Extensible Proof Engineering in Intensional Type Theory

A DISSERTATION PRESENTED
BY
GREGORY MICHAEL MALECHA
TO
THE HARVARD SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
NOVEMBER 2014

Thesis advisors: Greg Morrisett & Adam Chlipala Gregory Michael Malecha

Extensible Proof Engineering in Intensional Type Theory

Abstract

We increasingly rely on large, complex systems in our daily lives—from the computers that park our cars to the medical devices that regulate insulin levels to the servers that store our personal information in the cloud. As these systems grow, they become too complex for a person to understand, yet it is essential that they are correct. Proof assistants are tools that let us specify properties about complex systems and build, maintain, and check proofs of these properties in a rigorous way. Proof assistants achieve this level of rigor for a wide range of properties by requiring detailed certificates (proofs) that can be easily checked.

In this dissertation, I describe a technique for compositionally building extensible automation within a foundational proof assistant for intensional type theory. My technique builds on computational reflection—where properties are checked by verified programs—which effectively bridges the gap between the low-level reasoning that is native to the proof assistant and the interesting, high-level properties of real systems. Building automation within a proof assistant provides a rigorous foundation that makes it possible to compose and extend the automation with other tools (including humans). However, previous approaches require using low-level proofs to compose different automation which limits scalability. My techniques allow for reasoning at a higher level about composing automation, which enables more scalable reflective reasoning. I demonstrate these techniques through a series of case studies centered around tasks in program verification.

Contents

Author List

The work described in Chapters 4 and 5 was joint with Thomas Braibant and Adam Chlipala.

The work described in Chapter 7 was work with Jesper Bengston.

To my wife, who is always there with a smile.

And to my parents, who raised me with love and support.

Acknowledgments

I would like to thank my advisors, Prof. Greg Morrisett and Prof. Adam Chlipala, for their guidance and insight related to the work presented here and the other projects that I worked on during my graduate career. Greg has always been full of insights and remarks that helped me find where I was and led me in interesting directions. And it was through my work with Adam that the ideas in this disseratation first emerged.

My undergraduate reasearch advisor, Prof. Walid Taha, provided my first exposure to both computer science research and functional programming. He help lay the foundations of my knowledge which was the seed that started me on the path to graduate school.

My collaborators Ryan Wisnesky, who helped me disentangle "Coq" from dependent type theory, and Thomas Briabant, who helped me understand computational reflection. Jesper Bengston and Josiah Dodds have been great collaborators on $\mathcal{R}_{tac}$ and its application to imperative program verification. Nothing helps make things more useful than users. And I thank the people in Maxwell-Dworkin 309 for many years of interesting conversations, collaborations, and fun.

Finally, I would like to thank my wife, Elizabeth Smoot Malecha, who was and is a constant source of joy and inspiration. And I thank my parents, Michael and Karen Malecha, for constantly teaching, encouraging, and supporting me in my many endeavors.

1

Introduction

Software is a pervasive part of our lives, from the flight control systems on airplanes to "apps" that entertain us on the bus. Despite this, at a formal level, we do not understand what most of it does. Methodologies for software development espouse unit tests and test-driven development as engineering "best practices," but these do not convey a complete understanding of software. Rather, these methodologies extrapolate "correctness" from a collection of observations (test cases) in the style of random variables in statistics where, in reality, there is nothing random about correctness.

While this approach has served us well for many years, its flaws are becoming apparent. In days before network computing there was no incentive to break your software. Now, when global financial markets, utility grids, mass communication, and social media are run by computers, the importance of robust and correct software is more apparent than ever.

"Correctness" is one point on a spectrum of properties about systems. Because "correctness" is a system-dependent property, it is often difficult to build fully automatic tools that can express and prove these properties. Instead, programming languages research has focused, with great success, on shallow properties that can be stated and often proved in a program-agnostic way. For example, type systems, both static and dynamic, can guarantee run-time invariants such as type and memory safety [130]. These properties are so fundamental to abstraction and reasoning that researchers have developed tools to back-port some of these properties [5, 51, 79] to existing software using binary rewriting and instrumented compilers.

However, systems that provide high-level properties out of the box often come at a cost. High-level run-time systems such as Microsoft's .Net [4] and Oracle's Java [2] as well as type safe languages such as OCaml [146] and Haskell [145] prevent or make it difficult to exploit low-level properties of the underlying system. For example, only recently did real-time garbage collection make programming real-time systems in Java possible [20]. And even now, the overhead of features such as garbage collection can drain resources such as power, which prevents us from using these systems to their full potential. In many cases, optimization comes from exploiting problem-specific knowledge, but when working in high-level systems these properties cannot always be represented at the high level and are not always seen by general-purpose compilers. For example, algorithms such as union-find [64] can judiciously use imperative features to obtain asymptotic performance improvements over purely functional code. While this use of imperative features is safe, leveraging it in a purely functional language such as Haskell in a transparent way requires leveraging unsafe features of the language. An escape hatch for the brave is to enter the realm of the "under-specified."

Outside of these very general properties, researchers have developed analyses that automatically reason about software systems. For example, model checking and abstract interpretation provide ways to abstract systems to make exhaustive search possible in very large or unbounded systems. While often completely automatic, as systems grow these approaches become increasingly expensive. Further,

pieces of these systems are property-dependent, with entire areas of research dedicated to building new abstractions to capture specific properties. While frameworks exist for composing these properties, it is not clear that these domains can capture arbitrarily rich properties such as "a path exists through the graph" or "this compiler transformation is correct" while still being completely automated.

These approaches are focused on automated techniques for solving a problem. However, they achieve this automation by sacrificing expressivity. Since shallow properties are often sufficient, in practice this works well, but when more reasoning is necessary to go the extra mile from "safe" to "correct" these techniques often fall short. Take, for example, a program that manipulates matricies. Safety of the program might guarantee only that matrix sizes match up when performing addition or multiplication. Correctness, on the other hand might require reasoning that a particular function correctly implements LU-decomposition, which, in turn, requires stating what it means for a matrix to be lower-triangular.

What is needed to solve these problems is a foundational technique for building *and reasoning* about arbitrarily complex properties. While calculus is the *lingua franca* of most engineering disciplines, logic is the foundation of computer science. The last twenty years has seen a massive growth in automated reasoning systems. Logic tools such as SAT and SMT solvers [24, 68, 76], model checkers, and interactive proof assistants make it possible to approach problems that were previously too large too understand on our own.

In this dissertation I focus on the task of building domain-specific automation within foundational proof assistants. Proof assistants provide rich logical foundations, but their automation necessarily is unable to solve all problems that can be stated. However, the richness of the logic provides techniques for building extremely useful domain-specific automation. In this dissertation I develop a set of abstractions and building blocks for composing this domain-specific automation.

> **Thesis** Open computational reflection in intensional type theories can lower the cost of writing trustworthy, scalable, and customizable automation.

In the rest of this chapter I explain what this means, and in the rest of the dissertation I justify it. I begin with a brief overview of formal logic, presented from a computer science point of view (Section 1.1). Of particular interest is how formal logics capture meaning, which reduces reasoning, especially checking existing reasoning, to an algorithm. From this foundation I explain the term "intensional type theory" (Section 1.2), focusing on its relationship to logic and then what makes it intensional (as opposed to extensional). Finally, I give an overview of computational reflection and how it can facilitate scalable proofs of interesting properties within rich logical theories (Section 1.3.2). I conclude the chapter by outlining the remainder of the dissertation (Section 1.4).

1.1 Formal Logic

A logic is a system of inference rules that describes how to justify a conclusion from a collection of facts. Socrates's *modus ponens* example is probably the best known example of a logical inference:

> All men are mortal **and** Socrates is a man **implies** Socrates is mortal.

This sentence uses the two facts "All men are mortal" and "Socrates is a man" to conclude that "Socrates is mortal."

Making logic "formal" means giving it syntactic (i.e. based on the shape or form) rules for what constitutes a valid inference. In a formal syntax, I would write Socrates' statement as

$$(\forall m, \mathsf{Man}\ m \to \mathsf{Mortal}\ m) \wedge (\mathsf{Man\ Socrates}) \to \mathsf{Mortal\ Socrates}.$$

Matching this up with the English above illuminates the differences. First, I have eliminated as much of the English as possible. While useful as a communication mechanism, the nuances of the English language are difficult to make precise. For example, the previous statement uses both the singular "man" and the plural "men." Only our knowledge of the irregularity of the word "man" connects these two,

syntactically different, objects. As part of eliminating the English, I have replaced statements of being with predicates. For example, I converted "Socrates is a man" to Man Socrates. While less natural to read, the uniformity of the representation exposes high-level structure that will be essential for reasoning.

Continuing in the theme of syntax and building on its uniformity, it is useful to give a syntax to proofs in addition to logical assertions. A convenient way of writing and visualizing logical inferences is using inference rules phrased in the style of natural deduction. As an inference rule, Socrates' statement would be written as:

$$\frac{\mathsf{Man}\ m \to \mathsf{Mortal}\ m \qquad \mathsf{Man}\ m}{\mathsf{Mortal}\ m}\ \text{MP}$$

At the highest level, this rule can be read as an implication named using the text on the right (MP). It states that if the top two statements, "Man $m \to$ Mortal m" and "Man m", are derivable, then the bottom statement, "Mortal m", is also derivable. Free variables, e.g. m, are implicitly universally quantified, allowing us to instantiate the above rule to prove that I am mortal.

$$\frac{\mathsf{Man\ Gregory} \to \mathsf{Mortal\ Gregory} \qquad \mathsf{Man\ Gregory}}{\mathsf{Mortal\ Gregory}}\ \text{MP}$$

We can complete this proof by proving the premises. The left premise (statement above the line) is the interesting part of Socrates' statement, while the right hand side is an axiom of the system. Thus, we can combine the inference rules into the following proof tree.

$$\dfrac{\dfrac{}{\mathsf{Man\ Gregory} \to \mathsf{Mortal\ Gregory}}\ \textsc{Socrates} \qquad \dfrac{}{\mathsf{Man\ Gregory}}\ \textsc{Fact}}{\mathsf{Mortal\ Gregory}}\ \textsc{Mp}\ (\text{Gregory})$$

1.1.1 Machine Checkable Proofs & The Trusted Computing Base

While this level of rigor might seem excessively pedantic, it enables algorithmic manipulation of both propositions and their proof. This in turn enables us to scale reasoning to drastically larger problems by using computers to check, and sometimes even build, the proofs.

The critical factor to consider when trusting justifications attested by a computer is the size of the "trusted computing base." That is, what is the size and complexity of the software that must be trusted in order to trust that a statement is true?

The explicit syntax for propositions and proofs makes it possible to implement a checker for these rules in a relatively simple manner. This checker makes it possible to separate the complexities of finding a proof from the relatively simple process of checking it. Thus a relatively small proof checker can serve as the trusted computing base for arbitrarily large and sophisticated proofs as long as they are expressible in the logic. For example, recent verification results for proving full correctness properties for systems use proofs that are more than 5x the size of the system they verify [85, 96, 98, 103, 110]. Extrapolating these techniques to a multi-million line operating system results in hundreds of millions of lines of proof—far too much to verify by hand. Even a thousand line proof checker results in a 100,000x reduction in the size of the trusted code. This allows us to focus our attention on the checker's correctness so that we can avoid spending time on the details of individual proofs.

The draw-back of a small trusted computing base is the need for very detailed proofs. For example, to avoid trusting a sophisticated algorithm for reasoning about arithmetic, all of the arithmetic manipulation in the proof must be justified within the proof object itself. Abstraction may enable these proofs to be smaller, and automation (which I discuss in Section 1.3) may make it easier to build these proofs, but ultimately these proofs must exist somewhere within the logic.

1.2 Intuitionistic Type Theory as a Logic

There are many possible choices for the logic to use. Within the programming language verification community, intensional type theory has been gaining popularity as a useful, expressive logic. Several factors contribute to this. First, intensional type theory is higher-order, making it easier to state sophisticated theorems. Second, intensional type theory unifies the languages of propositions and proofs by way of the Curry-Howard correspondence (see the next paragraph).

This means that the several levels of syntax are merged into a single level and allows writing very generic properties about both. Beyond the theoretical benefits of the system are the practical ones. Intensional type theory has several implementations [8, 62] and some social processes (e.g. books [33, 56, 133] and active research programs [148]) behind it.

The Curry-Howard correspondence makes precise the relationship between a logic and a functional programming language. The correspondence states the following similarities:

Logical View		**Programming View**
Logic	$\sim$	Type system
Formula	$\sim$	Type
Constructive Proof	$\sim$	Typed Functional Program

To give a concrete example, consider the relationship between logical conjunction and pairs (tuples) in a functional programming language. A conjunction is proved by proving each of the two conjuncts individually, while a pair is constructed by combining the value for each part. In inference rules these are expressed as follows:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge\text{-I} \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \times\text{-I}$$

While the conclusions and premises look slightly more complex because the incorporate a context (Γ), these inference rules mean exactly the same as the simple inference rules that I showed for Socrates' example.

Note the similarity between the logical formulae to the right of the turnstile in the logical realm (on the left) and the types to the right of the colon in the programming realm (on the right). If we think of the Γ's as the same, the only difference is the e_1 and e_2 terms on the right. In the world of programming languages, these correspond to the program, while under the logical interpretation, these terms are the proofs. Intensional type theory's ability to represent and manipulate these proofs within the logic enables it to build procedures that construct proofs and to use proofs to implement procedures.

Intensional vs. Extensional Type Theory

The Curry-Howard correspondence explains how a type system can be viewed as a logic, but what makes the logic intensional? The distinction between intensionality and extensionality lies in the definition of equality that is native to the logical system. In extensional type theories, equality is interested in the objects themselves (i.e. the "extent" of the type). Intensional type theory, on the other hand, is interested in the description of the object. To see the difference concretely, consider two different ways to compute the sum of two integers n and m.

$$n + m \qquad\qquad m + n$$

Since plus is commutative, both expressions have the same value and can therefore be used interchangeably in an extensional type theory. In an intensional type theory, however, these two terms are not *freely* interchangeable. Instead the proof that witnesses their equality is necessary to convert one expression to the other.

The need for the proof as relevant to the *term* highlights the difference between two types of equality: definitional and propositional. Definitional equality (which I will notate using $\equiv$) is the equality that the entire theory internalizes. Within intensional type theory this equality is closely related to program reduction. For example, $(\lambda x.x)\ 3 \equiv 3$ and $1 + 1 \equiv 2$ simply by running the plus function[1].

Propositional equality (which I will notate using $=$) requires proofs to show not only that two terms are equal, but *how* they are equal. In intensional type theory, these proofs are themselves terms in the logic and thus have computational content. For example, a proof that $x = y$ should be viewed as a function that describes how to convert any proof of a property P about x into a proof of P about y.

While intensional type theory distinguishes between definitional and provable equality, extensional type theory unifies them using the equality reflection rule.

$$\frac{\Gamma \vdash A = B}{\Gamma \vdash A \equiv B} \equiv\text{-Ext}$$

[1]Here I use standard lambda calculus notation where $\lambda x.x$ is the identity function that takes any number and returns it.

This rule states that any propositional equality can be converted into a definitional equality. This makes sense in an extensional type theory because the meta theory is concerned with the actual values, and if two values are provably equal it will not be able to distinguish them.

The implication of equality reflection is most clearly seen in inconsistent contexts. In the following examples, $\bot$ denotes a false assertion. The following statement is perfectly valid in an extensional theory but complete nonsense in an intensional one.

$$\bot \to 3 = \text{`}x\text{'}$$

Here, equality requires two things of the same type, but 3 and 'x' clearly have different types since one is an integer and the other is a string. Using $\equiv$-Ext, however, we can prove that

$$\bot \to \mathbb{Z} = \mathsf{string}$$

by appealing to the contradictory premise. Thus the typing derivation for the above statement is the following:

$$\dfrac{\dfrac{\bullet \vdash \bot : \mathit{Prop} \quad \mathit{Eq}\text{-I}\ \dfrac{\bot \vdash 3 : \mathbb{Z} \quad \textsc{Conv}\ \dfrac{\dfrac{\bot \vdash \mathbb{Z} = \mathsf{string}}{\bot \vdash \mathbb{Z} \equiv \mathsf{string}}\ \equiv\text{-Ext} \qquad \bot \vdash \text{`}x\text{'} : \mathsf{string}}{\bot \vdash \text{`}x\text{'} : \mathbb{Z}}}{\bot \vdash 3 = \text{`}x\text{'} : \mathit{Prop}}}{\bullet \vdash \bot \to 3 = \text{`}x\text{'} : \mathit{Prop}}\ \to\text{-I}$$

By internalizing propositional equality into definitional equality, extensional theories provide the ability to write terms such as this directly.

Intensional theories can also express properties such as the one above except that the implicit uses of $\equiv$-Ext in the extensional proof must be converted into an explicit cast in the intensional proof. This means that, for example, instead of stating $3 = \text{`}x\text{'}$ in an intensional theory we would state `3 = cast pf 'x'` where `pf` is the proof that $\mathsf{string} = \mathbb{Z}$ given $\bot$. While seemingly tedious since we think of `cast` as simply being the identity function, considering the content of these proofs leads to interesting structures and is central to current work on proof-relevant mathematics which is championed by the emerging field of homotopy type theory [148].

While witnessing these casts explicitly in proofs may seem inconvenient, intensional theories enjoy several convenient properties that extensional theories do not. First, intensional theories have decidable, syntax-directed type checking because there is never a need to construct a proof out of thin air. Extensional theories, on the other hand, require the entire typing derivation rather than just the term to justify the well-formedness of terms to the proof checker. Second, intensional theories support reduction in inconsistent contexts. To see why this is a problem in extensional type theory, consider the following term:

$$\bot \to (\lambda x.x\,x)(\lambda x.x\,x)$$

Since $\bot$ can be used to prove anything, the term on the right-hand-side of the arrow is well typed; however, reducing it to a normal form will never terminate, thus the proof checker will never terminate. In an intensional theory, the cast necessary to make this term type check depends on the unknown proof of $\bot$ in the context. Thus, reduction stops when this cast needs to be inspected, which prevents the evaluation from diverging.

These theoretical properties, along with the existence of several practical tools [8, 62], have made intensional type theory a popular vehicle for formal developments in recent years. I survey alternatives in the related work at the end of Chapter 2.

1.3 Automation

Building explicit proofs from scratch is often labor intensive. To mitigate the burden, we use automation to build proofs for us. Fundamental limitations on the decidability of certain logics prevent us from solving all problems completely automatically. However, automation still plays a large role in building proofs at scale.

The degree of automation can vary wildly between problem domains. Type inference algorithms, such as those in OCaml [146] and Haskell [145], are excellent examples of a simple but powerful form of automation for proving a shallow property, i.e. the type safety of a program. At the other end of the spectrum, verify-

ing deeper properties—such as the functional correctness of an optimizing compiler [105] or deep mathematical results such as the Odd-Order Theorem [83]—is a much less automatic task. In instances such as these, automation is often relegated to filling in the leaves of the proof while human insight is necessary to guide the high-level proof structure. While not really automation itself, the ability to combine human guidance with automation is extremely useful in developing sophisticated proofs and is a feature unique to proof assistants.

A crucial property of automation is that it should lie outside of the trusted computing base. A proof generated by automation should be just as trustworthy as one that is painstakingly filled in by an expert. This interchangeability allows sophisticated heuristics and complex, highly optimized implementations of automation without the need to worry about compromising the soundness of the system. There are two high-level strategies for building trustworthy automation: have the automation produce a proof object and write a proof that the automation is sound.

1.3.1 Generate and Check Automation

A common automation technique in a rigorous formal system is to use procedures that build proof objects that can be checked after-the-fact by the trusted core. This approach avoids the need to trust, or even formalize, the procedures since the complete justification is captured by the self-contained proof. The independence of the automation from the trusted computing base provides considerable freedom in building this type of automation. For example, automation of this sort can leverage non-determinism, side effects, and even human hints without compromising the trustworthiness of the final proof.

Automation of this variety comes in two varieties: pre-packaged decision procedures that are tuned to solve particular problems and scripting-languages that provide building blocks for building problem-specific automation. In the former category, the Isabelle proof assistant [123] provides automation such as "sledgehammer" that invokes external tools such as SMT [24, 68, 76] solvers. While these external tools are large and may contain bugs, they are built to export proof

sketches that provide a skeleton that can often (but not always) be filled in to produce a full Isabelle proof object. Even in cases when such proof objects cannot be constructed, the results of these external tools can be helpful. For example, model checkers [78] can point out problems in proof obligations that make them false, e.g. detecting an off-by-one error or an incorrect sign in a theorem statement. This early detection can highlight problems with the current development before significant effort is invested in trying to prove a theorem.

Beyond external tools, some proof assistants provide "scripting" languages to help automated the construction of proofs. For example, the Coq proof assistant [62] provides $\mathcal{L}_{tac}$ [69], a domain-specific language for writing backtracking proof search procedures[2]. While often not as fast as external tools, the customizability makes these scripting languages well suited for building simple, domain-specific automation. In fact, the flexibility of $\mathcal{L}_{tac}$ coupled with the richness of dependent type theory has made $\mathcal{L}_{tac}$ a powerful building block for entire tools such as a synthesis engine [73] and numerous verification systems [14, 29, 54, 97] that live within Coq.

"Generate and check"-style automation is especially useful when a large amount of search is done to construct a relatively small proof witness. Classic satisfiability solvers have this property. The proof is simply a valuation of the variables, which can be easily checked, but the search needed to find those values is NP-complete [64]. In cases such as these, the proof search can be done once, and the results can be checked efficiently by any number of clients.

On the other end of the spectrum, some problems are very easily automated but produce very large proof terms. For example, reasoning in algebraic structures with associativity and commutativity often requires large proofs littered with appeals to permutation theorems. In cases such as these, compressing these large proofs can result in a substantial performance win.

---

[2] I discuss $\mathcal{L}_{tac}$ in more detail in Section 2.2.

1.3.2 Verified Automation: Computational Reflection

Within expressive logics that have a notion of computation another approach exists: implement automation within the logic and prove it sound. This style of proving is referred to as "computational reflection" [34] and it is the topic of the remainder of the dissertation. Using computational reflection, the automation does not need to build the problem-specific proof. Instead the general proof of the soundness of the automation can be instantiated to a particular instance and serve as the proof of a property. In essence, computational reflection allows us to extend the proof checker with custom proof checking procedures and use them, to avoid constructing and checking large proof objects. This ability to use computation enables us to exploit domain-specific knowledge to check properties more efficiently.

Example A simple, illustrative example of computational reflection in action is proving the evenness or oddness of numbers. Evenness and oddness can be defined by the following rules.

$$\frac{}{\vdash \mathsf{Even}\ 0}\ \mathrm{E0} \qquad \frac{\vdash \mathsf{Odd}\ n}{\vdash \mathsf{Even}\ (1+n)}\ \mathrm{EO} \qquad \frac{\vdash \mathsf{Even}\ n}{\vdash \mathsf{Odd}\ (1+n)}\ \mathrm{OE}$$

Using these rules it is trivial to prove that any constant natural number is either even or odd: simply apply the appropriate constructor until you reach the base case when the number is 0. Unfortunately while this algorithm is complete, it produces very inefficient proofs. For example, the proof tree justifying that 1024 is even requires 1025 steps.

Abstraction can help us build a shorter proof by showing that $2 \times n$ is even for any n and using that proof with $n = 512$. This solves the problem but only if we can figure out that 512 is the appropriate argument to n. Further, while this approach works nicely for numbers and evenness more complex domains such as propositional logic do not always have similarly nice properties.

To solve the even-odd problem reflectively we write a procedure `isEven` that takes a natural number and returns a boolean. We then prove a derived inference rule stating that if `isEven` returns `true` then the given number is even.

$$\frac{\vdash \mathsf{isEven}\ n = \mathsf{true}}{\vdash \mathsf{Even}\ n}\ \textsc{isEven-sound}$$

Using this rule we can convert a proof obligation of `Even n` into an equality. Leveraging the conversion rule, which is not manifest inside the proof, the logic can reduce `isEven` 1024 to `true` leaving us to prove `true = true`, which has a much shorter proof. Concretely, the large proof tree on the left can be converted into a small proof tree on the right.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{Even}\ 0}}{\vdots}}{\vdash \mathsf{Odd}\ 1023}}{\vdash \mathsf{Even}\ 1024} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{true} = \mathsf{true}}\ \text{=-}\textsc{Refl}}{\vdash \mathsf{isEven}\ 1024 = \mathsf{true}}\ \textsc{Conv}}{\vdash \mathsf{Even}\ 1024}\ \textsc{isEven-sound}$$

Assuming that checking conversion can be done efficiently, checking the small proof on the right will be significantly faster than checking the large proof on the left[3].

This example is purposefully simple. However, building these reflective procedures can be complex, and their scope is often limited. For example, while the technique described here is sufficient to prove that 1024 is even, it cannot prove

$$\vdash \forall x, \mathsf{Even}\ x \to \mathsf{Even}\ (4 + x)$$

even though the proof is arguably simpler. In Section 2.3 I discuss an enriched style of computational reflection that can solve this problem. In the rest of the dissertation I will present and apply my own techniques to make this style of automation more compositional.

A downside of computational reflection, however, is that it requires that the procedure be written and verified within the logic. This often means more upfront effort which is only justified if the automation is broadly applicable. Further, these requirements make some programming features such as side effects and randomness difficult to use since the implementation must fit within the evaluation rules of the logic.

[3]In Coq 8.4pl4 running on a 2.7Ghz Core i7 the proof on the right takes 0.1 seconds to check while the one on the left takes 1.8 seconds, an 18x speedup from using computational reflection.

1.4 Overview of the Dissertation

Large-scale computational reflection is a powerful technique that can be used to dramatically increase the usability of proof assistants. It offers a way to program proof assistants, enabling them to more efficiently reason about large, domain-specific problems while retaining access to rich logics that are essential when proving deep theorems. Rather than verifying stand-alone tools using proof assistants, we can leverage the engineering that makes interactive verification possible to build automation that runs within the proof assistant. This allows us to combine different automation procedures and for those procedures to degrade gracefully in the face of undecidable problems and new domains that pre-packaged tools do not support out of the box. Further, it allows us to accomplish all of these things in a foundational way without enlarging the trusted computing base.

Achieving this goal requires compositionality of automated procedures, something that current techniques for computational reflection in intensional type theory do not support. In the remainder of the dissertation I demonstrate several techniques that make it easier to write, compose, customize, and verify reflective procedures. These techniques lower the up-front cost of computational reflection, making it easier to build automation that scales to large problems.

Before diving into my contributions, I give a brief overview of the Coq proof assistant [62] and computational reflection (Chapter 2). In Chapter 3 I discuss the core technical contributions of the dissertation and how I realize them in the MirrorCore library for compositional computational reflection. I demonstrate the key insights of my techniques using small case studies for tautologies, monads, and generic proof search.

Next, I discuss the engineering decisions that contributed to the development of MirrorCore (Chapter 4). Rich types and first-class proofs provide a range of implementation choices for automation, and I describe some of the trade-offs between performance and verifiability that different choices have. I also discuss future directions that are promising ways to achieve the best of both worlds. In addition, I record some of the tricks necessary to make reflective verification fast

in Coq.

In Chapter 5 I demonstrate the system at scale by showing the degree of automation that my techniques enable for Bedrock [54], a Coq library for low-level imperative program verification. While Bedrock does not leverage MirrorCore directly, many of MirrorCore's techniques grew out of the Bedrock automation.

In Chapter 6, I develop a library of building blocks for assembling reflective procedures that greatly simplifies the verification task associated with computational reflection. In Chapter 7 I return to the task of imperative program verification and show how to combine the building blocks from Chapter 6 with custom automation to build a high-level verification framework within Coq.

I conclude in Chapter 8 by recapping my contributions and looking to future avenues to further the goal of extensible, scalable, and foundational automation.

2

Background

In this chapter I provide an overview of relevant background material. Much of the rest of the dissertation focuses on solving technical problems that arise from technical aspects of formal type theory. These details are in service of producing foundational proofs in a rich logic that is both decidable and efficient to check.

I begin with an overview of intensional type theory focusing in particular on Gallina, the type theory of the Coq proof assistant (Section 2.1). While in the dissertation I will avoid going into many of the technical details of proofs, a basic understanding of them is important to see how computational reflection fits into the larger picture of verification.

The Coq proof assistant also provides $\mathcal{L}_{tac}$ [69], a domain-specific language for generating proofs. In Section 2.2, I discuss the basics of $\mathcal{L}_{tac}$ to provide insight into the standard method of building proofs in Coq. This section also serves as background for the reflective tactic language that I present in Chapter 6.

In Section 2.3 I discuss, in greater detail than in the introduction, computational reflection, the core building block of the dissertation. In particular the tautology prover that I develop illustrates the techniques prior to the work I present in the remainder of the dissertation. I also explain the connection between proofs constructed by computational reflection and those constructed directly (Section 2.3.1). Finally, I cover some of the drawbacks to computational reflection as a proof technique (Section 2.3.2). I conclude the chapter by surveying alternatives to intensional type theory (Section 2.4).

> Much of the remainder of the dissertation is built in the Coq proof assistant. Therefore, the background focuses on the type theory and primitives that Gallina, Coq's logic, provides. In most cases, the ideas translate directly into other higher-order, dependently typed proof assistants/programming languages such as Agda [8].

2.1 Gallina: Coq's Logic

> **Nota Bene** *Advanced Topics in Types and Programming Languages* [131] provides a detailed introduction to dependent type theory, a task better suited for a book then a section of a chapter. This section provides a high-level overview of the core of Coq's logic predominantly for the purpose of highlighting notation and terminology.

2.1.1 The Calculus of Constructions

The logical core of the Coq proof assistant is the calculus of inductive constructions (CiC) [33]. CiC is a dependent type theory, the calculus of constructions (CoC), enriched with inductive definitions and universes.

Ignoring the "dependent" part for just a moment, the simply typed lambda calculus is defined inductively by the following definitions.

$$\begin{array}{rrcl} \text{(types/formulae)} & \tau & ::= & \tau_1 \to \tau_2 \mid \tau_{base} \\ \text{(terms/proofs)} & e & ::= & e_1\, e_2 \mid \lambda x : \tau . e^x \mid x \mid e_{base} \end{array}$$

Here, τ represents types and e represents terms. Types are either function types with domain τ_1 and range τ_2, written $\tau_1 \to \tau_2$, or any of a collection of base types, e.g. $\mathbb{N}$ the type of natural numbers[1].

At the term level, $e_1\ e_2$ represents the function application (call) of e_1 (the function) to e_2 (the argument). λ abstraction builds a function by "abstracting" over a term x of type τ. This abstracted term is referenced using the syntactic form for variables x. I use the superscript e^x to note that the variable x can occur "free" in e and say that references to x within e^x are *captured* by the lambda binder. For brevity, I will abbreviate "$\lambda x{:}\tau_x.\ (\lambda y{:}\tau_y.\ e^{xy})$" by "$\lambda(x{:}\tau_x)(y{:}\tau_y).\ e^{xy}$" and, when the types are clear, I will elide them entirely. Despite the syntactic convenience of eliding them, it is important to note that, in the style of Church, the terms actually carry their types.

Interpreting this language under the lens of logic via the Curry-Howard correspondence means that τ's are logical formulas, i.e. propositions, and e's are proofs. A proposition τ is provable when there exists a value of the type, e.g. 1 is a "proof" of $\mathbb{N}$.

The correspondence becomes more interesting for the function type. A function type corresponds to a constructive implication from the domain type to the co-domain type. With this understanding, function application corresponds to the logical rule of modus ponens that I showed in Chapter 1.

Adding Dependency

Things become more interesting when we wish to reason about terms. In order for logical formulae to mention terms, terms must be embeddable in the type/formula language. The foundation of Gallina is Coquand's calculus of constructions [63], which unifies the languages of types/propositions and terms/proofs in the follow-

[1] I include base types simply so the above language contains programs. Without base types there are no types at all.

ing way.

$$\tau, e \quad ::= \quad e_1\ e_2 \mid \lambda x{:}\tau.\ e^x \mid x \mid \Pi x{:}\tau.\tau^x \mid \texttt{Prop} \mid \texttt{Type}_i$$

Here, Π is the type of dependent functions, i.e. functions whose result types depend on their input. In the remainder of the dissertation I will write $\forall$ in place of Π when the function should be thought of as a proof and Π when it should be thought of as a function[2]. Or, in logical terms, propositions whose conclusions depend on universally quantified values. For example, a proof that all numbers are greater than or equal to 0 would be represented by the following type:

$$\forall n : \mathbb{N}.0 \leq n$$

The final two constructors are the universes in Gallina. `Prop` is the universe of propositions, which is impredicative, which means that it allows quantification over itself, for example $(\forall$ `P`$:$ `Prop`, `P` $\rightarrow$ `P`$)$: `Prop`. Coq also includes an hereditary, predicative heirarchy of universes indexed by natural numbers (the $\texttt{Type}_i$). `Type` being predicative means that types can only quantify over smaller types. For example, $(\forall$ `T`$:\texttt{Type}_i$, `T` $\rightarrow$ `T`$):\texttt{Type}_{i+1}$ but not $(\forall$ `T`$:\texttt{Type}_i$, `T` $\rightarrow$ `T`$):\texttt{Type}_i$. Their hereditary nature of the hierarchy means that it is cumulative, i.e. `x`$:\texttt{Type}_i \rightarrow$ `x`$:\texttt{Type}_{i+1}$. The inclusion of `Prop` is a fundamental difference from Agda [8], another popular implementation of dependent type theory. In the remainder of the dissertation I will elide the subscript on `Type`. Coq internally figures out appropriate universes.

Definitional Equality

Given its status as a programming language, the calculus of constructions has the same reduction rules as the lambda calculus. Within the dependent type theory, these reductions form the definitional equality in the system.

---

[2] In reality, Π and $\forall$ are synonymous in Gallina.

The primary interesting feature of definitional equality for computational reflection is that it occurs for free in proof terms. While it does take time to reduce terms, all type checking in Coq is done modulo definitional equality without any explicit proof terms stating how to perform reduction. This is important for computational reflection because the reflective procedures that I will write require thousands or millions of reductions to execute. Witnessing these reductions with explicit sequences of reduction steps would be prohibitively expensive in proof terms and would eliminate much of the benefit of computational reflection.

The reason why definitional reduction can be free without sacrificing decidable type checking is that convertibility of Gallina terms is decidable. This means that given any two well-typed Coq terms it is always possible to check whether the two are definitionally equal.

The downside of this restrictive, built-in notion of equality is that it is one of the few things in Gallina that is second-class. There is no way within Gallina to state that two terms are definitionally equal; they either are or they are not. In Chapter 3 I will present a powerful technique for circumventing this restriction in some circumstances.

Inductively Defined Data Types

The main extension in Gallina comes from inductively defined types. Inductive types are a way to extend Gallina with new types and values. For example, we can define a type representing natural numbers in unary using the following inductive definition.

```
Inductive ℕ : Type := O : ℕ | S : ℕ → ℕ. (* also written "S (_ : ℕ)" *)
```

In English, the inductive type $\mathbb{N}$ is freely generated by two constructors: `O` (representing 0) of type $\mathbb{N}$, and `S` of type $\mathbb{N} \to \mathbb{N}$ (representing one more than the number it is applied to). For example,

$$O = 0 \qquad S\,O = 1 \qquad S\,(S\,O) = 2$$

Constructing this type inductively corresponded to taking the least fixed-point of a function that generates the type. To ensure that the least fixed-point exists, Gallina places certain restrictions on the types of the constructors. In Gallina, the key restriction is that the recursive type must only occur in strictly positive positions (i.e. it cannot be mentioned to the left of an arrow) in the constructor's argument types. For example, Gallina rightly rejects the following inductive type

```
Inductive Bad := bad : (Bad → False) → Bad.
```

since it can be used to prove `False`. I will return to this point in the related work at the end of Chapter 3.

In addition to the constructors, used to build values, inductive types also induce two more features: pattern matching and structural recursion.

PATTERN MATCHING (CASE ANALYSIS) In the context of logic, pattern matching corresponds to case analysis and is written mimicking the syntax of OCaml. For example, the following code computes the predecessor of n or returns 0 if n is 0.

```
match n with
| O ⇒ O
| S n'' ⇒ n''
end
```

Since types can depend on values, it is important to be able to refine the type based on the value of the term being matched on (the discriminee). To do this, Gallina's pattern matching construct has dependency annotations `as` and `return`. The typing rule is quite complicated, but an example is illustrative.

```
match n as n' return n' ≥ 0 with
| O ⇒ ... : O ≥ 0
| S n'' ⇒ ... : (S n'') ≥ 0
end : n ≥ 0
```

Here, the `n'` after `as` is a binder which is referenced by the `return` clause. On the "outside" the `n'` in the return clause is substituted for the discriminee, meaning that

the entire `match` expression has the type `n≥0` (shown via the type ascription after the `end`). In each branch, however, some information is learned about the value *n*, allowing a more precise type to be given. In the first branch, the code knows that *n* is 0, so the type of . . . in the first branch should be $0 \geq 0$. Similarly, the type for the second branch should be `(S n'') ≥0`. Note that in both branches *n* is still in scope with the same type.

Recursive Functions (Induction) In addition to dependent pattern matching, inductive data types also admit structural recursion. This is introduced by the `Fixpoint` syntax (or `fix` when it is an expression). For example, the following is a definition of plus.

```
Fixpoint plus (n m : ℕ) {struct n} : ℕ :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
```

The need for soundness makes recursive functions in Gallina substantially more complex than they are in normal programming languages. Termination is necessary because divergent functions could be used to "witness" proofs of False. For example, without termination the following is a proof that `True → False`:

```
fix bad (x : True) : False := bad x.
```

In Gallina, termination is guaranteed extra-logically using a syntactic restriction on fixpoints that checks that all recursive calls are to syntactically structurally smaller terms. In the above implementation of `plus`, addition is defined on natural numbers by recursion on the first argument (denoted by the `{struct n}` annotation).

In addition to the somewhat restrictive definition rules, there is also a subtlety in defining reduction of fixpoints. Definitional equality is only allowed to unfold a fixpoint if the argument on which the function recurses (e.g. the *n* in `plus`) starts with a constructor, e.g. `O` or `S`. This restriction is necessary to retain decidable conversion.

Indexed Inductive Data Types Beyond the simple definition of $\mathbb{N}$ above, Gallina also supports indexed inductive types, sometimes called generalized algebraic data types or GADTs. The most common indexed data type is the one representing equality.

```
Inductive eq {T : Type} (a : T) : T → Prop :=
| eq_refl : eq T a a.
(* [eq T a b] is written [a = b] since [T] can be easily inferred.
 * The implicitness of [T] is notated by the { }'s *)
```

At an intuitive level, this type expresses that terms are only equal if they are syntactically identical. Here, the arguments `T` and `a` are "parameters" while the unnamed `T` to the right of the `:` is an "index." The main difference between parameters and indices is that parameters must be constant in the return types of all of the constructors while the indices of the inductive type can vary. In the type of `eq_refl` above the parameter is repeated in the index position stating that you can only prove $a = a$ for any value a. Note, however, since definitional equality is implicit in Coq, we can still prove, e.g. 3+2 = 5 since $3 + 2 \equiv 5$.

Pattern matching on inductive types with indices enables us to refine the type based on the knowledge that we learn about it in each branch of the pattern. To support this, Coq provides an `in` clause which provides binders for the indices of an inductive data type in the return type of a dependent match. For example, the following proof term uses a proof of equality to "rewrite" `b` into `a` in the type `b`$\geq$0.

```
match pf : a = b in _ = b' return b' ≥ 0 with
| eq_refl ⇒ ... : a ≥ 0
end : b ≥ 0
```

Here, the `in` is binding values from the *type* of `pf`. Again, on the outside `b'` unifies with `b` producing the type `F b`. On the inside, however, the `eq_refl` constructor uses `a` for both indices, so `b'` unifies with `a`, thus the ... requires a value of type `a`$\geq$0.

On top of Gallina, Coq provides a number of conveniences to help manage abstraction and build proof terms. I already showed implicit arguments as with the `T` in the type of `eq`. When they are important, I will notate them using subscripts on the function. For example, if I wish to call out the type of an equality I would write $\texttt{eq}_{\mathbb{N}}$ to represent equality of natural numbers. In more literal contexts where I explicitly spell out all arguments I will use Coq's standard `@` notation, e.g. `@eq` $\mathbb{N}$. Coq also provides top-level definitions using the syntax `Definition` (or `Fixpoint` if the function is recursive). When definitions should be thought of as local definitions I will use `Let` instead of `Definition`.

Coq also provides a section mechanism for implicitly quantifying many terms by the same set of unchanging arguments. Take lists, for example, which are polymorphic in the type of elements. Using sections and variables, lists and the length function can be defined as follows.

```
Section lists.
  Variable T : Type.
  Inductive list : Type := nil : list | cons : T → list → list.
  Fixpoint length (l : list) : ℕ :=
    match l with
    | nil ⇒ 0
    | cons _ l' ⇒ S (length l')
    end.
End lists.
```

Line 2 implicitly parameterizes the remaining definitions in the section by the variable `T` which is a type. For example, the above code is the same as:

```
Inductive list (T : Type) : Type :=
| nil : list T
| cons : T → list T → list T.
Definition length (T : Type) : list T → ℕ :=
  (fix length (l : list T) : ℕ :=
```

```
  match l with
  | nil ⇒ 0
  | cons _ l' ⇒ S (length T l')
  end).
```

In the dissertation, I will freely use `Variable`, but I will elide opening and closing sections.

2.2 Constructing Proofs with $\mathcal{L}_{tac}$

Section 2.1 described the language of proofs in Coq. All proofs in Coq have corresponding Gallina proof terms; however, writing proofs directly in Gallina can be cumbersome. To facilitate writing proofs, Coq comes packaged with a built-in tactic language, $\mathcal{L}_{tac}$ [69], that interprets "proof-like" commands and generates the corresponding Gallina proof terms.

The easiest way to explain $\mathcal{L}_{tac}$ is by a simple example. Consider proving the following statement about addition.

```
Goal ∀ n : ℕ, n + 0 = n.
```

Since addition is defined inductively on its first argument, which is a variable, the proof follows from induction on n. Induction is a combination of a fixpoint and a dependent pattern match. We could write this by hand, but in $\mathcal{L}_{tac}$ we can generate the Gallina code using the tactic `induction n`.

This results in two goals, one for the `O` case, and one for the `S` case.

```
Goal 1
  ========================
  0 + 0 = 0
Goal 2
  n : ℕ
  IHn : n + 0 = n
  ========================
  (S n) + 0 = (S n)
```

Focusing on the first goal for a moment, we can ask $\mathcal{L}_{tac}$ to simplify the expression producing the new goal: `0 = 0` which is solved using `eq_refl`. Using tactics[3]:

```
simpl. reflexivity.
```

This solves the first goal, leaving us with only the inductive case. Applying simplification reduces the call to `plus`, which changes the goal to the following.

```
S (n + 0) = S n
```

This goal exposes the ability to use the inductive hypothesis to rewrite the `n + 0` into `n`. Again, we could write the dependent match with the `in` clause, but instead we can have $\mathcal{L}_{tac}$ generate the code using the `rewrite` tactic. `rewrite IHn` generates a snippet of proof behind the scenes and leaves us to prove:

```
S n = S n
```

which is solved by `reflexivity`.

While we never wrote the proof term explicitly, we can ask Coq to show us the term that $\mathcal{L}_{tac}$ generated.

```
(fun n : ℕ ⇒
 ℕ_ind (fun n0 : ℕ ⇒ n0 + 0 = n0)
   eq_refl                                  (* Base case *)
   (fun (n0 : ℕ) (IHn : n0 + 0 = n0) ⇒ (* Inductive case *)
    eq_ind_r (fun n1 : ℕ ⇒ S n1 = S n0) (* rewrite IHn *)
    eq_refl IHn) n)
```

This proof uses defined functions rather than the primitive `fix` and `match` for doing both induction (`ℕ_ind`) and rewriting (`eq_ind_r`) but ultimately, the term reduces definitionally to the following term.

```
fun n : ℕ ⇒
  (fix F (n0 : ℕ): n0 + 0 = n0 :=
     match n0 as n1 return n1 + 0 = n1 with
```

[3]The `simpl` is optional since reduction in Coq occurs automatically during type checking.

```
  | 0 ⇒ eq_refl
  | S n1 ⇒
    match match F n1 in _ = y return y = n1 + 0 with
          | eq_refl ⇒ eq_refl
          end in _ = y return S y = S n1
    with
    | eq_refl ⇒ eq_refl
    end
  end) n
```

While this proof was generated completely using $\mathcal{L}_{tac}$, it is quite similar to a hand-constructed proof (not shown) of the same fact. In general, tactic-based proving produces less efficient proof terms than hand constructed terms. For this reason all functions that are meant to be computed on in the remainder of the dissertation are defined manually. Only proofs that are never reduced are defined using tactics.

2.3 Proof by Computational Reflection

I discussed the high-level ideas of computational reflection in Section 1.3.2. In this section I dig a little deeper with an example based on deciding tautologies.

The intuition of computational reflection is to prove a property by writing a function and using a soundness theorem about it coupled with the internal reduction mechanism of the logic to build the proof. At the high level a propositional tautology prover and its soundness lemma might look like this:

```
Definition rtauto' (H: list Prop) (G: Prop) : bool := ...

Theorem rtauto'_sound : ∀ (H: list Prop) (G: Prop),
  rtauto' H P = true →
  Forall id_Prop H → (* H_1 ∧ H_2 ∧ ··· ∧ H_n *)
  G.
```

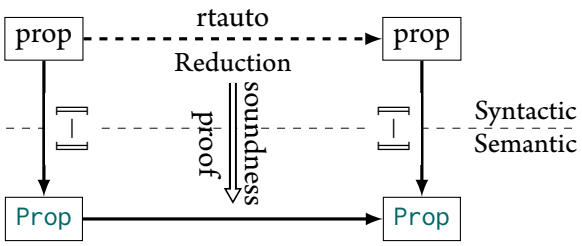

Figure 2.1: The basic components of computational reflection. The dashed line separates the semantic world (bottom) from the syntactic universe (top). $[\![-]\!]$ is the denotation function that connects the two. `rtauto` is the procedure that performs the reasoning, and `soundness` is the proof that translates the reduction into a proof object.

Intuitively, the `rtauto` function takes in a list of known facts `H: list Prop` and a goal `G: Prop` and returns `true` if `G` is provable under the conjunction of the facts in H.

While the above definitions are straightforward, there are no interesting implementations of `rtauto'` that satisfy the specification. This is because Gallina provides no way to inspect values of type `Prop`. That is, `rtauto'` cannot match on `G` or any of the values in `H` to see how they are built.

The standard technique to enable the function to inspect the goal is to introduce a level of indirection by defining a syntax that represents the structure of the propositions that we wish to inspect. Figure 2.1 shows the solution graphically. In the figure, the syntactic type `prop` represents a subset of all possible `Prop`s. A denotation function—represented in the figure using Oxford brackets ($[\![-]\!]$)—makes the meaning of the syntax precise by computing a proposition for each piece of syntax.

Figure 2.2 shows how this works using an inductive type and a denotation function. The `prop` inductive type provides constructors for each type of propositional fact that the automation will reason about. The denotation function (`propD`) shows how these syntactic objects map directly to their semantic meanings (lines

```
1  Inductive prop :=
2  | pRef (n : index)
3  | pTrue | pFalse
4  | pAnd (p q : prop) | pOr (p q : prop) | pImpl (p q : prop).
5
6  Variable ps : map index Prop.
7
8  Fixpoint propD (p : prop) : Prop := (* ⟦p⟧ *)
9    match p with
10   | pRef i ⇒ get i
11   | pTrue ⇒ True
12   | pFalse ⇒ False
13   | pAnd l r ⇒ propD l ∧ propD r
14   | pOr l r ⇒ propD l ∨ propD r
15   | pImpl l r ⇒ propD l → propD r
16   end.
```

Figure 2.2: Syntactic representation of propositional logic.

9-13). The `pRef` constructor is used to represent uninterpreted symbols which `rtauto` will know nothing about. The denotation of these symbols is provided by the `ps` map that is passed to the `propD` function using the section variable mechanism.

Adapting `rtauto` and its specification to use the syntactic propositions is simply a matter of replacing `Prop` with `prop` and using the denotation function to convert the syntactic values of type `prop` into semantic values of type `Prop` in the soundness theorem.

```
Definition rtauto (H: list prop) (G: prop) : bool :=
  ...
Theorem rtauto_sound : ∀ (H: list prop) (G: prop),
  rtauto' H P = true →
  ∀ (ps: map index Prop),
    Forall (propD ps) H → (* propD ps H1 ∧ ··· ∧ propD ps Hn *)
    propD ps G.
```

Since Gallina can match on values of type `prop`, we can now write, and prove sound, useful implementations of `rtauto`. Figure 2.3 shows a flavor of a simple

```
Fixpoint rtauto (hyps: list prop) (g: prop) : bool :=
  match g with
  | pTrue ⇒ true
  | pAnd g1 g2 ⇒ rtauto hyps g1 && rtauto hyps g2
  | pImpl g1 g2 ⇒ rtauto (learn g1 hyps) g2 (* assuming g1, prove g2 *)
  | g ⇒ findAssumption hyps g
  end.
```

Figure 2.3: A partial, simplified implementation of a reflective tautology prover. The `learn` function breaks conjunctions in the term apart and adds the conjuncts to the list of hypotheses. The `findAssumption` function searches for the proposition in the list of hypotheses.

reflective procedure. Note that the code relies crucially on the ability to inspect the input through pattern matching. Achieving this in an extensible way will be the topic of Chapter 3.

2.3.1 Computational Reflection and Proofs

While we often think of proofs as being "extra" objects that do not affect computation, proofs are actually an integral part of the computation in Gallina. For example, when we perform a rewrite we are matching on a proof term, therefore, computationally, there must be a proof to match on. How does this idea reconcile with one of the core motivations of computational reflection, to *not* generate proof terms?

To answer this question, consider the proof term for a simple tautology.

$$\vdash P \wedge Q \rightarrow Q \wedge P$$

In Gallina, a simple (manually constructed) proof term for this goal is the following[4]:

```
fun PQ: P ∧ Q ⇒ match PQ with
```

[4] `conj` is the constructor for the proof of a conjunction.

```
                | conj pfP pfQ ⇒ conj pfQ pfP
                end
```

The proof is a function that takes the proof of `P ∧ Q`, matches on it to extract the independent proofs of `P` and `Q`, and then reassembles them into a proof of `Q ∧ P`.

On the surface, the proof by computational reflection is quite a bit different, and longer.

```
rtauto_sound [] (* Hypotheses *)
   (pImpl (pConj (pRef 0) (pRef 1))
          (pConj (pRef 1) (pRef 0))) (* Goal *)
   eq_refl (* Computation proof *)
   {0 ↦ P; 1 ↦ Q} (* Environment *)
   Forall_nil (* Proof of hypotheses *)
```

The verbosity here is predominantly due to the need to repeat the problem in its syntactic form in order to call `rtauto_sound`. As the amount of reasoning increases relative to the size of the goal, the cost of repeating the goal becomes negligible compared to the size of the proof.

In the reflective proof, the call to `rtauto_sound` hides the constructors that were apparent in the manual proof. But, leveraging the constructive nature of Gallina's proofs, the reduction mechanism can cut through all of the abstraction and compute a proof that is not much different than from the hand-coded proof above.

```
fun (P Q : Prop) (x : P ∧ Q) ⇒
       conj_{P,Q} match x with
                  | conj _ x1 ⇒ x1
                  end match x with
                      | conj x0 _ ⇒ x0
                      end
```

The similarity to the manual proof is not surprising. Our soundness proof encodes exactly how to construct this proof for any particular problem instance. Further, since the soundness theorem is proved by structural induction on the reified syntax

it is not surprising that it is completely gone after reduction since the syntactic representation contains no opaque terms that would block reduction.

While the proof is simple, it requires some care to achieve the right computational properties. In particular, it is important to keep in mind that some proofs are universally quantified, which will block pattern matches, while others are built by the proof of soundness and can be freely pattern-matched on. Writing the soundness theorem in a continuation-passing style makes it possible to generate exactly the hand-crafted proof that uses a single `match` on the proof of `P∧Q`. While the proof terms do not match up exactly, they are provably equal.

2.3.2 The Drawbacks of Computational Reflection

While powerful and broadly applicable, computational reflection is not without some drawbacks. First, applying it requires *a priori* knowledge of all facts that could be used. This often results in building reflective procedures that require more facts than they actually need to prove any particular problem instance. While perfectly acceptable, this introduces false dependencies and might prevent generalization to situations where those false dependencies do not hold. For example, when proving `P → Q → P`, the proof of `P` is not necessary and, in a manually constructed proof, it would be ignored. In the reflective proof, on the other hand, the proof of `Q` would be passed to the reflective soundness theorem making it appear to be necessary to prove `P`. If the soundness proof of the reflective procedure is coded with care, these extraneous dependencies can be eliminated by completely reducing the proof as I showed in the previous section.

Second, computational reflection relies, in a fundamental way, on computation that can be reasoned about within the logic. In order to be efficient, this computation must be fast; however, the engineering effort necessary to make reduction fast often compromises simplicity and thus increases the size of the trusted computing base. In Coq, this trade-off manifests itself in the `vm_compute` [84] mechanism which performs fast reduction by compiling Gallina to a virtual machine and running the code there. However, neither the virtual machine nor the compila-

tion process is formalized, but both lie in the trusted computing base. A simpler implementation of reduction exists and is easier to trust but orders of magnitude slower.

Third, proofs by computational reflection often, but not always, mix proof search with proof objects. In cases where reflective procedures explore a space of potential proofs this exploration can make checking a proof as expensive as finding it. Techniques exist for easing this burden which I discuss in the related work of Chapter 6.

Finally, and related to the former point, the soundness proof (which might contain the heuristic search) is embedded directly in the proof. In cases where proprietary heuristics and procedures must be kept secret, proofs produced using them cannot be distributed in their efficient form. In dependent type theories, it is always possible to remove these generic lemmas from a development by inlining and reducing them. While this manifests the large proof object that we sought to eliminate by using computational reflection, it can easily be done off-line.

The last two points are addressed by using computational reflection to implement reflective proof checkers that fill in the boring details of high-level proofs. When designed this way, the high-level proof can be constructed by proprietary algorithms, and only their output needs to be included in the final, published proof. Since the high-level proof sketch exists explicitly in the proof it must be checked for well-typedness, which can be a burden if the sketch is long.

2.4 Related Work

In this chapter, I discussed the core underpinnings of the remainder of the dissertation. The related work incorporates the entire rest of the design space for proof assistants and automation techniques. I delay a comprehensive discussion of work related to computational reflection until the end of Chapter 3.

Proof Assistants Coq is one of many proof assistants and represents a single point in an incredibly large design space. A thorough evaluation of all the cur-

rent proof assistants[5] and dependently typed programming languages would be too much to include so I highlight some representative examples.

Agda [8] is perhaps the mostly closely related proof assistant to Coq and, while substantially younger, it has matured quite rapidly in comparison. While Coq picks a single, conservative point in the design space, Agda is parameterized by a range of choices enabling it to be used for exploring the ramifications of different choices. Some of the features relevant to this dissertation include: size types [6], which incorporate the guardedness check of structural recursion into the type system making recursion more modular and less syntactic; irrelevence [7], which can be useful to erase values that are not needed for reduction; and universe polymorphism, which allows explicit quantification over universes. Agda's support for positivity polymorphism, is of particular interest to extensible computational reflection because it offers a method to build inductive data types using Swierstra's technique from "Data types a. la. carte" [144]. I return to this observation in Chapter 3.

Adga's approach to proving is much more closely related to dependent type theory. It is only relatively recently that any work [74, 149] has investigated automation techniques such as computational reflection in the context of Agda. Due to the heavy use of dependent types in Agda, this work must address a harder representation problem than my own work solves. A big difference between the work in this dissertation and the work in Agda is the applications. My work has been used to automate large proof procedures with good performance. Much of the work on computational reflection in Agda suffers from a slower internal reduction mechanism and proof development methodology that stresses extremely fast type/proof checking.

Recent work inspired by homotopy type theory has started an investigation into theories with first-class definitional equality. The Andromeda proof assistant [27] implements one proposal for this idea. The ability to capture both intensional

[5]Including Zombie [50], Elf [128], Twelf [129], F* [143], L∃∀N [100], Dafny [104], Andromeda [27], Coq [62], Agda [8], Epigram [114], HOL [88], Isabelle [123], Nuprl [9], and PVS [126] just to name a few.

and extensional equality in the same system has significant potential for overcoming some of the limitations of intensional type theory, though it also introduces a variety of interesting questions and may make efficient reduction considerably more difficult or impossible. Further, like extensional type theories, Andromeda's proofs are not decidable to check from terms alone.

In the wider world of dependent type theories, the Nuprl proof assistant [9] implements an extensional type theory. The core of Nuprl lies on top of partial equivalence relations, universes, intersections, numbers, computational approximation, and equality at a type [12]. As I discussed in the introduction, extensional type theories also identify propositional and definitional equality. This makes type checking of terms undecidable in general though the Nuprl system has a considerable amount of automation built specifically to make the process of building proofs reasonably efficient. This insulates the developer from the corner cases that inevitably arise in the more expressive logic. For example, β-reduction is only sound in consistent contexts. Second, it is not as context-free as Gallina. For example, the term $F\,X$ may be a well-typed Nuprl term while F alone is not well-typed.

There are also proof assistants not directly built on type theories. Isabelle [123] is a generic proof assistant that can be applied to many logics. Its core insight is the use of a meta-language for constructing proof objects in a library-like fashion. For example, while Coq generates an explicit proof object, Isabelle's common implementations simply exposes an opaque type of "proofs" that are constructed via library functions. Those functions check that the terms match up on the fly, e.g. Isabelle would reject a construction like "x + True." One of the primary benefits of the library approach to a proof assistant is the ability to avoid re-implementing the machinery for performing proofs. ML code can be written to interact directly with the proof assistant kernel, which makes it easy to make use of computational resources such as parallelism and other system programs to perform proof search. The ability for these procedures to directly call Isabelle's kernel allows them to remain sound even in the presence of these sophisticated heuristics. Further, the fact that proofs are second-class in Isabelle means that the proof terms never need to be stored, thus making the proof process more memory efficient. At the same

time, however, this design eliminates the opportunity for writing an independent evaluator to check the proofs.

HOL [88], which stands for "higher-order logic," is an implementation of the proof rules for higher-order logic. It builds directly on top of eight rules for forming proofs: assumption , reflexivity, β-conversion, substitution, abstraction and type instantiation. β-reduction provides a way to do term simplification, for example allowing a proof of $\vdash 1+1=2$ to be reduced to a proof of $\vdash 2=2$. The ability to perform reduction within the logic is the essential component to making computational reflection possible. The fact that reduction steps are made explicit partially gets in the way here; however, reduction is actually quite fast in HOL since, like Isabelle, HOL does not produce proof terms.

Stampoulis's VeriML [139, 140] augments the logic to do *a priori* verification of tactics. The critical questions that VeriML answers are how to support pattern matching on terms and how to pre-check proof terms that automation produces. My work on $\mathcal{R}_{tac}$ in Chapter 6 bears some similarity to Stampoulis's work by defining a language of verified tactics within the logic. VeriML's logic does not handle the richness of dependent types, but it does support a higher-order logic similar to HOL's.

ACL2 [1] is probably the largest commercial success in program verification with uses in a variety of industry projects. For example, Intel's floating point unit has been verified in ACL2 since the notorious floating point bug [93]. ACL2 is built on the ideas of the Boyer-Moore verification infrastructure and is highly automated and customizable. The core logic is untyped and based on Lisp [141]. As with Isabelle and HOL, ACL2 does not construct proof terms and, unlike Isabelle and HOL, the trusted computing base is quite substantial both in terms of size and complexity.

The Milawa theorem prover [66] is like a foundational version of ACL2. Milawa has a core theory similar to ACL2's but then verifies each level of automation on top of previous layers. This approach is similar in many respects to computational reflection all the way down. From a trusted-computing-base point of view, Milawa is also quite impressive. The implementation runs on top of a verified runtime

system [120], and the implementation is itself verified within HOL [121]. The untyped nature of the logic greatly simplifies this verification.

Programming with Dependent Types Researchers have explored dependent types in several contexts outside of proof assistants. ATS [153], and its predecessor Dependent ML [154] (DML), incorporates dependent types into a semi-functional programming language. ATS also supports linear types which allows, it to perform imperative updates to functional data structures, which saves on memory and improves execution time.

Idris [44] grew out of the ideas in Agda and has been used to explored the more programming-oriented features of dependent types. It is one of the few dependently typed programming languages with a compiler [43]. To interact with the outside world, Idris has championed an effect system which has a variety of compositional properties that interact well with dependent types.

One of the major shortcomings of full dependent type systems is the need for strongly normalizing programs. The Trellys [50] project addresses this in an interesting way using a call-by-value semantics rather than Gallina's very liberal reduction semantics (which is only valid because its terms are both strongly normalizing and side-effect free). To get around issues with non-termination, β-reduction is explicit in proof terms using a syntactic form that is explicitly indexed by the maximum number of steps to run the computations before giving up. While β-reduction is witnessed explicitly in proof terms, casts using equalities are implicit, and the type checker uses congruence closure to decide whether two terms are equal [136].

3

Open Semantic Reflection

In Chapter 2 I fleshed out a tautology solver using standard techniques in computational reflection. The problem with this approach is that it relies on a fixed syntactic representation that is closed to further extension. This limitation requires that reflective procedures be written and proved with the entire reified language in mind. For example, enriching the propositional language with support for quantifiers requires a dramatic overhaul to both the term representation and its denotation function. Even less aggressive changes such as adding natural numbers and equality have a substantial impact on the amount of code needed. In this latter case, the code is related to multiple syntactic universes, one for propositions and one for numbers. When these features are not strictly layered, the syntactic representations become mutually inductive, which further complicates matters.

In this chapter I describe my solution to this problem. The core idea is to build a universal representation that can compose domain-specific symbols. Since the

goal is to reason within type theory, I take the simply-typed lambda calculus as this universal representation. This representation forms the foundation of MirrorCore [108], a Coq library for open semantic reflection.

In the remainder of the chapter I focus on the core representation of MirrorCore, motivating the choices with small case studies showing the features in action. The lambda-calculus core provides an extensible language of types and a generic representation of binders suitable for interleaving many domain-specific symbols (Section 3.1). These domain-specific symbols are included in the syntax parametrically, allowing after-the-fact extension via environments (Section 3.2). Next I revisit the representation technique generalizing it to handle second-class type constructors, polymorphic functions, and dependent types (Section 3.3). Using these techniques I develop automation for performing equational reasoning about monadic computations. Finally, I discuss unification variables and the role they play in building generic automation (Section 3.4). I conclude by surveying work related to deep embeddings of languages and unification (Section 3.5).

3.1 The Lambda Core

The core of MirrorCore's term representation is the simply typed lambda calculus enriched with base types and base symbols.

$$\begin{array}{lrcl} (\textit{types}) & \tau & ::= & \tau_1 \to \tau_2 \mid T \\ (\textit{terms}) & e & ::= & e_1\ e_2 \mid \lambda \tau.e^x \mid \#n \mid E \end{array}$$

The first line gives the representation of types which contains arrow types (represented $\tau_1 \to \tau_2$) and base types (represented with T). At the term level, expressions include function application, lambda abstraction, variable reference (notated using $\#n$), and base terms. The syntax encodes variables using de Bruijn indices [67], numbers corresponding to the number of binders to skip before the binding site. Figure 3.1 shows how these definitions map to Coq inductive data types.

```
Inductive typ : Type := (* the syntax of types *)
| tyArr : typ → typ → typ | tyInj : ℕ → typ.

Variable ts : list Type. (* meanings for injected types *)

Fixpoint typD (t: typ) : Type := (* the denotation of types *)
  match t with
  | tyArr l r ⇒ typD l → typD r
  | tyInj n ⇒ lookup_env ts n
  end.

Inductive expr : Type :=
| App : expr → expr → expr
| Abs : typ → expr → expr
| Var : ℕ → expr
| Inj : ℕ → expr.

Variable fs : list { t: typ & typD t }. (* meanings for injected terms *)

Definition ExprT (tvs: list typ) (T: Type) : Type :=
  hlist typD tvs → T.

(* the denotation of terms *)
Fixpoint exprD (tvs: list typ) (t: typ) (e: expr)
: option (ExprT tvs (typD t)) :=
  match e with
  | App f x ⇒ domT ← typeof_expr tvs x ;
              valF ← exprD tvs (tyArr domT t) f ;
              valX ← exprD tvs domT x ;
              Some (fun vs ⇒ (valF vs) (valX vs))
  | Abs t' e ⇒
    match t with
    | tyArr d r ⇒
      cast ← type_cast ts d t' ;
      val ← exprD (t' :: tvs) r e ;
      Some (fun vs ⇒ fun x ⇒ val (Hcons (Rcast_val cast x) vs))
    | _ ⇒ None
    end
  | Var v ⇒ (t,get) ← nth_error_get_hlist_nth tvs v ;
            cast ← type_cast t' t ;
            Some (fun vs ⇒ Rcast_val cast (get vs))
  | Inj i ⇒ val ← funcAs i t ;
            Some (fun _ ⇒ val)
  end.
```

Figure 3.1: MirrorCore's syntactic representation and denotation function.

I formalize the meaning of the syntactic objects using a pair of denotation functions, one for types (`typD`) and the other for terms (`exprD`). The meaning of types is unsurprising given the syntax. In the usual style of denotational semantics, the meaning of the arrow type is an arrow of the meaning of the arguments (line 8). The injection uses the same environment representation as the `prop`-language to give a meaning to injected terms. The denotation function for expressions is more complex for three reasons: different return types, potential for ill-typed terms[1], and open terms for denotating under binders.

First, there is no single type to return because the same type is used to represent terms of many different types. The denotation function captures this multi-typedness with the τ argument. Using dependent types, `typD` *computes* the return type of the function using the value of τ. For example, `exprD` will return a natural number if `typD ts` τ evaluates to $\mathbb{N}$. But it will return a function from $\mathbb{N}$ to $\mathbb{N}$ if given `tyArr` τ τ where τ denotes to $\mathbb{N}$. The computation is what makes the denotation above well-typed. The expression on the left has type "`typD (tyArr tyN tyN)`" while the expression on the right has type "$\mathbb{N} \to \mathbb{N}$."

Second, the syntax of terms does not guarantee that all terms are well typed. This means that not all expressions have corresponding denotations. In the type language we could fake this using `False` as a default value, but not all types are inhabited, e.g. `False`. Thus, the denotation of terms returns an `option` where `None` means the term was ill-typed at the given type. In the actual implementation, `exprD` is implemented using a mutual fixpoint where one function performs simultaneous type checking. That implementation is extensionally equal to this one (modulo the features that I will add in Section 3.3).

Finally, the fact that lambda introduces new binders means that the denotation function must be able to compute a meaning for open terms. For example, computing the denotation of `Abs tyN` *e* requires computing the denotation of *e* in a context that contains a natural number, the argument to the function. Since the

[1]The denotation function uses monad notation, which I describe in greater detail in Section 3.3.1. For now, it is sufficient to think of `x ← c ; k` as `let x := c in k` where the `let` is threading the potential for failure. If `c` or `k` fails then the entire expression fails.

types in the context affect the well-typedness of terms but the values in the context do not, the types are passed to `exprD` outside of the `option`, and the values are passed inside the `option` only if the term is well-typed. In addition to being necessary to properly implement the abstraction case, this type is also more precise because it makes explicit the fact that the well-typedness of a term does not depend on the values in the variable context. MirrorCore represents "terms in context" using the `ExprT` type, which represents functions from the context values (packed in a heterogeneous list, `hlist`) to the term type. Thus, taking the denotation of a term representing a natural number in a context containing both a natural number and a boolean would result in a value of type $(\mathbb{N} \times \texttt{bool}) \to \mathbb{N}$ if `hlist` was implemented as a tuple.

To make things concrete, the proposition `True` $\wedge$ `7 = 7` might be represented as the following syntax:

```
(* environments *)
Let types : env Type := [Prop; ℕ].
Let syms : env { t: typ & typD types t } :=
  [(tyInj 0, True);
   (tyInj 0 ⇛ tyInj 0 ⇛ tyInj 0,(∧));
   (tyInj 1 ⇛ tyInj 1 ⇛ tyInj 0,(=ℕ));
   (tyInj 1,7)].
(* meta constructors *)
Let eAnd l r := App (App (Inj 1) l) r.
Let eTrue := Ref 0.
Let eEq l r := App (App (Inj 2) l) r.
Let e7 := Ref 3.
(* the term *)
eAnd eTrue (eEq e7 e7) (* True ∧ 7 = 7 *)
```

It is important to note that Coq's definitional equality is essential in the typing of environments. Without it, the term `(tyInj 0; True)` would be ill-typed since the second component of the pair has type `Prop` while it requires a value of type `typD types (tyInj 0)`.

Universal Quantifiers with λ Using `Abs` and `Var`, MirrorCore can express many binding operations. The key is to use ideas from LF's abstract binding trees [86] to ascribe a meaning to a general function. For example, MirrorCore expresses a universal quantifier over propositions as a symbol applied to a lambda abstraction where the bound variable of the lambda is the variable introduced by the quantifier[2]. The following symbols achieves this:

```
Definition PForall (P: Prop → Prop) : Prop := ∀ x : Prop, P x.
```

Assuming that the denotation of `ALL` will be this function and the denotation of `AND` is conjunction, the following syntax represents the $\wedge$-introduction rule.

```
∀ A : Prop,
  ∀ B : Prop,
      A
    → B
    → A ∧ B.
```

```
App ALL (Abs tyProp (* A *)
  (App ALL (Abs tyProp (* B *)
     (App IMPL (Var 1)
     (App IMPL (Var 0)
               (App (App AND (Var 1) (Var 0)))))))))
```

Applying the denotation function to the syntactic term on the right produces the term on the left. All that remains is to fill the place holders `ALL`, `IMPL`, and `AND` with the appropriate terms. The one drawback to this representation is that it requires functional extensionality—which states $(\forall x, f\,x = g\,x) \to f = g$—to prove some theorems. While functional extensionality is not provable in Coq, it is one of the more accepted axioms, and many dependent type theories admit it [10, 148].

3.1.1 Tautologies: A Strawman

Before I continue, it is useful to have a concrete example of a reflective procedure using this syntax. Figure 3.2 shows how to translate `rtauto` from Chapter 2 to operate on the extensible representation.

[2] In Section 3.3, I discuss how to generalize this definition to be polymorphic over the quantified type.

Fixed Syntax

```
Fixpoint rtauto hs goal :=
  match goal with
  | eTrue ⇒ true
  | eAnd p q ⇒
    rtauto hs p && rtauto hs q
  | _ ⇒ find_assumption hs goal
  end.
```

Extensible Syntax

```
Fixpoint rtauto hs goal :=
  match goal with
  | Inj 0 ⇒ true
  | App (App (Inj 1) p) q ⇒
    rtauto hs p && rtauto hs q
  | _ ⇒ find_assumption hs goal
  end.
```

Figure 3.2: Converting `rtauto` into MirrorCore's generic syntax using environments requires knowing how the environment represents `True` and $\wedge$.

The crux of the translating lies in translating the `match` which must know the constructors of the data type. As we saw in the new syntax, these "constructors" have been replaced by definitions. However, as long as we know the representation of the symbols, in this case the indices in the environment, it is not difficult to translate `rtauto` to use MirrorCore's syntax.

Verifying the above procedure requires reasoning *semantically* about the environment, i.e. we must reason about the values contained in the symbol environment as more than opaque symbols. For example, to verify the `Inj 0` case our proof must use the fact that `exprD tvs (tyInj 0) (Inj 0) = Some True`, which itself requires that `typD (tyInj 0)` ≡ `Prop`.

To see the issue with extensible semantic reasoning, consider the proof for the new `rtauto` using the symbol environments from the example.

```
Theorem rtauto_sound : ∀ hyps goal,
  rtauto hyps goal = true →
  Forall (fun x ⇒ exprD_{types,syms} (tyInj 0) x) hyps →
  match exprD_{types,syms} (tyInj 0) goal with (* [tyInj 0] means Prop *)
  | None ⇒ True
  | Some P ⇒ P
  end.
```

Because the symbol environments are values and the indices are constants, veri-

fying this procedure is quite simple. To see why, focus on the "True" case. The salient pieces are represented in Coq as:

```
Hgoal : exprD_{types,syms} (tyInj 0) (Ref 0) = Some G
============================================
G
```

At first glance, we seem stuck; *G* is an opaque symbol, and all we know is that it is the result of the denotation function. However, since `syms` is a constant, the application of `exprD` to `Ref 0` is *definitionally equal* to `Some True`, leaving the goal

```
Hgoal : Some True = Some G
============================================
G
```

which is easily provable using the injectivity of `Some` to prove `G = True`, and then the proof of `True` is trivial.

This proof was simple. Since definitional equality is free in Coq, the proof needed no extra reasoning over the prover from Chapter 2. This is promising because it shows that if we can automate the reasoning about the denotation function, then reasoning about reflective procedures that use this universal representation will not be much more difficult than reasoning about the simple, non-extensible denotation functions.

3.2 Semantic Openness & Tautologies

The representation in Figure 3.1 achieves its extensibiltiy by quantifying over both type and symbol environments. The `rtauto_sound` theorem, on the other hand, does not have the same generality. It is intricately linked to the particular environments that I defined in the example. While the proof holds for any extension of the environments, the soundness statement does not make this explicit. To see the problem, assume that I modify `rtauto` to return `true` if it sees an index not in the function environment, e.g. 8. The theorem is still provable because the denotation

of `Ref 8` under the environments is `None` and the theorem holds vacuously. However, extending the environment with a symbol at position 8, e.g. `False`, would invalidate the proof.

The problem is that the statement of `rtauto_sound` is not sufficiently general. We need to prove the above theorem for any environments that satisfy certain properties. In this case, `Prop` at position 0 in the type table and `True` and $\wedge$ at positions 0 and 1 respectively in the symbol table. In the next two subsections I demonstrate two ways to state and prove correctness theorems to support this kind of extensibility

3.2.1 A Propositional Constraint Formulation

A natural way to represent constraints is with a partial environment. For example, I could represent the constraints for `True` and $\wedge$ with the following partial environment, for the moment ignoring the complexities of syntactic types.

$$\{0 \mapsto \texttt{True}; 1 \mapsto \wedge\}$$

I can then define a "satisfies" relation to express when an environment e satisfies a constraint C, written $C \models e$. Intuitively this is true when all mappings in the constraint are consistent with the environment. That is

$$C \models ts \iff \forall k\, s, (k \mapsto s) \in C \to \mathsf{lookup}\, k\, ts = s$$

Using these constraints and the satisfies relation I can start to phrase a more general soundness theorem for any environment that satisfies the constraints. Applying the constraints to both types and functions yields the following soundness statement.

$$\begin{aligned}\mathsf{rtauto\_sound} \quad : \quad & \forall hs\, g, \mathsf{rtauto}\, hs\, g = \mathsf{true} \to \\ & \forall ts\, fs, TC \models ts \to FC \models fs \to \\ & \quad \mathsf{Forall}\, \mathsf{Provable}\, hs \to \mathsf{Provable}\, g\end{aligned}$$

Here Provable takes the denotation of an expression as a `Prop`. Making this definition formal, and proving it, is subtle in intensional type theory due to the environment proofs. The issue can be seen in the definition of Provable.

Provable needs to use the denotation function to compute a `Prop` from a syntactic expression. However, the denotation function only allows us to compute values of type `typD ts X` for some syntactic type `X`. Using the following type constraint we can pick `tyInj 0` to stand for `Prop`.

```
Let TC : C Type := { 0 ↦ Prop ; 1 ↦ ℕ }.
```

Using this information, however, is slightly more difficult. Naïvely, we would like to write the following:

```
Let FC : C { t: typ & typD ?? t } :=
    { 0 ↦ (tyInj 0, True)
    ; 1 ↦ (tyInj 0 ⇛ tyInj 0 ⇛ tyInj 0, ∧) }.
```

But we need an environment satisfying `TC` to fill in for the `??`. Generalizing `FC` to be a function over such an environment provides a partial solution:

```
Let FC ts (pf: TC ⊨ ts) : C { t: typ & typD ts t } :=
  { 0 ↦ (tyInj 0; True)
  ; 1 ↦ (tyInj 0 ⇛ tyInj 0 ⇛ tyInj 0, ∧) }
```

but this term still does not type check in intensional type theory.

To see the problem, focus on the constraint for `True`. The rule for type checking the dependent pair requires that `True` (the second component) has type `typD ts (tyInj 0)`. Using conversion, this reduces to `lookup ts 0`, but this term is stuck since `ts` is universally quantified. Using the constraints, we can prove that `lookup ts 0 = Prop`, but to use this fact we must witness it in the term. With the casts, the term becomes the following:

```
Definition cast {T} {c: C T} {g: env T} (pf: c ⊨ g) (i: ℕ)
(F: T → Type) (val: F (lookup_C (default := ∅) c i))
: F (lookup_env g i) := ... (* proof here *)
```

```
Let FC ts (pf: TC ⊨ ts) : C { t: typ & typD ts t }:=
  { 0 ↦ (tyInj 0, cast pf 0 (fun x ⇒ x) True)
  ; 1 ↦ (tyInj 0 ⇛ tyInj 0 ⇛ tyInj 0,
         cast pf 0 (fun x ⇒ x → x → x) (∧)) }.
```

With these definitions we can write the soundness theorem using explicit casts.

```
Theorem rtauto_sound_constraints : ∀ hyps goal,
  rtauto hyps goal = true →
  ∀ (ts: env Type) (pf_ts: TC ⊨ ts)
    (fs: env { t: typ & typD ts t }) (pf_fs: FC ts pf_ts ⊨ fs)
  let Provable e := match exprD (tyInj 0) e with
                    | None ⇒ False
                    | Some P ⇒ cast pf_ts 0 (fun T ⇒ T) P
                    end
  in Forall Provable hyps → Provable goal.
```

Here `Provable` uses `cast` to convert a value from `typD ts (tyInj 0)` to `Prop` using the constraints. The proof of this theorem is the same as the proof of the previous, non-extensible, theorem except that the new proof must reason explicitly about the casts.

COMPSITIONALITY Using the propositional approach, it is easy to compose constraints, but composing theorems is slightly more complicated. The reason that composing theorems is difficult is that the *statement* of soundness uses the constraints to perform the cast. In an intensional type theory such as Coq's, this cast can have computational content, i.e. it can do more than just return the value it is given. Therefore, if the two specifications do not use the exact same proof to perform the cast then they could be making statements about two different interpretations of `Prop`, e.g. `Prop` and negated `Prop`. Fundamentally, the problem boils down to proving the following goal.

```
match pf1 in _ = x return x with
| eq_refl ⇒ P
```

```
end ↔ match pf₂ in _ = x return x with
      | eq_refl ⇒ P
      end
```

Here, $\texttt{pf}_1$ and $\texttt{pf}_2$ come from two different constraints so we do not know that they are the same. As counter-intuitive as it may seem, this goal is not provable in Coq without an axiom. In order to eliminate $\texttt{pf}_1$, we need to abstract the goal with respect to the syntactic type. Doing this removes the match on $\texttt{pf}_1$ but changes the type of $\texttt{pf}_2$ to `Prop = Prop`. Now, the well-typedness of the goal depends crucially on both sides of the equality that $\texttt{pf}_2$ witnesses being `Prop`. The problem lies in a lack of polymorphism.

Two solutions exist for this problem. First, we can add an axiom to Coq that states that all proofs of equality (of the same type) are equal. In math this assertion is captured by the following axiom.

$$\forall T\, a\, b\, (pf_1 : a = b)(pf_2 : a = b), pf_1 = pf_2$$

This approach is not consistent with models of Gallina based on homotopy theory where equality proofs correspond to paths and isomorphisms induce non-trivial equalities between types [148].

While the axiom is dissatisfying under a homotopy type theory, another solution is to require a propositional equality between $\texttt{pf}_1$ and $\texttt{pf}_2$, i.e. $\texttt{pf}_1 = \texttt{pf}_2$. This allows us to avoid the axiom essentially by requiring individual proofs in each instance where we would have used it. In practice all of these will be trivially provable by reflexivity but it does limit some of the generality since it is now harder to reason about equalities between equalities of constraint proofs.

3.2.2 A Definitional Constraint Formulation

The propositional constraint formulation provides maximal flexibility surrounding the statement and manipulation of constraints. However, it can be difficult to reason explicitly about the proofs. In addition, explicitly manifesting these proofs

can be cumbersome when applying a reflective soundness theorem.

We can avoid manifesting these proofs by phrasing them so that the provable equality becomes a definitional one. While not essential for the function environment, achieving definitional equality at the type level provides a cleaner phrasing of the soundness theorem with no need for explicit casts.

The key to stating the definitional equality in a type theory that does not have first-class definitional equality is to use computation. This need for computation requires that I be more concrete about the representations. For now, I represent constraints and environments as lists where the position in the list is the key[3]. In the more concrete representation, the typing context is the following.

```
Inductive Constraint (T: Type) : Type := Any | Exact (t: T).
Definition C (T: Type) := list (Constraint T).

Let TC : C Type := [Exact Prop; Exact ℕ].
```

Here, `Exact t` will mean that the current position is definitionally equal to `t`, while `Any` will place no restriction on the location.

Rather than requiring a proof, which would provide a propositional equality, I use the `applyC` function (in Figure 3.3) to compute a derived environment that manifestly satisfies the constraints. The structure of this definition is crucial. `applyC` is defined by structural recursion on the *constraints*, and the syntactic shape of the resulting list depends only on the constraints. This means that even when the environment is opaque, if the constraints are a constant then `applyC` will completely reduce. For example:

$$
\begin{aligned}
& \texttt{applyC}\ \emptyset\ [\texttt{Exact Prop};\ \texttt{Exact}\ \mathbb{N}]\ \textit{ts} \\
\equiv\ & \texttt{Prop} :: \texttt{applyC}\ \emptyset\ [\texttt{Exact}\ \mathbb{N}]\ (\texttt{tl}\ \textit{ts}) \\
\equiv\ & \texttt{Prop} :: \mathbb{N} :: \texttt{applyC}\ \emptyset\ []\ (\texttt{tl}\ (\texttt{tl}\ \textit{ts})) \\
\equiv\ & \texttt{Prop} :: \mathbb{N} :: \texttt{tl}\ (\texttt{tl}\ \textit{ts})
\end{aligned}
$$

[3]I will return to this point at the end of the section and discuss the necessary conditions for this definitional constraint formulation.

```
Fixpoint applyC (d: T) (c: C T) (e: list T) {struct c} : list T :=
  match c with
    | [] ⇒ e
    | c :: c' ⇒ match c with
                  | Any ⇒ hd (default := d) e
                  | Exact v ⇒ v
                 end :: applyC c' (tl e)
  end.
```

Figure 3.3: The definition of `applyC` recurses over constraints and ensures that the value of `e` does not affect the shape of the result.

Using `applyC`, the function constraints require neither explicit proofs nor explicit casts.

```
Let FC (ts: list Type) : C { t & typD (applyC ∅ TC ts) t } :=
  [ Exact (tyInj 0, True)
  ; Exact (tyInj 0 ⇛ tyInj 0 ⇛ tyInj 0, ∧) ].
```

To see why the problem is solved, consider once again the `True` constraint. Type checking needs to verify that `True` has type `typD (applyC ∅ TC ts) (tyInj 0)`, but this is *definitionally equal* to `Prop`.

$$
\begin{aligned}
& \texttt{typD}\ (\texttt{applyC}\ \emptyset\ \texttt{TC}\ \mathit{ts})\ (\texttt{tyInj}\ 0) \\
\equiv\ & \texttt{typD}\ (\texttt{Prop} :: \mathbb{N} :: \texttt{tl}\ (\texttt{tl}\ \mathit{ts}))\ (\texttt{tyInj}\ 0) \\
\equiv\ & \texttt{lookup}_{\mathit{env}}\ (\texttt{Prop} :: \mathbb{N} :: \texttt{tl}\ (\texttt{tl}\ \mathit{ts}))\ 0 \\
\equiv\ & \texttt{Prop}
\end{aligned}
$$

We can use the same technique to phrase an extensible version of the soundness theorem (Figure 3.4). Unlike the propositional formulation this formulation requires no propositional casts; computation alone makes the theorem statement type-check.

COMPLETENESS The computational formulation is easier to reason about because all casts are by definitional equalities. However, is the computational formulation less expressive than the propositional one? The answer is partially "no"

```
Theorem rtauto_sound : ∀ hyps goal,
  rtauto hyps goal = true →
  ∀ (ts': env Type), let ts := applyC TC ts' in
  ∀ (fs': env { t: typ & typD ts t }), let fs := applyC FC fs' in
  let Provable e := match exprD_{ts,fs} (tyInj 0) e with
                    | None ⇒ False
                    | Some P ⇒ P
                    end
  in Forall Provable hyps → Provable goal.
```

Figure 3.4: Using `applyC` allows us to phrase the soundness theorem extensionally without requiring propositional casts.

and partially "yes."

First, to address the "no." Any self-consistent constraint can be converted into a computational constraint such that every satisfying environment for the propositional constraint is also constructable via the computational formulation. This fact is justified by the following, easily proven, theorem.

Theorem 3.2.1 *Completeness of the Computational Formulation*

$$\forall d\ ts\ TC, TC \models ts \to \texttt{applyC}\ d\ (\texttt{to\_computational}\ TC)\ ts = ts$$

Here, `TC` is a propositional constraint that `to_computational` converts to a definitional constraint where holes are filled by `Any`.

However, the completeness theorem does not quite cover the entirety of expressivity. The computational formulation only avoids casts when the constraints are *values*. For example, we might like to parameterize `rtauto` by the representation of `Prop`, `True`, and $\wedge$[4] using the code in Figure 3.5. However, defining `FC` requires casts because `TC` is not a closed value since it was constructed using the universally quantified `id_prop`. The propositional constraint formulation, on the other hand, does allow for this kind of parameterization.

[4]The syntax x ?[=]y is ExtLib [3] syntax for the function that decides $x = y$ or $x \neq y$.

```
Variable id_prop : index.
Variable id_and id_true : index.

Fixpoint rtauto (hyps: list expr) (e: expr) : bool :=
  match e with
  | eRef n ⇒ if id_true ?[=] n then true else prove hyps e
  | eApp (eApp (eRef n) a) b ⇒
    if id_and ?[=] n then rtauto hyps a && rtauto hyps b
    else prove hyps e
  | _ ⇒ prove hyps e
  end.

Hypothesis and_not_true : id_and ≠ id_true.
Let TC : C Type := insert id_prop Prop.
Let FC ts : C { τ: typ & typD (applyC TC ts) τ } := ...
```

Figure 3.5: Definitional constraints, like definitional equality, are second class in Coq, which prevents us from certain types of quantification such as the quantification over indices shown here.

Composition A benefit of working with definitional equality is that the type theory guarantees that if a definitional equality exists between any two objects then it is unique. This means that we can computationally combine two constraints, and the order of composition is irrelevant as long as the constraints are consistent. Assuming the constraint composition is written $\circ$ and both C_1 and C_2 are values, the following are true *definitionally*.

$$\mathsf{applyC}\ C_1(\mathsf{applyC}\ C_2\ ts) \equiv \mathsf{applyC}\ (C_1 \circ C_2)\ ts \equiv \mathsf{applyC}\ C_2(\mathsf{applyC}\ C_1\ ts)$$

Leveraging this property, we can combine two proofs with different, compatible constraints that are values into a single proof by composition. We use the function `either` to combine the computational parts of the provers P_1 and P_2 to produce a prover that will solve the goal if either P_1 or P_2 would have solved it. Figure 3.6 highlights how this approach works works for types.

```
Let TC₁ := [Exact Prop].
Let TC₂ := [Any; Exact ℕ].

Hypothesis P₁_sound : ∀ ts, sound (applyC TC₁ ts) P₁.
Hypothesis P₂_sound : ∀ ts, sound (applyC TC₂ ts) P₂.

Theorem P₁_and_P₂_sound : ∀ ts,
  sound (applyC (TC₁ ∘ TC₂) ts) (either P₁ P₂).
Proof.
  intro ts.
  pose (P₁_sound (applyC TC₂ ts)).
    (* : sound (applyC TC₁ (applyC TC₂ ts)) P₁ *)
  pose (P₂_sound (applyC TC₁ ts)).
    (* : sound (applyC TC₂ (applyC TC₁ ts)) P₂ *)
  ...
```

Figure 3.6: Definitional constraints provide a simple way to perform composition without the need for propositional casting.

Leveraging the definitional equalities above, all that remains to show is:

$$\forall ts\, P_1\, P_2, (\mathsf{sound}\, ts\, P_1) \to (\mathsf{sound}\, ts\, P_2) \to (\mathsf{sound}\, ts\, (\mathsf{either}\, P_1\, P_2))$$

The difficult problem concerning proofs in the propositional formulation is completely gone.

Like the expressiveness of the definitional formulation, however, this type of composition relies fundamentally on the constraints being values. This means that composition must be done extra-logically, often by $\mathcal{L}_{tac}$ procedures which can build potentially ill-typed terms and ask the kernel to check them. While composition could fail, it is fairly easy to check whether two constraints are computationally compatible by seeing if the following term is well-typed.

```
fun ts ⇒ eq_refl : applyC C₁ (applyC C₂ ts) = applyC C₂ (applyC C₁ ts)
```

This simple test makes it easy to detect and report errors about incompatible contexts.

The Requirements for Definitional Reduction The key to making definitional reduction work lies in the interaction between `applyC` and `lookup`. The crucial aspect is the need for the structure of the container to be independent of its contents. This separation of structure and values is a stronger property than simply having a canonical representation. For example, the following alternative implementation of `applyC` is provably equal to one showed at the beginning of the section but does not have the same computational property.

```
Fixpoint applyC (d: T) (c: C T) (e: list T) : list T :=
  match c with
    | [] ⇒ e
    | Any :: c' ⇒ match e with
                  | [] ⇒ d :: applyC c' []
                  | e :: es ⇒ e :: applyC c' es
                  end
    | Exact c :: c' ⇒ match e with
                      | [] ⇒ c :: applyC c' []
                      | e :: es ⇒ c :: applyC c' es
                      end
  end.
```

The problem with this definition is that it matches on `e` before constructing the spine of the list. This match will block reduction when `e` is opaque. In the previous definition, `e` was only matched on to compute values under the spine of the resulting list, and since the `lookup` function only matched on the spine of the list it never gets stuck on a match in the constraints.

MirrorCore's actual implementation uses positive maps for environments, which achieves logarithmic time lookup while retaining the phase-split property that makes computation simple.

Non-monadic

```
Definition add_or_error
  (a b: option ℕ): option ℕ :=
match a with
  | None ⇒ None
  | Some aV ⇒
    match b with
      | None ⇒ None
      | Some bV ⇒ Some (aV + bV)
    end
end.
```

Monadic

```
Definition add_or_error
  (a b: option ℕ): option ℕ :=
  bind a (fun aV ⇒
  bind b (fun bV ⇒
  ret (aV + bV))).
```

```
Definition add_or_error
  (a b: option ℕ): option ℕ :=
  aV ← a ;
  bV ← b ;
  ret (aV + bV).
```

Figure 3.7: Code that adds two `option` ℕ's using explicit `match`es (left) and monad operations (right).

3.3 Meta-level Dependency & Monad Simplification

Purely providing extension through simple environments is not expressive enough to capture all of the interesting domains that I will reason about in the remainder of the dissertation. For example, monads, which I will discuss shortly, require both type constructors and polymorphism. In this section I show how to abstract the code in Figure 3.1 to support meta-level polymorphism, type functions, and type dependency in a second-class way without significantly complicating the language. For now, I will explain what these terms mean with a brief introduction to monads.

3.3.1 Reasoning About Monads

Fundamentally monads are about abstraction, often abstracting side effects. Ignoring their category-theoretic meaning, and thinking of them in a purely operational sense, monads essentially abstract the sequencing operator, i.e. `let x := ... in` ... in OCaml or ; in C. Doing this enables them to thread additional information through a computation, for example a piece of state or the possibility of an exception.

A simple example is the option monad which I showed at work in Figure 3.1. Figure 3.7 has a more minimal example which adds two optional natural numbers. Both columns perform the same computation, but the monadic implementation abstracts the `match`, `Some` and `None` into the monad operations `bind` and `ret`. In the option monad, `bind` abstracts the threading of the option through the computation, allowing the function to be coded oblivious to the potential of `None`. The `ret` for `option` simply injects the value using `Some`. The traditional monad notation is shown in the listing below the explicit definition with `bind` and `ret`.

Formally, a monad is a type function (`M : Type → Type`) that supports two operations:

```
ret : Π a: Type, a → M a
bind : Π a β: Type, M a → (a → M β) → M β
```

`ret` injects arbitrary values into the monad, and `bind` accumulates the "side-effects" in the first command and provides the value, of type a, to the second command, which may have additional side-effects.

To truly form a monad, `ret` and `bind` must satisfy the three monad laws. Eliding the types for simplicity, these are:

$$\begin{array}{rl} \text{right identity} & \forall x, \mathsf{bind}\ x\ \mathsf{ret} = x \\ \text{left identity} & \forall f\, x, \mathsf{bind}\ (\mathsf{ret}\ x)\ f = f\, x \\ \text{bind associativity} & \forall a\, b\, c, \mathsf{bind}\ (\mathsf{bind}\ a\ b)\, c = \mathsf{bind}\ a\ (\lambda x.\mathsf{bind}\ (b\, x)\ c) \end{array}$$

3.3.2 Generalizing the Syntax

What is missing from the environment representation is the ability to have additional structure within the type and function environments. For example, to express `option` in the environment encoding I would need to pick distinct indices for `option` $\mathbb{N}$ and `option bool` in the type environment. Further, in the symbol environment I would need different indices for different instantiations of polymorphic functions such as `Some` and `None`.

The solution to this problem is to abstract the representation of types and sym-

bols. This allows clients to choose a representation that can capture additional structure such as the polymorphism of `Some` and still use MirrorCore's core syntax and functions.

Figure 3.8 shows the parameterized code. The syntax also includes unification variables (`UVar`) which I will discuss in more detail in Section 3.4. In this representation, types and symbols are parameters (lines 1 and 2). Associated with these abstract types are the operations that the denotation requires to integrate them into the denotation (lines 10-12). `RType_typ` captures the requirements on types, namely a denotation function (`typD`) with the same type as before, and a decidable casting operation (`type_cast`) that checks for equality between syntactic types.

Merely a type language is not sufficient for the denotation function. To denote application and abstraction, the type language must include a representation of function types. In Figure 3.1 this requirement manifests itself in the pattern match in the `Abs` case and the use of `tyArr` in the `App` case. When we abstract the type, we lose access to its constructors and pattern matching facility. `Typ2_Fun` provides these operations. Figure 3.9 shows the definition for `Typ2` (representing two-argument type constructors) as well as the related theorems for reasoning about it. Three operations are necessary:

1. A "constructor" for building the syntactic type (`typ2`).

2. A propositional equality witnessing the computational behavior of the denotation function when applied to the constructor (`typ2_cast`).

3. A dependent eliminator that provides a way to determine if an arbitrary syntactic type represents the underlying semantic type (`typ2_match`). For polymorphic types such as functions this eliminator also extracts the constructor's type arguments (e.g. the domain and co-domain of the function type).

Two-argument type functions (such as function types) are just one particular way to build types. Generalizing these yields a family of possible constraints for representing types (such as $\mathbb{N}$), single-argument type functions (such as monads),

```
Variable typ : Type. (* the syntax of types *)
Variable sym : Type. (* the syntax of injected terms *)

Inductive expr : Type := (* the core syntax *)
| App : expr → expr → expr
| Abs : typ → expr → expr
| Var : ℕ → expr
| Inj : sym → expr
| UVar : ℕ → expr. (* see Section 3.4 *)

Variable RType_typ : RType typ.          (* type requirements *)
Variable Typ2_Fun : Typ2 RType_typ Fun. (* representation of arrows *)
Variable RSym_sym : RSym RType_typ sym. (* symbol requirements *)

Definition ExprT (tus tvs: list typ) (T: Type) : Type := (* open terms *)
  hlist typD tus → hlist typD tvs → T.

Fixpoint exprD (tus tvs: list typ) (t: typ) (e: expr)
: option (ExprT tus tvs (typD t)) :=
  match e with
  | App f x ⇒
    domT ← typeof_expr tus tvs x ;
    valF ← exprD tus tvs (typ2 domT t) f ;
    valX ← exprD tus tvs domT x ;
    ret (fun us vs ⇒
           (Rcast_val (typ2_cast domT t) (valF us vs)) (valX us vs))
  | Abs t' e ⇒
    typ2_match (fun T ⇒ option (ExprT tus tvs T)) ts t
               (fun d r ⇒
                   cast ← type_cast ts d t' ;
                   val ← exprD tus (t' :: tvs) r e ;
                   ret (fun us vs ⇒ fun x ⇒
                          val us (Hcons (Rcast_val cast x) vs)))
               None
  | Var v ⇒ (t,get) ← nth_error_get_hlist_nth  tvs v ;
            cast ← type_cast t' t ;
            ret (fun _ vs ⇒ Rcast_val cast (get vs))
  | Inj i ⇒ val ← funcAs s t ;
            ret (fun _ _ ⇒ val)
  | UVar u ⇒ (t,get) ← nth_error_get_hlist_nth  tus u ;
             cast ← type_cast t' t ;
             ret (fun us _ ⇒ Rcast_val cast (get us))
  end.
```

Figure 3.8: MirrorCore's core syntax and denotation function parameterized by an implementation of types (typ) and symbols (sym).

two-argument type functions (such as arrows), etc.

The denotation function uses these operations in exactly the places that the previous implementation (Figure 3.1) relied on pattern matching and definitional equality. The denotation for abstractions and applications shows how `exprD` uses `Typ2` to build and use functions constructed by the syntax. The abstraction case (lines 27-34) uses `typ2_match` to determine whether the requested syntactic type is a function type. It then uses `type_cast` (provided by `RType`) to check that the domain of the function type is convertible to the argument type annotated in the syntax. If the type cast succeeds, the returned proof is used to cast the result of the recursive call to the appropriate type. Having a simple, transparent proof for this cast is essential because the denotation function will actually inspect the proof to perform the cast. The `App` case is the converse. The denotation function determines the type of the function and uses it to build the appropriate syntactic type to use when computing the denotation of the function. The result then uses `typ2_cast` to convert the result, which has type `typ_arr domT t`, to a function type.

3.3.3 Automating the Monad Laws

Now that MirrorCore's syntax can support arbitrary type and term algebras, it is much easier to build automation for monads. First, I embed the monadic language in MirrorCore by defining the appropriate algebra of types and symbols (Figure 3.10). In addition to the `tyArr` constructor, the type language also includes a constructor for the monad type (`tyM`). The fact that `m` is a unary type constructor is captured by the fact that the syntactic constructor `tyM` takes an argument, the return type of the monadic computation. All other types, e.g. numbers, strings, etc., are represented using the environment encoding that I discussed in Section 3.2.

At the term level, I represent the type arguments to polymorphic terms such as `bind` and `ret` as arguments to the symbol's constructor. For example, the type of `bind` depends on two types, α and β, so the `mBind` constructor takes two arguments

```
(* The semantic meaning of the type *)
Variable F : Type → Type → Type.

Class Typ2 : Type :=
{ typ2 : typ → typ → typ
; typ2_cast : ∏ a b, typD (typ2 a b) = F (typD a) (typD b)
; typ2_match : ∏ (T: Type → Type) t,
    (∀ a b, T (F (typD a) (typD b))) → T (typD t) → T (typD t) }.

Class Typ2Ok (TI : Typ2) : Type :=
{ (* typ2_match recognizes typ2 *)
  typ2_match_iota
  : ∀ T a b tr fa,
    typ2_match T (typ2 a b) tr fa =
    match eq_sym (typ2_cast a b) in _ = t return T t with
    | eq_refl ⇒ tr a b
    end
  (* the domain and range are ``smaller'' than the arrow *)
; tyAcc_typ2L : ∏ a b, tyAcc a (typ2 a b)
; tyAcc_typ2R : ∏ a b, tyAcc a (typ2 b a)
  (* typ2 is injective *)
; typ2_inj
  : ∀ a b c d, (typ2 a b) = (typ2 c d) → a = c ∧ b = d
  (* a type either is or is not an arrow *)
; typ2_match_case
  : ∀ x, (∃ d r (pf: x = typ2 d r),
           ∀ T tr fa,
             typ2_match T x tr fa =
             Relim T pf
             (match eq_sym (typ2_cast d r) in _ = t return T t with
             | eq_refl ⇒ tr d r
             end)) ∨
         (∀ T tr fa, typ2_match T x tr fa = fa)
  (* typ2_match respects the equivalence relation on types *)
; typ2_match_Proper
  : ∀ T t t' (pf: t' = t) tr fa,
      typ2_match T t tr fa =
      Relim T (eq_sym pf) (typ2_match T t' tr (Relim T pf fa)) }.
```

Figure 3.9: The extensional (constraint) representation of two-argument type constructors. typD is the denotation function on types introduced by the definition of RType.

```
Variable m : Type → Type.
Variable Monad_m : Monad m.

Inductive typ : Type := (* the type algebra *)
| tyArr : typ → typ → typ  (* l ⇛ r *)
| tyM   : typ → typ
| tyInj : ℕ →  typ.

Fixpoint mtypD (x: mtyp) {struct x} : Type :=
  match x return Type with
  | tyArr l r ⇒ typD l → typD r
  | tyInj x ⇒ getType ts x
  | tyM x ⇒ m (typD x)
  end.

Inductive mfunc : Type := (* monad specific operations *)
| mBind : mtyp → mtyp → mfunc
| mReturn : mtyp → mfunc.

Definition typeof_mfunc (m: mfunc) : typ :=
  match m with
  | mBind a b ⇒ tyM a ⇛ (a ⇛ tyM b) ⇛ tyM b
  | mReturn a ⇒ a ⇛ (tyM a)
  end.

Definition mfuncD (f: mfunc) : match typeof_mfunc f with
                               | None ⇒ unit
                               | Some t ⇒ typD m ts t
                               end :=
  match f as f return typD m (typeof_mfunc f) with
  | mBind a b ⇒ bind
  | mReturn a ⇒ ret
  end.

(* join with environments for non-monad symbols *)
Definition func := mfunc + ℕ.
```

Figure 3.10: Representing monads using MirrorCore's support for arbitrary type and term algebras.

```
Notation "'BIND' [ a ,  b ]" := (Inj (inr (mBind a b))).
Notation "'RET' [ a ]" := (Inj (inr (mReturn a))).
Notation "a @ b" := (App a b).

Definition mexpr := expr typ func.

Fixpoint reduce_m (t: typ) (e: mexpr) {struct e} : mexpr :=
  match e with
  | BIND [ _ , _ ] @ e @ RET [ _ ] ⇒ reduce_m t e
  | BIND [ t' , _ ] @ (RET [ _ ] @ e) @ e' ⇒
    let e' := reduce_arrow t' (tyM t) e' in
    let e := match t' with
             | tyM z ⇒ reduce_m z e
             | tyArr a b ⇒ reduce_arrow a b e
             | _ ⇒ e
             end in
    red_app e' e
  | BIND [ t' , _ ] @ (BIND [ t'' , _ ] @ a @ b) @ c ⇒
    let a := reduce_m t'' a in
    let b := reduce_arrow t'' (tyM t') b in
    let c := reduce_arrow t' (tyM t) c in
    red_bind t' t a (Abs t'' (red_bind t' t (red_app b (Var 0)) c))
  | _ ⇒ e
  end
with reduce_arrow (d r: typ) (e: mexpr) {struct e} : mexpr :=
  match e with
  | Abs t (App x (Var 0)) ⇒ (* η-reduction *)
    match lower 0 1 x with
    | None ⇒ e
    | Some e ⇒ e
    end
  | Abs t e' ⇒
    match r with
    | tyM m ⇒ Abs t (reduce_m m e')
    | _ ⇒ e
    end
  | _ ⇒ e
  end.
```

Figure 3.11: A simple monad reducer built on top of MirrorCore. red_app and red_bind are "smart" constructors that perform single-level reduction such as reducing $(\lambda x.f^x)y$ into f^y.

(α β : typ). The `mRet` constructor is similar. In the denotation function, the `bind` and `ret` functions come from the monad instance (defined as a section variable on line 2) while the type arguments are computed from `a` and `b`. Again, I use an environment to represent all other symbols. It is worth noting that while we can use type constructors such as sum to combine disjoint symbol languages, the same is not true for types because the type algebra is recursive. The symbol representation does not need to be recursive because MirrorCore's expression representation takes care of the recursive portions of the expression language.

Using the syntax, it is not too difficult to implement monad reduction in Coq (see Figure 3.11). The algorithm relies on two mutually recursive functions that simplify terms of different types.

- `reduce_m` simplifies syntactic expressions that have type "`m t`" using the monad laws. The cases of the function can be read directly from the laws with recursive calls being made to either `reduce_m` or `reduce_arrow` depending on the type of the sub term.

- `reduce_arrow` simplifies syntactic functions that have type "`d` $\rightarrow$ `m r`" by performing η-reduction (the first case) and reducing monadic expressions that occur inside of λs (the second case).

In both cases, the fall-through branch could be extended to do additional processing. For example, it could be extended to handle other monad functions (such as `join` or `fmap`) or other monad structures such as monads with a zero or monads with state.

Verifying `reduce_m` and `reduce_arrow` is similar to the verification of `rtauto` in the previous section. It is also possible to generalize the implementation to work over abstract type and symbol algebras, e.g. by requiring a `Typ1` representing the monad and classes (not shown) analogous to `Typ0`, `Typ1`, etc. for representing symbols extensionally. This abstraction provides more opportunities to reuse the automation.

Meta-level Dependency The same technique can be used to achieve meta-level dependency of terms. For example, the following type and symbol algebras allow arbitrary-length bit-vectors.

```
Inductive typ :=
| Arr (d r : typ)
| Bv (s : ℕ)
| ...
```

```
Inductive sym :=
| Plus (n : ℕ) (* Bv n → Bv n → Bv n *)
| SExt (n m : ℕ) (* Bv n → Bv m *)
| ...
```

3.4 Unification Variables & Backward Reasoning

The final syntactic form of MirrorCore expressions is unification variables. Unlike MirrorCore's other syntactic forms, unification variables are not part of type theory. Rather, they act as place holders to assist the user or the automation in incrementally building terms.

MirrorCore's representation of unification variables is similar to the environment representation of variables. In fact, the only difference is that there are no syntactic forms that introduce new unification variables. In the denotation function, the value environment carries the unification variables representing the placeholders for the term. This allows MirrorCore to reify an expression such as ?1+?2 into "`App (App PLUS (UVar 0)) (UVar 1)`" using `[?1;?2]` as the environment of unification variables.

Rather than using a substitution semantics for reasoning about instantiating a unification variable, MirrorCore expresses instantiation using provable equalities. In MirrorCore, these learned facts are stored in a table (called a substitution) that maps unification variables to syntactic expressions. The denotation of a substitution is a conjunction of equations between each unification variable and the denotation of the expression that is corresponds to. For example, the denotation of a substitution mapping `UVar 0` to 7 and `UVar 1` to $x+y$ would be the following.

$$?1 = 7 \quad \wedge \quad ?2 = x + y$$

```
Theorem eprover_sound : ∀ tus tvs goal s s',
  eprove goal sub = Some sub' → (* the procedure returns success *)
  WellFormed_subst sub →   (* good initial substitutions *)
  WellFormed_subst sub' ∧ (* produce good final substitutions *)
  ∀ subD goalD,
    (* if the initial goal and substitution are well-typed *)
    exprD tus tvs goal = Some goalD →
    substD tus tvs sub = Some subD →
    ∃ subD', (* then the final substitution is well-typed *)
      substD tus tvs s' = Some subD' ∧
      ∀ us vs,
        subD' us vs → (* if the final substitution is provable, then *)
           goalD us vs (* the goal and *)
        ∧ subD us vs. (* initial substitution are provable *)
```

Figure 3.12: The soundness statement for a reflective procedure (eprove) with the ability to instantiate unification variables.

Since the meaning of a substitution is a first-class value (computed by substD) it is easy to integrate substitutions into soundness theorems. Figure 3.12 shows a high-level soundness theorem for a function eprove that uses, and possibly extends, a substitution. The procedure eprove takes a goal and an initial substitution and returns an optional final substitution. If eprove returns Some sub', eprove succeeded in solving the goal but only if the denotation of the final substitution (sub') is provable. To ensure that the final substitution is compatible with the initial one, the soundness theorem requires provability of the final substitution to imply the provability of the initial substitution in addition to the goal. The provability of the initial substitution is essential when chaining provers together, but the entry points for reflective procedures drop these in their specifications since the denotation of an empty substitution is True.

3.4.1 Case Study: A Generic Backward Prover

To demonstrate unification variables in action I will show how to use them to build an extensible prover similar to $\mathcal{L}_{tac}$'s eauto tactic. This tactic takes a collection of lemmas and repeatedly applies them until it finds a proof or fails to do so. While

the implementation and the heuristics are quite simple, the strategy works in a surprising number of cases.

Building `eauto` requires using unification variables in several interesting ways. The procedure will have to introduce new unification variables, solve existing ones, and soundly remove instantiated unification variables completely reflectively. Before diving into each of these tasks, I discuss MirrorCore's representation of lemmas.

Representing Lemmas

Using raw expressions to represent lemmas can be quite inefficient since doing so would require repeatedly parsing the expression to extract the quantifiers, premises and conclusion. Instead, MirrorCore uses the structured representation of lemmas shown in Figure 3.13.

A lemma is a record that contains a list of universally quantified variables (here only their types), a list of premises (represented as `expr`s) and finally a conclusion. For generality, the type of lemmas is parametric in the type of the conclusion all that is required is a denotation function that produces a `Prop`. The backwards prover picks the conclusion to be an expression and uses `exprD` for the denotation function. This allows the conclusion of the lemma to match up exactly with the goal that it is applying to. On the other hand, a rewriter would choose the conclusion to be triple of a type and two expressions where the denotation is the equality of the two expressions at the given type.

The denotation function for lemmas is expressed using the same two-stage form as `exprD`. In particular, the option expresses whether the lemma is well-typed and the value of the `Some` constructor will express the actual meaning. While global lemmas never have any variables or unification variables, writing the denotation function to permit them makes it easy to weaken the entire lemma pulling it into the context where it will be applied.

While seemingly complex, `open_lemmaD` and `closed_lemmaD` have nice computational properties that make the meaning of the lemma exactly the type of the

```
Variable concl : Type. (* the type of conclusions *)

Record lemma : Type := (* lemmas are Coq records *)
{ foralls: list typ ; premises : list expr ; conclusion : concl }.

Variable Typ0_Prop : Typ0 Prop.
Variable conclD : Π (tus tvs: tenv typ), concl → option (ExprT tus tvs Prop).

Definition open_lemmaD (tus tvs: tenv typ) (lem: lemma)
: option (ExprT tus tvs Prop) :=
  let vars := lem.( foralls) in
  match mapT (fun e ⇒ exprD tus (vars ++ tvs) typ0 e) lem.( premises)
      , conclD tus (vars ++ tvs) lem.( conclusion)
   with
   | Some prems , Some concl ⇒
     Some (fun us vs ⇒
       forallEach vars (fun vs' : hlist (typD ts) vars ⇒
         fold_right (fun P conc ⇒ P us (hlist_app vs' vs) → conc)
                    (concl us (hlist_app vs' vs)) prems
   | _ , _ ⇒ None
   end.

Definition closed_lemmaD (lem: lemma) : Prop :=
  match open_lemmaD [] [] lem with
  | None ⇒ False
  | Some lD ⇒ lD Hnil Hnil
  end.
```

Figure 3.13: MirrorCore's generic representation of lemmas.

lemma itself. That is,

```
Definition lem_AI : lemma :=
{ foralls := [tyProp; tyProp]
; premises := [Var 0; Var 1]
; conclusion :=
  App (App (Inj ∧) (Var 1)) (Var 0) }.

Lemma AI : ∀ P Q, P → Q → P ∧ Q. (* ≡ closed_lemmaD lem_AI *)
```

Applying a Lemma

The heart of the `eauto` prover is applying lemmas. While an efficient implementation would use a discrimination tree to speed up the process, the algorithm that I present here does not. Since the procedure is implemented within Gallina, nothing prevents us from building a more sophisticated implementation except for the effort required to prove it sound.

The general procedure for applying a lemma has four steps. To be concrete, I will explain how they apply the $\wedge$-introduction theorem to the following goal

$$y = x + 3 \wedge \mathsf{Even}\, x$$

Introducing the Lemma First, the procedure pulls the global lemma (`lem_AI`) into the current context, i.e. under the x and y variables. Semantically this leads to the following goal

$$(\forall P\, Q, P \to Q \to P \wedge Q) \to y = x + 3 \wedge \mathsf{Even}\, x$$

Since environment weakening is free in MirrorCore's representation, this step is purely logical and introduces no computational overhead.

Next, new unification variables are introduced for P and Q and the lemma in the context is instantiated with them. This results in the following goal

$$(?1 \to ?2 \to ?1 \wedge ?2) \to y = x + 3 \wedge \mathsf{Even}\, x$$

Here, weakening the goal with new unification variables is free but instantiating the lemma requires real work.

Unification Once the lemma is in the context and has been specialized with unification variables, the conclusion of the lemma $(?1 \wedge ?2)$ is unified with the current goal using a reflective implementation of unification. Unification answers the question: "What values for $?1$ and $?2$ ensure that $?1 \wedge ?2$ equals $y = x + 3 \wedge$

Even x?" In this case, unification produces the following instantiation:

$$?1 = (y = x + 3) \quad \wedge \quad ?2 = \mathsf{Even}\, x$$

If unification succeeds, then the lemma applies, and proving the premises is sufficient to prove the goal. While this unification solved all of the newly introduced unification variables, this is not always the case. For example, when applying a transitivity lemma, the intermediate value is not mentioned in the conclusion and will therefore only be solved when solving the premises.

The current MirrorCore implementation of unification is first-order and only supports structurally identical terms. However, the specification permits much richer unification algorithms since it is stated using propositional, rather than definitional, equality. For example, while Coq is unable to unify $x+y$ with $y+x$ for opaque values of x and y, MirrorCore's unification specification admits this type of reasoning since the terms are provably equal. Due to the need to have the type of the terms in order to express the provable equality, MirrorCore's unification algorithm is also type-aware, which allows it to unify modulo η-conversion, e.g. unifying x with y for any x and y of type unit (the type inhabited by a single element).

Strengthening After all the premises have been solved, it is necessary to check that all of the unification variables have been instantiated. Proof theoretically this corresponds to checking that the proof that the procedure "constructed" (though the procedure did not actually construct it) has no remaining holes.

Ensuring that there exists a term for each unification variable requires that instantiations do not contain cycles. To see the issue, consider the following two unifications:

$$?1 = 1+?2 \qquad\qquad ?2 = ?1 + 1$$

While it appears that both ?1 and ?2 are instantiated, there is no term that satisfies both constraints. The problem lies in the cyclic dependence between ?1 and ?2. Since expressions are defined inductively, all expressions must be of finite size, and

given the two constraints no such expression exists (either syntactically or semantically) since unfolding the constraints results in the following equation..

$$?1 = 1 + (1 + (...?1... + 1) + 1)$$

In its phrasing, this property is orthogonal to the soundness of unification, but the invariant must still be represented and tracked.

To avoid complex reasoning about acyclicity MirrorCore expresses this property using the `WellFormed_subst` predicate that I showed in the specification for `eprove`. The following theorem expresses the meaning of well-formedness of substitutions

```
Theorem no_infinite_terms : ∀ sub,
  WellFormed_subst sub →
  ∀ u e u',
    lookup u sub = Some e → lookup u' sub ≠ None →
    mentionsU u' e = false.
```

where `mentionsU u' e = false` states that the unification variable `u'` does not occur *syntactically* within `e`. Using this lemma allows the algorithm to conclude that all variables are instantiated if the substitution has a mapping for each one.

Conclusion In the previous three subsections, I described the algorithm for applying an arbitrary lemma. This is the core component of the reflective `eauto` procedure. Around this core, MirrorCore's backward prover includes support for calling custom reflective procedures similar to Coq's `Hint Extern`. Using the reflective procedure is quite simple. Create a list of reified lemmas and pass them to the procedure along with the goal to solve and voilà! The result is sound as long as all of the lemmas are provable.

While not conceptually new (the algorithm is a simplification of the one implemented in Coq and many other proof assistants) this is the first reflective procedure that has made use of a fully internalized unification algorithm within inten-

sional type theory[5]. Ultimately, it demonstrates the degree of meta-reasoning that is possible when given the ability to manipulate unification variables reflectively.

3.5 Related Work

In Chapter 2, I discussed a variety of work related to computational reflection in general. That work is relevant to this chapter and the rest of the dissertation. In this section I discuss work related to syntactic representations of languages and unification.

3.5.1 Term Representation

Unlike most work in computational reflection which defines a very narrow syntax for terms, MirrorCore chooses a term representation that closely resembles the simply typed lambda calculus. Using this term representation, operators in the more narrowly defined syntaxes become symbols that are embedded inside the generic reprsentation [87]. This representation naturally allows for partial application and higher-order terms.

MirrorCore's representation is similar to the representation discussed in Garrilot et. al.'s work on representing simple types in type theory [80]. Their work highlights some of the difficulties that I highlighted in the chapter, predominantly in the phase distinction necessary for handling abstractions. The work does not have a true implementation. Their paper remarks "if we locally switch off the type checker;" however, the central insights are clear. MirrorCore's representation makes these ideas precise while retaining much of the simplicity. The authors highlight the theoretical interest of the construction in terms of completeness of a syntactic logic, which is closely related to the strength of a reflective procedure that operates on the syntax.

[5]The original unification algorithm was implemented in MirrorShard [111] which used it for rewriting.

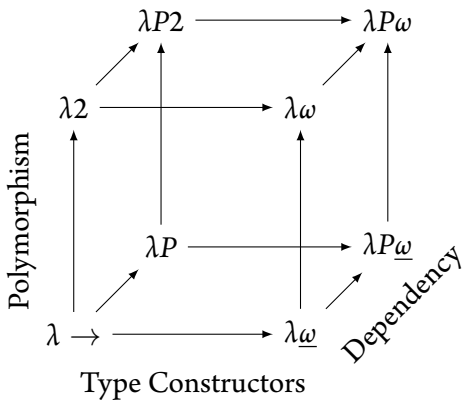

Figure 3.14: Barendregt's lambda cube. Moving to the right allows types to depend on types, e.g. type functors like `list`. Moving up allows terms to depend on types, e.g. polymorphic functions like $\lambda x.x$. Moving into the page enables types to depend on terms (dependent types), e.g. `vector 3`.

Lescuyer's reflective SMT solver [107] also uses a representation similar to MirrorCore's encoding of terms using environments. This representation is well-suited for SMT-style reasoning about uninterpreted functions, but he does not provide any way to reason semantically about these environments. A common way to avoid this problem is to generalize the appropriate lemmas and reify them with the problem. While this works for first-order data, it does not work well for custom procedures because their soundness proofs are non-trivial. This solution also introduces additional cost of reflecting lemmas and processing them on each reflective invocation. The key insight of MirrorCore is the constraint formulations that enable custom procedures and first-order data to be reused across reflective calls. Lescuyer's representation also does not provide lambda abstraction or unification variables.

While MirrorCore does not handle all of the complexities of dependent type theory, it forms a sweet-spot that avoids much of the complexity while still being useful. For comparison, recall that HOL [88] and Isabelle [123] are popular proof assistants with many users and many successes [98, 120, 121], and neither of these

systems contains dependent types at all.

MirrorCore's choice of the simply-typed lambda calculus makes it simple to embed but means that some features such as polymorphism and type constructors are second-class. Barendregt's lambda cube (Figure 3.14) demonstrates three dimensions of extension: polymorphism, type constructors, and dependent types. The MirrorCore repository [108] contains toy implementations enriching the core language to support (independently) polymorphic symbols and type constructors in a first-class way. The implementations retain the simple pieces of the representation and are therefore likely to also have good computational properties. What makes these extensions feasible while retaining much of the simplicity is that neither requires tightly interwoven contexts since the type context does not depend on the term context. De Bruijin showed how this dependency can be accomplished using "telescopes" [48] but the encodings in Gallina become significantly more complex.

Reflecting Dependent Types While dependent types are not necessary, they are useful, and reflectively reasoning about them provides an impressive system for automation. Several researchers have built formalizations of a dependent type theory within itself [23, 53, 65, 115, 152]. Much of this work relies fundamentally on more advanced features than Coq currently supports, for example, inductive-inductive [11] definitions which allow simultaneously defining an inductive type and its inductively defined index.

In 2007, Danielsson [65] formalized a core dependent type theory inside of AgdaLite, a precursor to Adga [8]. Unlike MirrorCore, Danielsson's work uses strongly typed terms, essentially expressing the typing derivation as opposed to the actual term. The formulation relies on induction-recursion and non-positive inductive types, neither of which was fully formalized within AgdaLite, though Dybjer and Setzer proved that they have a set-theoretic model [77]. Beyond this, the formalization includes results about normalization by evaluation and witness substitutions as opposed to functions that compute substitutions.

Two years later, Chapman developed another formalization of dependent type

theory [53] inside of Agda. Chapman's work is significantly heavier than Danielsson's. For example, the entire formalization is relational, never defining an actual denotation *function*. This allows Chapman to avoid the inductive-recursive definitions that lie at the heart of Danielsson's work. While the lack of an interpretation function may be considered a non-starter for computational reflection, Chapman suggests that the Bove-Capretta method [39] can be used to derive (somewhat mechanically) a functional implementation from a proof of strong normalization. The intricacies of the encoding, however, are likely to be expensive in practice.

A year later, McBride revisited the problem using the meta-language's (in his case Agda's) equality to handle the complexity of reduction and typing [115]. He achieves this by avoiding a complete functional representation of types, instead indexing terms by their types and leaving the type checking to Agda. This approach works well for McBride's motivation of embedding domain specific languages within dependent type theory. It would be interesting to see how these ideas could be adapted to MirrorCore. The core ideas are fairly similar to those that make MirrorCore possible though the implementation relies heavily on dependent types and advanced features not available in Coq.

Shulman also recently raised the question of self-representation in the context of Homotopy Type Theory [135]. Shulman's question is slightly closer to that of MirrorCore because he desires a weakly typed representation of terms where previous formalisms have opted for a completely dependent style. This has desirable properties from a computational point of view (since dependent types do not have to be carried around). However, previous work gives a strong indication that the complexity of the denotation function will rely in a fundamental way on deep normalization arguments that are expensive in their own right.

Unlike the previous proposals, Shulman provides something of a negative result though also an open question as to whether it is truly possible. For him, a mathematician and homotopy type theorist, the core question is in representing the coherence conditions for higher types. In an ideal world, everything would be defined up to the equivalence relation induced by the next universe, but such a definition seems to be too extensional to work.

Barzilay's work [25, 26] on computational reflection in Nuprl is a notable success in full reflection of a dependently typed language used for practical purposes. Significant pieces of the Nuprl system are based on the reflective automation developed in that work. The key power that Barzilay is leveraging is the extensional nature of Nuprl where computation is inherently untyped. In the extensional setting, the complexities associated with casting that I discussed in Sections 3.2.1 and 3.2.2 disappear. In addition, since computation is untyped the strong normalization proof that all of the above formalisms work hard to build and maintain in the denotation function is kept separately in the typing derivation. This means that it incurs no additional overhead during computation the way it would in intensional theories.

Another recent success in self-representation is Brown's work on self-representation in Girard's System U [47]. While not dependent, System U does provide polymorphism and type functions that would be useful additions to MirrorCore's representation. Brown's work builds on top of higher-order abstract syntax (popularized by Twelf [129]) and is focused on meta-programming.

3.5.2 Representing Constraints

New ideas in proof assistants have been proposed to find a middle ground between intensional and extensional equality. There are important reasons to have both, but the theory of a system involving both is not completely worked out yet in a type theory as rich as Coq's. Andromeda [27] is one attempt to unify the two based on a proposal by Voevodsky [150]. The price paid for this generality is decidability of type checking of terms.

3.5.3 Composable Syntax

Swierstra presented an alternative solution to the composable syntax problem in his work on "Datatypes a. la. carte" [144]. His approach relies on describing inductive data types as functors and using a special fixpoint inductive type to "tie the knot." Figure 3.15 shows the merging of two languages, one with plus and the

```
Inductive Eplus (E : Type) : Type := (* plus language *)
| Plus : E → E → Eplus E.
Inductive Econst (E : Type) : Type := (* minus language *)
| Const : ℕ → Eminus E.
Inductive Either (F G : Type → Type) (E : Type) : Type :=
  (* disjunction *)
| Left : F E → Either F G E
| Right : G E → Either F G E.
Inductive Mu (F : Type → Type) : Type := (* fixpoint *)
| MuF : F (Mu F) → Mu F. (* not accepted by Coq *)

(* the combined language *)
Definition plus_and_const := Mu (Either Eplus Econst).
(* or *)
Definition const_or_plus := Mu (Either Econst Eplus).
```

Figure 3.15: Combining two languages using Swierstra's technique from "Datatypes a. la. carte."

other with constants. Here, we define two type functors (`Eplus` and `Econst`) that abstract over the type of expressions. We can then combine them into the complete language using `Either` and take the fixpoint of the functor using `Mu`.

Coq rejects the above definition of `Mu` because in the `MuF` constructor `Mu` does not occur strictly positively, a restriction related to domain theory that justifies that the inductive type is well-behaved. While the general definition is rejected, the idea works for particular choices of `F`. In fact, while the requirements on `F` are not expressible in Coq, Agda does admit this type of definition [7] by allowing quantification over strictly positive functions[6].

Beyond being expressible in the current version of Coq, MirrorCore's universal representation has several additional benefits. First, note that `Plus` and `Minus` are second-class. This means that they cannot be partially applied or passed to functions. Prior to Coq 8.4pl4 this restriction prevented MirrorCore from rea-

[6]Since Agda does not have an impredicative universe, the requirement of strict positivity is relaxed to simple positivity.

soning about partial function application, e.g. `map (plus 1) ls`. Using η for functions enables representing this expression as `map (fun x ⇒ plus 1 x) ls`. Second, note that the exact order of constructing the final type is essential. While `plus_and_const` is isomorphic to `const_or_plus`, the two types have different representations. This means that procedures or proofs written with one composition in mind must be rewritten for the other representation or the terms must be translated. MirrorCore's representation choice, on the other hand, provides a canonical representation of terms that can be carved up using the techniques in Sections 3.2.1 and 3.2.2.

Delaware [70–72] showed how to use Church encodings [130] to circumvent the strict positivity limitation and encode Swierstra's extensible representation in Coq. While this approach still has the same problems for computational reflection that Swierstra's approach has, the ideas in this work are useful when writing generic functions over the syntax that exposes high-level structure.

3.5.4 Unification

Unification is a core procedure used in the checking of both standard programming languages and proof assistants and has thus been a topic of research since the 1970s. Some of the seminal work on higher-order unification was done by Huet [91] while developing the foundations of proof assistants. In the higher-order case, which Huet addresses, there is no principle unifier of two terms, so completeness is impossible. MirrorCore's unification algorithm avoids the complexity of higher-order unification by requiring terms to match up exactly, i.e. it does not do any reduction. MirrorCore's interface for unification does allow for higher-order unification as well as unification modulo theories [19]. This generality is due to the specification being defined using propositional, rather than definitional, equality.

McBride [113] showed an elegant algorithm for expressing the acyclicity check for first-order unification using dependent types. By making the acyclicity proof manifest, he is able to use it to prove termination, thus providing a complete, first-

order unification algorithm. MirrorCore's algorithm, on the other hand, explicitly avoids constructing this proof object, which my experiments have shown can be costly when running within the proof assistant (see Chapter 4). The ability to erase proofs during execution would enable a cleaner interface for substitutions without sacrificing performance.

McBride's algorithm handles a "uniform prefix," where all existential quantifiers have the same context. Miller discusses the more general problem of unification with a mixed prefix [119]. Many of these techniques involve skolemization, which tries to lift quantifiers and place restrictions on them, for example, converting $\forall x : \mathbb{N}, \exists y : \mathbb{B}, \ldots y \ldots$ into $\exists y : \mathbb{N} \to \mathbb{B}, \forall x : \mathbb{N}. \ldots y\, x \ldots$. While not necessary for the backwards prover presented in this chapter, MirrorCore does support mixed prefixes though in an incomplete way.

Very recent work on computational reflection has also built a backwards proof procedure in Agda [99]. While the authors remark that the implementation is quite slow, it side-steps the complexity of defining it within the theory.

4

Engineering Reflective Automation

The principal motivation for computational reflection is often performance. In this chapter I discuss the engineering considerations that contributed to the design of MirrorCore. In some cases this performance comes at the cost of more complex proofs, and throughout the chapter I point to promising avenues of current and future work that might ameliorate some of the pain points.

I begin with an overview of the relevant reduction mechanisms that Coq provides (Section 4.1). Two features are important for computational reflection. First, in some cases it is important to control reduction by only reducing certain symbols. This feature plays a crucial role in constructing proof terms that interface with non-reflective proofs. And second, some reduction mechanisms are annotated in the proof term and can therefore be used during proof checking, while other mechanisms are used by tactics but not included in the proof term, which can cause proof checking and proof generation to have very different run-time properties.

Next I discuss empirical results related to the use of dependent types in computation (Section 4.2). Coq's rich type theory makes it possible to express precise types that blur the line between programming and proving. While this is not *a priori* a bad thing—in fact strong types can simplify programming dramatically—they can incur run-time overhead both by constant factors and asymptotically.

Achieving good performance with computational reflection also requires constructing good proof terms for invoking the reflective procedure. While tactics usually construct proof terms in a relatively *ad hoc* manner, it is worthwhile tuning both the proof terms and the scripts that generate them. In Section 4.3, I discuss both of these challenges as they relate to the current version of Coq. The techniques highlight the importance of low-overhead interfaces to the proof kernel since both interpretive overhead and redundant computation can be devastating to the goal of building fast proofs.

Finally, I describe some technical details related to reification, the process of building a syntactic representation of a semantic term (Section 4.4). Reification is often overlooked in formal presentations, but fast reification is essential to achieving good performance with computational reflection.

> **Nota Bene** As Coq evolves the need for some of these choices may lessen and the need to make other choices may arise. All performance evaluations in this chapter are based on Coq 8.4pl4, and all of the suggestions are true for earlier 8.x versions of Coq as well.

4.1 Coq's Reduction Mechanisms

The workhorse of proof by reflection is reduction, and having efficient computation is essential to making computational reflection fast. While definitional equality is a single relation, there are several implementations of it in Coq. These implementations trade efficiency for customizability and are useful for different tasks. In this section I discuss three of Coq's implementations of reduction with an eye towards how they affect computational reflection.

Virtual Machine Reduction The most efficient implementation of reduction currently available in Coq is virtual machine (VM) reduction [84][1]. This strategy compiles Gallina code to a small virtual machine implemented in C and runs it there. All-in-all virtual machine reduction is often approximately 100x faster than Coq's next fastest reduction technique (which I will discuss shortly).

There are two draw-backs to virtual machine reduction. First, VM reduction completely reduces terms. This works very well when computing on values, e.g. when computing the result of the reflective procedure. However, when opaque terms or quantified variables occur they block reduction. When this happens, reduction often builds a very large term that is expensive to represent. Second, VM reduction fails on terms that mention unification variables. From an engineering point of view, this is because VM reduction is implemented in the kernel, which does not understand unification variables (since they are not part of the type theory). This limitation re-enforces the need to maintain a strong separation between the semantic meaning of terms in computational reflection and their syntactic representation. However, the kernel implementation also has a large benefit: VM reductions are noted as such in the proof term so the kernel uses them when checking the final proof.

In many cases, though not all due to dependent types, both of these limitations can be overcome by abstracting unification variables and terms that we wish to hide from reduction. Braibant's `evm` plugin [45] provides useful tactics for performing this abstraction and is discussed in more detail in [111].

Delimited Call-by-Value Next in the efficiency spectrum is delimited call-by-value. Unlike VM reduction, delimited call-by-value is programmable by specifying particular reduction rules, e.g. β- (function call) or ι-reduction (match reduction). In addition, δ-reduction (replacing names with their definitions) is customizable by either a black-list or a white-list of identifiers to reduce. This is useful

[1]Coq 8.5 will also feature a reduction strategy that compiles terms to native code via OCaml [37]. It is currently unclear if compilation will incur too much overhead to make native compilation effective for computational reflection.

when reducing a denotation function but not the semantic terms inside of it. Even this customizability is sometimes insufficient. For example, when both the denotation function and the term use the same function it is desireable to reduce only certain instances of the function, which is not currently possible.

Unlike VM reductions, the customizations to call-by-value reduction are not included in the final proof term, so the kernel falls back on lazy reduction (which I discuss shortly) to perform the actual check. This can have a substantial impact on the proof checking time compared to the time it takes to build the proof.

Lazy Unification The kernel's core algorithm for determining if two terms are definitionally equal performs lazy reduction of the terms until their head symbols match and then continues unifying the arguments pointwise[2]. In the common case when terms match up exactly, or very closely, lazy unification behaves very well. However, as more computation is needed the overhead of the lazy reduction becomes significant.

In addition, when checking that two terms are equal Coq must select which term to try to reduce to the other. For example, consider testing the following two terms for equality:

```
id (fib 100)        fib 100
```

If Coq chooses to reduce the former to the latter it only requires a small amount of work to evaluate `id`. On the other hand, if Coq attempts to reduce the latter to the former Coq will reduce the entire term before it attempts to reduce the other. Emperically, reducing the second to the first takes 0.001 seconds while reducing the later to the former takes almost 20 seconds.

[2]This is a significant simplification of the algorithm, but the performance characteristics are similar.

Non-Dependent

```
Inductive expr : Type :=
| Const : ℕ → expr
| Var : ℕ → expr
| App : expr → expr → expr
| Abs : typ → expr → expr
```

Dependent

```
Inductive wtexpr
: list typ → typ → Type :=
| wtConst : Π ts, ℕ → wtexpr ts tyNat
| wtVar : Π ts t, member t ts → wtexpr ts t
| wtApp : Π ts d r, wtexpr ts (d ⇛ r) →
    wtexpr ts d → wtexpr ts r
| wtAbs : Π ts d r, wtexpr (d :: ts) r →
    wtexpr ts (d ⇛ r).
```

Figure 4.1: Two representations for lambda terms. The dependent representation guarantees that terms are well-typed.

4.2 Engineering Verifiable, Executable Code

A central question in developing code with both efficiency and verifiability in mind is the representation of information. Tight integration of data and invariants often makes it easier to verify code, while looser integration often allows for more efficient computation. These invariants are necessary to prove soundness. The essential difference is where the information is required, constructed, and maintained.

For illustrative purposes consider the two representations of a simple λ-like language similar to MirrorCore's shown in Figure 4.1. On the left, the type conveys minimal information about the meaning of the term. On the right, the `wtexpr` type conveys the entire typing derivation including the variable context and the type of the term. This means that any term of type `wtexpr ts t` is guaranteed to be well-typed at type `t` in a context that contains `ts` variables. This makes the denotation function for `wtexpr` total and avoids the need to interleave type inference and denotation since all of the typing information is embedded in the term. Indexing the type of terms also makes it easy for functions to express the contexts of terms that they accept and compute. This makes it harder to make simple mistakes when manipulating terms.

Figure 4.2 shows how the choice of representation affects the performance of computation. Terms are constructed according to the following syntax to mini-

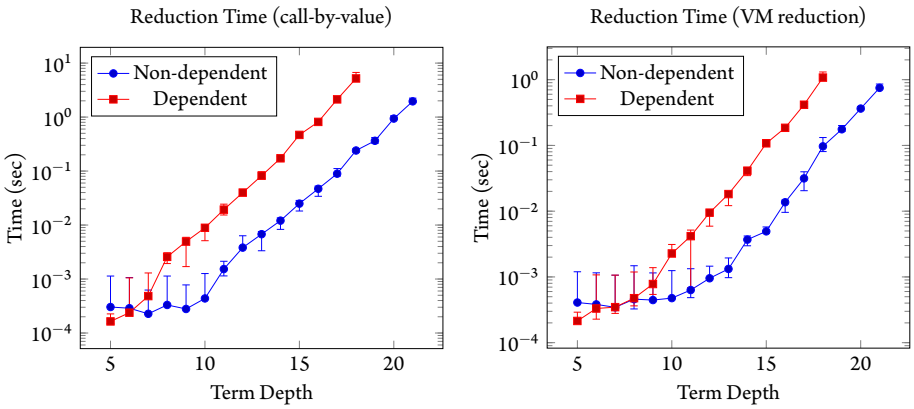

Figure 4.2: Performance characteristics under call-by-value and VM reduction of the dependent and non-dependent term representations. Coq cannot represent the dependent term for depths larger than 16.

mize the amount of term reduction and maximize the amount of syntax.

$$e_0 = 0 \qquad e_{n+1} = (\lambda x.e_n)\, e_n$$

The denotation function for the dependent syntax is a straightforward translation of the denotation function in Chapter 3. The non-dependent denotation is the subset of the denotation function that MirrorCore uses which performs simultaneous type checking using mutual induction. The dependent representation stops early (at depth 16) when Coq can no longer construct the representation in a reasonable amount of time (Coq takes almost 5 minutes to construct the term of depth 16 using VM reduction).

A quick look at the axes of the two graphs shows that VM reduction is about 10x faster than Coq's call-by-value reduction strategy on this problem. Unfortunately, VM reduction is too aggressive to be used when reducing the denotation function in computational reflection. Using it unfolds all of the symbols, building terms that take too long to unify.

In both cases, the non-dependent representation is an order of magnitude faster than the dependent representation. This may seem counter-intuitive because the non-dependent code must do a fair amount of error checking since the denotation

function is not total. However, the extra data associated with the constructors in the dependent representaiton is quite substantial, and carrying around this information is expensive.

Another property of the representation that is important to consider is the size of terms since the terms will appear within the final proof. The additional indicies makes the size of terms in the dependent representation grow significantly more quickly than the size of terms in the non-dependent representation. This can be mitigated to some extent by introducing local names for pieces of the term that are repeated. For example, constructors can be partially applied to parameters to avoid repeating the parameters throughout.

The larger worry, however, is the cost of type-checking the term. Even seemingly small terms can take time to type check. For example, if `wtexpr` is extended with symbol injection, any use of this constructor would require type checking the symbol. This would require either large terms or reduction which the kernel will perform using the lazy evaluation strategy.

Asymptotics & Dependency

Not all of the cost of dependent types is accurately captured by the size of the term and the performance of the denotation function. By reducing the amount of information captured by the type, a single term can represent many terms, and converting between them becomes a zero-cost operation.

The easiest way to see this is with a function that weakens the context of a term. The type of the dependent weakening function is the following:

```
Definition wtweaken : Π ts ts' t, wtexpr ts t → wtexpr (ts ++ ts') t.
```

This function recurses over the term and rebuilds it using the new value for `ts`. The cost of this is linear in the size of the term. In the non-dependent representation, on the other-hand, weakening is free because well-typed terms weaken to themselves.

```
Definition weaken : expr → expr := fun x ⇒ x.
```

The computation done by `wtweaken` is not completely lost however. To see where it has gone, consider the soundness theorem that states that weakening an expression does not change its meaning. Using the dependent representation the soundness theorem is the following.

```
Theorem wtexprD_weaken : ∀ ts t (e : wtexpr ts t) ts' vs vs',
  (exprD ts t e) vs =
  (exprD (ts ++ ts') t (wtweaken ts ts' t e)) (vs ++ vs').
```

Using the non-dependent representation the soundness is a bit more complicated:

```
Theorem exprD_weaken : ∀ ts t (e : expr) ts' eD,
  exprD ts t e = Some eD →
  ∃ eD', exprD (ts ++ ts') t e = Some eD' ∧
    ∀ vs vs', eD vs = eD' (vs ++ vs').
```

Note that in the non-dependent version the theorem is forced to prove that if the given term has a denotation in the stronger context then it also has a denotation under the weaker context. This property is exactly what `wtweaken` is spending its linear time to compute. Further, note that the non-dependent weakening lemma requires an explicit proof that the term being weakened is well-typed (i.e. the fact that the denotation returns `Some`). Changing `exprD_weaken` to directly state the equality like `wtexprD_weaken` results in a theorem that is not true because weakening an ill-typed term can produce a well-typed term, which will clearly have a different denotation.

A Fully Dependent Representation In addition to the types, terms can also be indexed by their contextualized values. Doing this allows the weakening function capture the entirety of the weakening lemma.

```
Definition dweaken : Π ts ts' t (val : hlist typD ts → typD t),
   dexpr ts t val →
   dexpr (ts ++ ts') t (fun ctx ⇒ val (fst (split ctx))).
```

	`expr`	`wtexpr`	`dexpr`
function	O(1)	O(n)	O(n)
soundness	O(n)	O(n)	O(1)

Figure 4.3: The distribution of work between the function and the soundness theorem. `dexpr` does all of the work in computation, `expr` does all of the work in soundness, and `wtexpr` does the well-typedness during the computation and the semantic equivalence in the soundness theorem.

In this definition, `split` extracts the beginning of the extended context which it passes to `val`.

While quite cumbersome to work with, the fully dependent representation does complete the picture of computational asymptotics (Figure 4.3). Note that all of the implementation choices do the same amount of work. What changes between the implementations is *where* the work is done. With the non-dependent representation all of the work is done in the soundness proof, while in the fully dependent representation all of the work is done in the function. The well-typed representation does half of the work in the function (the well-typedness part) and the other half in the soundness proof (the denotation part).

Beyond the computation there is another crucial property to note here: the soundness theorem relies on functional extensionality, which must be an axiom in Coq. Since functional extensionality is only necessary when reasoning about the contextualized value, it is not necessary when implementing `weaken` or `wtweaken`. In the fully dependent representation, on the other hand, functional extensionality is necessary for the weakening function itself. This means that reducing a call to `dweaken` will get stuck on the axiom and not fully reduce. Here the division of labor effectively hides the axiom from the computation.

Bi-directional Type Checking One improvement that would greatly improve the term representation is bi-directional type checking [132]. Bi-directional type checking more closely integrates terms with their types which allows some pieces of them to be elided. For example, all of the parameters except for the mem-

bership proof of `wtVar` are completely determined by its type. Making the type checker aware of this dependency enables these redundant terms to be omitted entirely rather than just hidden by Coq's mechanism for implicit arguments.

The forthcoming Coq 8.5 will feature first-class records which will support a restricted form of bi-directional typing for record projections. While useful this feature is not sufficient, since it does not erase the values from the constructors.

4.2.1 Proofs, Compression & Computation

The representation in the previous section integrates the data and its properties. This is computationally useful, but it makes the division of labor that I highlighted at the end of the last section less obvious. An alternative is to split the property from the data that it holds on. The following alternate definition captures the well-typedness of terms using Coq's dependent pair type.

```
Definition wtexpr' ts t : Type := { e : expr | WellTyped ts t e }.
```

If `WellTyped` is an inductive type, it is the same size as `wtexpr`, and there is extra data (but no extra information) in the `expr`.

But there is a potential benefit to this formulation if the property is decidable. Using computational reflection we can compress the large inductive proof into a small proof of equality.

```
(is_well_typed ts t e = true) ⟺ WellTyped ts t e
```

Substituting equals for equals in the definition of `wtexpr'` allows us to build an equivalent type that is much smaller.

```
Definition wtexpr'' ts t : Type :=
  { e : expr | is_well_typed ts t e = true }.
```

Values of this type are essentially the same size as `expr` but also carry the additional information that they are well-typed, and therefore the denotation function can be made total.

While more space-efficient to represent the value, computation on the value demonstrates where the work has actually gone. In each case, rather than performing the error handling, the denotation must use the well-typedness proof to prove that the sub-terms are well-typed before making the recursive calls. Thus we have traded error handling for manipulating the proof term, which is often more expensive since the type of subterms cannot be easily extracted from the computational proof term.

4.2.2 Reasoning About Simple Types

One place where proof objects are constructed and often ignored is in predicate testing functions such as those that test for equality and inequality. For example, consider the task of determining whether two natural numbers are equal. Coq's standard library provides two functions for this, one that returns a boolean and the other that returns a sum type carrying proofs.

```
beq_nat : ℕ → ℕ → bool
eq_nat_dec : Π n m : ℕ, {n = m} + {n ≠ m}
```

The type of `eq_nat_dec` is much more informative since it states exactly what "true" and "false" mean. However, to satisfy the type the function must construct the corresponding proof. Figure 4.4 shows how this can affect performance. While the times seem small, the frequency of comparing terms makes this optimization valuable in practice. In the MirrorShard system that I present in the next chapter moving from `eq_dec` to `eq_bool` resulted in a 40% reduction in time[3].

Post-facto Reasoning While slower, the more informative type makes the dependently typed function significantly easier to reason about. For example, a non-dependent match on "`eq_nat_dec a b`" provides either a proof that "`a = b`" or that "`a ≠ b`". Matching on "`beq_nat a b`", on the other hand, provides no useful information. In fact, to get the result we need to appeal to another proof that

[3]At this point, MirrorShard did not use VM reduction.

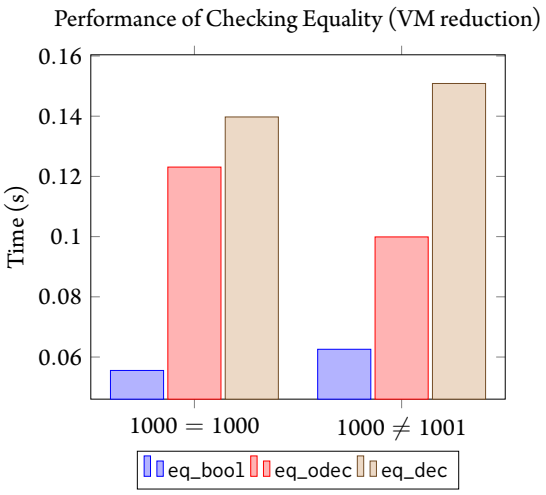

Figure 4.4: The performance cost of constructing proof terms with `vm_compute`. Numbers correspond to the time it takes to run the function 1000 times.

connects `beq_nat` to the equality or inequality proofs.

Reasoning about less informative types after the fact can be cumbersome. There are specialized tactic patterns for learning information from case analysis. For example, the SSreflect [81] tactic library uses the `case` tactic which generates a dependent match on a datatype indexed by the term (such as `beq_nat`) rather than matching on the term directly. While the idea is simple, it is still burdensome to remember the names of theorems.

The ExtLib library [3] on which MirrorCore is built alleviates this burden of remembering the extra name by using Coq's dependent type classes [137] to associate soundness theorems with their functions. Traditional type classes carry additional information about *types*, e.g. an equality decider. Dependent types enable type classes to carry information about *values*. For example, a type class indexed by a function can carry a proof about that function:

```
Class EqOk {T : Type} (f : T → T → bool) : Type :=
{ eq_ok : ∀ x y, f x y = true ↔ x = y }.
```

When proofs and automation reference the symbol `eq_ok`, Coq's type class reso-

lution will attempt to find an appropriate instance. This approach is similar to the use of type classes in the math classes project [138]. With this resolution under the hood, ExtLib provides a tactic `consider` that finds the most useful information to give when performing case analysis on a term. This technique has proven quite useful. Using it, it is possible to write general type classes that combine information allowing tactics to learn useful information by considering complex terms such as "`eq_b a b && eq_b c d`" to get either "$a = b \wedge c = d$" or "$a \neq b \vee c \neq d$".

Beyond performance, loosely coupling functions with their specifications can enable greater reuse. For example, the same function can have several spefications allowing it to be used in different contexts.

4.2.3 Conclusions

Performance evaluation showed that the dependent representation was too expensive without additional kernel functionality. This result is directly related to the combination of the smaller representation and zero-cost weakening, which is a common operation especially when using lemmas. While the size of the dependent representation can be mitigated through certain techniques that I discussed earlier, zero-cost weakening is an asymptotic improvement that is more difficult to regain. Introducing a weakening constructor and lazily weakening terms is one potential solution, but orchestrating this while avoiding repeating large amounts of work seems difficult. Haskell-style lazy evaluation could be a saving grace here, but the complexity that goes into efficient implementations of lazy evaluation seems ill-suited for the kernel of a proof assistant.

4.3 Crafting Proof Terms for Computational Reflection

Even after all of the Gallina code is written and all of the proofs are done, there are still important techniques for crafting the right proof term and efficiently communicating it to Coq. This topic is not usually discussed in the literature, but getting it right can translate into minutes or more saved when checking large developments.

4.3.1 Phrasing Soundness

The key to efficiently using computational reflection is the statement of the correctness theorem. The general soundness theorem for reflective procedures comes in two forms depending on whether the result of the reflective procedure is known before hand or not.

```
Theorem known_result : ∀ hs g,
  rtauto hs g = true →
  ∀ ps, Impls ps hs g.
  (* ⟦hs1⟧ → ⟦hs2⟧ → .. → ⟦g⟧ *)
```

```
Theorem unknown_result: ∀ hs g g',
 rtauto_simplify hs g = Some g' →
 ∀ ps, propD ps g' →
       Impls ps hs g.
```

Here, `ps` is the environment of symbols that the denotation function will use to fill in terminal values and the `Impls` function sets up an iterated implication between the denotation of each of the hypotheses (`hs`) and the goal (`g`). These two theorem statements demonstrate two guidelines to follow when building theorems that can be efficiently used to invoke reflective procedures within proofs.

First, both theorems have a separate equation stating the reduction of the reflective procedure. This makes it possible to use VM reduction to perform the bulk of the work and still be able to use the customizable call-by-value reduction to reduce only the denotation function on the final result if necessary. Note that the term being reduced is a closed term with no opaque symbols that could interfere with reduction.

Second, both theorem statements avoid extraneous data by passing multiple values as separate function arguments rather than building a tuple of proofs. When conjunctions are necessary as premises to the theorem, convert them into implications using a definition such as `Impls` that produces a chain of implications. When they are "returned" to the user, write the new obligation as an implication using `Impls` so that the new premises can be introduced by the `intros` tactic without requiring explicit matching in the remainder of the proof.

```
1 let hs := pRef 0 :: pRef 1 :: nil in
2 let g := pAnd (pRef 0) (pRef 1) in
3 @known_result
4   hs (* syntactic hypotheses *)
5   g (* syntactic goal *)
6   (* The computation proof with a vm_cast *)
7   (@eq_refl bool true <: rtauto hs g = true)
8   (P :: Q :: nil) (* values for [pRef 0] and [pRef 1] *)
```

Figure 4.5: The proof script for applying reflective procedures that have known results.

4.3.2 Building the Proof Term

Beyond the phrasing of the soundness theorem, the precise structure of the final proof term and how it is communicated to Coq are also important.

The proof term for applying the reflective procedure (Figure 4.5) is, not surprisingly, closely related to the soundness theorem. Two pieces of this are important. First, the proof uses `let`-declarations (lines 1 and 2) to avoid repeating terms. More than space savings, this avoids re-typechecking the terms. Second, the proof of the equality is stated explicitly using `eq_refl`, and an explicit VM type ascription is used to asssert it also has a type that matches the premise of the soundness theorem (notated <:). The term on the left has type `true = true` while the type on the right states that it must have type `rtauto hs g = true`.

Known Result Proof Scripts

Figure 4.6 shows the $\mathcal{L}_{tac}$ proof script used to build this proof efficiently. The first step is to manipulate the goal to match the type of the above term. To do this, we use $\mathcal{L}_{tac}$'s `generalize` or `revert` tactic to move the premises below the line. This results in the following goal which is convertible to the type of the proof term above.

```
P → Q → P ∧ Q
```

```
generalize HQ ; generalize HP ; (* revert the hypotheses *)
set (ps := P :: Q :: nil) ;   (* introduce names for common terms *)
set (hs := pRef 0 :: pRef 1 :: nil) ;
set (g := pAnd (pRef 0) (pRef 1)) ;
change (Impls ps hs g) ; (* change the goal *)
(* build the proof *)
set (pf := known_result hs g (@eq_refl bool true <: rtauto hs g = true));
exact pf. (* use the proof *)
```

Figure 4.6: The proof script used to apply the `known_result` soundness theorem.

Next the script uses the `set` tactic to introduce names for the large pieces of the proof that will be used multiple times. In the example, this is the environment for the opaque terms and the syntactic representation of the hypotheses and goal.

Once the names are introduced, the script uses the `change` tactic to hint to Coq that the goal matches exactly the conclusion of the goal. While this step is not required since Coq will perform this check automatically, the hint tells Coq which way to reduce. In the above goal, "`Impls ps hs g`" simplifies to the implication, while the implication does not simplify to `Impls`. Without this hint, Coq (version 8.4) will attempt to simplify "`P` $\rightarrow$ `Q` $\rightarrow$ `P` $\wedge$ `Q`" into "`Impls ps hs g`" and only try the other way once this fails.

Finally, the script uses the proof term to solve the goal. Naïvely it would use `exact` to solve the goal, but in the 8.4 family of Coq this is not optimal due to extra checks that are performed by `exact`. The solution is to use `set` to introduce the proof and `exact` to use the introduced name to solve the goal. While the proof term is slightly longer, building it is much faster.

Timing results vary considerably based on how much time the reflective procedure and denotation function take to compute, but each component that I described above is important to the final proof. The peculiarities of what checks certain tactics perform should be better documented so that it is easier to determine what tactics should be used. A more "assembly-like" tactic language would be useful for this purpose and possibly also for some of the more aggressive uses of $\mathcal{L}_{tac}$ that currently suffer from performance problems [54, 73].

```
generalize HQ ; generalize HP ; (* revert the hypotheses *)
set (hs := pRef 0 :: pRef 1 :: nil) ; (* introduce names for common terms *)
set (g := pAnd (pRef 0) (pRef 1)) ;
set (ps := P :: Q :: nil) ;
let result := constr:( rtauto_simplify hs g) in
let resultV := eval vm_compute in result in (* call the procedure *)
match resultV with
| Some ?G' ⇒
  (* compress G' if necessary before introducing it into the proof *)
  set (g' := G') ;
  let GD' := eval cbv β ι ζ δ [ ... ] in propD ps g' in (* whitelist reduction
    *)
  cut (GD') ;
  [ (* the goal to solve using the soundness theorem *)
    change (propD ps g' → Impls ps hs g) ;
    generalize ps ;
    cut (result = resultV) ;
    [ (* solve the goal with the soundness theorem *)
      set (pf := @unknown_result hs g) ; exact pf
    | (* insert a vm_cast but do not check it *)
      vm_cast_no_check (@eq_refl _ resultV) ]
  | (* clean up the goal that is returned to the user *)
    clear g' hs g ps ]
end.
```

Figure 4.7: The proof script used to apply the unknown_result soundness theorem.

Computed Result Proof Scripts

When reflective procedures return interesting information that describes additional proof obligations, additional work is necessary to build an efficient proof. The proof term itself is essentially the same as above. The difference is that there is now an additional hole that corresponds to the remainder of the proof. Since this hole depends on the result of the reflective procedure, we need to reduce the procedure before constructing the proof. Figure 4.7 shows the daunting script that applies the `unknown_result` soundness theorem.

The beginning of the proof script is quite similar to the previous proof script. For extra efficiency the script can generalize the hypotheses only if the computation

will succeed, but this introduces no overall difficulty.

To continue, the script needs to reduce the reflective procedure. The script sets up the call in line 4 and performs the reduction in line 5 using VM reduction.

The $\mathcal{L}_{tac}$ match on line 5 parses the result of the computation to extract the pieces that need to be passed to the lemma. In this case, the reflective procedure is only meant to succeed if it returns `Some`, so the `None` case is absent[4]. As before, the script introduces names for potentially large terms (line 10). Note that if `result` can be compressed, e.g. by replacing sub-terms with variables in the context, then that should be done before the `set` statement.

Line 11 computes the resulting goal that will be returned to the user. Here we cannot use VM reduction because it would over-reduce the term. All that we want to reduce is the denotation function. To achieve this, the script uses a delimited form of call-by-value reduction programmed with a whitelist of symbols to reduce. While it can be frustrating to construct the white-list, using `cbv` is often substantially faster than using Coq's `simpl` reduction strategy, which does extra checks on terms to see if unfolding definitions is "useful."

Once the result is known it can be `cut` (line 12), which produces two sub-goals, the second of which will be returned to the user. As before, the script uses `change` to introduce a cast hinting to the type checker the direction to perform the reduction in. In an ideal world, we would be able to hint the entire whitelist, but Coq's kernel does not currently support this kind of annotation.

At this point our problem looks exactly the way that it did before; however, we want to avoid running the reflective computation again. This prevents us from directly applying the `set-exact` strategy that I showed previously. Instead the script needs to use the undocumented `vm_cast_no_check` tactic to build the cast without doing the computation. To use it, we further generalize the predicate list `ps` and then `cut` the equality[5]. The goal for the first branch of the `cut` now matches exactly

[4]The semantics of $\mathcal{L}_{tac}$ means that this tactic will fail if the procedure returns `None`, which is exactly the behavior that we want.

[5]An alternative approach is to lift the `ps` before the equality, which trades the generality of the result for the ease of applying the theorem.

with the partially applied soundness theorem, and the script solves it using `set-exact` (line 18). In the second branch, the script uses `vm_cast_no_check` which acts like `exact` but introduces a VM type ascription without checking it. For example, if the goal is G then "`vm_cast_no_check pf`" produces the proof term "`pf <:` G". This is exactly the proof term that we want, and by using `vm_cast_no_check` the script avoids computing the reflective procedure twice.

4.4 Reification: Building Syntax for Semantic Terms

Most work on computational reflection glosses over the problem of reifying terms. While not particularly glamorous, the reification process can dramatically affect verification time. It is possible to implement reification of terms in $\mathcal{L}_{tac}$ using tricks for recursing under binders; however, the solution is not very elegant.

There are two issues with the semantics of $\mathcal{L}_{tac}$ that make it ill-suited for implementing reification. First, $\mathcal{L}_{tac}$ is built for backtracking proof search, not term manipulation. For example, a typo in a branch of the reification algorithm often leads to exponential backtracking, which can be difficult to debug. This is compounded by the fact that term-producing tactics cannot use `idtac`, $\mathcal{L}_{tac}$'s version of `printf`. To get around this latter limitation, term manipulating tactics are often written in continuation passing style, making them that much more difficult to write.

The second issue deals with type checking terms. In the 8.4 releases, $\mathcal{L}_{tac}$ does not support a way to manipulate terms without eagerly type-checking them. The default of eagerly type checking terms works very well for the use case that $\mathcal{L}_{tac}$ was envisioned for but it is too inefficient for reification. For example, consider the two $\mathcal{L}_{tac}$ programs that build the natural number 1000.

```
constr:(1000)
```

```
let x := constr:( O) in
let x := constr:( S x) in ...
```

The term on the left invokes the type checker once on the term 1000, while the term on the right invokes the type checker 1001 times, once on each of the terms

between 0 and 1000. While one would never write the program on the right, it is exactly the type of behavior that the reification process has. We know that the full term will be type checked once when it is inserted into the proof script, so there is no reason to type check it a second time. In Coq 8.5 $\mathcal{L}_{tac}$ will feature a way to build terms without type checking them immediately, which will solve this problem.

Coq 8.5 will also feature native support for more sophisticated plugins. This support makes the $\mathcal{M}_{tac}$ tactic language [155] which I will discuss in Section 6.4 usable with the standard Coq distribution. $\mathcal{M}_{tac}$'s implementation allows it to construct terms without the overhead of checking them and would allow for a convenient way to reify terms in a type-safe manner.

4.4.1 Plugin-based Reification

When performance is crucial and other solutions come up lacking, we can fall back on writing Coq plugins. When reifying large terms it can be useful to write custom plugins to achieve good performance. For example, in performing reification for a domain similar to MirrorCore, $\mathcal{L}_{tac}$-based reification was 88x slower than plugin-based reification for the same problems. The draw-back of custom plugins is that they become a system-level dependency, and care must be taken to ensure that they run on different platforms, e.g. Windows, Mac, and Linux, as well as different versions of OCaml. Further, as Coq evolves these plugins must be updated to understand the internals of the Coq implementation (as opposed to the type theory that it implements).

One solution to this problem is to try to write generic plugins that can solve many reification problems. For example, Coq's built-in Quote plugin [62] attempts to invert a `Fixpoint` definition using heuristics. While it does support environments, it does not understand two-level languages such as MirrorCore's representation. Agda's AutoQuote library [149] provides a programmable version of Quote where a table of patterns can be associated with terms and used to reify general syntax. This approach avoids the need to use heuristics at the cost of being a little more verbose.

Another approach is to reify the entire core syntax of the proof assistant and then write Gallina functions that process it. TemplateCoq [109] is a Coq plugin that builds a representation of Coq terms that is closest to the kernel's own representation. While TemplateCoq's representation is not the reflected syntax, it can be processed by Gallina functions into the appropriate syntax, though the cost of this could be non-trivial.

MirrorCore's Reification Plugin To deal with the added complications of nested denotations and dependent functions, MirrorCore includes a Coq plugin for programmable reification similar in spirit to Agda's AutoQuote plugin. The type of customization that the plugin supports is best illustrated through an example. Figure 4.8 shows the plugin in action on a syntax definition for natural numbers with equality.

The core feature of the plugin is a reification function which is declared using `Reify Declare Syntax.` Reification functions are built from a collection of declarative combinators. `CApp`, `CAbs`, and `CVar` handle the core primitives: application, lambda abstraction, and variable reference. In each case their last argument is the syntactic constructor that corresponds to the semantic object.

Terms, such as local variables, that are reified using the analog of `pRef` are handled by tables declared using `CTable` and `CTypedTable.` The reification process constructs these tables using positive maps and returns them so they can be used to fill the environment maps in the proof.

The main customizability of the plugin is achieved through the use of actions and patterns which are similar to the patterns in AutoQuote. Patterns provide a simple, semi-declarative way to perform syntactic matching on terms. Pattern branches are added to pattern tables using "`Reify Pattern`" which takes a pattern and an "action" describing the term it should produce. Patterns are defined using a Coq inductive data type which includes:

$$p \quad ::= \quad p_1 @@ p_2 \mid !!e \mid ?n \mid ?!n \mid \mathsf{RImpl}\, p_1\, p_2$$

These represent, respectively: function application, exactly the term *e*, a pattern variable that binds the current term to the n^{th} argument, a pattern variable that binds only constants, and Coq implication. To avoid the cost of reduction, pattern matching is purely syntactic.

Variables bound in patterns are processed by the action associated with the pattern. Actions are Coq functions where the types of the arguments tell what to do with the given values. For example, the `id` term (line 29) says that no further processing is necessary and the term should be passed for the argument. `function f` states that the term should be passed to the reification function `f` and the result should be passed to the action. After the arguments are processed, the final term is produced by substitution. No other normalization occurs, which is often important to avoid large terms and the overhead of reduction.

Programmable reification procedures such as this one provide good performance and are well-suited to perform reification for generic syntactic representations such as MirrorCore's. Customizing them to support specialized symbol algebras only requires adding patterns to capture the known symbols. Unknown terms will be safely hidden in the opaque symbol environment. Further, while they are necessary to use the reflective tactics, they are not, in theory, necessary to check the proofs constructed using them.

```
(* Declare pattern tables for custom types and functions *)
Reify Declare Patterns ptrn_simple_typ := typ.
Reify Declare Patterns ptrn_simple := (expr typ func).

(* Declare a quoting function for types *)
Reify Declare Syntax reify_simple_typ :=
  { (CPatterns ptrn_simple_typ ) }.

(* Declare a table that will reify to an environment *)
Reify Declare Typed Table table_terms : BinNums.positive ⇒
     reify_simple_typ.

(* Declare a quoting function for terms *)
Reify Declare Syntax reify_simple :=
  { (@CFirst (expr typ func)
       [(CPatterns ptrn_simple ) (* patterns first *)
       ;( CApp (@App typ func))      (* function application *)
       ;( CAbs reify_simple_typ (@Abs typ func)) (* functions *)
       ;( CVar (@Var typ func))      (* variables *)
       ;( CTypedTable table_terms otherFunc )]) (* other terms *) }.

(* Add patterns to the pattern tables *)
Reify Pattern ptrn_simple_typ += (!! ℕ) ⇒ tyNat. (* ℕ becomes tyNat *)
Reify Pattern ptrn_simple_typ += (!! Prop) ⇒ tyProp.

(* call [reify_simple_type ] to reify the bound arguments [0] and [1] *)
Reify Pattern ptrn_simple_typ += (@RImpl (?0) (?1)) ⇒
  (fun (a b : function reify_simple_typ ) ⇒ tyArr a b).
Reify Pattern ptrn_simple += (?! 0) ⇒ (* constants *)
  (fun (n : id ℕ) ⇒ @Inj typ func (N n)).
Reify Pattern ptrn_simple += ((!! (@eq)) @@ ?0) ⇒
  (* polymorphic equality: reify the type using [reify_simple_typ ] *)
  (fun (t : function reify_simple_typ ) ⇒ Inj (typ:=typ) (Eq t)).

(* reify a term *)
Goal expr typ func.
  reify ((fun x y ⇒ x) 1 3).
(* App (App (Abs tyNat (Abs tyNat (Var 1))) (Inj (N 1))) (Inj (N 3)) *)
```

Figure 4.8: Reification rules for a simple language with constant natural numbers and equality.

5

Case Study: Program Verification in Bedrock

In the introduction I motived mechanized reasoning for large-scale program verification. In this chapter I use my techniques to build compositional reflective automation that verifies programs written in Bedrock [54], a Coq library for low-level imperative program verification. Program verification is an interesting application of computational reflection because it requires reasoning about several layers of abstractions. First, it requires reasoning about the semantics of the programming language in order to understand what the code is doing. Second, it requires reasoning about program specifications including higher-order features such as function pointers and assertions about the current state of the program. Third, large programs introduce program-specific abstractions such as lists, finite maps, thread queues, and memory allocators.

Building automation to solve any one of these problems is difficult but doable. However, architecting these pieces so that they can be developed independently requires building them in a compositional way. Handling program-specific abstractions highlights the major difficulty. The predicates used to describe these abstractions are known only by the program developer, who wants to be able to use the automation predominantly as a client.

Bedrock includes a generic, assembly-like language that builds higher-level abstractions through principled macros [55] that come equipped with their own proof rules and automation. The latter is essential because Bedrock programs are proved correct from first principles. This choice unshackles the developer from the constraints of a conservative type system at the cost of making him or her prove the code safe. Clearly this approach is only feasible when such proofs can be done with minimal overhead. In the rest of the chapter I demonstrate how I used the ideas that I presented in Chapter 3 to achieve this[1].

The automation infrastructure that I describe in this chapter has been used to verify thousands of lines of Bedrock code ranging from data structures [111], to low-level OS-like services such as garbage collection [124] and a thread scheduler and web server stack [57]. It has also been used to verify a compiler that compiles a higher-level language into Bedrock [151].

I begin with an end-to-end example developing a small module in Bedrock (Section 5.1). In this context, I outline the verification task and focus on the aspects that are well-suited to automation. In the next section, I concentrate on the automation tasks that I solve using computational reflection: symbolic execution (Section 5.2.1) and entailment checking (Section 5.2.2). Each task is solved by its own reflective procedure that invokes both application- and framework-specific automation. I conclude with a discussion of the performance characteristics of the reflective procedures (Section 5.3) before surveying other program verification tools (Section 5.4).

[1]Historically speaking the Bedrock automation (called MirrorShard) was developed prior to MirrorCore.

5.1 Bedrock by Example

The code in Figure 5.1 gives a representative example of typical Bedrock code[2]. The figure is broken down into two parts: on the left is the executable code, and on the right is its verification. The code implements the standard binary search tree insertion algorithm without balancing.

The Bedrock Example

The code on the left begins by declaring a Bedrock module for binary search trees that contains a single `add` function for imperatively inserting new elements into the tree. The implementation is mostly standard; what is interesting is how Bedrock integrates programming and verification (on the right).

Take the function signature for example. There are no high-level types for the code, e.g. the two procedure arguments `s` and `k` are not declared to be a pointer and an integer. Rather, all of the assertions about the arguments are made by the procedure specification (lines 4-8) which is an assertion about the state of the system when the procedure runs. This allows us to express deeper properties, e.g. about the relationship between different values. In Bedrock, the function specification is written in separation logic [134]. It states that `s` is a pointer to the root of a tree that represents the mathematical set `sM` (expressed by the predicate "`bst sM s`"). The post-condition (inside of "`POST [] ...`") states that if the procedure returns, the same pointer `s` points to a mathematical set containing all of the elements that were in the initial set and also the new element `k`, expressed using "`sM ∪ {V "k"}`". In this style, Bedrock specifications are similar to other Coq verification systems [17, 58] where the imperative implementation is specified using a functional implementation written as a Gallina function.

The $*$ in Bedrock assertions is the separating conjunction from separation logic. The formula $P * Q$ states that the assertions P and Q are true about disjoint frag-

[2]This code was written by Adam Chlipala and published in [111].

Program

```
Definition bstM : bmodule := { ...
(* Method implementation *)
bfunction add(s, k)
   [SPEC("s", "k") reserving 7
    Al sM,
    PRE[V] bst sM (V "s") * mallocHeap
    POST[_] bst (sM ∪ {V "k"}) (V "s")
             * mallocHeap]
  tmp := *s ;;
  [∀ s, ∀ t,
    PRE[V] V "sM" ↦ V "tmp" * bst' sM t (V "tmp")
             * mallocHeap
    POST[_] ∃ t', ∃ p', V "sM" ↦ p' *
      bst' (sM ∪ {V "k"}) t' p' * mallocHeap]
  While (tmp ≠ 0) {
    tmp := *(tmp + 4) ;;
    If (k = tmp) { (* Key matches! *)
      Return 0
    } else {
      s := *s ;;
      If (k < tmp) {
        (* Searching for a lower key *)
        Skip
      } else {
        (* Searching for a higher key *)
        s := s + 8
      };;
      tmp := *s
    }
  };;
  (* Found a spot for a new node.
   * Allocate and initialize it. *)
  (* Argument is two less than size to
     allocate. *)
  tmp := Call "malloc"!"malloc"(1)
  [PRE[V, R] s ↦[1] ? * R ↦[3] ?
   POST[_] s ↦ R * (R ↦ $0, k, $0)];;

  *s   := tmp ;;
  *tmp := 0 ;;
  tmp  := tmp + 4;;
  *tmp := k;;
  tmp  := tmp + 4;;
  *tmp := 0;;
  Return 0
end ... }.
```

Verification

```
(* "Spine" type to define the representation
 * predicate recursively *)
Inductive tree :=
| Leaf : tree
| Node : tree → tree → tree.

(* Recursive rep. predicate for BSTs *)
Fixpoint bst' (s : set) (t : tree) (p : W)
: HProp := (* details omitted *).

(* Main representation predicate, which wraps
 * the above with a mutable pointer to its
 * root *)
Definition bst (s : set) (p : W) := [ freeable
      p 2 ]
  * Ex t, Ex r, Ex junk, p ↦ r *
    (p ⊹ $4) ↦ junk * bst' s t r.

(* A standard tree refinement hint *)
Theorem nil_fwd : ∀ s t (p : W), p = 0 →
  bst' s t p ⟹ [ s ≃ empty ∧ t = Leaf ].
Proof.  destruct t; sepLemma.  Qed.

(* ...omitting a few other hints... *)

(* Extend the generic hints with hints about
 * trees *)
Definition hints : HintPackage.
prepare (nil_fwd, bst_fwd, cons_fwd)
        (nil_bwd, bst_bwd, ...).
Defined.

(* Prove the implementation correct. *)
Theorem bstMOk : moduleOk bstM.
Proof.
  vcgen; abstract (sep hints; auto).
Qed.
```

Figure 5.1: A BEDROCK program for insertion into a binary search tree and its verification.

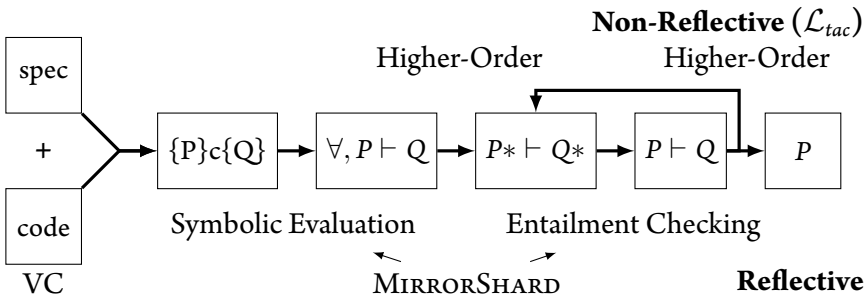

Figure 5.2: The process of verifying a Bedrock program.

ments of the heap. The separation greatly simplifies reasoning about aliasing and the independence of certain operators.

The implementation of add uses the loop (lines 15-30) to determine the correct position to insert the new element. This property is captured by the loop invariant (lines 10-14). The "pre-condition" of the loop expresses what is true at the beginning of the loop, while the post-condition expresses the updated post-condition of the function. The part of the tree that has already been visited has been moved into the specification of the continuation, a feature that is available in Bedrock because its verification is based on Hoare doubles and continuations. The code after the loop allocates a new memory block by calling `malloc!malloc` (line 34) and initializing the new memory contents accordingly.

5.1.1 Verifying Bedrock Programs

Figure 5.2 shows the five steps of verifying a Bedrock function. The verification combines a custom reflective procedure and two reflective procedures built on top of MirrorShard, the first-order predecessor to MirrorCore which contains custom handling for separation logic [111].

First, the verifier breaks the function down into a set of verification conditions (VCs) that collectively ensure that the function implements its specification. Each path through the program generates (roughly) two verification conditions. The

first states that the code will not produce an error if run from a state satisfying the pre-condition (the progress condition). The second VC states that any terminating execution starting from a state satisfying the pre-condition will finish in a state satisfying the post-condition (the preservation condition). In Bedrock, a path finishes when it performs an indirect or backwards jump.

After VC generation, the next step is to reason about the semantics of the code. The structure of separation logic coupled with the fact that all verification conditions are about straightline code makes both the progress and preservation conditions solvable by symbolically executing the code to compute a post-condition.

While progress is proven simply by symbolic execution, presevation additionally needs to prove that the computed post-condition implies the stated one. In these cases, at the end of symbolic execution the goal often has the following form.

$$\{P * [k \rightsquigarrow Q]\}\texttt{goto}\, k$$

Where the $\rightsquigarrow$ represents that k is a function pointer with pre-condition Q. While this goal is simple, resolving the appropriate continuation in a Bedrock specification could require complex higher-order reasoning. To avoid the need to implement and prove reflective procedures to automate domain-specific obligations such as these, the automation uses proof-generating $\mathcal{L}_{tac}$ to convert proof obligations such as the one above into entailments between separation logic formulae. For example, the automation would reduce the above obligation into $P \vdash Q$.

Next, MirrorShard's reflective entailment checker runs (usually) solving the separation logic part of the entailment and returning any unsolved pure propositions or higher-order formulae to $\mathcal{L}_{tac}$. Problem-specific $\mathcal{L}_{tac}$ automation solves the non-separation logic obligations, e.g. about the elements in the set. And the higher-order separation logic assertions are reduced to first-order entailments that are passed back to the entailment checker.

Including $\mathcal{L}_{tac}$ in the verification loop makes it easy to add domain-specific hints and automation. However, as I will show in Section 5.3, this design has a significant impact on performance.

5.2 Reflective Verification in Bedrock

The reflective automation for Bedrock is built on top of MirrorShard, the first-order predecessor of MirrorCore. I will return to its differences in Section 5.3; until then it is sufficient to think of it as MirrorCore.

Of the five verification steps, two are implemented using the compositional techniques that I discussed in Chapter 3. These are the procedures for symbolic execution and entailment checking, which are often the bottlenecks in program verification. To be concrete, I follow the verification of a minimal verification condition from the following Bedrock code snippet, which increments the first element of a linked list.

```
bfunction (ls)
  [SPEC("ls") reserving 0
   Al ls, PRE[V] sll ls (V "ls")
          POST[_] sll (inc_first ls) (V "ls")]{
  IF (p ≠ 0) { ++*p }
  Return 0
}
```

The preservation verification condition for the true branch of the conditional is the following:

$$\vdash \left\{\mathsf{llist}\, xs\, p\right\}\ \texttt{assume}\ (\texttt{p} \neq 0); \texttt{++*p}\ \left\{\exists L, \mathsf{llist}\, L\, p\right\}$$

Figure 5.3 shows the full process for solving this verification condition.

5.2.1 Symbolic Execution

Symbolic execution takes a pre-condition and a path through the code and computes a post-condition. The core reasoning about the code is relatively straightforward. In the example, the first "instruction" in the path is an assertion that the conditional `p = 0` returned `false`. Combining this with Bedrock's semantics, the

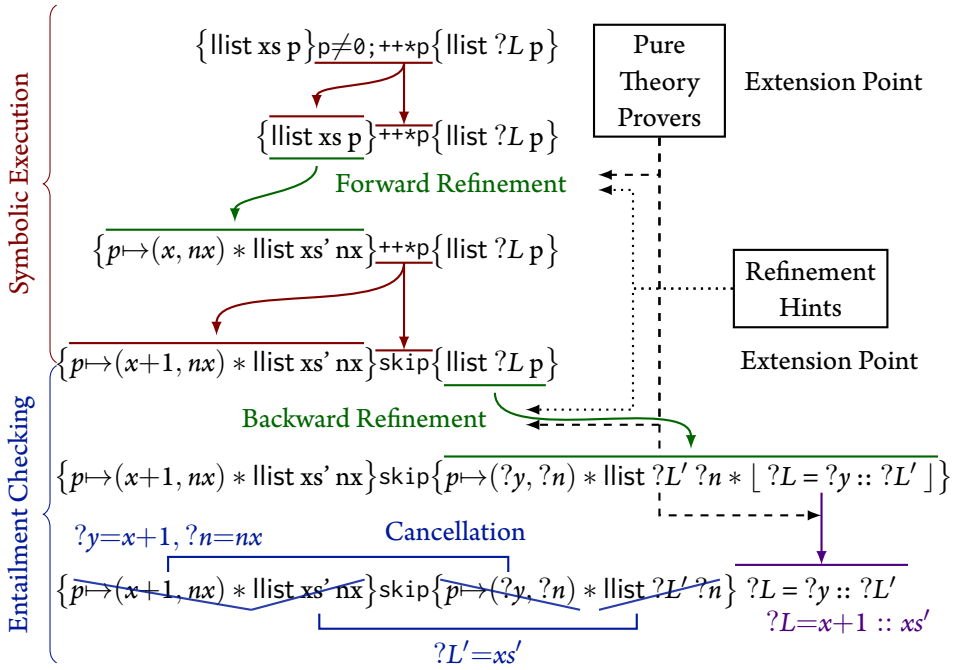

Figure 5.3: The high-level verification strategy applied to a simple program that increments the first cell of a non-empty list.

symbolic evaluator is able to determine that the symbolic value `p` does not equal zero, which it adds to its context of known facts. Learning this fact triggers the symbolic executor to attempt to refine the pre-condition. This refinement utilizes MirrorShard's *refinement hints*.

Rewriting with Refinement Hints

Refinement hints are stylized Coq theorems that express predicated heap entailments. These are represented reflectively using a specialization of MirrorCore's lemma syntax where the conclusion is a pair of separation logic assertions. The lemma that applies in the example states that any linked list whose head pointer is provably not null can be split into a first cell and a rest of the list.

$$\forall xs\, p, p{\neq}0 \rightarrow \mathsf{llist}\, xs\, p \vdash \exists x\, xs'\, np, p \mapsto (x', np) * \mathsf{llist}\, xs'\, np$$

To use this theorem, MirrorShard's reflective rewrite procedure attempts to find a (separating) conjunct on the left-hand side of the entailment that unifies with `llist` *xs p*. If a match can be found and the side-conditions can be discharged, the algorithm joins the right-hand-side of the rewrite with the remainder of the pre-condition. Newly introduced existential quantifiers are introduced into the context and are *appended* to the environment of universal quantifiers. Appending is slightly counter-intuitive when viewing variables as de Bruijn indices, but it allows the refiner to avoid lifting all other variables in the goal and known facts, which would be expensive. Instead the zero-cost weakening admitted by the non-dependent representation means there is no global manipulation to do.

Pure Theory Solvers

Similar to the backward prover in Chapter 3, pure theory solvers discharge `expr`-encoded propositional facts that arise from side conditions to rewriting lemmas, e.g. the obligation that `p` $\neq 0$ from the above rewriting theorem. MirrorShard provides four provers that are built to discharge the sort of obligations that arise in

verification conditions about pointer-based code.

- A **reflexivity prover**, which proves statements of the form $e = e$ for any e.
- An **assumption prover**, which maintains a list of known facts and attempts to use them to solve the goal directly. It is not difficult to adapt this procedure to perform unification rather than equality checking, but in practice it has not yet been necessary.
- A **linear arithmetic prover for equalities and inequalities on width-32 bitvectors**. This prover makes inferences by combining hypotheses representing expressions of the form $e_1 = e_2 + k$ for constants k. Understanding these facts is crucial when reasoning about array- and structure-oriented code.
- A prover oriented toward **array bounds checks**, which understands that array writes preserve length.

MirrorShard also supports composing provers in a simple disjunctive style which proves a proposition if either of the two provers can prove it. This composition leverages the environment-based composition that I discussed in Section 3.2.2. Other composition strategies are possible, for example a bounded recursive strategy that allows procedures to call one another, or one that dispatches to certain provers based on the structure of the goal, but the simple composition prover has been sufficient for Bedrock's current applications.

In the running example, the assumption prover discharges the side-condition of the linked list refinement lemma using the fact that symbolic execution learned from the code. Lifting the variables and pure facts into the context leaves the automation to verify the following:

$$p \neq 0 \wedge xs = x :: xs' \vdash \{p \mapsto (x, n) * \mathsf{llist}\, xs'\, n\}{+}{+}{*}\mathsf{p}\{\exists L, \mathsf{llist}\, L\, p\}$$

Memory Evaluators

The next step is to interpret the read and write of `*p`. Separation logic enables an effective algorithm for reasoning about memory operations based on *memory evaluators*. Memory evaluators are custom reflective procedures that reason about reads from and writes to heaps satisfying a separation logic assertion. This approach enables the symbolic evaluator to interpret memory operations in terms of many different data structure predicates without the need to expose individual points-to assertions. In addition to the simple composition memory evaluator, which combines two memory evaluators in a disjunctive fashion, MirrorShard includes memory evaluators for 32-bit points-to, arrays (of both words and bytes), and local variable stack frames. The latter two are very important for controlling the size of separation logic formulae. Without them we would need to break arrays and stack frames down to individual points-to assertions, which could result in very large formulae.

In the example, the memory evaluator uses the provers to inspect the separation logic formula and determines that the value read from `p` is `x` (because a sub-formula `p` $\mapsto$ `(x,nx)` appears in the pre-condition). The writer part of the memory evaluator is then able update the memory cell contents directly with the expression `x+1` that was constructed syntactically when interpreting the addition. This reasoning further evolves the pre-condition into the following formula, which happens to be the strongest.

$$\lfloor p \neq 0 \rfloor * p \mapsto (x + 1, nx) * \mathsf{llist}\ nx\ xs'$$

A Note About Completeness In principle, computational reflection allows the developer to prove that the computed post-condition is the strongest. While proving completeness would give a strong statement that the automation does the most possible, the first-order nature of MirrorShard's representation means not only that there must be a proof, but that there must be a first-order proof. A proof of completeness would require the completeness of the symbolic executor and all of the extension points including memory evaluators, the pure provers, and the

unification algorithm. Since BEDROCK is built to handle complex, user-defined assertions completeness in general seems unattainable.

5.2.2 ENTAILMENT CHECKING VIA CANCELLATION

After $\mathcal{L}_{tac}$ automation runs to handle the higher-order reasoning, the next task is to check whether the computed post-condition entails the stated post-condition. Continuing with the example, the higher-order $\mathcal{L}_{tac}$ will produce the following entailment.

$$\lfloor p \neq 0 \rfloor * p \mapsto (x{+}1, nx) * \mathsf{llist}\, xs\, nx \vdash \exists L, \mathsf{llist}\, L\, p$$

Note that this $\vdash$ is different than the one above. In particular, it states an entailment in separation logic, while the above stated a propositional entailment.

Since there is nothing left to do in the premise, MIRRORSHARD next massages the right-hand side by replacing existential quantifiers with new unification variables. This process results in the following formula.

$$\lfloor p \neq 0 \rfloor * p \mapsto (x{+}1, nx) * \mathsf{llist}\, xs\, nx \vdash \mathsf{llist}\, ?L\, p$$

To solve this entailment, MIRRORSHARD must reason about the `llist` predicate, this time when it occurs in the conclusion. To do this, MIRRORSHARD uses another lemma to expand the definition of `llist` in the post-condition and expose the points-to and the existential quantifiers[3]. After converting the new existentials into unification variables, the resulting entailment is the following.

$$\lfloor p \neq 0 \rfloor * p \mapsto (x{+}1, nx) * \mathsf{llist}\, nx\, xs \vdash p \mapsto (?y, ?n) * \mathsf{llist}\, ?L'\, ?n * \lfloor ?L = ?y :: ?L' \rfloor$$

From here, MIRRORSHARD can perform the actual entailment checking using the cancellativity property of the separating conjunction.

[3]It is no coincidence that the combination of these two lemmas states a bi-entailment which is very close to the definition of the predicate itself.

$$\frac{Q \vdash R}{P * Q \vdash P * R}\ \textsc{Cancel}$$

In a non-reflective procedure, the difficulty of entailment checking using cancellation comes from permuting the conjuncts until the Cancel theorem can be applied. For example, if the conjuncts were reversed on the right-hand side, we would have to apply the commutativity of the separating conjunction to get the common $p \mapsto (x{+}1, nx)$ at the front of the conjunction. This reasoning about permutation is exactly the place where computational reflection shines. Permutation proofs are large because Coq requires that the entire problem is re-stated for each step of the permutation. For a moderately sized problem of about 8 conjuncts, this can result in hundreds of conjuncts that must be checked. I will return to this with empirical results in Section 5.3.2.

Cancellation Heuristics While the problem looks like a vanilla permutation problem, the existence of unification variables complicates things. To see the problem, consider a simple entailment with unification variables:

$$p \mapsto q * q \mapsto v \vdash ?1 \mapsto ?2 * p \mapsto ?1$$

The cancellation lemma applies directly to this goal, but using it immediately would leave us in the following unprovable state:

$$q \mapsto v \vdash p \mapsto p$$

To avoid this predicament, the entailment checker sorts the right-hand-side conjuncts lexicographically, placing unification variables at the end before beginning cancellation. While woefully incomplete, this simple heuristic is amazingly useful for the vast majority of Bedrock programs.

5.2.3 Computational Reflection and Custom Representations

In addition to small proof terms, another benefit of computational reflection is the ability of select custom representations and algorithms that are better suited to

solving a particular problem. Custom representations showed up several times in the previous section, e.g. the special representation of lemmas and the facts known by pure provers. Continuing in this vein, MirrorShard also uses a custom representation of separation logic assertions represented mathematically as follows.

$$(\text{formulae}) \quad s \quad ::= \quad P\,\vec{e} \mid s_1 * s_2 \mid \mathsf{emp} \mid \exists x : \tau, s^x \mid \lfloor e \rfloor$$

In particular, note that the $P\,\vec{e}$ is the only place where new separation logic formulae can be added, and since the arguments to P are expressions, this syntax cannot be used to express additional separation logic connectives such as $\wedge$ or $-\!\!*$.

In addition to this custom syntax, MirrorShard builds an even more specialized representation based on discrimination trees [40] to simplify the implementation of MirrorShard's custom procedures. The specialized representation factors out the associativity and commutativity of separation logic formulas as well as the pure facts. Concretely, the implementation is a pair containing a list of pure facts and a finite multi-map associating each user-defined predicate to the list of arguments the predicate is applied to. For example, the following separation logic assertion would be represented as follows.

$$p \mapsto q * q \mapsto r * \lfloor x = y \rfloor \quad \Leftrightarrow \quad ([x = y], \{\text{`}\mapsto\text{'} \mapsto [[p; q]; [q; r]]\})$$

This representation is tailored to make two operations efficient:

1. Extracting "pure" facts, i.e. those facts that do not depend on the heap. These facts express properties like the equality or inequality of two values or the existence of an element within a mathematical set.

2. Extracting all of the separation logic assertions with a given head symbol. This makes it easy to find all potentially matching conjuncts when searching for a particular symbol, for example when rewriting by a lemma or when performing cancellation.

File	Program	Invar.	Tactics	Other	Overhead
LinkedList	42	26	27	31	2.0
Malloc	43	16	112	94	5.2
ListSet	50	31	23	46	2.0
TreeSet	108	40	25	45	1.0
Queue	53	22	80	93	3.7
Memoize	26	13	56	50	4.6

Figure 5.4: Case study verifications, with data on annotation burden, in lines of code

5.3 Evaluation

In this section I evaluate the usefulness and performance of Bedrock's reflective automation, comparing it to the traditional $\mathcal{L}_{tac}$-style verify-and-check approach to mechanized verification. I begin with a brief analysis of the level of automation that the reflective procedures achieve by discussing several medium-sized modules for data structures in detail and then some of the larger, more organically constructed programs. Afterwards, I consider the performance characteristics that MirrorShard's reflective automation has, in particular comparing them to $\mathcal{L}_{tac}$-based verification methodologies.

> **Nota Bene** MirrorShard's internal details are slightly different than MirrorCore's. MirrorShard is first-order and uses n-ary function application. The latter makes a syntactic representation of function types unnecessary. Also, extensibility in MirrorShard is based solely on the environment extension method that I presented in Section 3.2.

5.3.1 Usability, Expressivity & Automation

Figure 5.4 shows some code statistics from six data-structure case studies that have been verified with MirrorShard. In order, the columns count the executable

part of the module being verified, the function specifications and invariants asserted in code, the $\mathcal{L}_{tac}$-tactic proof scripts (including commands to register hints), all the remaining lines, and finally the ratio of verification lines to program lines. The lines categorized under "Other" are almost all definitions of data structure representation predicates and statements of theorems about them.

The case studies are: LinkedList, consisting of the classic functions is-empty, length, reverse, and concatenate (the latter two performed in-place with mutation); Malloc, a naïve memory allocator, based on an unsorted free list with no coalescing, used by all the later case studies; ListSet and TreeSet, implementations of a common finite set interface specified with mathematical sets, respectively using unsorted lists and binary search trees; Queue, a standard FIFO queue specified mathematically using bags; and Memoize, a higher-order function that memoizes Bedrock code that implements a mathematical, effect-free, deterministic function.

The proof overhead is slightly lower than with the same case studies used in the fully $\mathcal{L}_{tac}$-based Bedrock [54]. This decrease arises mostly from MirrorShard's hint databases. Modulo loop invariants and function specifications the programmer only needs to add new reflective hints when he or she develops new representation predicates. Once he or she does this once, the automation is able to apply them whenever necessary. In the original $\mathcal{L}_{tac}$-version, annotations were needed everywhere that lemmas were needed to refine abstract predicates.

Ultimately, MirrorShard's ability to verify the same examples demonstrates that it achieves a similar level of automation. Further, its ability to verify higher-order code demonstrates its ability to integrate with Coq's built-in support for higher-order reasoning.

MirrorShard's procedures have also been used in a larger case study that built a verified cooperative threading library and then verified a Web server running on top of the library [57]. The thread library includes about 400 lines of implementation code and 3000 additional lines for its verification, while the Web server has 200 lines of implementation and 500 more for the proof, which establishes that representation invariants are maintained for key data structures. Most of this ver-

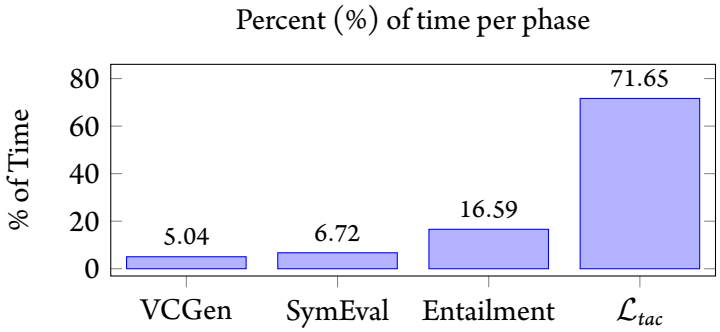

Figure 5.5: The amount of time spent in different pieces of automation. $\mathcal{L}_{tac}$ still accounts for a significant amount of the total time.

ification overhead in the thread library is due to higher-order logic that MirrorShard does not provide automation for.

5.3.2 Performance

Beyond expressive power, a crucial benchmark for verification tools is performance. End-to-end performance is approximately the same as the previous, $\mathcal{L}_{tac}$-based Bedrock. On the surface, this result contradicts much of the work claiming significant performance improvements from computational reflection. However, there are good reasons for this to be the case.

Figure 5.5 shows the amount of time spent in the various components of the automation. Note that only about 23% of the total verification time is spent inside MirrorShard's two reflective procedures. More than 70% of the time is still spent in $\mathcal{L}_{tac}$, mostly in higher-order reasoning. Two experiments better elaborate the reasons for this. First, I discuss a micro-benchmark that demonstrates the scaling potential of MirrorShard's reflective automation. Second, and more importantly, I discuss integration performance, i.e. the cost incurred by repeatedly solving small problems with computational reflection rather than solving a single, large problem. These results set up the challenges for the rest of the dissertation.

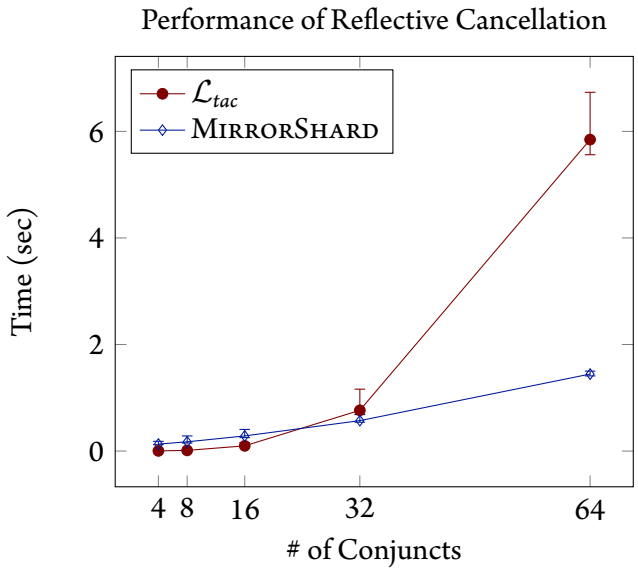

Figure 5.6: A microbenchmark comparing scaling characteristics of reflective and non-reflective proofs for cancellation in separation logic.

Microbenchmarks

Microbenchmarks confirm the potential for large performance improvements over $\mathcal{L}_{tac}$-based solutions, especially in scaling and proof checking. Figure 5.6 shows the time required for cancellation in both $\mathcal{L}_{tac}$ and MirrorShard's reflection implementation.

While the reflective proofs are faster and have the best scaling properties, performance numbers are comparable for small problems. This is due to the cost required to reify the goal and apply the more sophisticated reflection lemma. Since many of Bedrock's goals are well-abstracted and therefore have few conjuncts, Bedrock does not benefit from the large performance wins that emerge with more conjuncts.

This benefit-only-in-the-limit behavior is not the case for all of the verification tasks. Computational reflection yields a substantial performance benefit for the rewriting phase that happens at the beginning of the cancellation algorithm. To

demonstrate the behavior, we use rewriting to unfold the linked lists predicate and hoist existential quantifiers and pure facts in the precondition into the goal. For example, for a list of length 2, the input problem is the following:

$$\mathsf{llist}(x_0 :: x_1 :: [], p_0) \vdash emp$$

and the resulting goal is the following

$$\begin{array}{lll} p_0 \neq 0 & p_1 \neq 0 & p_2 = 0 \\ p_0 \mapsto (x_0, p_1) * p_1 \mapsto (x_1, p_2) \vdash emp & & \end{array}$$

where the first line are premises above the line in Coq context and the second line is the goal. By setting the right hand side to `emp` we avoid extraneous cancellation.

The key to solving these goals is the following two refinement lemmas, which are applied in the forward direction.

$$\begin{array}{rcl} \mathsf{llist}([], p) & \vdash & \lceil p = 0 \rceil \\ \mathsf{llist}(x :: \ell, p) & \vdash & \lceil p \neq 0 \rceil * \exists p'.\ p \mapsto x, p' * \mathsf{llist}(\ell, p') \end{array}$$

Beyond rewriting using these lemmas, the automation must also lift the extistentials and the pure facts ino the context. In $\mathcal{L}_{tac}$ this actually improves performance because it shrinks the size of the goal, which makes future reasoning faster.

Figure 5.7 shows the performance characteristics of both $\mathcal{L}_{tac}$ and MirrorShard. Proving this family of theorems using $\mathcal{L}_{tac}$ automation is painfully slow both to find a proof ($\mathcal{L}_{tac}$) and to check it ($\mathcal{L}_{tac}$ Qed). The reflective implementation, on the other hand, sees near constant performance independent of the length of the list. The key to MirrorShard's significant performance win in this case is the custom representation of separation logic formulas that lifts existential quantifiers and pure facts to the top immediately and locally. The $\mathcal{L}_{tac}$ implementation, on the other hand, relies on `autorewrite`, which is a global operation that is unable to lift existentials into the context and unable to rewrite under binders meaning that it must be run interleaved with $\mathcal{L}_{tac}$automation to do both of these.

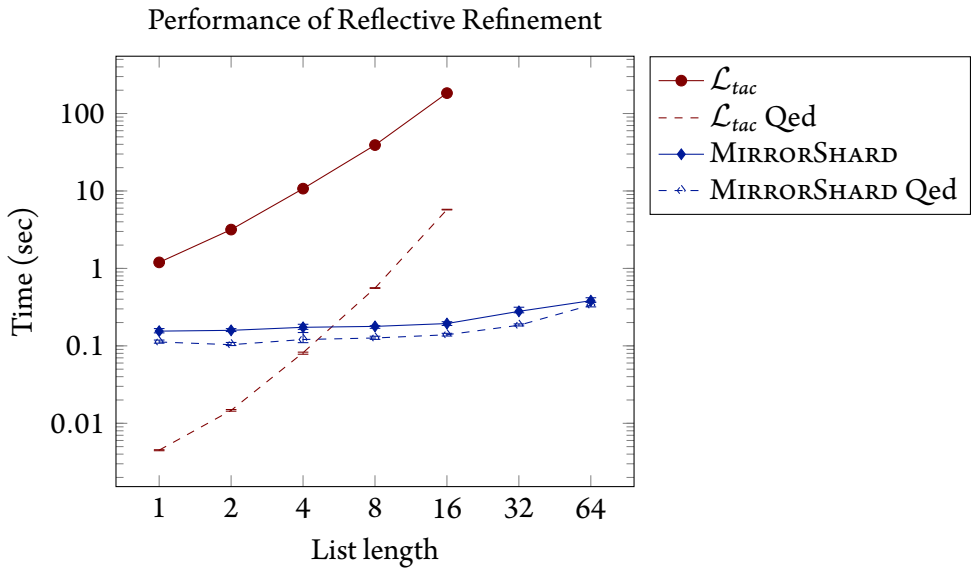

Figure 5.7: Reflective refinement is significantly more efficient than non-reflecive refinement.

Integration Performance

The microbenchmarks show that the win from computational reflection comes from solving large problems. While MirrorShard is used to solve some large problems, it turns out that the verification process does not actually use reflection to solve these large problems because the higher-order reasoning is not performed reflectively. Thus, the large verification task is split up into many small tasks that are solved independently.

To understand the repercussions of alternating computational reflection and $\mathcal{L}_{tac}$-based automation, consider the following path through the length function for linked lists:

```
assume( *(Sp+4) ≠ 0 );     (* not at the end of the list *)
*(Sp+8) := *(Sp+8) + 1 ;   (* increment the length counter *)
Rv := *(Sp+4) ;            (* get the next pointer *)
*(Sp+4) := *Rv             (* update "current" *)
```

The references from the stack pointer Sp are to local variables. Sp+8 is the location

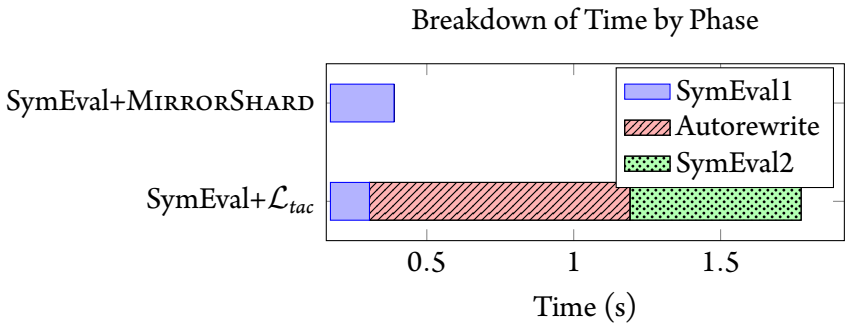

Figure 5.8: Profiling of symbolic execution with reflective rewriting (SymEval+MirrorShard) versus the same symbolic execution engine using `autorewrite` (SymEval+$\mathcal{L}_{tac}$).

of the length counter, and `Sp+4` is the location of the "current" pointer. The first line is the result of knowing that the conditional comparing `current` to `null` returned false, implying that evaluation is not at the end of the list. This fact justifies the memory dereference on the last line where the code reads the `next` pointer of the current linked-list cell (`**(Sp+4)`).

In order to exploit this information during symbolic execution, Bedrock's symbolic evaluator uses the following refinement hint (which I showed during the example in Section 5.2.1), provided in a hint database, to expose the $\mapsto$ predicate which the memory evaluator knows how to interpret:

```
Lemma llist_cons_fwd : ∀ ls (p : W), p ≠ 0
  → llist ls p ⊢
    ∃ x ls', [| ls = x :: ls' |] * ∃ p', p ↦ (x, p') * llist ls' p'.
```

Without this mechanism, the automation could achieve the same result by running an $\mathcal{L}_{tac}$ loop bouncing between our reflective symbolic execution and the `autorewrite` tactic to perform this rewriting. In the above example, this loop would call symbolic execution, which would get stuck on the final instruction, falling back on `autorewrite` to expose the cons cell, enabling a second call to symbolic evaluation to complete the task.

Figure 5.8 shows how the loop approach (SymEval+$\mathcal{L}_{tac}$) compares to Mir-

rorShard's fully reflective procedure (SymEval+MirrorShard). Using the latter, the entire symbolic execution takes 0.39 seconds, less than half the amount of time (0.89 seconds) taken by `autorewrite` to perform just the rewriting. Overall the reflective composition results in a 4.6x speedup over the hybrid reflective and $\mathcal{L}_{tac}$-based composition on this goal. On the entire linked list module this translates to a savings of 44s in verification time.

While MirrorShard's reflective rewriter is not as powerful as `autorewrite`, it is customizable by adding new lemmas in the same way. Further, because it is written in Gallina rather than hardcoded inside Coq, we can extend it with domain-specific knowledge. For example, unification can be made to reason up to provable equality rather than just definitional equality. For a domain such as separation logic, this would enable unification modulo associativity and commutativity in the style of Braibant's AAC tactics [46].

While MirrorShard achieves this level of end-to-end reflective reasoning for symbolic execution, its algorithm for solving higher-order entailments is more piecemeal. During entailment checking, MirrorShard alternates between computational reflection, to solve first-order entailments, and $\mathcal{L}_{tac}$-tactics to solve higher-order constraints that can produce additional entailments. While most entailments are relatively small, the phrasing of Bedrock's logic and verification conditions means that it is not uncommon for a single verification condition to require between three and eight calls to the entailment checker even though all calls completely solve their goals. Coupled with the results above, and considering that the higher-order reasoning is more expensive than $\mathcal{L}_{tac}$ rewrites, it is not difficult to see the performance bottleneck. MirrorCore addresses this problem by reifying higher-order syntax and general binders. I show how this works in the case study in Chapter 7.

5.4 Related Work

Chlipala's original version of Bedrock [54] supported similar automation to the work described in this chapter. The work in this chapter moves the automation

from the *proof-generating* style using $\mathcal{L}_{tac}$ to the reflective style which is the theme of this dissertation. Bedrock has grown considerably since this automation was developed, but the extensibility of the automation had enabled it to adapt to much larger developments than the ones I presented in this chapter. Oe built a verified mark-and-sweep garbage collector [124] in Bedrock. The implementation required reasoning about aliasing data structures in Bedrock's separation logic, a task which has been widely studied [38, 41, 42, 75, 95, 101, 110, 127] but mostly avoided in large-scale verification. Bedrock also contains many core pieces of an operating system and has been connected to the POSIX API via axiomatized invocations of C functions that can be linked with Bedrock code. This work has been used to build a web-server, and the performance of the code is surprisingly good considering the lack of compiler optimizations.

Finally, Wang et al. [151] built a compiler for a higher-level language based on abstract data types to Bedrock. The compiler is based on the idea of compiling high-level languages to low-level languages, targeting a program logic for the low-level language rather than directly targeting the language semantics. This enables them to reuse much of the automation that we developed in this setting. The integration with Coq's automation is essential in this setting because the verification conditions that arise necessarily compose opaque code fragments produced by recursing over an abstract syntax tree. While this feature is not supported by MirrorShard's representation, MirrorCore's more extensible representation can directly support this type of reasoning.

Verified Verification Procedures Several projects [18, 35, 118] have studied translation of SMT-solver proof traces into forms that can be verified by a proof assistant. Some of these projects are based on reflectively checking a proof trace that is generated by an external tool rather than verifying the prover itself. Separate proof generation and reflective proof checking has non-obvious trade-offs with pure computational reflection. Verifying the prover removes the need for potentially expensive proof generation and checking, but the proof-generating approach is compatible with using efficient low-level languages and optimizing compilers to

implement the provers.

Lescuyer and Conchon [106] built a reflective SAT solver tactic for Coq, and Nanevski et al. [122] and Oe et al. [125] have verified efficient low-level code for a part of an SMT solver and a full SAT solver, respectively. None of this past work supports sound, modular extension with new procedures nor do they support a rich formula language that includes quantifiers and is extensible by user-defined predicates with associated axioms.

Verified Separation Logic Automation A few past projects have proved the correctness of non-extensible separation logic proof procedures. Marti and Affeldt [112] verified a simplification of Smallfoot [31] using Coq. Stewart et al. did a Coq verification of a Smallfoot-style program verification tool, VeriSmall [15], that relies on a verified heap theorem prover, VeriStar [142]. VeriStar implements an entailment checking algorithm based on paramodulation. Adapting this algorithm to to MirrorShard's richer logic would avoid the need for some of the heuristics, which would ultimately make for a more robust procedure.

None of this work has tackled the, often more difficult, entailments that arise when proving functional correctness, as opposed to memory safety. To solve these more difficult verification conditions, MirrorShard's reflective algorithms return obligations to the user that it (and the extensions) cannot solve. This approach enables the combined system to discharge interesting verification conditions that are out of reach of fully integrated provers. This synergy with the proof assistant is lost when the automation is run as stand-alone applications.

Other researchers have built non-reflective verification infrastructure on top of proof assistants. Several proof assistant libraries provide support for separation logic proofs, including the tactic libraries of Appel [13] and McCreight [117], Holfoot [147], Ynot [58], and Charge! [29]. Some of these libraries provide proof automation comparable to that of the standalone tools, though support for extensible automation in all of them is limited.

All of these procedures, including MirrorShard's, handle restricted forms of separation logic formulae. For example, MirrorShard's representation does not

support extension by additional separation logic operators such as separating implication [134] or classical conjunction. Recent procedures [89, 102] have sought to prove entailments in these fragments based on the idea of enriching the logical language with an explicit way to name heaps. Integrating these ideas into the separation logic would require substantial effort to retro fit definitions in MirrorShard. MirrorCore's representation, on the other hand, would support the extension in a fairly natural way, leaving only the difficulty of implementing the procedures themselves.

Non-verified Verification Tools Many standalone tools do efficient, automated analysis of large low-level code bases for memory safety, using separation logic, outside of the context of proof assistants. Examples include Smallfoot [31], SpaceInvader [49], and SLAyer [32]. Xisa [52] and VeriFast [92] bear a special relationship to MirrorShard, as they are extensible with new predicate definitions in separation logic. Further, while both support new abstract predicates, their verification engines are fixed, meaning that they cannot be adapted to reason outside of the logics that they were built to address, without serious modifications or rewrites. MirrorShard, on the other hand, can easily be extended with new domain-specific pure provers, and algorithms can be tweaked for particular instances while retaining foundational proofs.

There are also several tools that do non-separation logic verification mostly based on SMT solvers. Boogie [22] is a program verification framework built on top of the Z3 SMT solver [68]. It supports a variety of tools and forms the foundation for Dafny [104]. While Dafny is proof assistant-like, it is substantially more automated than most existing proof assistants. However, it lacks an underlying formal semantics.

Significant progress has also been made on using Z3 in the verification of concurrent code. VCC [61] is a powerful tool that has been used to verify components of Microsoft's hypervisor, Hyper-V [103]. While much of this project has been verified, VCC alone is not powerful enough to perform the full verification. Gu et al. [85] showed that Coq's logic is rich enough to verify a smaller, custom

hypervisor.

In practice, standalone tools often achieve significantly better performance than their proof assistant counterparts. In theory, computational reflection should be able to match the performance of these tools; however, the compilation and optimization infrastructure currently available for modern proof assistants does not match that of industrial-strength compilers for low-level programming languages. The advantage of building tools within a proof assistant, however, is that these tools can fall back on the features of the proof assistant when their own features are insufficient. The Why3 tool [36] takes a middle ground by offering the ability to translate verification conditions into both Coq and SMT solvers such as Z3 [68], Yices [76], and CVC3 [24]. Proofs done in Coq are guaranteed by its kernel, but the verification condition generation and verification conditions that are solved by un-verified solvers do not have the same foundational guarantees. However, using SMT solvers often results in substantially faster proofs than when verification conditions do not require deep higher-order reasoning.

6

$\mathcal{R}_{tac}$: A Reflective Tactic Language

Up to this point, I have demonstrated how to develop compositional reflective procedures. However, developing these procedures can still be difficult and time consuming, especially for very application-specific problems. In this chapter I aim to ease the development of reflective procedures by developing a small library of tactics and tactic combinators that can be composed into higher-level automation. MirrorCore's tactic language, $\mathcal{R}_{tac}$, aims to achieve four goals.

1. *Easy interoperability with reflective procedures.* Both calling reflective procedures from $\mathcal{R}_{tac}$ and calling $\mathcal{R}_{tac}$ from custom reflective procedures is important. A major limitation of reflective procedures is that they are unable to call generate-and-check style automation techniques since the proof checking process cannot be internalized in the logic. $\mathcal{R}_{tac}$ is not meant as a drop-in replacement for $\mathcal{L}_{tac}$, though in some cases it could be used as one. Instead, $\mathcal{R}_{tac}$ should be viewed as high-level glue for stitching together custom deci-

sion procedures such as the cancellation algorithm from Chapter 5 or writing automation to fill in the leaves of an otherwise reflective proof.

2. *No overhead building proofs.* $\mathcal{R}_{tac}$ maintains a strong separation between computation terms and proofs. In domains such as program verification where large proofs are constructed and then immediately abstracted by a `Qed`, explicitly building the terms can be avoided entirely.

3. *Be fully customizable within the logic.* While $\mathcal{L}_{tac}$ can be customized and extended via Coq plugins, these plugins are not always easy to build and are a system-level dependency. These tie your development to Coq rather than Gallina and, while there are not many competing implementations of the logic, plugin compatibility is not guaranteed between versions. Further, this extension point is brittle to low-level implementation choices inside of Coq, and might not work in the same way between two versions of Coq or between different implementations of the type theory.

4. *Easy to develop and verify simple reflective procedures.* Using tactics should be natural and require as little proof overhead as possible. Achieving this goal means that soundness theorems of individual tactics should compose naturally, and the details of the specific proof should be well encapsulated.

$\mathcal{R}_{tac}$'s design is based heavily on $\mathcal{L}_{tac}$'s [69] which I described in Chapter 2. Simple $\mathcal{L}_{tac}$ tactics translate directly into $\mathcal{R}_{tac}$ with roughly the same semantics. Figure 6.1 shows a simple $\mathcal{L}_{tac}$ tactic for proving numbers even and a corresponding $\mathcal{R}_{tac}$ tactic which achieves the same goal using only standard $\mathcal{R}_{tac}$ tacticals. Notice that the tactic itself is a direct translation of the $\mathcal{L}_{tac}$ code, and the proof is completely automated by the custom $\mathcal{L}_{tac}$ tactic `rtac derive soundness`, which blindly applies the soundness theorems for the underlying tactics. The call to `auto` proves the two applied lemmas sound using their semantic counterparts.

In the rest of the chapter I discuss the definition and compositional specification of tactics (Section 6.1). Using the definition, I describe several simple tactics and

$\mathcal{L}_{tac}$

```
Ltac tac_even :=
  repeat first
    [ assumption
    | apply even_0
    | apply even_odd
    | apply odd_even
    ].
```

$\mathcal{R}_{tac}$

```
Def tac_even : rtac :=
  REPEAT 10 (FIRST
    [ ASSUMPTION
    ; APPLY even_0_syn
    ; APPLY even_odd_syn
    ; APPLY odd_even_syn
    ]).
```

$\mathcal{R}_{tac}$ Proof

```
Theorem even_sound
: rtac_sound tac_even.
Proof.
  rtac derive soundness;
  eauto using even_0,
    even_odd, odd_even.
Qed.
```

Figure 6.1: Proving evenness in $\mathcal{L}_{tac}$ and $\mathcal{R}_{tac}$.

tactic combinators (Section 6.2). In Section 6.3 I evaluate the performance of $\mathcal{R}_{tac}$ reflective procedures compared to both $\mathcal{L}_{tac}$ and custom reflective procedures. I conclude with a discussion of related work (Section 6.4).

6.1 Compositional Tactics

Before diving into the implementation, I begin with an overview of the two high-level challenges that $\mathcal{R}_{tac}$ must overcome: local reasoning and incremental global reasoning. To see these challenges consider proving the following proposition.

$$\exists x : \mathbb{N}, (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)$$

While the reflective tactics will not actually construct this proof, the soundness proof of the tactics must demonstrate how to build the proof. With this in mind, it makes sense to explain the tactics as if they were constructing the proof explicitly.

Local Reasoning Proving the above proposition first requires some local reasoning about the existential quantifier. A first logical step is to introduce a new unification variable and use it to instantiate the existential. This reasoning step produces the following proof skeleton.

$$\dfrac{\dfrac{(1)}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x) \wedge (\exists z : \mathbb{N}, z = ?x)}}{\vdash \exists x : (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)}\ \exists\text{-I}$$

In the new goal (notated "(1)"), $?x$ is bound in the context (separated from the goal by $\vdash$) as a unification variable notated by the $?$.

Next, a tactic applies $\wedge$-introduction, splitting the goal into two parallel subgoals.

$$\dfrac{\dfrac{\dfrac{(1)}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x)} \quad \dfrac{(2)}{?x : \mathbb{N} \vdash (\exists z : \mathbb{N}, z = ?x)}}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x) \wedge (\exists z : \mathbb{N}, z = ?x)}\,\wedge\text{-I}}{\vdash \exists x : (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)}\,\exists\text{-I}$$

Now things begin to become interesting. Note that filling in the (1) and (2) can be done mostly independently; however, both proofs must agree on the instantiation of ?x.

Focusing on (1) for a moment, introducing the universal quantifier updates the proof tree but leaves (2) unchanged.

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)}{?x : \mathbb{N}, y : \mathbb{N} \vdash y \geq ?x}}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x)}\,\forall\text{-I} \quad \dfrac{(2)}{?x : \mathbb{N} \vdash (\exists z : \mathbb{N}, z = ?x)}}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x) \wedge (\exists z : \mathbb{N}, z = ?x)}\,\wedge\text{-I}}{\vdash \exists x : (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)}\,\exists\text{-I}$$

Introducing the existential quantifier in (2) further evolves the proof tree resulting in the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)}{?x : \mathbb{N}, y : \mathbb{N} \vdash y \geq ?x}}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x)}\,\forall\text{-I} \quad \dfrac{\dfrac{(2)}{?x : \mathbb{N}, ?z : \mathbb{N} \vdash ?z = ?x}}{?x : \mathbb{N} \vdash (\exists z : \mathbb{N}, z = ?x)}\,\exists\text{-I}}{?x : \mathbb{N} \vdash (\forall y : \mathbb{N}, y \geq ?x) \wedge (\exists z : \mathbb{N}, z = ?x)}\,\wedge\text{-I}}{\vdash \exists x : (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)}\,\exists\text{-I}$$

Global Reasoning Returning to (1), we can instantiate $?x$ with the value 0 and solve the goal. This leads to the following proof term.

$$\dfrac{\dfrac{\dfrac{}{y : \mathbb{N} \vdash y \geq 0}\,\geq\text{-0}}{\vdash (\forall y : \mathbb{N}, y \geq 0)}\,\forall\text{-I} \quad \dfrac{\dfrac{(1)}{?z : \mathbb{N} \vdash ?z = 0}}{\vdash (\exists z : \mathbb{N}, z = 0)}\,\exists\text{-I}}{\vdash (\forall y : \mathbb{N}, y \geq 0) \wedge (\exists z : \mathbb{N}, z = 0)}\,\wedge\text{-I}}{\vdash \exists x : (\forall y : \mathbb{N}, y \geq x) \wedge (\exists z : \mathbb{N}, z = x)}\,\exists\text{-I=0}$$

Notice that all occurrences of $?x$ have been replaced by 0. While the new proof term is obviously valid, this transformation must be justified with a proof.

The key to ensuring that semantic proofs are robust to global effects is to ensure that tactics guarantee sufficiently general proofs. This insight corresponds to the strengthened inductive hypothesis that is necessary to prove the soundness of $\mathcal{R}_{tac}$ compositionally. $\mathcal{R}_{tac}$ achieves this generality semantically rather than syntactically.

6.1.1 Tactics, Goals & Contexts

The core of the computational piece of $\mathcal{R}_{tac}$ is focused on maintaining the goal state, and the core of $\mathcal{R}_{tac}$'s soundness surrounds stitching together proof fragments produced by individual tactics. This separation of concerns means that the core of $\mathcal{R}_{tac}$ is parametric in the underlying representation of propositions. To use $\mathcal{R}_{tac}$ with MirrorCore we can instantiate the expression type using MirrorCore expressions. This parameterization not only simplifies the proofs by hiding irrelevant details behind universal quantifiers, but also makes $\mathcal{R}_{tac}$ retargetable to other representations of goals, e.g. MirrorShard's separation logic entailments or its first-order expression language. Figure 6.2 presents the core pieces of $\mathcal{R}_{tac}$ mathematically.

Tactics An $\mathcal{R}_{tac}$ tactic is a Coq function that accepts a goal and the context that it resides in (i.e. $\mathcal{C}$) and produces a new goal and a possibly updated context that entails the given one. Drawing a comparison with standard $\mathcal{L}_{tac}$-style verification, the context represents the part of the goal above the line and the goal represents the part below the line. I will discuss more about these shortly.

$$
\begin{array}{rrcl}
\text{Goals} & \mathcal{G} & ::= & \forall \tau, \mathcal{G} \mid \exists \tau \{= e\}?, \mathcal{G} \mid e \to \mathcal{G} \mid \mathcal{G}_1 \wedge \mathcal{G}_2 \mid \lfloor e \rfloor \mid \mathsf{True} \\
\text{Context} & \mathcal{C} & ::= & \mathcal{C} \forall \tau \mid \mathcal{C} \exists \tau \{= e\}? \mid \mathcal{C} \to e \mid \varepsilon
\end{array}
$$

Tactics

$$
\begin{array}{rcl}
\mathsf{rtac} & = & \mathcal{C} \to e \to \mathsf{option}\big(\mathcal{G} \times \mathcal{C}\big) \\
\mathsf{rtacK} & = & \mathcal{C} \to \mathcal{G} \to \mathsf{option}\big(\mathcal{G} \times \mathcal{C}\big)
\end{array}
$$

Goal Denotation

$$
\begin{array}{rcl}
[\![-]\!]^{\mathcal{G}}_{tus,tvs} & : & \mathcal{G} \to \mathsf{ExprT}\ tus\ tvs\ \mathsf{Prop} \\
[\![\forall \tau, g]\!]^{\mathcal{G}}_{tus,tvs} & = & \forall x{:}\tau, [\![g]\!]^{\mathcal{G}}_{tus,tvs \uplus [x]} \\
[\![\exists \tau, g]\!]^{\mathcal{G}}_{tus,tvs} & = & \exists x{:}\tau, [\![g]\!]^{\mathcal{G}}_{tus \uplus [x],tvs} \\
[\![\exists \tau = e, g]\!]^{\mathcal{G}}_{tus,tvs} & = & [\![g]\!]^{\mathcal{G}}_{tus \uplus [[\![e]\!]_{tus,tvs}],tvs} \\
[\![e \to g]\!]^{\mathcal{G}}_{tus,tvs} & = & [\![e]\!]_{tus,tvs} \to [\![g]\!]^{\mathcal{G}}_{tus,tvs} \\
[\![g_1 \wedge g_2]\!]^{\mathcal{G}}_{tus,tvs} & = & [\![g_1]\!]^{\mathcal{G}}_{tus,tvs} \wedge [\![g_2]\!]^{\mathcal{G}}_{tus,tvs} \\
[\![\lfloor e \rfloor]\!]^{\mathcal{G}}_{tus,tvs} & = & [\![e]\!]_{tus,tvs} \\
[\![\mathsf{True}]\!]^{\mathcal{G}}_{tus,tvs} & = & True
\end{array}
$$

Context Denotation

$$
\begin{array}{rcl}
[\![-]\!]^{\mathcal{C}}_{tus,tvs} - & : & \Pi c{:}\mathcal{C}, \mathsf{ExprT}\ (tus \uplus c_U)\ (tvs \uplus c_V)\ \mathsf{Prop} \\
 & & \quad \to \mathsf{ExprT}\ tus\ tvs\ \mathsf{Prop} \\
[\![c \forall \tau]\!]^{\mathcal{C}}_{tus,tvs}\ P & = & [\![c]\!]^{\mathcal{C}}_{tus,tvs}(\lambda us\ vs.\ \forall x{:}\tau, P\ us\ (vs \uplus [x])) \\
[\![c \exists \tau]\!]^{\mathcal{C}}_{tus,tvs}\ P & = & [\![c]\!]^{\mathcal{C}}_{tus,tvs}(\lambda us\ vs.\ \forall x{:}\tau, P\ (us \uplus [x])\ vs) \\
[\![c \exists \tau = e]\!]^{\mathcal{C}}_{tus,tvs}\ P & = & [\![c]\!]^{\mathcal{C}}_{tus,tvs}(\lambda us\ vs.\ \forall x{:}\tau, x = [\![e]\!]^{\tau}_{tus,tvs}\ us\ vs \to P\ (us \uplus [x])\ vs) \\
[\![\varepsilon]\!]^{\mathcal{C}}_{tus,tvs}\ P & = & P
\end{array}
$$

Figure 6.2: $\mathcal{R}_{tac}$'s core definitions presented mathematically. Goals denote to propositions and contexts denote to propositions with "holes."

The `option` in the tactic signals the potential to fail. For example, if a tactic attempts to apply a lemma that does not apply to the current goal, then the apply tactic will fail. I will discuss failure in more detail in Section 6.3.2. For now it is sufficient to assume that tactics always succeed, which is the interesting case for verification.

GOALS The goal data type (also called the goal tree) expresses the remaining proof obligations, which are represented by *e*s in the syntax. In $\mathcal{L}_{tac}$, these are flattened into a list of obligations, but the need to correctly maintain a cactus of substitutions means that it is easier to represent this sharing in $\mathcal{R}_{tac}$ using a representation of goals that mirrors the proof tree. It also means that tactic continuations (`tacK`) can manipulate entire sub-proofs, which is a useful feature that is necessary to support goal minimization (see Section 6.3.3).

Goals are built from a subset of the core logical connectives in an intuitionistic logic, e.g. `True`, $\wedge$, $\forall$, $\exists$, and $\rightarrow$. Exposing this structure allows $\mathcal{R}_{tac}$ to reason underneath hypotheses and binders introduced by the quantifiers. Existential quantifiers can optionally carry expressions corresponding to their actual instantiations. To reason semantically about goals, $\mathcal{R}_{tac}$ provides a denotation function for goals (parameterized by the denotation of the underlying proposition syntax) that maps them directly to contextualized propositions in the natural way.

To be concrete, consider the goal at different stages in the example proof earlier. Initially, the entire goal is a single, opaque expression from $\mathcal{R}_{tac}$'s point of view (represented as $\lfloor e \rfloor$). When the new unification variable is used to instantiate the existential, some structure is revealed in the goal and the new goal becomes "$\exists x : \mathbb{N}, \lfloor e' \rfloor$" where e' represents the conjunction under the existential. Since $\mathcal{R}_{tac}$ knows about the existential, it can run tactics underneath it. This allows $\wedge$-I to expose the conjunction in the underlying goal resulting in "$\exists x : \mathbb{N}, \lfloor e'' \rfloor \wedge \lfloor e''' \rfloor$". Eventually the goal expands and tactics solve the left side, resulting in the goal "$\exists x = 0 : \mathbb{N}, (\forall y : \mathbb{N}, \mathsf{True}) \wedge \lfloor e''' \rfloor$ where the value of x has been instantiated with 0.

Contexts Contexts form a zipper-like structure [90] that encodes the path from the root of the goal (represented as ε) to the current goal. The denotation of a context is a transformer of contextualized propositions where the proposition to fill the hole can additionally mention the variables introduced by the context. This is captured by the type of $[\![c]\!]^{\mathcal{C}}_{tus,tvs}$, which accepts a semantic value with variables equal to those in the ambient environment (tus and tvs) as well as those that come from the context (written c_U and c_V) and returns a semantic value with variables that come only from the ambient environment. For example,

$$\begin{aligned}
[\![\varepsilon \exists \tau_a \forall \tau_b \to e]\!]^{\mathcal{C}}_{tus,tvs} \quad &: \quad \mathsf{ExprT}\ (tus \uplus [\tau_a])\ (tvs \uplus [\tau_b])\ \mathsf{Prop} \\
&\quad \to \mathsf{ExprT}\ tus\ tvs\ \mathsf{Prop} \\
&= \quad \lambda P.\lambda us\ vs.\forall a : [\![\tau_a]\!], \forall b : [\![\tau_b]\!], [\![e]\!]_{(us \uplus [a]),(vs \uplus [b])} \to \\
&\qquad\qquad P\ (us \uplus [a])\ (vs \uplus [b])
\end{aligned}$$

The most interesting part of contexts is the way that the denotation function handles existential quantification. Recall that the key to incrementally constructing proofs is the need to construct maximally parametric proofs with respect to the values of uninstantiated existential variables. To capture this property, $\mathcal{R}_{tac}$ uses a *universal quantifier* as the denotation for *existential quantifier*. In this approach, the existential quantifier is instantiated by adding a new equation stating an equality between the unification variable and the term to instantiate it. This technique is captured by the difference between the second and third equations in the context denotation function in Figure 6.2.

Contexts as Logics While $\mathcal{R}_{tac}$ is primarily interested in a single logic, contexts actually induce derived logics on top of the base logic, e.g. going under a hypothesis induces another logic where the hypothesis becomes an axiom. Two properties are important for proving facts in these derived logics. First, tautologies can be embedded into contexts for free. Formally,

$$\forall P, \left(\forall us\ vs, P\ us\ vs\right) \to [\![c]\!]^{\mathcal{C}}_{tus,tvs} P$$

Note that P is a semantic object that is not necessarily representable syntactically. This is related to the `pure` morphism of applicative functors [116].

In addition to embedding tautologies, contextual facts can also be combined under the context. Formally, this combination is captured by the following proof rule.

$$\forall P\, Q,\, [\![c]\!]^{\mathcal{C}}_{tus,tvs}\big(\lambda us\ vs.P\ us\ vs \to Q\ us\ vs\big) \to [\![c]\!]^{\mathcal{C}}_{tus,tvs}\, P \to [\![c]\!]^{\mathcal{C}}_{tus,tvs}\, Q$$

Again, P and Q are semantic objects, and the contextualized function (the first unnamed argument) shows how to manipulate a proof of P into a proof of Q in the context. This operation is related to the `ap` morphism of applicative functors and to cut-elimination in logic.

In the remainder of the chapter I will abuse notation and use standard logical notation for contextualized properties. For example, when P and Q are contextualized propositions:

$$P \to Q \quad \equiv \quad \lambda us\ vs.P\ us\ vs \to Q\ us\ vs$$

6.1.2 Local Reasoning

The most common type of reasoning that $\mathcal{R}_{tac}$ tactics perform is local, backwards reasoning. The tactic inspects the goal (G) and computes a new goal (G'). The soundness of this reasoning step is justified by an implication from the post-goal to the pre-goal under the current context (c). Mathematically,

$$[\![c]\!]^{\mathcal{C}}_{tus,tvs}\big(G' \to G\big)$$

It is important to note that this statement is stronger, i.e. it says more, than

$$[\![c]\!]^{\mathcal{C}}_{tus,tvs}G' \to [\![c]\!]^{\mathcal{C}}_{tus,tvs}G$$

The extra strength of this statement comes in the knowledge that the values introduced by the context are the same values used to fill in both G and G'.

Proving the implication inside the context gives it access to facts stored in the context. For example, the following theorem allows using facts learned from hypotheses that occur in the context.

$$\forall c\, h, h \in c \to [\![c]\!]^{\mathcal{C}}_{tus,tvs}\, [\![h]\!]_{tus \uplus c_U, tvs \uplus c_V}$$

Note that in addition to accessing the context, proving this theorem also relies on weakening of expressions since new universal and existential quantifiers can be introduced between the hypothesis and the goal.

In addition, all of the equations implied by the instantiation of unification variables are also implied by the context using similar reasoning. Formally,

$$\forall c, [\![c]\!]_{tus,tvs}\, [\![\mathsf{substFor}\ c]\!]_{tus \uplus c_U, tvs \uplus c_V}$$

where substFor extracts the substitution from the context, and the denotation converts it to a conjunction of equations using the denotation function for contexts.

6.1.3 Global Reasoning

Global (or context) reasoning is slightly more complicated than local reasoning. Global reasoning needs a way to convert any semantic proof under the old context into a semantic proof in the new context. The following definition captures this.

$$\mathsf{Evolves}\ c\ c' \quad \triangleq \quad \forall P, [\![c]\!]^{\mathcal{C}} P \to [\![c']\!]^{\mathcal{C}} P$$

Note that here P is the same in both places, but the context changed from c to c'.

Uncurrying the definition makes it clear that the the proof is purely about the context. $[\![c]\!]^{\mathcal{C}} P$ is an arrow type with a co-domain of P, so we can rewrite it into "$[\![c]\!]^{T} \to P$" where $[\![c]\!]^{T}$ is a telescope [48] that carries all of the values in the context. Using this definition, Evolves becomes

$$\forall P, ([\![c]\!]^{T} \to P) \to ([\![c']\!]^{T} \to P)$$

This is isomorphic to the following direct implication, which is the same as the reverse implication that arose in the `eprove` soundness lemma in Section 3.4.1.

$$[\![c']\!]^T \to [\![c]\!]^T$$

It is illustrative to see an actual proof of Evolves. Consider the evolution of "$\forall x : \mathbb{N}, \exists y : \mathbb{N}, \forall z : \mathbb{N}$" when y is instantiated with the value of x. Using the definition of Evolves the obligation is the following (recall the meaning of the existential quantifier is a universal quantifier):

$$\begin{aligned}\forall P, (\forall x : \mathbb{N}, \forall y : \mathbb{N}, \forall z : \mathbb{N}, P\,x\,y\,z) \to \\ (\forall x : \mathbb{N}, \forall y : \mathbb{N}, y = x \to \forall z : \mathbb{N}, P\,x\,y\,z)\end{aligned}$$

The proof takes the facts from the right-hand side and uses them to prove the premises on the left. Even the proof term is simple.

```
fun P pf x y pfY z ⇒ pf x y z
```

The equality between y and x is simply dropped because it is not necessary to prove P.

NATURAL PROOFS OF EVOLUTION While correct, the definition of Evolves above is not quite strong enough for $\mathcal{R}_{tac}$. The problem is a lack of parametricity which rears its head when $\mathcal{R}_{tac}$ needs to escape from under a quantifier. The proof above is exactly the one that we want, but in inconsistent contexts there is another, less parametric proof. For illustrative purposes, take the above example but convert the type of z from $\mathbb{N}$ to $\emptyset$. Revisiting the obligation demonstrates the problem:

$$\forall P, (\forall x : \mathbb{N}, \forall y : \mathbb{N}, \forall z : \emptyset, P\,x\,y\,z) \to (\forall x : \mathbb{N}, \forall y : \mathbb{N}, y = x \to \forall z : \emptyset, P\,x\,y\,z)$$

The above proof still solves the goal, but there is another, less natural[1] proof of this proposition that leverages the $\emptyset$ to avoid its obligation to prove P. The unnatural

[1] In the category theoretic sense [87]

proof is the following:

```
fun P pf x y pfY z ⇒ elim∅ z
```

This proof is not strong enough to let us escape from under an inconsistent context. To see why, suppose that an $\mathcal{R}_{tac}$ begins reasoning in the above context and offers the above proof as evidence that it preserved the context. After the tactic returns, $\mathcal{R}_{tac}$ needs to escape from under the $\forall z : \emptyset$ by using the above proof to build a proof of the following:

$$\forall P, (\forall x : \mathbb{N}, \forall y : \mathbb{N}, P\,x\,y) \to (\forall x : \mathbb{N}, \forall y : \mathbb{N}, y = x \to P\,x\,y)$$

A simple manipulation of the natural proof above (dropping all occurrences of z) makes it easy to build the natural proof of this obligation (though this is rightfully not possible to do in Gallina).

```
fun P pf x y pfY ⇒ pf x y
```

But there is no such simple manipulation of the unnatural proof that has the same property. Ultimately, each piece of the context introduces a new "level" of the proof, and it is necessary to ensure that higher-level knowledge (facts that are introduced by deeper quantifiers) cannot be used to justify lower facts. This prevents, for example, proving $(P \to \top) \wedge P$ by using the negative proof of P on the left of the conjunction to prove the positive occurrence of P to the right of the conjunction.

One way to solve this problem would be to abandon MirrorCore's insistence on semantic objects and require syntactic evolution of contexts. Bindings in substitution cannot change except for through further instantiation, and no existing bindings can be removed. While this would work, it is somewhat restrictive. For example, it prevents any form of semantic reasoning within the substitution's denotation, for example replacing `0 + x` with `x`.

$\mathcal{R}_{tac}$ takes a more semantic approach. The soundness of a tactic requires an appropriate Evolves proof at every level of the context[2]. Recalling the example that suggested the problem, under $\mathcal{R}_{tac}$'s solution a tactic is obligated to prove both of the following:

$$\forall P, (\forall x : \mathbb{N}, \forall y : \mathbb{N}, \forall z : \emptyset, P\,x\,y\,z) \rightarrow (\forall x : \mathbb{N}, \forall y : \mathbb{N}, y{=}x \rightarrow \forall z : \emptyset, P\,x\,y\,z)$$
$$\forall P, (\forall x : \mathbb{N}, \forall y : \mathbb{N}, P\,x\,y) \rightarrow (\forall x : \mathbb{N}, \forall y : \mathbb{N}, y{=}x \rightarrow P\,x\,y)$$

While this may seem wasteful, since $\mathcal{R}_{tac}$ is fully reflective all of this overhead takes place when the tactic is proved, not when the tactic is run. Further, since this approach is purely semantic, it provides the full richness of the underlying proof language rather than imposing arbitrary restrictions based on the underlying syntax.

These two types of reasoning form the semantic "assembly language" of tactics. To be easy to use, $\mathcal{R}_{tac}$ contains a range of tactics that capture slightly higher-level reasoning steps at the granularity of the core $\mathcal{L}_{tac}$ tactics.

6.2 Core Tactics

These core tactics can be categorized into two groups: tactics for proving which manipulate goals (Section 6.2.1) and tactics for combining other tactics into automation procedures (Section 6.2.2).

6.2.1 Tactics for Proving

Proving tactics are meant to express finer granularity operations such as splitting a conjunction or applying a lemma. Their small nature makes them good building blocks for developing larger automation. For example, the backward reasoning proof procedure described in Chapter 3 is essentially repeated use of $\mathcal{R}_{tac}$'s EAPPLY tactic. This tactic implements the lemma application algorithm that I described in

[2]In actuality lower proofs can be used to build higher proofs when there are no changes to the context between them. Therefore it is only necessary to witness proofs in places where the context can change.

detail in Section 3.4.1.

Goal manipulation tactics are also easy to implement. Any function that preserves the semantic meaning of an expression can be converted into a sound tactic. The entire tactic is a one-liner in $\mathcal{R}_{tac}$[3]:

```
Definition SIMPLIFY : rtac := fun c e ⇒ Some (GGoal (reduce e c), s).
```

This tactic can be used to rewrite by equalities, perform β-reduction, and instantiating unification variables, just to name a few. Making these functions into tactics using `SIMPLIFY` is simply a matter of proving the transformation from P to Q implies that $[\![P]\!] \leftrightarrow [\![Q]\!]$ (in fact, the proof can be weakened to $[\![Q]\!] \rightarrow [\![P]\!]$).

Other tactics such as `INTRO` and `EEXISTS` are equally simple because they convert syntactic expressions into their corresponding $\mathcal{R}_{tac}$ representations. For example, using the `prop` language from Chapter 2, the tactic for introducing a hypothesis is trivial to write:

```
Definition INTRO_hyp : rtac :=
  fun _ e ⇒ match e with
             | pImpl P Q ⇒ Some (GHyp P (GGoal Q), s)
             | _ ⇒ None
             end.
```

Implications are exposed to $\mathcal{R}_{tac}$ through the `GHyp` constructor, and the remainder of the goal is included via `GGoal`. The soundness theorem essentially amounts to the following trivial facts:

$$\begin{array}{rc} \text{Soundness} & \forall c, [\![c]\!]^{\mathcal{C}} \left(([\![P]\!] \rightarrow [\![Q]\!]) \rightarrow ([\![P]\!] \rightarrow [\![Q]\!]) \right) \\ \text{Context Soundness} & \forall c\, G, [\![c]\!]^{\mathcal{C}} G \rightarrow [\![c]\!]^{\mathcal{C}} ([\![e]\!] \rightarrow G) \end{array}$$

A Note About Parametricity Since $\mathcal{R}_{tac}$ is fully parametric in the underlying term language, proving tactics must also be parametric in the operations that they require on terms. For example, the generic `EAPPLY` tactic requires expres-

[3]The actual code has a few more variables but also has a similarly simple implementation.

sions to have a representation of unification variables and functions that manipulate them. MirrorCore provides default instantiations for many of these tactics when the underlying language is MirrorCore's extensible syntactic representation. Some tactics, such as introducing a quantifier or splitting a conjunction, require deeper semantic knowledge about the domain, for example the representation of conjunction. In these cases, $\mathcal{R}_{tac}$ exposes procedures that are parameterized by functions that inspect the term and take care of the $\mathcal{R}_{tac}$ term manipulation. For example, the generic `INTRO` tactic requires a function (`open`) that satisfies the following specification in order to introduce hypotheses:

```
∀ tus tvs g g', open g = Hyp h g' →   (* the function succeeded *)
∀ gD, propD tus tvs g = Some gD →   (* initial goal means [gD] *)
  ∃ eD' hD, propD tus tvs h = Some hD ∧ (* the hypothesis means [hD] *)
      propD tus tvs g' = Some g'D ∧ (* the new goal denotation [g'D] *)
      (* the implication holds in all contexts *)
      ∀ us vs, (hD us vs → g'D us vs) → (gD us vs)
```

In English this specification says "if `open e` returns `Hyp h g'` and `e` is a well-typed proposition then both `h` and `g'` are well-typed and $(\llbracket h \rrbracket \to \llbracket g' \rrbracket) \to \llbracket g \rrbracket$."

6.2.2 Combinators for Proof Search

The $\mathcal{R}_{tac}$ programming model is based around a similar backtracking search procedure which is similar in many respects to $\mathcal{L}_{tac}$'s model. As I mentioned previously, $\mathcal{R}_{tac}$ tactics can either succeed or fail. It is failure that triggers backtracking.

In addition to the core parameterized reasoning tactics, $\mathcal{R}_{tac}$ also provides a collection of tactic combinators including: identity, failure, alternation, bounded recursion, and sequencing. These are easy to implement in $\mathcal{R}_{tac}$.

```
IDTAC : rtac
FAIL  : rtac
ALT   : rtac → rtac → rtac
REC   : (* fuel : *) ℕ → (rtac → rtac) → rtac → rtac
AFTER : rtac → rtac → rtac
```

The soundness of these tactics is derived directly from the soundness of their components. Take `ALT` as an example. First it runs the first tactic, and if the first tactic fails, then `ALT` will run the second tactic. The soundness statement is the following:

```
Theorem ALT_sound
: ∀ t1 t2, rtac_sound t1 → rtac_sound t2 → rtac_sound (ALT t1 t2).
Proof. ... Qed.
```

The proof follows immediately from the (very simple) implementation. In fact, in general, the combinators are significantly easier to prove than the actual tactics.

Tactic Continuations

In almost all cases automation applies to an individual goal. However, there are some cases when we want or need to process an entire goal tree. Implementing the `AFTER` tactic combinator is one such instance. After the first tactic runs on a local goal, it produces a goal tree, not just a single local goal. What is the correct semantics for running the second tactic? A logical choice is to run the second tactic on each leaf goal produced by first tactic. This choice corresponds directly to $\mathcal{L}_{tac}$'s semicolon operator.

While this choice makes a good default, there are cases when we want to leverage more information to improve the proof search. Take applying the following lemma for example:

```
lem : ∀ x y, Even x → Odd y → Odd (x + y).
```

If this lemma applies to the goal, two goals are produced, one for `Even` and the other for `Odd`. If we have separate automation for `Even` and `Odd`, we would like to set up the automation so that it calls the tactic to prove evenness on the first goal and the tactic to prove oddness on the second goal. In $\mathcal{L}_{tac}$ this pattern of combination is supported by the "tac; [tac | tac]" construct.

$\mathcal{R}_{tac}$ tactic continuations provide a unifying way to describe the semantics of both the standard chaining and the hybrid chaining feature above. Tactic continuations have exactly the same type as tactics except that they accept goal trees in-

stead of single goals.

$$\mathsf{rtacK} \quad := \quad \mathcal{C} \to \mathcal{G} \to \mathsf{option}\,(\mathcal{G} \times \mathcal{C})$$

Using this definition $\mathcal{R}_{tac}$ provides `ON_ALL`, which applies a single tactic to each goal in a goal tree. This behavior is the same as $\mathcal{L}_{tac}$'s semi-colon operator. The `USE_EACH` tactic continuation takes a list of tactics and runs one on each leaf goal in the goal tree. As with $\mathcal{L}_{tac}$, `USE_EACH` fails if the provided number of tactics is not exactly equal to the number of leaf goals in the reflective procedure. This makes it simple to, for example, fail if more than one subgoal remains using a script such as.

```
AFTER_K (tac_to_run) (USE_EACH [ IDTAC ])
```

Here, `AFTER_K` generalizes `AFTER` by taking a tactic continuation as its second argument. Using `AFTER_K`, `AFTER` has a simple implementation

```
Definition AFTER (tac1 tac2: rtac) : rtac :=
  AFTER_K tac1 (ON_ALL tac2).
```

Beyond providing a compositional semantics for the various forms of chaining, tactic continuations also provide access to the goal tree, which gives them the ability to, e.g. clear hypotheses or instantiate unification variables. Tactic continuations are also key to implementing first-class goal minimization, which I will discuss in Section 6.3.3.

6.3 Performance

The primary goal of $\mathcal{R}_{tac}$ is not blindingly fast tactics, though with appropriate support accomplishing this might be feasible. More importantly, $\mathcal{R}_{tac}$ is useful primarily for two purposes. First, it makes building simple tactics relatively straightforward. Second, $\mathcal{R}_{tac}$ makes it easy to glue together more specialized automation. For example, replacing some of Bedrock's $\mathcal{L}_{tac}$ for higher-order reasoning with $\mathcal{R}_{tac}$ could avoid the performance penalties of reflecting and reifying terms.

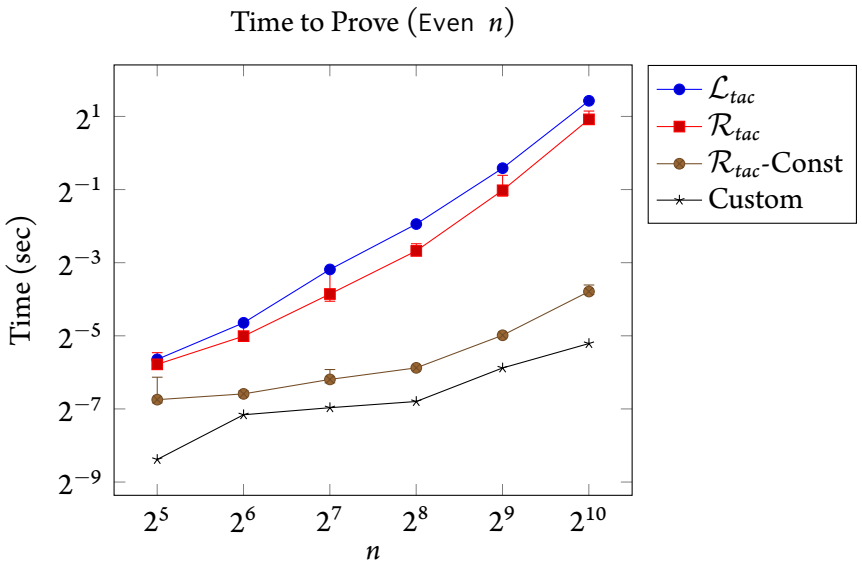

Figure 6.3: Comparison of $\mathcal{L}_{tac}$, $\mathcal{R}_{tac}$, and a custom decision procedure for proving constants Even.

6.3.1 Evenness

In Chapter 1 I demonstrated a very simple procedure for proving evenness of large constants and compared it to proving the same property in $\mathcal{L}_{tac}$. $\mathcal{R}_{tac}$ provides another way to solve this goal which I showed in Figure 6.1. Figure 6.3 shows the performance comparison for proving various numbers even in $\mathcal{L}_{tac}$, $\mathcal{R}_{tac}$ and using the custom decision procedure.

First, notice the significant difference between $\mathcal{L}_{tac}$ and the custom decision procedure. While $\mathcal{L}_{tac}$ is quadratic in the size of the number, the custom decision procedure is linear. The direct translation of $\mathcal{L}_{tac}$ to $\mathcal{R}_{tac}$ has performance closer to $\mathcal{L}_{tac}$ than the custom Gallina decision procedure. This poor scaling is due in part to the large terms that appear during the computation. For example, in MirrorCore's representatation, the size of 1024 contains more than 2048 constructors. Constructing and representing the term is one thing, but computing on such large terms can also be expensive. For example, the occurs check is linear in the size of

the term.

To avoid some of the overhead of reifying and computing on large terms, we can leverage MirrorCore's general term algebra to lazily reify constants in the goal. To do this, we include a special representation of constants into the symbol language and have its denotation simply be the value stored in its argument. To be concrete, to apply this technique to the evenness problem, we would augment our reflective constants as follows:

```
Inductive func :=
| ConstNat : ℕ → func
| Zero | Succ
| Even | Odd.
```

```
Definition funcD (f : func) :=
  match f with
  | ConstNat n ⇒ n
   ...
```

Using this approach, we reify 1024 into "`Inj (ConstNat 1024)`" rather than exposing all of the individual applications. Since our procedures know that all terms inside of `ConstNat` are constants, they can compute on the constants without worrying about becoming stuck on opaque symbols. The automation can then use custom "parametric" lemmas to implement the same proof search much more efficiently. The performance of this custom representation is shown in the $\mathcal{R}_{tac}$-Const line. Figure 6.4 shows the relevant code excerpts. The tactic uses the `AT_GOAL` tactic combinator which is similar to $\mathcal{L}_{tac}$'s `match goal with` tactical and allows the developer to inspect the goal and the context before picking the tactic to run. In this case, I use `AT_GOAL` to determine the number to use to instantiate the parametric lemma. The tactic also uses the derived `REPEAT` tactical, which runs the tactic at most `fuel` times and chains each with an implicit recursive call.

Supporting Variables

The above evenness experiment shows $\mathcal{R}_{tac}$ in a poor light for two reasons. First, the size of the problem is directly proportional to the size of the proof. We see this in the results when we note that the reification process accounts for a substantial piece of the overall verification time. Second, the problem domain is so simple

```
(* parametric lemma *)
Definition pEO_syn (n : ℕ) : lemma :=
{ foralls := nil
; premises := App Odd_syn (Const n) :: nil
; conclusion := App Even_syn (Const (S n)) }.

Definition even_odd_tac_nrec (fuel : ℕ) : rtac :=
  REPEAT fuel (AT_GOAL (fun e ⇒
                          match e with
                          | App (Inj fEven) (Const 0) ⇒ APPLY pE0_syn
                          | App (Inj fEven) (Const (S n)) ⇒ APPLY (pEO_syn n)
                          | App (Inj fOdd) (Const (S n)) ⇒ APPLY (pOE_syn n)
                          | _ ⇒ FAIL
                          end)).
```

Figure 6.4: Parametric lemmas enable more efficient reflective procedures by avoiding the need to reify large constants.

that it does not even require a syntactic representation. Thus any use of "standard" reflection will incur a cost in building this representation. The second $\mathcal{R}_{tac}$ implementation skirts this issue by reifying the entire constant into a single piece of syntax, and the resulting procedure is substantially faster.

Making the problem just slightly more complicated, however, shows off $\mathcal{R}_{tac}$'s generality. Introducing variables requires that the entire Gallina procedure be rewritten, while the $\mathcal{R}_{tac}$ procedure is easy to adapt since MirrorCore's representation already handles variables. To demonstrate the effect that adding variables has on the reasoning, consider enriching the automation to be able to solve the following problems:

$$\mathsf{Even}\ x \vdash \mathsf{Even}\ (\mathsf{S}\ (\mathsf{S}\ x))$$
$$\mathsf{Odd}\ x \vdash \mathsf{Odd}\ (\mathsf{S}\ (\mathsf{S}\ x))$$

Since x in both of these goals is opaque the custom reflective procedure is unable to solve these. Because $\mathcal{R}_{tac}$ is built on top of a reflected term syntax, it can solve interesting problems such as these. For example, the naïve procedure that I showed in Figure 6.1 is able to solve both of these goals.

Even when only constants are involved it can be useful to exposed the additional

structure in the goal. For example, consider trying to prove the following

$$\vdash \mathsf{Even}\ (\mathsf{pow}\ 2\ 20)$$

Since everything is a constant the simple tactic should work. To use the reflective procedure, however, the exponentiation must be completely reduced. In Coq 8.4, this reduction leads to a stack overflow. We can avoid this problem by exploiting extra structure in the goal, essentially hiding the large number behind the universal quantifier in the following lemma:

$$\vdash \forall n\, m, \mathsf{Even}\ n \to m > 0 \to \mathsf{Even}\ (\mathsf{pow}\ n\ m)$$

A one-line change to an $\mathcal{L}_{tac}$ script or an $\mathcal{R}_{tac}$ definition enables the automation to use this lemma to avoid producing the large term. Solving the goal after that simply requires a small amount of additional automation for the $m > 0$ obligation.

6.3.2 Backtracking Proof Search

In Chapter 3, I demonstrated MirrorCore's unification variables by implementing a backtracking proof search that combined lemmas and custom reflective procedures. That entire algorithm is easy to implement using the parameterized $\mathcal{R}_{tac}$ in Figure 6.5. The soundness proof for this tactic is also easy to prove—though not completely automatic since the `try_lems` tactic is constructed from an unknown list of sound lemmas.

While not completely automatic, the ability to implement this procedure within Gallina highlights the ability to define new abstractions such as hint databases directly within the logic. For example, we can easily modify the above tactic to perform a depth-bounded depth-first search, or, with a bit more work, a breadth-first search. While the first can be coded in $\mathcal{L}_{tac}$ using `eauto 1; eauto 2; eauto 3`, there is no easy implementation of the latter.

In addition to other search strategies, it is also possible to perform offline pre-

```
Definition EAUTO_USING (fuel: ℕ)(tac: rtac) (lems: list lemma)
: rtac :=
  let try_lems rec :=
    map (fun x ⇒ SOLVE (THEN (EAPPLY x) rec)) lems in
  TRY (REC fuel (fun rec ⇒ FIRST (ASSUMPTION :: tac :: try_lems rec))).

Theorem EAUTO_USING_sound : ∀ tac lems,
   rtac_sound tac → Forall Provable lems →
   rtac_sound (EAUTO_USING fuel tac lems).
Proof. ... Qed.
```

Figure 6.5: The implementation of eauto in $\mathcal{R}_{tac}$ takes a reified "hint database" (lems) and applies the lemmas using EAPPLY. Failure is handled by the backtracking semantics of FIRST.

processing of search strategies. For example, we can compile a collection of lemmas into a discrimination tree [40] to perform simultaneous unification with many lemmas.

6.3.3 The Importance of Minimizing Goals

$\mathcal{R}_{tac}$'s task of maintaining the goal state can become expensive, especially as unification variables pile up. Part of this is due to the current implementation of the occurs check, which is linear in the size of all of the terms in the unification table. Coq's own implementation is able to make judicious use of side-effects and laziness to avoid rescanning lists, but Gallina does not support this kind of programming.

To achieve good performance when applied to large problems, it is essential to minimize goal representations in places where known unification variables are likely to stack up. Take a naïve implementation of EAPPLY for example. The implementation introduces a new unification variable for each quantified variable and unifies the lemma's conclusion with the goal. In many cases, applying a lemma will instantiate many of the newly introduced unification variables, and it is essential that these variables be eliminated quickly to keep the substitution small. Concretely, minimization would convert a goal such as "$\exists y = 3, (\mathsf{Odd}\, y \wedge \mathsf{True})$"

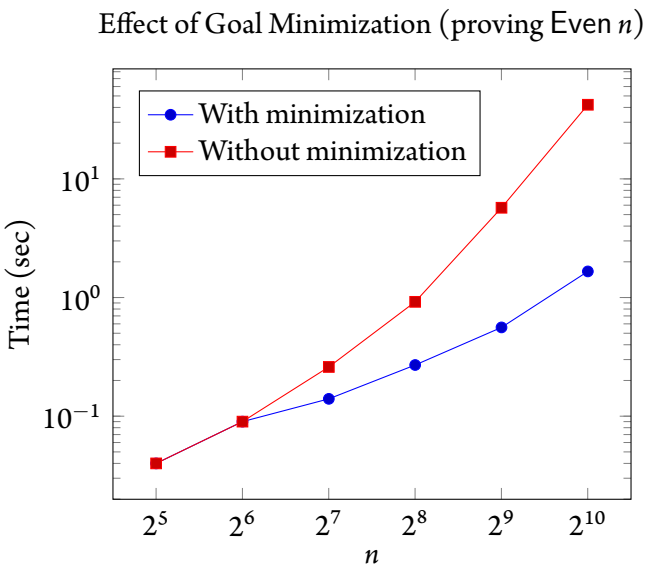

Figure 6.6: Minimizing goals is essential to achieve good asymptotic performance when tactics generate many unification variables. The time represents only computation time, not the time for reification.

into "Odd 3."

Figure 6.6 shows how important goal minimization is in practice. In the line without goal minimization, I have removed the internal minimization steps from the evenness prover for constants described in Section 6.3.1. This change increases the complexity from linear (with a coefficient of approximately $\frac{1}{1000}$) to quadratic (with a coefficient of almost 6).

6.4 Related Work

In this chapter I discussed the implementation of a fully-reflective, $\mathcal{L}_{tac}$-like tactic language. Tactic-based proving is a common approach in Coq and has facilitated the development of impressive proof artifacts [54, 58, 85, 96, 105, 110]. Recent work has even demonstrated how $\mathcal{L}_{tac}$ can be used for tasks such as program synthesis [73].

Despite its prevalence, the semantics of $\mathcal{L}_{tac}$ has historically been somewhat of

a mystery [21] in part due to implementation bugs obscuring some of the details. Recent work by Jedynak [94] has built an abstract machine semantics for $\mathcal{L}_{tac}$. The semantics includes $\mathcal{L}_{tac}$'s `match goal`, which allows tactics finer granularity access to the hypotheses and the structure of the goal. $\mathcal{R}_{tac}$ supports this type of goal inspection but reasoning about it has proven difficult. A promising approach to solving this problem would be a program logic for tactic verification. The difficulty here lies in building a flexible enough logic to reason about back-tracking failures.

Recent work by Devriese and Piessens [74] developed some reflective tactics for Agda [8]. The work aims to do strongly-typed meta-programming using computational reflection. As an example, they develop an assumption tactic whose type encodes its correctness property. Translated into Coq the specification is the following:

```
Definition assumptionTactic : ∀ n T {Γ : Context n} (ty_t : Γ ⊢ T : Type)
    (ty_Γ : ⊢ Γ) (asmpts : interpCtx ty_Γ),
ifYes (inContext Γ T) (interpSet ty_t ty_Γ asmpts).
```

Here, `inContext` is the executable part of the tactic, and the body of the definition is the proof of its correctness. As is usual in Agda developments, their work leverages dependent types heavily, which can incur heavy execution costs, though Agda contains some features to mitigate this [7]. The authors report no performance numbers to hint at the overhead or larger case studies like the ones I discussed in Section 6.3. Further, their work does not support many of the conveniences such as unification variables and unification that $\mathcal{R}_{tac}$ inherits from MirrorCore and are essential to building more sophisticated tactics such as `APPLY` while maintaining compositionality.

Several lines of recent research have looked into alternative tactic languages for Coq. Claret's work [59] axiomatizes a "tactic monad" that provides search-oriented features that can be more efficiently implemented outside of Coq. This allows leveraging imperative algorithms such as union-find [64] to efficiently perform the proof search while retaining soundness by translating traces of the proof search back into Coq. To skirt the issue of needing to reason about the monadic compu-

tation, the computation produces explicit proof objects. The idea of producing a trace could mitigate the problem of re-performing the proof search during proof checking by threading through a trace of the "right" choices that would be produced by the initial reflective procedure.

$\mathcal{M}_{tac}$ [155] adapts some of the ideas in Claret's work to build an alternative tactic language that reuses Gallina. $\mathcal{M}_{tac}$ is built to run in Coq and provides the developer with access to Coq's unification algorithm via term matching. This is useful because it completely avoids the need to build a syntactic representation of the goal. Despite the fact that $\mathcal{M}_{tac}$ terms are written in Gallina, $\mathcal{M}_{tac}$'s evaluation strategy constructs proofs at run time which are later checked by the Coq kernel. This is essential due to the reliance on algorithms such as higher-order unification that are outside of Coq's trusted computing base. Further, the ability for $\mathcal{M}_{tac}$ terms to inspect values syntactically means that $\mathcal{M}_{tac}$ computations do not respect propositional equality, i.e. if *tac* is an $\mathcal{M}_{tac}$ tactic then it is *not* necessarily the case that

$$\forall x\, y, x = y \rightarrow \mathsf{run}\ tac\ x = \mathsf{run}\ tac\ y$$

This property prevents us from using Gallina to reason post-facto about $\mathcal{M}_{tac}$ tactics since these tactics are *not* Gallina functions.

The ability to implement tactics in the proof assistant also opens up questions about better, more programmable designs for tactic languages. Brady's Idris [44] language is built around a domain-specific tactic language (written in Haskell) for elaborating Idris programs into a core type theory. This approach has proven quite modular, enabling Idris to rapidly add features such as type classes and implicit arguments with relatively little new code. Brady's tactics are very low level, e.g. create a new unification variable, focus on a unification variable, and construct a term. $\mathcal{R}_{tac}$ could support tactics at this level of generality and use them to build the higher-level features. This approach however may require some optimization or partial evaluation to avoid performing fine-grained checks that are guaranteed to succeed when used to build larger tactics.

6.4.1 Future Work

The primary limitation in $\mathcal{R}_{tac}$ lies in the difficulty of expressing context manipulation tactics. For example, splitting a conjunction which occurs in a hypothesis into two premises is often useful, but in order to keep from doing it repeatedly, it is desirable to clear the conjunction afterwards. In $\mathcal{R}_{tac}$'s current implementation this sort of manipulation can only occur in tactic continuations.

7

Case Study: Embedded Logics for Imperative Programs

Up until this point I have shown a multitude of relatively restricted examples of building on top of MirrorCore predominantly for illustrative purposes. The Bedrock case study was substantial but used MirrorShard which was custom-built for the application. In this chapter I demonstrate the expressivity and flexibility of MirrorCore and $\mathcal{R}_{tac}$ operating together. My application is once again imperative program verification, this time for a simple imperative programming language. The problem will be simpler but the automation will be easier to develop and extend than with the custom procedures that I implemented for Bedrock.

Exercising the modularity and extensibility of MirrorCore, I build the formalism in this chapter using Bengtson's Charge! logic library [29]. This library provides axiomatic interfaces for logics and definitions for layering logics on top

of one another. Charge! was originally built to abstract the logics used to verify Java programs [28] but it has also been applied to x86 machine code [30] and is very similar to Appel's logic [17] for verifying CompCert C [105] programs.

Charge! enables building custom logics that internalize the features of the programming language, e.g. local variables or recursion. This feature makes it more natural to read and write specifications because the logic contains simple ways to talk about the constructs in the program. It also provides modularity by allowing other users to embed orthogonal features into the logic without having to revisit all of the old functionality. Appel's work on C [16] shows how to use this extensibilty to capture features of modern programming languages such as non-computational recursive predicates.

Beyond the automation for the program logics is the need to reason about the program. In Bedrock I built custom automation to perform symbolic evaluation of programs. However, this approach relied crucially on a fixed programming language. Bedrock macros were expanded by verification condition generation before being passed to the symbolic evaluator. Bedrock uses a stylized form of macro to make this work without needing to duplicate all of the work, but it would be nicer to have the symbolic evaluator understand macros from the offset. This approach also avoids the need to generate many verification conditions that must be individually reified, solved, and reflected.

The combination of all of these features stands as strong evidence for my thesis that compositional computation reflection is a powerful technique for quickly building efficient, foundational automation. Core reflective building blocks such as $\mathcal{R}_{tac}$ as well as extensible custom reasoning procedures are essential components to realizing this vision. These pieces provide a gentle ramp for converting inefficient $\mathcal{L}_{tac}$ procedures into more efficient reflective procedures by overcoming the initial hurdle that requires all pieces of a reflective procedure to be reflective. This allows gradual optimization of individual pieces, which enables evolving the reasoning from a *useful* prototype to a performant verification framework.

I begin the chapter with a brief discussion of Charge!'s approach to specifying logics using type classes [137] (Section 7.1). Next I demonstrate two techniques

for using MirrorCore's extensibility to represent type classes (Section 7.2). Building on top of the type classes, I develop automation for a program logic for a simple language using $\mathcal{R}_{tac}$ (Section 7.3). I focus on the ease of translating an $\mathcal{L}_{tac}$ prototype into an $\mathcal{R}_{tac}$ tactic and, despite being as naïve as possible, end up with two orders of magnitude performance improvement for a few hours of work. I conclude the section by considering augmenting the automation to support additional features such as opaque code (Section 7.3.2) and memory (Section 7.3.3). Stepping back I reflect, no pun intended, on some of the inconveniences in the development and consider promising solutions. I conclude by discussing ongoing collaborations that extend the case study presented in this chapter to work with full-fledged programming languages (Section 7.4).

7.1 Describing Axiomatic Logics with Charge!

The core abstraction in the Charge! framework is an intuitionistic logic. Intuitionistic logics provide the core logical operators as well as a notion of entailment ($\vdash$) which is often closely related to implication in the meta logic. These definitions are expressed by the type class definition shown in Figure 7.1[1]. The type index `L` is the type of propositions in the logic, and the fields correspond to the logical operators. The `ILogicLaws` class expresses the laws that these operators adhere to. For example, `landR` states the standard and-introduction rule ($\wedge$-I). When we reason abstractly about a logic, these laws will be the only way to manipulate the symbols.

Phrasing the interface for intuitionistic logics as a set of functions leaves the type open to adding new symbols. For example, we can reason abstractly about any step-indexed intuitionistic logic (L) by quantifying over L and over the corresponding operators.

```
Variable L : Type.
Variables (ILogic_L: ILogic L) (Later_L: Later L).
```

[1] Charge!'s ILogic interface also includes operators for $\forall$ and $\exists$.

```
Class ILogic (L:Type) :=
{ lentails : L → L → Prop (* entailment: "P ⊢ Q" *)
; ltrue : L               (* truth:     "TT" *)
; lfalse : L              (* falsehood: "FF" *)
; land : L → L → L        (* conjunction: "P /\ Q" *)
; lor : L → L → L         (* disjunction: "P \/ Q" *)
; limpl : L → L → L       (* implication: "P -->> Q" *) }.

Class ILogicLaws (L:Type) (LL:ILogic L) :=
{ Preorder_lentails :> Preorder lentails
; ltrueR : ∀ P, P ⊢ TT
; lfalseL : ∀ P, FF ⊢ P
; landR : ∀ P Q R, R ⊢ P → R ⊢ Q → R ⊢ P /\ Q
; landL1 : ∀ P Q, P /\ Q ⊢ P
; landL2 : ∀ P Q, P /\ Q ⊢ Q
; lorR1 : ∀ P Q R, P ⊢ Q → P ⊢ Q \/ R
; lorR2 : ∀ P Q R, P ⊢ R → P ⊢ Q \/ R
; lorL  : ∀ P Q R, P ⊢ R → Q ⊢ R → P \/ Q ⊢ R
; landAdj: ∀ P Q C, C ⊢ (P -->> Q) → C /\ P ⊢ Q
; limplAdj: ∀ P Q C, C /\ P ⊢ Q → C ⊢ (P -->> Q) }.
```

Figure 7.1: The Charge! interface for intuitionistic logics.

This style is convenient because it abstracts the underlying logic. For example, L could also be a separation logic over a heap structure or contain other modalities. In addition, the carrier L could be defined by a deep embedding, where the proof rules of the logic are written as an inductive type; or as a shallow embedding, where the type L is defined in terms of primitive Coq definitions and the proof rules are theorems.

When we verify programs using Charge!'s logics there are a surprising number of logics floating around. For example, standard Gallina propositions form a logic but there are also custom logics for the programming language.

- **Lifted logics** can represent additional state information such as a local variable store or a heap. They can be modeled as a function space where the domain is the type of the extra information and the co-domain is any intu-

itionistic logic. The entailment relation is the pointwise lifting of entailment in the underlying logic.

- **Step-indexed logics** can represent time. These are interesting because step-indexing provides a way to do induction which allows us to conveniently reason about programs and logical assertions that may otherwise not be well-founded. Here, the implication is a generalized weakening that supports arbitrarily decreasing the step index.

7.2 Reifying Type Classes

Type classes are not really anything special within Coq; in fact, adding them does not change the trusted computing base at all. In Coq, type classes are just records[2]. What makes type classes special is the integration with elaboration (the process of taking a Coq term as written by the user and building the core term that Coq reasons about). The problem with reasoning about elaboration is that elaboration is a partial operation. For example if we write the term $1 \vdash 2$, Coq will attempt to find an instance of `ILogic` $\mathbb{N}$, but one of these is unlikely to exist since $\mathbb{N}$ is not an intuitionistic logic.

There are two solutions for representing type classes reflectively in MirrorCore. Both rely heavily on the ability to customize MirrorCore's representation of symbols. The first follows the style of Haskell [145] and reifies the type class resolution algorithm (Section 7.2.1). The second solution more closely follows Coq's model of type classes [137] and reifies type classes and type class instances as types and values respectively (Section 7.2.2).

```
1  (* Type class functions, τ is the logic's type, e.g. Prop *)
2  Inductive ilogic_sym := ilsTrue (τ:typ) | ilsAnd (τ:typ) | ...
3
4  (* The reified resolution algorithm *)
5  Variable getInstance : Π τ : typ, option (ILogic (typD τ)).
6
7  (* symbols are only well-typed if there is an instance *)
8  Definition typeof_ilogic_sym (s: ilogic_sym) : option typ :=
9    match s with
10   | ilsTrue τ ⇒ _ ← getInstance τ ; ret τ
11   | ilsAnd τ ⇒ _ ← getInstance τ ; ret (τ ⇛ τ ⇛ τ)
12   end.
13
14 (* The denotation uses dependent types to return the value
15  * only if the term is well-typed *)
16 Definition ilogic_symD (s: ilogic_sym)
17 : match typeof_ilogic_sym s with
18   | None ⇒ unit
19   | Some τ ⇒ ILogic (typD τ)
20   end :=
21   match s with
22   | ilsTrue τ ⇒ match getInstance τ with (* dependent types omitted *)
23                 | Some l ⇒ @lTrue _ l (* use the returned instance *)
24                 | None ⇒ tt
25                 end
26   | ...
```

Figure 7.2: Encoding second-class type classes in MirrorCore by reifying type class resolution as a partial function. The use of the option monad in `typeof_ilogic_sym` (lines 10-11) ensures that a type class symbol is well-typed only if an appropriate type class instance exists.

7.2.1 Second-class Type Classes

We can represent type classes in a second-class way by reifying the type class resolution function. This allows the denotation function to invoke type class resolution and fail if an appropriate type class does not exist. Figure 7.2 demonstrates the key aspects of this approach. The type of symbols (`ilogic_sym`) includes paramet-

[2]In Coq 8.4 and earlier, records are themselves simply non-recursive inductive data types with special syntax.

ric symbols for each type class function (line 2). The `getInstance` function is the reified type class resolution function which returns an option with the appropriate instance. Since type class functions only exist when a corresponding type class instance exists, symbols are only well-typed if the type class resolution function finds an appropriate instance.

What makes this encoding second-class is that there is no expression that represents a type class instance. This prevents this representation of type classes from multiple instances of a type class at a single type. While the assumption of non-overlapping instances is often true (and enforced in Haskell) it is not guaranteed true in Gallina[3]. Accommodating overlapping instances requires a trick common in Haskell programs where we introduce type aliases. However, care must still be taken to keep the aliases straight, especially during reification. In practice, this kind of type alias is generally not a problem since we are often reasoning opaquely about the type class instances.

7.2.2 First-class Type Classes

To address the problem of overlapping instances, the first class approach splits instances from their uses by introducing a new type constructor to the type algebra and a new symbol for each instance. Figure 7.3 shows the core of the solution. Here, the first argument to all of the type class functions is the appropriate type class instance. This solution requires no additional error propagation or fancy dependent types because MirrorCore handles all of it through its denotation function.

There are two downsides to the first-class representation. First it is slightly larger, which affects both the proof term and the amount of computation needed to compare terms. Second, the algebra of types is more difficult to extend with polymorphic symbols than the algebra of terms. A first-class treatment of polymorphism

[3]Coq provides canonical structures [82] as a more "principled" type class-like mechanism that prevents overlapping instances; however, this property is not codified in the type system.

```
Inductive typ :=
| tyArr : typ → typ → typ
| tyILogic : typ → typ
| ...

Inductive ilogic_sym :=
| ilsTrue : typ → ilogic_sym
| ilsAnd : typ → ilogic_sym
| ilsFun : typ → typ → ilogic_sym.

Definition typeof_ilogic_sym (s:ilogic_sym) : option typ :=
  match s with
  | ilsTrue τ ⇒ Some (tyILogic τ ⇛ τ)
  | ilsAnd τ ⇒ Some (tyILogic τ ⇛ τ ⇛ τ ⇛ τ)
  | ilsFun d c ⇒ Some (tyILogic c ⇛ tyILogic (d ⇛ c))
  end.
```

Figure 7.3: Encoding first-class type classes in MirrorCore by introducing a parameterized type for the class and symbols for the instances.

and type functions in MirrorCore would likely make this definition much easier to use in practice.

7.3 Case Study: Verifying Imperative Programs

Combining the logic automation with $\mathcal{R}_{tac}$ makes it fairly easy to build a fully reflective program verifier for a simple language. I start with a brief overview of a core imperative language and build a simple, non-reflective verification procedure on top of its axiomatic semantics. Next, I show how this simple procedure can be iteratively translated, in a mostly straightforward way, to an efficient reflective procedure using $\mathcal{R}_{tac}$ (Section 7.3.1). I highlight how $\mathcal{R}_{tac}$ retains the ease of extension that is inherent in $\mathcal{L}_{tac}$ by showing how to extend the procedure to reason about conditionals and opaque code. Finally, I discuss how more aggressive extensions such as separation logic assertions can be integrated into the system.

$$\begin{array}{rrcl} \text{(commands)} & c & ::= & x := e \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c \\ \text{(arith expressions)} & e & ::= & x \mid n \mid e_1 + e_2 \end{array}$$

Core Commands

$$\frac{}{\{P\}\mathsf{x} := e\{\exists X, P[\mathsf{x} \rightsquigarrow X] \wedge \mathsf{x} = e[\mathsf{x} \rightsquigarrow X]\}}\ \text{C-Assign}$$

$$\frac{}{\{P\}\mathtt{skip}\{P\}}\ \text{C-Skip} \qquad \frac{\{P\}c_1\{R\} \qquad \{R\}c_2\{Q\}}{\{P\}c_1; c_2\{Q\}}\ \text{C-Seq}$$

Simple Extensions

$$\frac{\{P \wedge \lfloor e \neq 0 \rfloor\}c_1\{Q\} \qquad \{P \wedge \lfloor e = 0 \rfloor\}c_2\{Q\}}{\{P\}\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\{Q\}}\ \text{C-If}$$

$$\frac{P \vdash I \qquad \{I \wedge \lfloor e \neq 0 \rfloor\}c\{I\}}{\{P\}\mathtt{while}_I\, e\, c\{I \wedge \lfloor e = 0 \rfloor\}}\ \text{C-WhileI}$$

$$\frac{P \vdash Q}{\{P\}\mathtt{assert}\ Q\{Q\}}\ \text{C-Assert}$$

Figure 7.4: Axiomatic semantics for a simple imperative programming language with mutable local variables.

The Language The core of the language is a minimal imperative programming language with integer values, mutable local variables, and sequential composition. The first part of Figure 7.4 summarizes the language's axiomatic semantics. By convention I will use upper-case variables in math font to represent logical variables, e.g. variables introduced by an existential quantifier, and lower-case sans-serif names, e.g. x or y, for program variables. In the formal presentation the logical connectives and applicative functor operations thread the store around. These details can be hidden by a combination of Coq notation and coercions to make the textual presentation quite readable.

We compute the post-condition by methodically applying the axiomatic seman-

tics to the program. The $\mathcal{L}_{tac}$ code that implements this "algorithm" is trivial to write.

```
repeat first [ apply triple_exL ; apply ILogic.l∀R ; intro
             | apply CAssign_seq
             | apply CSkip_seq
             | apply CAssign_tail
             | apply CSkip_tail ] ; (* solve postcondition *)
```

The first line introduces existentials in the pre-condition into the Coq context to avoid them piling up in the pre-condition. The derived lemmas, e.g. `CAssign_seq`, are simply for convenience to avoid needing to apply the rule of consequence.

7.3.1 Reflecting Verification

Unfortunately, the simple $\mathcal{L}_{tac}$ implementation of post-condition-based verification has poor performance even for relatively small programs. The blue line in Figure 7.5 shows the amount of time it takes to verify a program that increments n variables for n ranging from 3 to 26. For example, the 2 point (if it existed) would be verifying the following program.

$$\{\mathsf{a} = 0 \wedge \mathsf{b} = 1\}\mathsf{a} := \mathsf{a} + 1; \mathsf{b} := \mathsf{b} + 1\{\mathsf{a} = 1 \wedge \mathsf{b} = 2\}$$

To get better performance we can start incrementally translating the $\mathcal{L}_{tac}$ code into $\mathcal{R}_{tac}$. Converting the post-condition calculation portion of the verification results in the code in Figure 7.6. Both the syntax definition (which is just types and symbols layered on top of MirrorCore) and its reification procedure are omitted. At the beginning of the code each lemma is translated into MirrorCore using the reification plugin with a bit of post-processing. These theorems are then combined by the reflective postcondition tactic to produce the heart of the algorithm. The looping portion of the algorithm is accomplished using $\mathcal{R}_{tac}$'s `REC` combinator for building recursive tactics. Notice that a simplification step is run before each recursive call to perform some reasoning using the McCarthy memory ax-

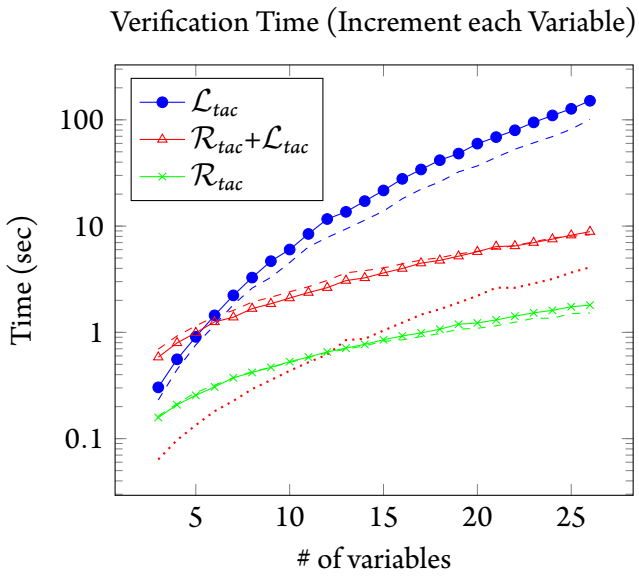

Figure 7.5: Performance for each version of the program verifier. The dotted red line is the amount of time the hybrid approach spends in $\mathcal{L}_{tac}$.

ioms which essentially perform the substitution. Finally, the reflective procedure is proved sound using the generic `derive_rtac_soundness` proof combined with the lemmas that the tactic uses.

The red lines in Figure 7.5 show the effect of translating this portion of the algorithm into $\mathcal{R}_{tac}$. Note that the slope of the curve has dropped substantially, corresponding to an exponential speedup in the implementation. The dotted red line shows the amount of time that we still spend in $\mathcal{L}_{tac}$ processing, which is still substantial, suggesting that we can get more benefit from making more of the verification procedure reflective.

The last part of the problem is to discharge the final entailment. If we were working directly in `Prop` then this task would be easy using Coq's built-in tactics. Since we are working in an embedded logic, however, $\mathcal{L}_{tac}$ tactics do not work immediately for us. Tactics like `intro` need to be translated to applications of Charge! lemmas. This algorithm is quite intricate, but it is just $\mathcal{L}_{tac}$, and therefore it can (almost) be translated to $\mathcal{R}_{tac}$. The only hiccup is that $\mathcal{R}_{tac}$ does not come packaged

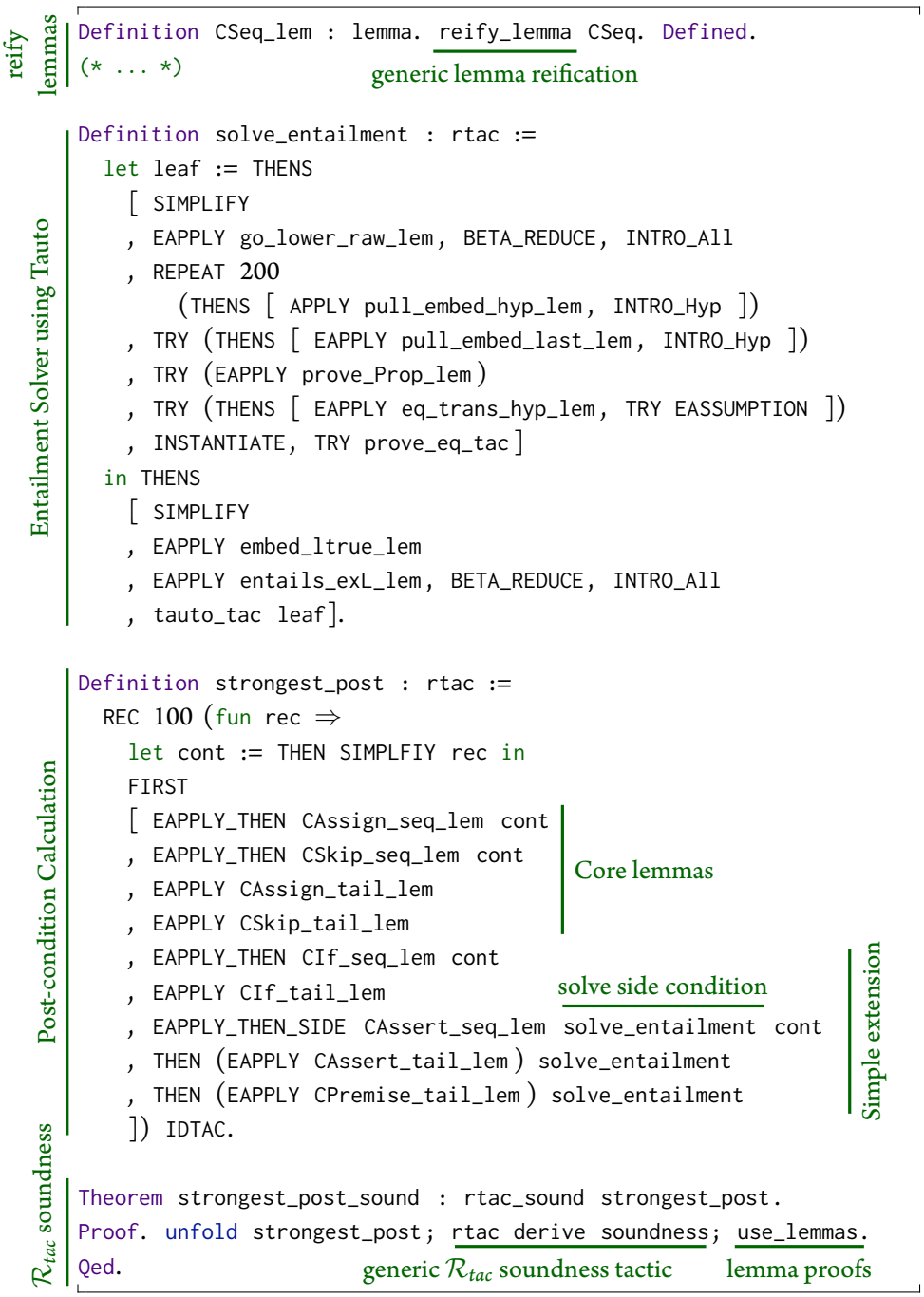

Figure 7.6: $\mathcal{R}_{tac}$ implementations of strongest post-condition calculation for the core imperative language.

with a pre-built substitution tactic. In the spirit of rapid prototyping I specialized a lemma to perform exactly the type of reasoning that we need. Using the lemma (and a little bit of reduction that we have to code reflectively) solves the remainder of the verification condition and gives us our final speedup (Figure 7.5).

From $\mathcal{L}_{tac}$ prototype to naïve $\mathcal{R}_{tac}$ implementation we have achieved roughly a 100x speedup. Further, this translation relied on a very minimal set of tactics, essentially only `EAPPLY`, `INTRO`, and reduction. It did use custom procedures for doing some rewriting by equalities, but procedures like those tend to be extremely easy to verify. Further, we could imagine using Charge!-specific automation to handle more of the details of Charge!'s embedded logics.

7.3.2 Simple Extensions: Conditionals & Quantified Code

Despite the 100x speedup, this reflective solution is not optimal. However, we arrived at this solution with a minimal amount of effort over the $\mathcal{L}_{tac}$-based verification. In addition to being easy to write (and prove), augmenting the $\mathcal{R}_{tac}$-based solution to handle more features is similarly easy to do. For example, we can easily support conditionals and assertions by adding them to the list of lemmas to use when computing the post-condition (see Figure 7.6). The four-line change to the tactic and the slight tweak to the proof script is all we need.

We can also use the specification logic to support quantified code fragments similar to the support that Bedrock has for code-pointers. To do this, we need to augment our post-condition calculus with a specification logic. In most Coq developments we would take Gallina as the specification logic and re-use Coq's context. However, if we wish to add support for general recursive functions we will need a step-indexed specification logic. Making the specification logic explicit changes the post-condition calculation to have the following form:

$$G \vdash \{P\}c\{Q\}$$

Updating the post-condition calculator to handle these rules is trivial; we just need to add the specification logic entailment to the front of the rule. For example, the

updated rule for sequencing is the following:

$$\frac{G \vdash \{P\}c_1\{R\} \qquad G \vdash \{R\}c_2\{Q\}}{G \vdash \{P\}c_1; c_2\{Q\}}\ \text{C-Seq}$$

Using a program specification on the left of the turnstile comes from the laws about intuitionistic logics. However, it is useful to phrase a rule and add it to the post-condition calculation. A simple rule would be the following:

$$\frac{G \vdash \lfloor P \vdash P' \rfloor \qquad \{P'\}c\{Q'\} \vdash \{Q'\}c'\{Q\}}{\{P'\}c\{Q'\} \vdash \{P\}c; c'\{Q\}}\ \text{C-Premise}$$

The use of $\lfloor - \rfloor$ embeds the assertion logic entailment (between P and P') into the specification logic, which allows specification logic facts to be used to prove the assertion.

7.3.3 Adding Memory and Pointers

At the offset, enriching the language with memory and pointers seems pretty simple. Once again we have to enrich the program logic, this time the assertion logic, to include heaps, but all of the previous rules continue to work. Unlike the store, it makes sense to specify the heap structure using Charge!'s separation logic type class.

Adding the Hoare triples for read and write is also completely straightforward.

$$\frac{P \vdash e \mapsto val * True}{\{P\}\mathsf{x} \leftarrow [e]\{\exists X, P[\mathsf{x} \rightsquigarrow X] \wedge \lfloor \mathsf{x} = val[\mathsf{x} \rightsquigarrow X] \rfloor\}}\ \text{C-Read}$$

$$\frac{P \vdash \exists z, e_1 \mapsto z * Q}{\{P\}[e_1] \leftarrow e_2\{e_1 \mapsto e_2 * Q\}}\ \text{C-Write}$$

What is not so easy is automating the side conditions that arise during these rules. In the classic style, these entailments would be solved by an entailment checker like the one coded for Bedrock. It takes a little bit of pre-processing to handle the rich entailments that come from the several layers of logics. Even after this, MirrorShard's separation logic solver has had difficulty solving the premise of the

write rule since it involves two different quantifiers, the implicit universal quantifier over Q and the explicit existential quantifier over z. The algorithm seems to need a heuristic for a very particular form of higher-order unification. An alternative approach would be to use Bedrock-style memory evaluators as custom entailment checkers.

7.4 Future Avenues and Ongoing Applications

Experiences developing reflective automation for the simple language were positive overall. For the relatively minimal work it took to translate the tactic, we improved performance of the verification by more than 100 fold on large problem instances. In addition, unlike most reflective procedures, the reflective procedure that we wrote was generic enough to support after-the-fact extension with new language constructs (or macros) as well as reasoning about higher-order code.

Even further extension such as separation logic, however, requires a bit more work. It took only a few hours to perform post-condition computation for a limited set of read instructions, but the current limitations of the reflective entailment checker make it insufficient to handle the premise of the C-Write rule. It is feasible to code Bedrock's memory evaluators as custom procedures in MirrorCore, and doing so might take some of the pressure off of the entailment checker, though an alternative would be to improve the entailment checker enough to solve these enriched entailments. $\mathcal{R}_{tac}$ does not include all of the bells and whistles that $\mathcal{L}_{tac}$ features, which prevents it from achieving a direct transformation when more complex $\mathcal{L}_{tac}$ tactics such as `subst` are used. Despite this, using only `EAPPLY` and the tactic combinators can get the user quite far.

Beyond more combinators it would be interesting to consider generalizing $\mathcal{R}_{tac}$ to support arbitrary Charge! logics. The implementation of $\mathcal{R}_{tac}$ makes it fairly easy to achieve basic functionality by replacing `Prop` with any `ILogic`. In its current incarnation however, $\mathcal{R}_{tac}$ cannot support embedded logics because the definition of the soundness of an $\mathcal{R}_{tac}$ tactic is indexed by the implementation of the logic, in this case `Prop`. Supporting embedded assertions would require this quantifica-

tion to be local, essentially requiring the proofs of procedures to hold for any intuitionistic logic. This universal quantifier complicates semantic reasoning about the particular logic that a tactic is working within, especially since $\mathcal{R}_{tac}$ does not support communicating this information between tactics.

Another, related feature, that would have been useful in the development of these tactics is the ability to maintain external information, e.g. a data structure tracking equalities and inequalities for a congruence closure algorithm. Bedrock's pure provers make good use of this type of information, but $\mathcal{R}_{tac}$'s maintenance of multiple goals makes it somewhat difficult to ensure that locally justified information does not escape.

7.4.1 Ongoing Work

Much of the work in this chapter arose from building automation for Charge!'s Java formalism [29]. Java has a variety of complexities that do not occur in Imp, for example pointers and functions. Like Imp's symbolic execution, symbolic execution in Java is built directly using $\mathcal{R}_{tac}$.

Beyond Java, Kennedy and Benton [30] have developed a program logic for assembly using separation logic and a previous version of Charge!. Extending the work in this chapter to support these logics is likely to provide good automation for the system.

Appel and collaborators have also begun applying MirrorCore to the Verified Software Toolchain project [17]. VST's logical rules are significantly more complex than Java's because they work on a typed language rather than working on an untyped core. By reasoning on a typed language, however, the verification results can be connected to Leroy's CompCert [105] compiler, which can guarantee that high-level properties are maintained all the way down to the machine code.

8

Conclusions

Many authors [34, 46, 107] have shown computational reflection to be an efficient way to build both large- and small-scale automation. This automation enables users to reason more effectively and efficiently at higher levels of abstraction, which are chosen by them rather than fixed by the proof assistant. This dissertation focused on the following thesis:

> **Thesis** Open computational reflection in intensional type theories can lower the cost of writing trustworthy, scalable, and customizable automation.

In the past five chapters I justified this thesis through the development of two frameworks, MirrorCore (Chapter 3) and MirrorShard (Chapter 5), that encapsulate techniques for building compositional reflective procedures. These frameworks focus on the problem of composition by building extensible represen-

tations and phrasing the soundness of procedures in extensible ways. By supporting extension, these frameworks take care of much of the repetitive work of building reflective syntax and providing a denotation function. In addition to the syntax, these frameworks provide useful support for term-manipulation that would otherwise have to be coded for each new problem. Procedures such as lifting and lowering are essential building blocks and are necessary when reasoning about binders. Higher-level procedures such as unification and β-reduction are useful when building higher-level tasks such as applying generic lemmas.

The core of extensibility in these frameworks comes from leveraging proofs and reasoning about them in deep ways. Definitional environment constraints (Section 3.2.2) sacrifice some compositionality for ease of use and are a good way to achieve one-off customization, for example when reasoning about a particular, isolated abstraction. Propositional environment constraints (Section 3.2.1) are more expressive but can be difficult to work with in axiom-free ways. Finally, truly extensional formulations (Section 3.2.1) provide a way to abstract further, offering the ability to simulate dependency in an otherwise non-dependent representation.

I have also applied my frameworks to a number of case studies, both large and small. The largest leveraged MirrorShard, the first-order predecessor of MirrorCore, to do program verification for the Bedrock structured programming library. That automation has been used to verify thousands of lines of low-level code in a higher-order program logic. I also presented some results applying MirrorCore to the Charge! library for embedded logics. While the application is currently much smaller than the Bedrock automation, $\mathcal{R}_{tac}$ has enabled us to make progress much more rapidly than we were able to in Bedrock. This experience demonstrates $\mathcal{R}_{tac}$'s usefulness for rapid prototyping of reflective automation.

8.1 Avenues for Future Work

This thesis is a strong step in the direction of verification at scale within intensional type theory. Many avenues exist to extend the work described here.

Are there better features for modular computational reflection? Delaware's work on modular meta-theory [71, 72] suggests an encoding of extensible syntax within Coq's type theory, but it is unclear whether this approach readily adapts to effective computational reflection. In particular, the approach uses Church encodings of data, which tend to be both more difficult to reason about and have a higher computational burden than natively defined datatypes. Enriching Coq's type system to support compositional arguments for strict postivity may make this approach feasible. While a step in the right direction, fully leveraging this approach requires better support for reasoning up to isomorphism, since changing the order of composition will result in different, but isomorphic, representations.

MiniAgda [6] already contains support for this type of polymorphism, and translating some of the ideas into Agda could prove beneficial, especially since the state of automation in Agda is not as mature as Coq's still quite *ad hoc* approach.

Can intensional type theory be effectively compiled? Some of the suggestions in Chapter 4 are due specifically to the lack of a good optimizing compiler for Gallina. The absence of side effects provides a number of opportunities for standard compiler techniques. In addition, the existence of proofs makes it possible to perform more domain-specific optimization in a sound way. For example, proving a contextual equality between two code fragments allows the compiler to substitute one for the other perhaps eliminating intermediate data structures or enabling some compile-time reduction.

Other languages, e.g. Idris [44], already have compilers [43] for dependent type theory, but they are not verified, and the need to include such a complicated piece of machinery in the trusted computing base should give pause. One could argue that the computational mechanisms such as Coq's `vm_compute`, which computational reflection leverages heavily, are already too complicated to be completely trustworthy. A verified compiler would open up the possibility of a formally verified kernel within Coq itself, providing a methodology to justify that the core is faithfully implementing the type theory, in a similar manner to the achievements of Davis with the Milawa theorem prover [66].

Related to compilation is the technique of heterogeneous refinement [60]. This technique enables the semi-automatic construction of refinement proofs between two implementations of a data type. Current examples focus on erasing proof terms, but this technique could also be applied to refining the implementation of abstractions such as monads to use the imperative state of the underlying machine. Performing this kind of reasoning would require even better ways to reason equationally about code, and computational reflection may be a key enabling technology to scale this up to large developments.

Can we build reflection into the meta-theory of our logic? In Chapter 4, I touched on the reification problem which converts semantic terms into syntactic values. Given the similarity between MirrorCore's representation and Coq's own, it would be interesting to explore running reflective procedures directly on Coq's internal term representation. With appropriate support, this would provide a way to do incremental reification, only reifying as much structure as the reflective procedure needs. From a proof-theory point of view, the minimally useful representation corresponds to the most general proof.

Even from a completely practical point of view ignoring the proof objects entirely, this is interesting because the size of the term affects how much computation is needed. For example, to determining whether two syntactic objects are equal requires a linear scan over the two objects. Thus, if the two objects are simply environment references to the same location, the comparison will return true immediately. On the other hand, if the terms are actually large (but equal) syntactic terms, the cost of comparing them can be substantially higher.

Making this notion of incremental observation fit nicely into a computational model is not immediately obvious. What is needed is a notion of continuity for functions that forbids them from returning different answers if given more observations. Some operations, such as equality and less than, respect this property, but it is not immediately clear how this generalizes to other functions. Higher-inductive types [148] could be a potential solution to this problem since they can be used to enforce a weaker notion of equivalence on data.

Can we effectively build automation on top of a richer representation? I discussed a variety of attempts to achieve self-representation within intensional type theory [53, 65, 115]. These works focus on self-representation from a predominantly theoretical point of view. In this dissertation, I focused on the practical aspects of using self-representation to build automation, but I applied my techniques to a language with only simple types. While parameterizing the representation allowed us to achieve second-class polymorphism and type functors, making this support first-class would simplify using the system in two ways. First, it would make the syntax more closely mirror the native syntax that it represents thus lowering the cognitive overhead of a new representation. And second, it would enable reasoning about problems that use more features of the logic in a natural way, e.g. verifying polymorphic programs.

8.2 Final Thoughts

Proof assistants provide a rigorous framework for integrating automation and human insight to solve problems that neither can achieve alone. To be truly useful, automation must be customizable, to enable us to easily write more, compositional to allow us to combine it to solve more complex problems, and efficient to enable it to scale. My techniques enable all of these features together and are a step towards understanding how to develop truly programmable and foundational proof assistants for intensional type theory. This ability to rapidly build trustworthy, domain-specific automation has the potential to fundamentally change the manner and rate of invention and discovery in the years to come.

References

[1] ACL2 Version 6.5. 2014. URL http://www.cs.utexas.edu/users/moore/acl2/.

[2] java.com: Java + You, 2014. URL http://www.java.com/en/.

[3] Coq ExtLib. 2014. URL https://github.com/coq-ext-lib/coq-ext-lib.

[4] .NET Downloads, Developer Resources & Case Studies | Microsoft .NET Framework, 2014. URL http://www.microsoft.com/net.

[5] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[6] Andreas Abel. MiniAgda: Integrating sized and dependent types. *arXiv preprint arXiv:1012.4896*, 2010.

[7] Andreas Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 57–71. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19804-5. doi: 10.1007/978-3-642-19805-2_5. URL http://dx.doi.org/10.1007/978-3-642-19805-2_5.

[8] Agda Development Team. The Agda proof assistant reference manual, version 2.4.2. 2014.

[9] Stuart F Allen, Mark Bickford, Robert L Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[10] Thorsten Altenkirch and Conor McBride. Towards observational type theory. *Manuscript, available online*, 2006.

[11] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 70–84. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22943-5. doi: 10.1007/978-3-642-22944-2_6. URL http://dx.doi.org/10.1007/978-3-642-22944-2_6.

[12] Abhishek Anand, Mark Bickford, Robert L. Constable, and Vincent Rahli. A Type Theory with Partial Equivalence Relations as Types. 2014.

[13] Andrew W. Appel. Tactics for separation logic, 2006. Draft of January 2006.

[14] Andrew W. Appel. Verified software toolchain. In *Proc. ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer-Verlag, 2011.

[15] Andrew W. Appel. Verismall: Verified Smallfoot shape analysis. In *Proc. CPP*, 2011.

[16] Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007.

[17] A.W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, X. Leroy, S. Blazy, and G. Stewart. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN 9781107048010. URL http://books.google.com/books?id=ABkmAwAAQBAJ.

[18] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25378-2. doi: 10.1007/978-3-642-25379-9_12. URL http://dx.doi.org/10.1007/978-3-642-25379-9_12.

[19] Franz Baader and Klaus U Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation*, 21(2):211–243, 1996.

[20] David F Bacon, Perry Cheng, and VT Rajan. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices*, 38 (1):285–298, 2003.

[21] Bruno Baras. The dark side of $\mathcal{L}_{tac}$. June 2009.

[22] Mike Barnett, Bor-Yuh Chang, Robert DeLine, Bart Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-36749-9. doi: 10.1007/11804192_17. URL http://dx.doi.org/10.1007/11804192_17.

[23] Bruno Barras and Benjamin Werner. Coq in coq. Technical report, 1997.

[24] Clark Barrett and Cesare Tinelli. CVC3. In *Proc. CAV*, volume 4590 of *LNCS*, pages 298–302. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_34. URL http://dx.doi.org/10.1007/978-3-540-73368-3_34.

[25] Eli Barzilay. *Implementing Reflection in Nuprl*. PhD thesis, Cornell University, Ithaca, NY, USA, 2005. AAI3195788.

[26] Eli Barzilay, Stuart Allen, and Robert Constable. Practical reflection in nuprl. In *Short paper presented at 18th Annual IEEE Symposium on Logic in Computer Science, June*, pages 22–25, 2003.

[27] Andrej Bauer. Andromeda, 2014. URL https://github.com/andrejbauer/andromeda.

[28] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 22–38. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22862-9. doi: 10.1007/978-3-642-22863-6_5. URL http://dx.doi.org/10.1007/978-3-642-22863-6_5.

[29] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *Interactive Theorem Proving*, pages 315–331, 2012. ISBN 978-3-642-32346-1. doi:

10.1007/978-3-642-32347-8_21. URL http://dx.doi.org/10.1007/978-3-642-32347-8_21.

[30] Nick Benton, Jonas B. Jensen, and Andrew Kennedy. High-level separation logic for low-level code. In *Proc. POPL*, pages 301–314. ACM, 2013.

[31] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. FMCO*, FMCO'05, pages 115–137, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36749-7, 978-3-540-36749-9. doi: 10.1007/11804192_6. URL http://dx.doi.org/10.1007/11804192_6.

[32] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In *Proc. CAV*, 2011.

[33] Yves Bertot and Pierre Casteran. *Coq'Art: The Calculus of Inductive Constructions*. 2004. ISBN 3-540-20854-2.

[34] Frederic Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_4. URL http://dx.doi.org/10.1007/978-3-540-74464-1_4.

[35] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In *Certified Programs and Proofs*, pages 151–166. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25378-2. doi: 10.1007/978-3-642-25379-9_13. URL http://dx.doi.org/10.1007/978-3-642-25379-9_13.

[36] Francois Bobot, Jean-Christophe Filliatre, Claude Marche, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *BOOGIE'11*, 2011.

[37] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. *Lecture notes in computer science*, 2011. doi: 10.1007/978-3-642-25379-9_26. URL https://hal.inria.fr/hal-00650940.

[38] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM. ISBN

1-58113-830-X. doi: 10.1145/1040305.1040327. URL http://doi.acm.org/10.1145/1040305.1040327.

[39] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–125. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42525-0. doi: 10.1007/3-540-44755-5_10. URL http://dx.doi.org/10.1007/3-540-44755-5_10.

[40] Bob Boyer. Rewrite rule compilation. Technical Report AI-194-86-P, Micro-electronics and Computer Technology Corporation (MCC), June 1986.

[41] John Boyland. Checking interference with fractional permissions. In *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 1075–1075. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40325-8. doi: 10.1007/3-540-44898-5_4. URL http://dx.doi.org/10.1007/3-540-44898-5_4.

[42] John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32:22:1–22:33, August 2010. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1749608.1749611. URL http://doi.acm.org/10.1145/1749608.1749611.

[43] Edwin Brady. Epic—a library for generating compilers. In Ricardo Peña and Rex Page, editors, *Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32036-1. doi: 10.1007/978-3-642-32037-8_3. URL http://dx.doi.org/10.1007/978-3-642-32037-8_3.

[44] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X. URL http://journals.cambridge.org/article_S095679681300018X.

[45] Thomas Braibant. evm_compute. 2013. URL https://github.com/braibant/evm_compute.

[46] Thomas Braibant and Damien Pous. Tactics for Reasoning Modulo AC in Coq. In *Certified Proofs and Programs*, pages 167–182, 2011.

[47] Matt Brown and Jens Palsberg. Self-Representation in Girard's System U. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676988. URL http://doi.acm.org/10.1145/2676726.2676988.

[48] N. G. De Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, 1991.

[49] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proc. POPL*, pages 289–300. ACM, 2009.

[50] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 33–45, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535883. URL http://doi.acm.org/10.1145/2535838.2535883.

[51] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 45–58. ACM, 2009.

[52] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proc. POPL*, pages 247–260. ACM, 2008.

[53] James Chapman. Type theory should eat itself. *Electronic Notes Theoretical Computer Science*, 228:21–36, January 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.12.114. URL http://dx.doi.org/10.1016/j.entcs.2008.12.114.

[54] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993526. URL http://doi.acm.org/10.1145/1993498.1993526.

[55] Adam Chlipala. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500592. URL http://doi.acm.org/10.1145/2500365.2500592.

[56] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, December 2013. ISBN 9780262026659.

[57] Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 609–622, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677003. URL http://doi.acm.org/10.1145/2676726.2677003.

[58] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-order Imperative Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596565. URL http://doi.acm.org/10.1145/1596550.1596565.

[59] Guillaume Claret, Lourdes Del Carmen Gonzalez Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *Interactive Theorem Proving*, Rennes, France, July 2013. URL http://hal.inria.fr/hal-00870110.

[60] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Proofs and Progams*, 2013. URL http://www.maximedenes.fr/download/refinements.pdf.

[61] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proc. TPHOLs*, 2009.

[62] Coq Development Team. The Coq proof assistant reference manual, version 8.4. 2012. URL http://coq.inria.fr/distrib/V8.4/refman/.

[63] Thierry Coquand, Jean Gallier, and Le Chesnay Cedex. A proof of strong normalization for the theory of constructions using a kripke-like interpretation. In *In Workshop on Logical Frameworks–Preliminary Proceedings*, 1990.

[64] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001. ISBN 9780262032933. URL https://books.google.com/books?id=NLngYyWFl_YC.

[65] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_7. URL http://dx.doi.org/10.1007/978-3-540-74464-1_7.

[66] Jared Curran Davis. *A self-verifying theorem prover*. PhD thesis, University of Texas at Austin, December 2009.

[67] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[68] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[69] David Delahaye. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning*, pages 85–95. Springer, 2000.

[70] Benjamin Delaware. *Feature Modularity in Mechanized Reasoning*. PhD thesis, The University of Texas at Austin, December 2013.

[71] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429094. URL http://doi.acm.org/10.1145/2429069.2429094.

[72] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500587. URL http://doi.acm.org/10.1145/2500365.2500587.

[73] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 689–700, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677006. URL http://doi.acm.org/10.1145/2676726.2677006.

[74] Dominique Devriese and Frank Piessens. Typed syntactic meta-programming. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 73–86, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500575. URL http://doi.acm.org/10.1145/2500365.2500575.

[75] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*, pages 161–177. Springer ENTCS, 2009. URL http://msl.cs.princeton.edu/fresh-sa.pdf.

[76] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver. Technical report, SRI International, 2006.

[77] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:2000, 1998.

[78] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. QuickChick: Property-Based Testing for Coq. July 2014.

[79] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX Association, 2006.

[80] François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74590-7. doi: 10.1007/978-3-540-74591-4_27. URL http://dx.doi.org/10.1007/978-3-540-74591-4_27.

[81] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008. URL http://hal.inria.fr/inria-00258384.

[82] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 163–175, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034798. URL http://doi.acm.org/10.1145/2034773.2034798.

[83] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, 2013. Springer. doi: 10.1007/978-3-642-39634-2_14. URL http://hal.inria.fr/hal-00816699.

[84] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 235–246, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581501. URL http://doi.acm.org/10.1145/581478.581501.

[85] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676975. URL http://doi.acm.org/10.1145/2676726.2676975.

[86] R. Harper. *Practical Foundations for Programming Languages*. Practical Foundations for Programming Languages. Cambridge University Press, 2012. ISBN 9781107029576. URL https://books.google.com/books?id=YhZ2yMHwLm0C.

[87] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity - a reynolds programme for category theory and programming languages. *Electronic Notes Theoretical Computer Science*, 303: 149–180, March 2014. ISSN 1571-0661. doi: 10.1016/j.entcs.2014.02.008. URL http://dx.doi.org/10.1016/j.entcs.2014.02.008.

[88] HOL4 Development Team. HOL4. 2014. URL http://hol.sourceforge.net/.

[89] Zhé Hóu, Ranald Clouston, Rajeev Goré, and Alwen Tiu. Proof search for propositional abstract separation logics via labelled sequents. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 465–476, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535864. URL http://doi.acm.org/10.1145/2535838.2535864.

[90] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

[91] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11 (1):31–55, 1978. ISSN 0001-5903. doi: 10.1007/BF00264598. URL http://dx.doi.org/10.1007/BF00264598.

[92] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20397-8. URL http://dl.acm.org/citation.cfm?id=1986308.1986314.

[93] Mark Janeba. The Pentium Problem. 2011. URL http://www.willamette.edu/~mjaneba/pentprob.html.

[94] Wojciech Jedynak. Operational Semantics of Ltac. Master's thesis, Uniwersytet Wrocławski, the Netherlands, 2013.

[95] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 377–396. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_19. URL http://dx.doi.org/10.1007/978-3-642-28869-2_19.

[96] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676966. URL http://doi.acm.org/10.1145/2676726.2676966.

[97] Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. Coq: The world's best macro assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2154-9. doi: 10.1145/2505879.2505897. URL http://doi.acm.org/10.1145/2505879.2505897.

[98] 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*, pages 207–220. ACM, 2009.

[99] Pepijn Kokke and Wouter Swierstra. Auto in Agda. Under submission. URL https://github.com/pepijnkokke/AutoInAgda.

[100] Soonho Kong and Leonardo de Moura. L∃∀N Theorem Prover. URL http://leanprover.net/.

[101] Neel R Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 63–76. ACM, 2010.

[102] Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 477–490, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.

1145/2535838.2535871. URL http://doi.acm.org/10.1145/2535838.2535871.

[103] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 806–809. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_51. URL http://dx.doi.org/10.1007/978-3-642-05089-3_51.

[104] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL http://dl.acm.org/citation.cfm?id=1939141.1939161.

[105] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf.

[106] S. Lescuyer and S. Conchon. Improving Coq propositional reasoning using a lazy CNF conversion scheme. In *Proc. FroCos*, 2009.

[107] Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, January 2011.

[108] Gregory Malecha. MirrorCore. The MirrorCore repository. URL https://github.com/gmalecha/mirror-core.

[109] Gregory Malecha. TemplateCoq. 2014. URL https://github.com/gmalecha/template-coq.

[110] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/1706299.1706329. URL http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/1706299.1706329.

[111] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *Interactive Theorem Proving*, 2014.

[112] Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008.

[113] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13:1061–1075, 11 2003. ISSN 1469-7653. doi: 10.1017/S0956796803004957. URL http://journals.cambridge.org/article_S0956796803004957.

[114] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170. Springer, 2005.

[115] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863497. URL http://doi.acm.org/10.1145/1863495.1863497.

[116] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008. ISSN 0956-7968. doi: 10.1017/S0956796807006326. URL http://dx.doi.org/10.1017/S0956796807006326.

[117] Andrew McCreight. Practical tactics for separation logic. In *Proc. TPHOLs*, 2009.

[118] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *Proc. PDPAR*, 2005.

[119] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992. ISSN 0747-7171. doi: 10.1016/0747-7171(92)90011-R. URL http://dx.doi.org/10.1016/0747-7171(92)90011-R.

[120] Magnus O Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving*, pages 265–280. Springer, 2011.

[121] Magnus O Myreen and Jared Davis. The reflective Milawa theorem prover is sound. 2012.

[122] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proc. POPL*, 2010.

[123] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[124] Duckki Oe and Adam Chlipala. A verified garbage colletor for bedrock. 2013.

[125] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. Versat: A Verified Modern SAT Solver. In *Proc. VMCAI*, 2012.

[126] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL http://www.csl.sri.com/papers/cade92-pvs/.

[127] Matthew Parkinson. The next 700 separation logics. In *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-15056-2. URL http://dx.doi.org/10.1007/978-3-642-15057-9_12.

[128] Frank Pfenning. Elf: A meta-language for deductive systems. In *Automated Deduction—CADE-12*, pages 811–815. Springer, 1994.

[129] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16*, pages 202–206. Springer, 1999.

[130] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 9780262162098. URL http://books.google.com/books?id=ti6zoAC9Ph8C.

[131] B.C. Pierce. *Advanced topics in types and programming languages*. MIT Press, pub-MIT:adr, 2005. ISBN 0-262-16228-8.

[132] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100.

[133] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014. URL http://www.cis.upenn.edu/~bcpierce/sf.

[134] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55 – 74, 2002. doi: 10.1109/LICS.2002.1029817.

[135] Mike Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow), March 2014. URL http://homotopytypetheory.org/2014/03/03/hott-should-eat-itself/.

[136] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. *Submitted for publication*, 2014.

[137] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-71065-3. doi: 10.1007/978-3-540-71067-7_23. URL http://dx.doi.org/10.1007/978-3-540-71067-7_23.

[138] Bas Spitters and Eelis Van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(04):795–825, 2011.

[139] Antonios M Stampoulis. *VeriML: A Dependently-Typed, User-Extensible and Language-Centric Approach to Proof Assistants*. PhD thesis, Yale University, New Haven, CT, 2013.

[140] Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In *Proc. ICFP*, pages 333–344. ACM, 2010.

[141] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 231–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155373. URL http://doi.acm.org/10.1145/154766.155373.

[142] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *Proc. ICFP*, 2012.

[143] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011*, pages 266–278, September 2011.

[144] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18:423–436, 7 2008. ISSN 1469-7653. doi: 10.1017/S0956796808006758. URL http://journals.cambridge.org/article_S0956796808006758.

[145] The Haskell Development Team. Haskell – haskell wiki. 2014. URL http://www.haskell.org.

[146] The OCaml Development Team. Ocaml. 2014. URL http://ocaml.org/.

[147] T. Tuerk. A formalisation of Smallfoot in HOL. In *Proc. TPHOLs*, 2009.

[148] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[149] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4. doi: 10.1007/978-3-642-41582-1_10. URL http://dx.doi.org/10.1007/978-3-642-41582-1_10.

[150] Vladimir Voevodsky. A simple type system with two identity types. Feb 2013. URL https://uf-ias-2012.wikispaces.com/file/view/HTS.pdf/410120566/HTS.pdf.

[151] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 675–690, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660201. URL http://doi.acm.org/10.1145/2660193.2660201.

[152] Benjamin Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997.

[153] Hongwei Xi. Applied type system. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22164-7. doi: 10.1007/978-3-540-24849-1_25. URL http://dx.doi.org/10.1007/978-3-540-24849-1_25.

[154] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227. ACM, 1999.

[155] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500579. URL http://doi.acm.org/10.1145/2500365.2500579.

Colophon

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. A template, which can be used to format a PhD thesis with this look and feel, has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.