘rawexpr‘s, the output of ‘eval_rewrite_rules is ‘wf‘-related:`coq Lemma wf_eval_rewrite_rules (do_again1 : forall t : base.type, @expr.expr base.type ident (@value var1) t -> @UnderLets var1 (@expr var1 t)) (do_again2 : forall t : base.type, @expr.expr base.type ident (@value var2) t -> @UnderLets var2 (@expr var2 t)) (wf_do_again : forall G (t : base.type) e1 e2, (exists G', (forall t v1 v2, List.In

(existT _t (v1, v2)) G' -> Compile.wf_value G v1 v2) / expr.wf G' e1 e2) -> UnderLets.wf (fun G' => expr.wf G') G (@do_again1 t e1) (@do_again2 t e2)) (d : @decision_tree raw_pident) (rew1 : rewrite_rulesT1) (rew2 : rewrite_rulesT2) (Hrew : rewrite_rules_goodT rew1 rew2) (re1 : @rawexpr var1) (re2 : @rawexpr var2) {t} G e1 e2 (Hwf : @wf_rawexpr G t re1 e1 re2 e2) : UnderLets.wf (fun G' => expr.wf G') G (rew [fun t => @UnderLets var1 (expr t)] (proj1 (eq_type_of_rawexpr_of_wf Hwf)) in (eval_rewrite_rules1 do_again1 d rew1 re1)) (rew [fun t => @UnderLets var2 (expr t)] (proj2 (eq_type_of_rawexpr_of_wf Hwf)) in (eval_rewrite_rules2 do_again2 d rew2 re2)). - The interpretation correctness is also the expected one: for a "rewrite again" function that preserves interp-relatedness, a good-for-interp list of rewrite rules, a ‘rawexpr‘ whose types are ok and which is interp-related to a value ‘v‘, the result of ‘eval_rewrite_rules is interp-related to ‘v‘:coq Lemma interp_eval_rewrite_rules (do_again : forall t : base.type, @expr.expr base.type ident value t -> UnderLets (expr t)) (d : decision_tree) (rew_rules : rewrite_rulesT) (re : rawexpr) v (Hre : rawexpr_types_ok re (type_of_rawexpr re)) (res := @eval_rewrite_rules do_again d rew_rules re) (Hdo_again : forall t e v, expr.interp_related_gen ident_interp (fun t => value_interp_related) e v -> UnderLets_interp_related (do_again t e) v) (Hr : rawexpr_interp_related re v) (Hrew_rules : rewrite_rules_interp_goodT rew_rules) : UnderLets_interp_related res v. “‘

- Only one piece remains (other than defining particular rewrite rules and proving them good). If you were following carefully, you might note that `eval_rewrite_rules` takes in a `rawexpr` and produces an `UnderLets expr`, while `rewrite_bottomup` expects a function `rewrite_head : forall t (idc : ident t), value_with_lets t`. From a PHOAST identifier, we must construct a `value_with_lets` which collects all of the `value` arguments to the identifier and performs `eval_rewrite_rules` once the identifier is fully applied. We call this function `assemble_identifier_rewriters`, and it is built out of a small number of pieces.

- We define a convenience function that takes a `value` and an `expr` at the same type, and produces a `rawexpr` by using `rExpr` on the expr if the type is a base type, and `rValue` on the `value` otherwise. Morally, the `expr` and the `value` should be the same term, modulo `reify` and/or `reflect`:

```
Definition rValueOrExpr2 {t} : value t -> expr t -> rawexpr
ă := match t with
ă ă ă| type.base _ => fun v e => @rExpr _ e
ă ă ă| type.arrow _ _ => fun v e => @rValue _ v
ă ă ăend.
```

- We take in a context variable (eventually autogenerated by python) which eta-iota-expands a function over an identifier by producing a `match` on the identifier. Its specification is that it is pointwise-equal to function application; it exists

entirely so that we can perform rewrite-rule-compilation time reduction on the rewrite rules by writing down the cases for every head identifier separately. The context variable is `eta_ident_cps : forall {T : type.type base.type -> Type} {t} (idc : ident t) (f : forall t', ident t' -> T t'), T t`, and we require that `forall T t idc f, @eta_ident_cps T t idc f = f t idc`.

- We can now carefully define the function that turns `eval_rewrite_rules` into a thing that can be plugged into `rewrite_head`. We take care to preserve thunked computation in `rValue` nodes, while describing the alternative structure via `reify`. In general, the stored values are only interp-related to the same things that the "unrevealed structure" expressions are interp-related to. There is no other relation (that we've found) between the values and the expressions, and this caused a great deal of pain when trying to specify the interpretation correctness properties.

```
Section with_do_again.
ă Context (dtree : decision_tree)
ă ă ă ă ă (rewrite_rules : rewrite_rulesT)
ă ă ă ă ă (default_fuel : nat)
ă ă ă ă ă (do_again : forall t : base.type, @expr.expr base.type ident value t -

ă Let dorewrite1 (e : rawexpr) : UnderLets (expr (type_of_rawexpr e))
ă ă := eval_rewrite_rules do_again dtree rewrite_rules e.

ă Fixpoint assemble_identifier_rewriters' (t : type) : forall e : rawexpr, (fora
ă ă := match t return forall e : rawexpr, (forall P, P (type_of_rawexpr e) -> P
ă ă ă ă| type.base _
ă ă ă ă ă=> fun e k => k (fun t => UnderLets (expr t)) (dorewrite1 e)
ă ă ă ă| type.arrow s d
ă ă ă ă ă=> fun f k (x : value' _ _)
ă ă ă ă ă ă => let x' := reify x in
ă ă ă ă ă ă ă ă@assemble_identifier_rewriters' d (rApp f (rValueOrExpr2 x x') (k
ă ă ă ăend%under_lets.

ă Definition assemble_identifier_rewriters {t} (idc : ident t) : value_with_lets
ă ă := eta_ident_cps _ _ idc (fun t' idc' => assemble_identifier_rewriters' t' (
End with_do_again.
```

  - The `wf`-correctness condition for `assemble_identifier_rewriters'` says that if two `rawexpr`s are `wf`-related, and both continuations are extensionally/pointwise equal to the identity function transported across the appropriate equality proof, then the results of `assemble_identifier_rewriters'` are `wf`-related, under the assumption that the "rewrite again" functions are appropriately `wf`-related and the list of rewrite rules is good. "'coq Section with_do_again. Context (dtree : @decision_tree raw_pident) (rew1 : rewrite_rulesT1) (rew2 : rewrite_rulesT2) (Hrew :

rewrite_rules_goodT rew1 rew2) (do_again1 : forall t : base.type, @expr.expr base.type ident (@value var1) t -> @UnderLets var1 (@expr var1 t)) (do_again2 : forall t : base.type, @expr.expr base.type ident (@value var2) t -> @UnderLets var2 (@expr var2 t)) (wf_do_again : forall G G' (t : base.type) e1 e2, (forall t v1 v2, List.In (existT _t (v1, v2)) G' -> Compile.wf_value G v1 v2) -> expr.wf G' e1 e2 -> UnderLets.wf (fun G' => expr.wf G') G (@do_again1 t e1) (@do_again2 t e2)).

Lemma wf_assemble_identifier_rewriters' G t re1 e1 re2 e2 K1 K2 (He : @wf_rawexpr G t re1 e1 re2 e2) (HK1 : forall P v, K1 P v = rew [P] (proj1 (eq_type_of_rawexpr_of_wf He)) in v) (HK2 : forall P v, K2 P v = rew [P] (proj2 (eq_type_of_rawexpr_of_wf He)) in v) : wf_value_with_lets G (@assemble_identifier_rewriters‘var1 rew1 do_again1 t re1 K1) (@assemble_identifier_rewriters' var2 rew2 do_again2 t re2 K2). - The ‘wf‘-correctness condition for ‘assemble_identifier_rewriters‘ merely says that the outputs are always ‘wf‘-related, again under the assumption that the "rewrite again" functions are appropriately ‘wf‘-related and the list of rewrite rules is good.coq Lemma wf_assemble_identifier_rewriters G t (idc : ident t) : wf_value_with_lets G (@assemble_identifier_rewriters var1 rew1 do_again1 t idc) (@assemble_identifier_rewriters var2 rew2 do_again2 t idc). Proof. - The interpretation correctness condition says that for a good "rewrite again" function, a good-for-interpretation list of rewrite rules, a ‘rawexpr‘ ‘re‘ whose types are ok and which is interp-related to an interpreted value ‘v‘, the result of ‘assemble_identifie is interp-related to ‘v‘. The actual statement is slightly more obscure, parameterizing over types which are equal to computed things, primarily for ease of induction in the proof.coq Lemma interp_assemble_identifier_rewriters' (do_again : forall t : base.type, @expr.expr base.type ident value t -> UnderLets (expr t)) (dt : decision_tree) (rew_rules : rewrite_rulesT) t re K (res := @assemble_identifier_rewriters' dt rew_rules do_again t re K) (Hre : rawexpr_types_ok re (type_of_rawexpr re)) (Ht : type_of_rawexpr re = t) v (HK : K = (fun P v => rew [P] Ht in v)) (Hdo_again : forall t e v, expr.interp_related_gen ident_interp (fun t => value_interp_related) e v -> UnderLets_interp_related (do_again t e) v) (Hrew_rules : rewrite_rules_interp_goodT rew_rules) (Hr : rawexpr_interp_related re v) : value_interp_related res (rew Ht in v). - The interpretation correctness condition for ‘assemble_identifier_rewriters‘ is very similar, where the ‘rawexpr_interp_related‘ hypothesis is replaced by an pointwise equality between the interpretation of the identifier and the interpreted value.coq Lemma interp_assemble_identifier_rewriters (do_again : forall t : base.type, @expr.expr base.type ident value t -> UnderLets (expr t)) (d : decision_tree) (rew_rules : rewrite_rulesT) t idc v (res := @assemble_identifier_rewriters d rew_rules do_again t idc) (Hdo_again : forall t e v, expr.interp_related_gen ident_interp (fun t => value_interp_related)

e v -> UnderLets_interp_related (do_again t e) v) (Hrew_rules : rewrite_rules_interp_goodT rew_rules) (Hv : ident_interp t idc == v) : value_interp_related res v. “‘

- We have not talked about correctness conditions for the functions we looked at in the very beginning, `rewrite_bottomup` and `repeat_rewrite`, but the correctness conditions for these two are straightforward, so we state them without explanation.

  – The `wf` correctness conditions are “‘coq Section with_rewrite_head. Context (rewrite_head1 : forall t (idc : ident t), @value_with_lets var1 t) (rewrite_head2 : forall t (idc : ident t), @value_with_lets var2 t) (wf_rewrite_head : forall G t (idc1 idc2 : ident t), idc1 = idc2 -> wf_value_with_lets G (rewrite_head1 t idc1) (rewrite_head2 t idc2)).

  Local Notation rewrite_bottomup1 := (@rewrite_bottomup var1 rewrite_head1). Local Notation rewrite_bottomup2 := (@rewrite_bottomup var2 rewrite_head2).

  Lemma wf_rewrite_bottomup G G’ {t} e1 e2 (Hwf : expr.wf G e1 e2) (HG : forall t v1 v2, List.In (existT _t (v1, v2)) G -> wf_value G’ v1 v2) : wf_value_with_lets G’ (@rewrite_bottomup1 t e1) (@rewrite_bottomup2 t e2). End with_rewrite_head.

  Local Notation nbe var := (@rewrite_bottomup var (fun t idc => reflect (expr.Ident idc))).

  Lemma wf_nbe G G’ {t} e1 e2 (Hwf : expr.wf G e1 e2) (HG : forall t v1 v2, List.In (existT _t (v1, v2)) G -> wf_value G’ v1 v2) : wf_value_with_lets G’ (@nbe var1 t e1) (@nbe var2 t e2).

  Section with_rewrite_head2. Context (rewrite_head1 : forall (do_again : forall t : base.type, @expr (@value var1) (type.base t) -> @UnderLets var1 (@expr var1 (type.base t))) t (idc : ident t), @value_with_lets var1 t) (rewrite_head2 : forall (do_again : forall t : base.type, @expr (@value var2) (type.base t) -> @UnderLets var2 (@expr var2 (type.base t))) t (idc : ident t), @value_with_lets var2 t) (wf_rewrite_head : forall do_again1 do_again2 (wf_do_again : forall G’ G (t : base.type) e1 e2 (HG : forall t v1 v2, List.In (existT _t (v1, v2)) G -> wf_value G’ v1 v2), expr.wf G e1 e2 -> UnderLets.wf (fun G’ => expr.wf G’) G’ (do_again1 t e1) (do_again2 t e2)) G t (idc1 idc2 : ident t), idc1 = idc2 -> wf_value_with_lets G (rewrite_head1 do_again1 t idc1) (rewrite_head2 do_again2 t idc2)).

  Lemma wf_repeat_rewrite fuel : forall {t} G G’ e1 e2 (Hwf : expr.wf G e1 e2) (HG : forall t v1 v2, List.In (existT _t (v1, v2)) G -> wf_value G’ v1 v2), wf_value_with_lets G’ (@repeat_rewrite var1 rewrite_head1 fuel t e1) (@repeat_rewrite var2 rewrite_head2 fuel t e2). - `The interpretation correctness conditions are`coq Section with_rewrite_head. Context (rewrite_head

: forall t (idc : ident t), value_with_lets t) (interp_rewrite_head : forall t idc v, ident_interp idc == v -> value_interp_related (rewrite_head t idc) v).

Lemma interp_rewrite_bottomup {t e v} (He : expr.interp_related_gen (@ident_interp) (fun t => value_interp_related) e v) : value_interp_related (@rewrite_bottomup var rewrite_head t e) v. End with_rewrite_head.

Local Notation nbe := (@rewrite_bottomup var (fun t idc => reflect (expr.Ident idc))).

Lemma interp_nbe {t e v} (He : expr.interp_related_gen (@ident_interp) (fun t => value_interp_related) e v) : value_interp_related (@nbe t e) v.

Lemma interp_repeat_rewrite {rewrite_head fuel t e v} (retT := value_interp_related (@repeat_rewrite _rewrite_head fuel t e) v) (Hrewrite_head : forall do_again (Hdo_again : forall t e v, expr.interp_related_gen (@ident_interp) (fun t => value_interp_related) e v -> UnderLets.interp_related (@ident_interp) (expr.interp_related (@ident_interp)) (do_again t e) v) t idc v, ident_interp idc == v -> value_interp_related (@rewrite_head do_again t idc) v) (He : expr.interp_related_gen (@ident_interp) (fun t => value_interp_related) e v) : retT.
“‘

Chapter 7

Reification by Parametricity

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

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.

7.1 Introduction

Proof by reflection [7] 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 7.2, 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 7.3, which explains our alternative technique. We benchmark our approach against 18 competitors in Section 7.4.

7.1.1 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_0 : is_even 0
| even_SS : forall x, is_even x -> is_even (S (S x)).
Goal is_even (let x := 100 * 100 * 100 * 100 in
```

```
  let y := x * x * x * x in
  y * y * y * y).
```

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

Even on smaller terms, though, proof size can quickly become an issue. If we give a 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.

7.1.2 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 7-1: Evenness Checking

A simple evenness checker can just operate on the unary encoding of natural numbers (Figure 7-1). We can use its correctness theorem to prove goals much more quickly:

```
Theorem soundness : forall n, check_is_even n = true -> is_even n.
Goal is_even 2000.
  Time repeat (apply even_SS || apply even_0). (* 1.8 s *)
  Undo.
  Time apply soundness; vm_compute; reflexivity. (* 0.004 s *)
```

The tactic `vm_compute` tells Coq to use its virtual machine for reduction, to compute the value of `check_is_even 2000`, after which `reflexivity` proves that `true =`

`true`. Note how much faster this method is. In fact, even the asymptotic complexity is better; this new algorithm is linear rather than quadratic in `n`.

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

7.1.3 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 7-2. Simple Ltac expression

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 7.3, we get the inductive type of Figure 7-2.

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) [30], parametric higher-order abstract syntax (PHOAS) [9], and de Bruijn indices [8]. 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 [9]. 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.
```

7.2 Methods of Reification

We implemented reification in 18 different ways, using 6 different metaprogramming facilities in the Coq ecosystem: Ltac, Ltac2, Mtac [14], type classes [33], canonical structures [13], and reification-specific OCaml plugins (quote [10], template-coq [3], ours). Figure 7-3 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 λ nodes.

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.

```
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 7-3: Reification Without Binders in $\mathcal{L}_{ex}$

7.3 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

[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`

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 7-4.

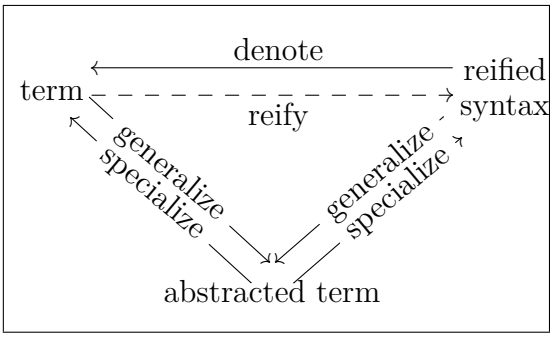


Figure 7-4: Abstraction and Reification

7.3.1 Case-By-Case Walkthrough

Function Applications And Constants.

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

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

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

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

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

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

Wrapped Primitives: "Let" Binders, Eliminators, Quantifiers.

The above procedure can be applied to a term that contains "let" binders to get a PHOAS 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 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 $\forall$/Π 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 ($\lambda n\ f.$ `SumUpTo` n (`Abs` ($\lambda v.\ f$ (`Var` v)))) for `sum_upto`. Note that the expression plugged in for `sum_upto` differs from the one plugged in for `Let_In` only in the use of a deeply embedded abstraction node. If we wanted to reify `LetIn` as just another higher-order function (as opposed to a distinguished wrapper for a primitive), the code would look identical to that for `sum_upto`.

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

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

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

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

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.

7.3.2 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 shortly.

[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.

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.
```

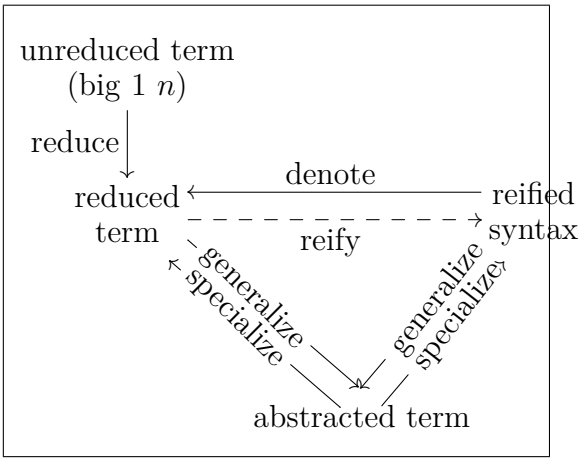
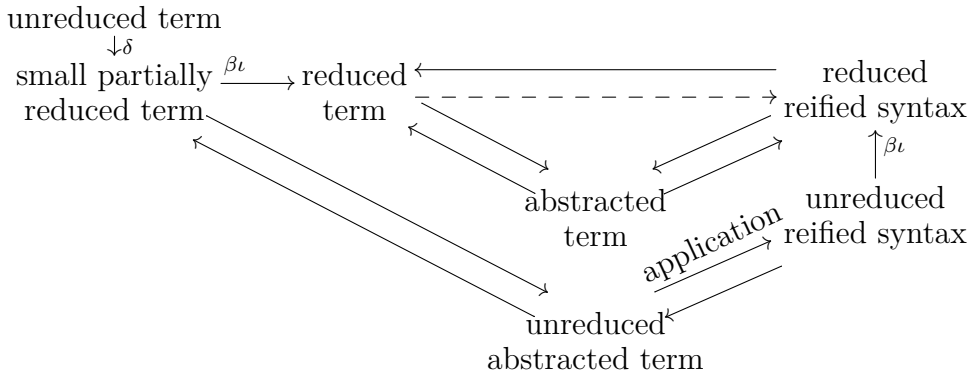


Figure 7-5: Abstraction, Reification, Reduction

Our commutative diagram in Figure 7-4 now has an additional node, becoming Figure 7-5. 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 δ or unfolding step, followed by a $\beta\iota$ step that reduces applications of λs and evaluates recursive calls. In specialization, there is an application step, where the λ is applied to arguments, and a β-reduction step, where the arguments are substituted. To obtain reified syntax, we may perform generalization after δ-reduction (before $\beta\iota$-reduction), and we are not required to perform the final β-reduction step of specialization to get a well-typed term. It is important that unfolding `big` results in exposing the body for generalization, which we accomplish in Coq by exposing the anonymous recursive function; in other languages, the result may be a primitive eliminator applied to the body of the fixpoint. Either way, our commutative diagram thus becomes



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

Our first step is to unfold `big`, rendered as the arrow labeled δ in the diagram. In Coq, the result is an anonymous fixpoint; here we will write it using the recursor `count_rec`

of type $\forall T.\ T \to (\texttt{count} \to T \to T) \to \texttt{count} \to T$. Performing δ-reduction, that is, unfolding `big`, gives us the small partially reduced term

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

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

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

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 [12] for a couple hundred different choices of numeric parameters (e.g., prime modulus of arithmetic). A couple hundred is enough to beat the overhead.

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

`ExampleMoreParametricity.v` in the code supplement mirrors the development of reification by parametricity in Subsection 7.3.1.

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 β-reduction. By chaining `pattern` with an $\mathcal{L}_{tac}$-`match` statement to peel off the application, we can get the abstracted function.

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

Note that if `@expr var` lives in `Type` rather than `Set`, 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
         | ?P' _ => revert n; change (_forall nat P')
```

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

```
  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`.

7.3.4 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.

7.4 Performance Comparison

We have done a performance comparison of the various methods of reification to the PHOAS language `@expr var` from Figure 7.1.3 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 consider (1), because it provides the most fine-grained analysis of the actual reification

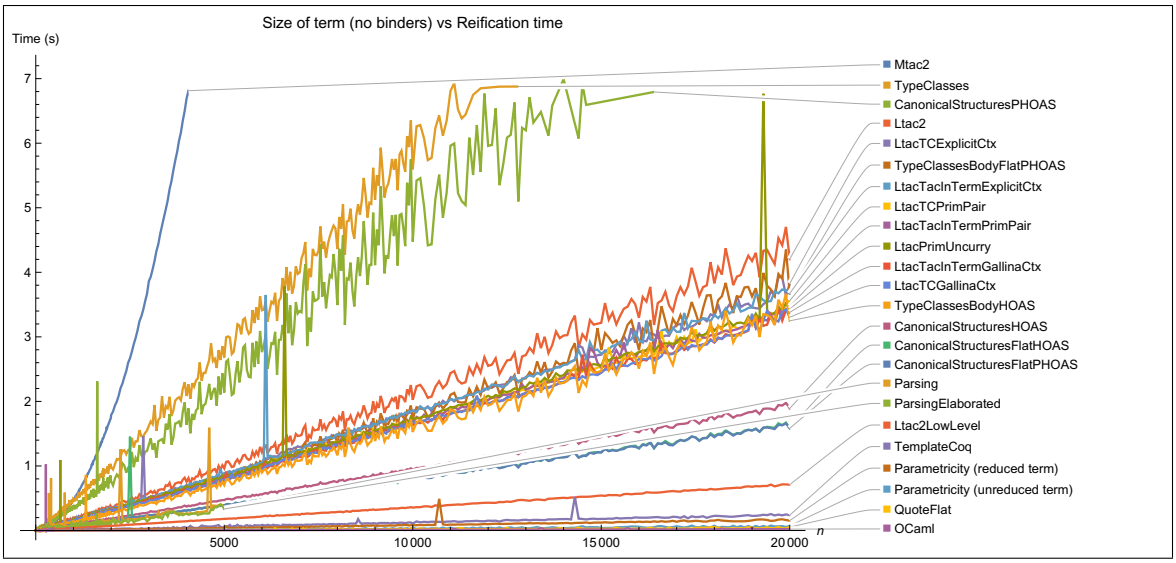


Figure 7-6: Performance of Reification without Binders

method.

7.4.1 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 7-6 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. Un-

like 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.

7.4.2 With Binders

We look at terms of the form `dlet` $\mathtt{a}_1$ `:= 1 * 1 in dlet` $\mathtt{a}_2$ `:=` $\mathtt{a}_1$ `*` $\mathtt{a}_1$ `in ... dlet` $\mathtt{a}_n$ `:=` $\mathtt{a}_{n-1}$ `*` $\mathtt{a}_{n-1}$ `in` $\mathtt{a}_n$, 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 7-7 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 [29].

7.5 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 7.3.2. We perform δ on an unreduced term to obtain a small, partially reduced term; we then perform abstraction to get an abstracted, unreduced term, followed by application to get unreduced reified syntax. These steps are all fast. Finally, we perform $\beta\iota$-reduction to get reduced, reified syntax and perform $\beta\iota\delta$ reduction to get back a

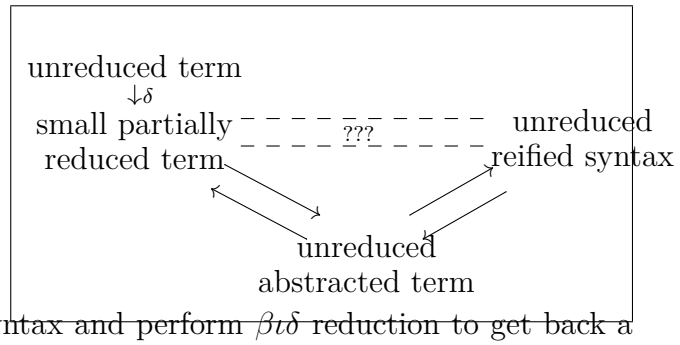


Figure 7-8: Completing the commutative triangle

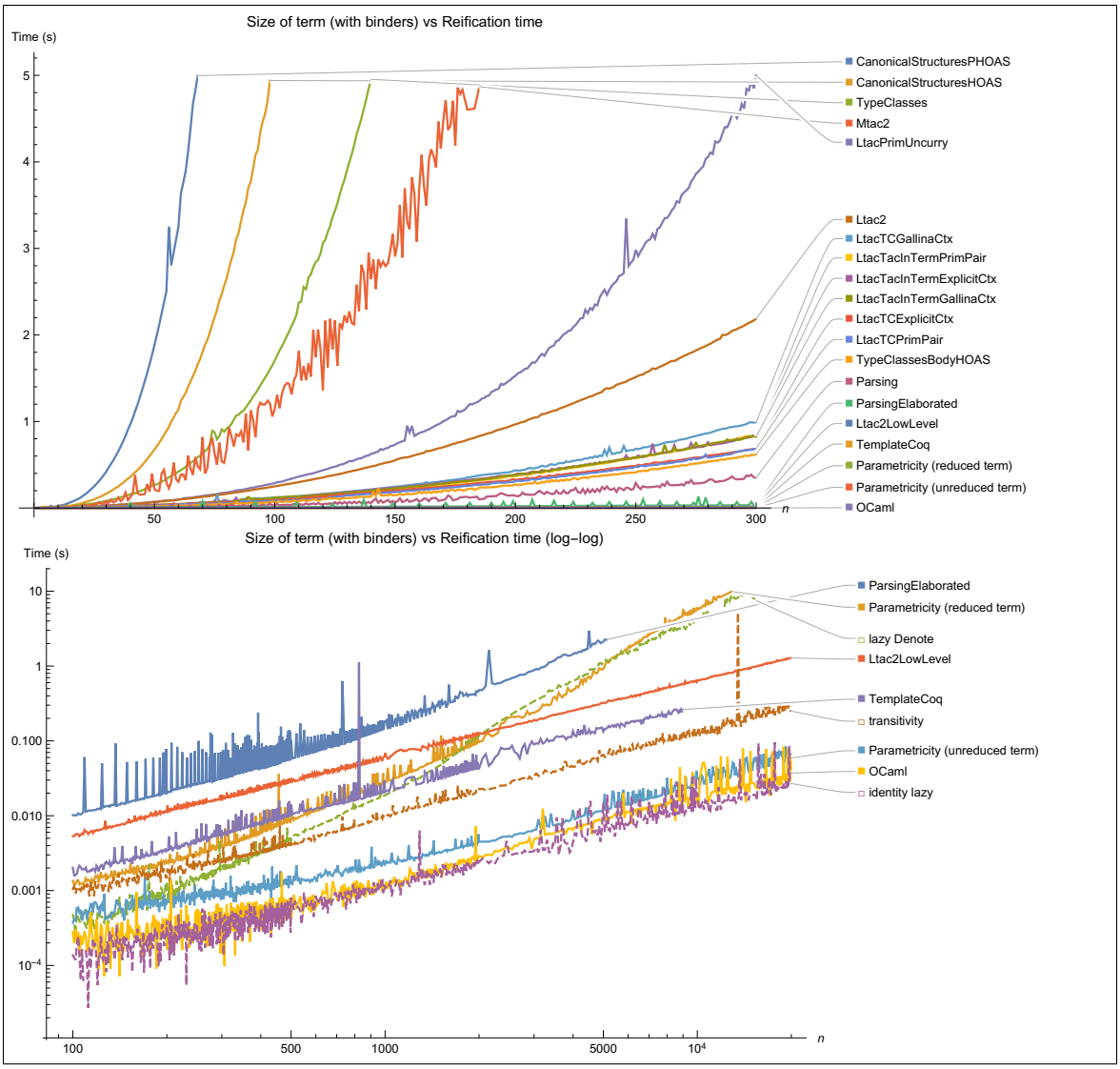


Figure 7-7: Performance of Reification with Binders

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 7-8.

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 $\mathbb{N} \times$ `expr`. Then we would need a proof that for any function F of type

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

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

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

This theorem is a sort of parametricity theorem.

Despite this remaining open question, we hope that our performance results make a strong case for our method of reification; it is fast, concise, and robust.

7.6 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 [25]. We first made use of `pattern` to allow `vm_compute` to replace `cbv`-with-an-explicit-blacklist when we discovered `cbv` was too slow and the blacklist too hard to maintain. We then noticed that in the sequence of doing abstraction; `vm_compute`; application; β-reduction; reification, we could move β-reduction to the end of the sequence if we fused reification with application, and thus reification by parametricity was born.

This work was supported in part by a Google Research Award and National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584.

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

Part IV

Conclusion

Chapter 8

Tooling fixes (improvements in Coq)

8.A Fragments from the category theory paper

For reasons that we present in the course of the paper, we were unsatisfied with the feature set of released Coq version 8.4. We wound up adopting the Coq version under development by homotopy type theorists [32], making critical use of its stronger universe polymorphism (**??**) and higher inductive types (**??**). We hope that our account here provides useful data points for proof assistant designers about which features can have serious impact on proving convenience or performance in very abstract developments. The two features we mentioned earlier in the paragraph can simplify the Coq user experience dramatically, while a number of other features, at various stages of conception or implementation by Coq team members, can make proving much easier or improve proof script performance by orders of magnitude, generally by reducing term size (Subsection 3.5.2): primitive record projections (Subsection 3.5.4), internalized proof irrelevance for equalities (Subsection 3.5.2), and η rules for records (Section 3.4.1) and equality proofs (Subsection 3.4.2).

8.B transcript bits from Adam

Ah a sort of like preconclusion chapter that's like let's now look at how cock is evolved performance wise over the years like here places that we've actually improved performance and this will be one that draws a bunch of other examples from the category theory paper, like look universe polymorphism is a thing that was implemented and helps here and like sort of like presenting a bunch of little things.

8.C transcript bits from Rajee

So those are the two main sections the thesis. And then there's another section of other small. Miscellaneous things that come up better like performance bottlenecks. Through like can or performance concerns, let's say these are things like decide design decisions that can have quadratic impacts. Um, Decisions about like what parts of cop to use for what and like why some bits might be more or less slow than others.

Yeah, that's that's that section and then I think I'm going to have seconds last section. Be a sort of retrospective of like places where cocks performance has gotten better in the past like decade or so of like I started with a bunch of ways that solving performance issues improved systems is heard but here are some successes and things where like we've managed to improve things and you can actually like, Leverage this for faster performance.

[**TODO:** this chapter]

Chapter 9

Concluding Remarks

9.A transcript bits from Adam

And then they'll be the conclusion which I'm thinking of having as a like what are the next steps in performance of previous distance and I think this will. My current inclination is to like, Sort of point towards the paper that under has been talking about writing that's like okay, so there's a sense in which in order to do program transformation and rewriting we took the entire non-trusted part and we threw it out.

The like. And like part of that is because most uses of the non-trusted part you just cobble something together that works but if you're like tracking every single time you invoke conversion and you're like carefully piecing together something it should be possible to make something that scales. And it's not clear if that's currently even the case.

Unlike investigating that it's sort of the next wave of. Performance issues to look at. Okay. Well when you get to the conclusion of this sort of document you've pretty freehand to speculate on things and go where your heart takes you so I'm not too worried about I haven't feel like that part of the relatively quick for you to write and free of difficult choice points.

Yeah, that seems that seems mostly true. I feel like I'll have a little bit of trouble with the like first paragraph on the last paragraph of the conclusion. I'd like the transition points and they'll like actually tying it up but the body of it seems I don't expect to have that much trouble, okay?

9.B transcript bits from Rajee

[**TODO:** Decide between options, maybe add more text]

Option 1 Perhaps this thesis has inspired you to write your own performance system and we remind you about the things you should look out for when implementing it.

Option 2 The End

[**TODO:** insert category theory diagram of an End here]

Option 3 (best so far) What are the next steps in proof assistant performance. There's a paper that Andres has floated writing that I think is a good next paper to write.

Ah, that is something like okay, so you've like, Followed all the tenants that I've laid out to like have fast APIs you're like very careful about where you're having called two things. And then you start hitting so brief historical perspective. I've described a bunch of like quadratic or exponential behaviors where like you're hitting.

Areas of the system that aren't scaling nicely. There was a previous generation to this where pretty much everything was quadratic or exponential in like everything and so you couldn't do anything beyond a certain scale because everything would start blowing up on you I see and there was someone before me named George got there who when working on he was the one who led the team at Microsoft Research to you formalized the four color theorem.

I think now not the four color theorem the odd order theorem. In call okay took them about ten years. I think you've mentioned this yeah oh and he went on the cocktails and they fixed these like everything is terrible and everything. So now we're heading like problems that maybe maybe are more fundamental to proof assistance.

But like then you you design your things carefully and you're careful about which parts the system you use and you'd like count for every step. And then you start hitting the next class of problems, which is I have a couple thousand things. A couple thousand variables and I want to introduce them all oops adding a couple thousand like adding n variables is quadratic or cubic in the number of variables that I'm introducing that's unfortunate.

Um, or you're like I want to like change my goal state oops making a new goal state is linear in how many variables there that's sad now. I'm now by running time is quadratic in the number of goal states or something mm-hmm.

And like you hit all of these like the fundamental building blocks. Are too slow. And. That's sort of the next area to investigate of like how do you build a proof assistant so like what are the fundamental building blocks? How are they too slow? Huh the how do we know there are two slow what what are the factors that they're too slow and

like can we show that there's like no way to get anything to actually scale without completely re-implementing the profanion because that's basically what I what I said for program transformation.

I'm like look the existing thing it's quadratic it's real sad let's throw it out and write a new one and stick it in the part of the system that's fast. So like yeah, you can do that for all your proofs you can throw out the entire pure system and write a new one.

But like, Would be nice if you didn't have to do that we say that again. So like you're like, okay, I was trying to do this thing no just the last sentence, oh it would be nice if we didn't have to do that yeah. So the alternative is to the to the alternative is that you figure out what the primitives are what they're too slow and why they're too slow and how do you design a proof assistant like a proof engine with primitives that are actually performant that if you're carefully accounting for all of the primitive steps that you're doing in your proof then you can actually get a proof with reasonable performance.

Like all the things that I've been describing are. You slap something together and it works on small things and then you increase your you try to scale it and it's suddenly stops working because of exponential behavior. And like, Maybe there isn't a hope of fixing that if you slap something together.

But if you're like carefully engineering your proof, you should be able to avoid that. What is the careful part like can you describe that or is it just like the thing so? Okay, so here's how here's how beginners pure things in caulk. Their teacher tells them what they're trying to prove.

They look at what they're trying to prove they look at the list of things they can do they're like, oh I'm trying to prove for all X something. I know a tactic to use. I'm gonna use interest. Oh I'm trying I have a conjunction in my hypothesis that I know a tactic to use I'm gonna use destruct.

I'm trying to prove something about natural numbers, how do I prove something about natural numbers by induction? Where you have this very simple pattern match that are matching program that's running in a brain that you're like how do I do this thing one step at a time? I'm just gonna try a thing and see what works we have some arithmetic, let's try simple let's see if cock knows how to make it simpler there's a tactic called simple without the, Okay, um, sometimes it makes things much nicer.

Sometimes it makes things explode, sometimes it runs forever and gives you nothing it doesn't actually ever run forever pretty much. But running for a year is about as good as running forever.

And so you'll try it and if it works then you're like great it worked. I can keep going

yeah and if it doesn't work then you're like, oh I guess it didn't work, let me try something else instead. And like this is this is how beginners implement proofs and like the way I do proofs is I'm like, okay, let me figure out why this thing should be true.

And let me figure out what gets me closer to my understanding of why it should be true and then I run the same kind of simple program that um that beginners run that's like, oh this should be true by induction on this variable. I'm not just doing induction randomly.

I know why I'm doing induction on what and I'm like, oh I have this conjunction. I can split it apart. I have this disjunction I can split it apart and like I keep making steps and at each point. I'm like, am I still convinced that this theorem is true?

And if I have ever I'm like, oh doesn't these seem like this true anymore that I'd like backup but otherwise I just keep going as long as I'm convinced that the theorem is still reasonable. Where you say something like you do things by figuring out why something should be true is that like.

Is that like constructing approved sketching your head and then doing it versus someone being like oh I know what tactic to implement them, therefore. I will try to construct yeah it's like using a proof method to generate a proof versus knowing what you want to prove and then writing it need to or something oh where is this something different like how does it apply to the engineering case?

I think it's something like that okay, so the thing that I'm doing is I'm like do I believe that this is true when I explain to a very intelligent five-year-old why this is true. And then I'll make steps unlike if at any point. I hit a theorem that I or like I hit a state where I'm like.

This is doesn't seem true anymore. That I'll like back up but I and like I have a big sense of the proof in my head okay, oh but it's like I'm like, okay this this should follow by arithmetic. So then I do a bunch of arithmetic like things and eventually hopefully I get out a thing that's true, but it's like if I want to prove that something is true by arithmetic.

I can just like look at my thing take a step that makes the thing simpler and if the thing still seems true that I'm like great I made it simpler now what and like I can keep taking steps to make it simpler until it's done and I don't have to have a like entire proof in my head.

That's interesting. Don't yeah yeah, this is because I got lined by line feedback on my prefixes. I go along it's great. Yeah. The problem with doing things this way is that they don't scale yeah it seems hard like. It seemed like I feel like with most problems you have to kind of have a proof in your head and then use the syntax to

like.

Make it so let me let me also clarify yeah the sorts of proofs currently that you need to do in caulk or way simpler and the more tedious and the sorts of proofs that you're thinking of here's an example of a proof that you might have to do in caulk, um, this is this is like on the interesting end of proofs okay, oh if you, Have a loop that adds up all the numbers between one and m.

It's the same thing as multiplying n times, m, plus one dividing by two, okay? This is the interesting proof here's another interesting proof, we're merged sword and bubble sort give you the same list if you give them the same list then. In both cases do just do it you you prove so the things you need to prove is you need to prove that they're included or do you just run both things and say no you can't run both things because you need to prove that it's true for every single list, right?

So yeah so the way that you would prove this is you define what it means to be sorted better to be what it means to be a stable sorting of a particular list, maybe you don't need that. I think you can just define what it means to be sorted and what it means for like two lists to be the same up to permutation and you're like for any list there's a unique list, that is the same up to permutation and also sorted.

Look both of these sorting methods produce that list. Okay, oh and like this is at the interesting end the like standard end or things like, um,

If I have a binary tree that holds numbers. And I add one to all the leaves then I take the sum of all the leaves. The number that I get is the number of leaves plus the sum of all the leaves before I added one.

You know, it's these sort of like trivial structural properties.

A lot of time is that proving trivial structural properties. I see so so it's not like you like there's like a it's often not like you're missing a concept and understanding how to generate the proof or something but you need like an elegant way to like structure the proof or something and that's the part where you're saying that the beginner would just be like here's tactic.

I will apply it anywhere like oh what's the good structure to do this or something? I mean, even I'm not what's the good structure to this? I'm like what's what's the structure to do this that isn't wrong? Okay, the beginner is like, Like cargo culting Margo what cargo holding that means?

I maybe it's originated from Richard Feynman and that. There are some places where the. Ah, like livelihood of the tribes depended on like airplane deliveries of cargo. And. Like there was always a ritual associated with the cargo showing up where you like wave the lights in the air so that the airplane can land on the landing strip.

And so you can like there there were some cults so I hear I don't know how accurate this is that developed around this where people would waive the lights in the air hoping that this would make the airplanes in the cargo show up. I see right so you can do the same sort of thing with programming you're like, oh I found the program that does the thing.

I want maybe I can take the code and maybe it'll do the thing. I want. And like you'll also include all the other care and like you're like, why is this other code here? Well, because it was in this other program that did the thing that I want. I don't know if I need it.

So like why are you doing this proof like this because the example proof that I saw had this structure and it worked through the system was like, okay. Yeah, right. So I am I'm at a more advanced level where I'm like, okay, I know that's a proof things about binary trees.

I'm gonna go by induction on the binary tree and I'll do something with the number of leaves and I'll figure the rest of the details out as I go.

And this is how most proofs get done. You do them piecewise. And you like don't account for like how much work call has to do at each step, you're like try it. So work it works then it's good doesn't work that it's bad. And like you could carefully in your head design the entire proof and like carefully account for how much work you expect cock to have to do at each step and make sure that cock shouldn't have to do any work that you don't think it should do.

But very few people designed proofs like this. But it should be possible to get a proof that is fast if you design it like this and like the next wave of performance issues is going to be that even when you're designing proofs like this things are still not fast enough.

And so then that's Cox problem. That's the like, how do you design a proof assistant with good enough primitives? Right, yeah, right. I'm like, look you can just. Throughout throw out the prevention. Do this other thing instead works nicely.

But like it would be nice if you don't have to do that. To get things to scale.

And yeah, that's that's the sort of next step. Right. Now, that's not like a nicer conclusion. Next step is all these very inspiring.

Bibliography

[1] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. "A Compiled Implementation of Normalization by Evaluation". In: *Proc. TPHOLs*. 2008.

[2] Nada Amin and Tiark Rompf. "LMS-Verify: Abstraction without Regret for Verified Systems Programming". In: *Proc. POPL*. 2017.

[3] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. "Towards Certified Meta-Programming with Typed Template-Coq". In: *Proc. ITP*. 2018.

[4] Bruno Barras and Bruno Bernardo. "The implicit calculus of constructions as a programming language with dependent types". In: *FoSSaCS*. 2008. URL: http://hal.archives-ouvertes.fr/docs/00/43/26/58/PDF/icc_barras_bernardo-tpr07.pdf.

[5] U. Berger and H. Schwichtenberg. "An inverse of the evaluation functional for typed λ–calculus". In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. July 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: http://www.mathematik.uni-muenchen.de/~schwicht/papers/lics91/paper.pdf.

[6] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *Proc. CPP*. 2011.

[7] 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.

[8] 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.

[9] 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. URL: http://adam.chlipala.net/papers/PhoasICFP08/.

[10] 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.

[11] 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.

[12] 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. URL: http://adam.chlipala.net/papers/FiatCryptoSP19/.

[13] 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/.

[14] 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.

[15] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *Proc. ICFP*. 2002.

[16] 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. eprint: 1401.7694. URL: https://people.csail.mit.edu/jgross/personal-website/papers/category-coq-experience-itp-submission-final.pdf.

[17] 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. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2018-reification-by-parametricity-itp-camera-ready.pdf.

[18] Florian Haftmann and Tobias Nipkow. "A Code Generator Framework for Isabelle/HOL". In: *Proc. TPHOLs*. 2007.

[19] John Harrison. *Formalized mathematics*. TUCS technical report. Turku Centre for Computer Science, 1996. ISBN: 9789516508132. DOI: 10.1.1.47.8842. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.8842&rep=rep1&type=pdf.

[20] 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.

[21] 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*. ACM, 2009, pp. 207–220.

[22] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (Dec. 2009), pp. 363–446. ISSN: 0168-7433. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[23] Saunders Mac Lane. *Categories for the working mathematician*. URL: http://books.google.com/books?id=MXboNPdTv7QC.

[24] 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.

[25] Gregory Malecha, Adam Chlipala, and Thomas Braibant. "Compositional Computational Reflection". In: *ITP'14: Proceedings of the 5th International Conference on Interactive Theorem Proving*. 2014. URL: http://adam.chlipala.net/papers/MirrorShardITP14/.

[26] Luc Maranget. "Compiling Pattern Matching to Good Decision Trees". In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM. 2008, pp. 35–46. URL: http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf.

[27] Erik Martin-Dorel. *Implementing primitive floats (binary64 floating-point numbers) - Issue #8276 - coq/coq*. Aug. 2018. URL: https://github.com/coq/coq/issues/8276.

[28] Alexandre Miquel. "The Implicit Calculus of Constructions". In: *Typed Lambda Calculi and Applications*. Vol. 2044. Springer. 2001, p. 344. URL: http://www.pps.univ-paris-diderot.fr/~miquel/publis/tlca01.pdf.

[29] Pierre-Marie Pédrot. *Fast rel lookup #6506*. Dec. 2017. URL: https://github.com/coq/coq/pull/6506.

[30] Frank Pfenning and Conal Elliot. "Higher-order abstract syntax". In: *Proc. PLDI*. Atlanta, Georgia, United States, 1988, pp. 199–208. URL: https://www.cs.cmu.edu/~fp/papers/pldi88.pdf.

[31] Tiark Rompf and Martin Odersky. "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs". In: *Proceedings of GPCE* (2010). URL: https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf.

[32] 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`.

[33] Matthieu Sozeau and Nicolas Oury. "First-class type classes". In: *Lecture Notes in Computer Science* 5170 (2008), pp. 278–293. URL: `https://www.irif.fr/~sozeau/research/publications/First-Class_Type_Classes.pdf`.

[34] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. URL: `http://homotopytypetheory.org/book/`.