Using Reflective Separation-Entailment Solvers for Reasoning Formally About C: Integrating the Verified Software Toolchain with the MIRRORSHARD Solver

Mario M. Alvarez

May 2014

In partial fulfillment of the requirements for a Bachelor of Arts in Computer Science.

I pledge that this document represents my own work, in accordance with University regulations.
Abstract

Formal software verification is a growing field of research in computer science. The aim of software verification research is to create technologies for reasoning formally about software, and to use such technologies to prove interesting and useful results. By proving programs correct, with respect to particular specifications of what correctness means for the programs in question, we can derive much stronger guarantees about how the software will behave when run than would be possible through other means, such as testing. In this paper we discuss improvements we have made to the Verified Software Toolchain (VST), a system for formal reasoning about C programs, and we distill lessons from this project that might be more widely applicable to similar projects in the future. In particular, we describe the process of integrating VST with MirrorShard, a reflective solver for expressions in separation logic. Separation logic is used internally by VST to express properties of programs that relate to their usage of heap memory. After this discussion, we reflect on the future work that remains to be done in VST, and on how the work described in this paper can be built upon to further increase VST's power and usability, as well as possibly serving as an example for other projects.
Acknowledgements

This paper would not have been possible without the efforts and aid of many wonderful people.

First, I'd like to thank my parents, for bringing me into this world and for providing me with immeasurable support during my time in it.

Next, I would like to thank my research advisor, Andrew Appel. First, for showing me the light: his course on programming languages was what first sparked my interest in theorem-proving as a method of software development, and my final project in the course, which used VST, helped lead me to this thesis topic. Next, for making this thesis happen: he was willing to advise me on this project, despite its rather ambitious nature and despite having numerous other demands on his time (including a sabbatical and chairing Princeton’s computer science department).

I also owe a debt of gratitude to Josiah Dodds, who has been a combination of advisor, mentor, and senior partner to me in this project. He has been an amazing collaborator and an incredible source of support. Being able to draw on his knowledge of and expertise in Coq was invaluable to me. Despite having a son at home to (literally) babysit, he was always willing to make time to help me in this project, even if that sometimes meant (figuratively) babysitting me as well.

I owe thanks to Gregory Malecha, the creator of and ultimate authority on MirrorShard. He gave a great deal of his time making himself available as a resource on the system, and gave us crucial guidance about its limitations and how we could work around them.

I would like to thank E. Lennart Beringer, for serving as an important resource and a sort of proxy advisor to me in Andrew’s absence, and for being an extremely accessible and helpful resource on Coq. J. Gordon Stewart deserves mention as well, for his help in understanding Navarro Perez’s theory of linked lists and with other aspects of the project.

Princeton’s Programming Languages group as a whole deserves mention, for providing an amazingly welcoming and supportive environment for me, and for making room for me this spring, while I’ve occupied a desk in their office on only a semi-official basis[1].

Finally, I’d like to thank everyone who read drafts of my thesis: Andrew Appel, Josiah Dodds, Gregory Malecha, E. Lennart Beringer, Jamie Titus, and Anna K. Simpson. The feedback I received was incredibly helpful, and has made this thesis significantly better.

[1] It’s not squatting if the department chair says it’s OK, right?
1 Introduction

Formal software verification is an exciting field of computer science which has seen a great deal of progress and innovation in recent years. This paper, describing improvements in an approach to applying these technologies to the verification of real-world imperative programs, represents just a small part of one of verification's many fascinating frontiers. Here we will describe the main thrust of this work, as well as the verification technologies on which it has been built.

Software verification of the kind we are performing aims to prove full functional correctness rather than more lightweight, partial, safety analysis. This is particularly difficult to apply to imperative programs, such as systems code. This code is often in C; therefore, it is useful to have a platform for formally verifying C programs, which can resolve these difficulties, making it easier to prove C code correct. The Verified Software Toolchain (VST) \cite{appel2008verified}, developed largely by Andrew Appel’s group at Princeton, aims to provide such a platform. One crucial part of VST is its proof automation, which enables it to generate parts of the complex proofs required to prove the correctness of C code on the user’s behalf. Good automation is central to the usability of a system like VST; the better the automation, the wider the range of projects for which verification becomes worthwhile.

The \textsc{MirrorShard} system\cite{malecha2017mirrorshard}, built at Harvard by Gregory Malecha, is a solving framework that can more efficiently solve separation logic entailments, expressions in a specialized logic which appear frequently in proofs about C programs in VST. Our aim was to use \textsc{MirrorShard} to improve a piece of VST’s proof-automation system. VST contains a routine for simplifying and solving separation logic entailments, written in the imperative \texttt{Ltac} tactic language. This solver has all the problems common to \texttt{Ltac}-based proof automation, discussed in Section 2.1 and it lacks the ability to solve many entailments that could be discharged automatically with a more powerful solver. In particular, the \texttt{Ltac}-based entailer has only a limited knowledge of data structures such as linked lists, and so is unable to use properties of these data structures that could make many entailments automatically solvable. Though the existing infrastructure could be extended to accommodate such knowledge by writing new tactics for rewriting goals, we aimed to eliminate the need for this by providing a standalone system that could more efficiently solve goals related to facts about linked lists and that does not depend on \texttt{Ltac}.

In augmenting VST’s automation with the \textsc{MirrorShard} solver, we hoped to accomplish the following:

1. Improve the expressive power of the automation; that is, to enable the automation to solve goals that it previously could not. In this paper, we focused on improving its ability to solve goals involving linked lists, but hope that our work can serve as an example for others wishing to improve automation for other data structures (such as, for instance, trees).

2. Improve the speed of automation, taking advantage of the comparatively
high performance of MirrorShard’s approach to solving, reflection (see Section 4.1 for details)

3. Reduce the size of the proofs the automation produces (again, see Section 4.1 for details) \(^2\)

4. Lay the groundwork for reified symbolic execution, which can amplify the benefits of reflective solving by reducing the overhead involved in using a solver like MirrorShard (see Section 5.2.2). A similar approach has been detailed in [5].

Thus, we hoped to use MirrorShard to improve the automation of separation logic entailment solving, relative to the Ltac-based entailment system. We considered both the speed of proof generation and the size of the proofs generated. Additionally, we used MirrorShard’s hints system to provide the solver with more detailed knowledge of linked lists, enabling it to solve entailments relating to linked lists that the previous entailment system could not. These hints, we hope, can serve as an example for others wishing to encode facts about other C data structures as hints for MirrorShard. We used these criteria - speed, memory usage, and generality - to evaluate the success of this project.

1.1 Contributions

The integration of VST with MirrorShard was a collaboration between me and Josiah Dodds, with guidance from Gregory Malecha and Andrew Appel. My primary contributions to the project were the following:

- Creating a system for safely simplifying terms by computation within MirrorShard, as well as a “prover” that enable MirrorShard to take advantage of this infrastructure to use an expanded notion of equality (equality of terms that compute to the same result, not just terms that are syntactically identical) as part of its proving process. This is the Computation Prover, described in 5.1

- Stating a version of Juan Antonio Navarro Perez’s [16] theory of linked list segments for C, as well as creating procedures for reifying the lemmas comprising the theory so that they could be used with MirrorShard (5.2). With access to these lemmas, MirrorShard is able to automatically perform certain kinds of reasoning about linked lists that would otherwise require user intervention.

A more detailed discussion of this work follows (Section 5), but some contextualization is necessary before it can be explained fully; that will be the role of the next few sections.

\(^2\)This might seem like a secondary concern, but it is actually quite important. The current entailment builds proof objects that are in some cases so large that they exceed the amount of memory that some versions of Coq can address, leading to the need for somewhat ugly workarounds that we hope to render unnecessary. This has been the case, for instance, in Andrew Appel’s work on verifying SHA256. [1]
1.2 This Thesis

To summarize, computational reflection can improve proof automation for program verification, in terms of both power and performance. We believe that the potential of computational reflection is exemplified by our project of integrating the Verified Software Toolchain with MirrorShard. In the following sections, we outline the details of this project. We first cover the Coq proof assistant [2], summarizing its workings and its usefulness for verifying programs (and for building platforms for verifying programs, such as VST and MirrorShard). We then discuss the features of VST [3] and MirrorShard [4], in turn. With this background in place, we discuss our integration project in detail [5]. We describe the primary contributions of my research in integrating these two systems, illustrating them using a simple example. Then, we compare our MirrorShard-based reflective solver with VST’s non-reflective entailment solver [6], discussing the improvements and challenges that the new approach brings. We conclude with a discussion of related work [7].

2 The Coq Proof Assistant

The Verified Software Toolchain and MirrorShard, the systems involved in this project, are both built on the Coq proof assistant. Here we will give a brief overview of Coq’s goals, capabilities, and uses.

By a proof assistant, we mean that Coq is a collection of tools for describing and proving mathematical facts. These proofs are checked by a piece of software called the proof checking kernel, a small program that verifies the correctness of proofs. Coq also contains standard libraries useful in a wide range of developments as well as infrastructure for automating the process of building proofs.

Proof-checking in Coq is very similar to type-checking in languages such as ML and Haskell, the primary difference being that, in Coq and proof assistants like it, terms are dependently typed. This means that types can depend on values, and can be computed on in much the same way that “normal” values can. This enables the expression of complex facts that can be checked by the type-checker (proof-checker); proofs can even be encoded in such a system. Essentially, facts to be proved are types, and proofs of that fact are terms (syntactic objects) that typecheck to that type.

2.1 An Example Proof in Coq

For instance, let’s examine a proof of one of De Morgan’s laws. The statement of this theorem (its type) is the following:
\[ \forall a \ b : \text{bool}, \neg b (a \land b) = (\neg a) \lor (\neg b) \]

One proof of this theorem (an object fulfilling that type) is the following:

\[ (\forall a \ b : \text{bool}, \neg b (a \land b) = (\neg a) \lor (\neg b)) \]
fun a b : bool ⇒
if a as b₀ return (negb (b₀ && b) − (negb b₀ || negb b))
  then if b as b₀ return (negb (true && b₀) − (negb true || negb b₀))
    then eq_refl
    else eq_refl
  else if b as b₀ return (negb (false && b₀) − (negb false || negb b₀))
    then eq_refl
    else eq_refl

Figure 1: A proof of the theorem described above.

Without going into detail, it is worth observing that this proof’s syntactic form should not be too unfamiliar to a reader familiar with ML or Haskell. This is a very familiar-looking object; in fact, it is a function! The if statements correspond to a case-analysis on the possible values of a and b.

This approach to theorem proving is particularly amenable to proving facts about software, because it is a natural extension of expressing properties through types. These properties are expressed in Coq’s term language, called Gallina, in which the objects Coq’s kernel can accept are expressed. Gallina is a programming language that is powerful and general enough to represent the facts we want to prove, the objects we want to prove them about, and the proofs themselves. In addition to allowing the user to define data-types representing mathematical objects, Gallina also allows the expression of computational objects, such as programs, their inputs, and their results. Coq also includes Ltac, a so-called tactic language that enables the user to define programs to build proofs in an automated fashion, so that she does not have to do so by hand. As we will see, Ltac has its problems, but it is very convenient. In the example above, the following Ltac script would produce the result listed above:

Theorem demorgan : ∀ a b : bool, negb (andb a b) − orb (negb a) (negb b).
Proof.
  intros.
  destruct a; destruct b; simpl; congruence.
Qed.

While convenient, Ltac has many problems. Primarily, it is untyped, which makes constructing the typed terms that Coq accepts more challenging. It lacks among many others.

4Here, negb is boolean negation, || is boolean OR, and && is boolean AND.
5eq_refl is the proof of facts of the form a = a.
6The fact that the proof is (in a sense) actually a function is perhaps surprising. It is due to the Curry-Howard correspondence, an important duality between proofs and programs with deep implications for theorem-proving of the kind Coq is used for.
7For example, there is special syntactic sugar for writing recursive functions: the Fixpoint command in the Coq vernacular (the “outer” language interpreted by Coq that allows the construction of Gallina terms, using Gallina and the Ltac tactic language as sub-languages).
8Theorem, Proof, and Qed are all actually statements in the Coq vernacular.
powerful debugging facilities, compounding this problem. Its performance is also often lackluster, particularly when trying to build large proofs using complex heuristics. In part this is because Coq type-checks the intermediate terms Ltac produces far more often than it needs to, but there are other more fundamental causes. Because Ltac works by building proof terms out of ones that already exist, and often does so iteratively, the proofs it produces can become quite large, posing a significant burden in terms of time and space. Manipulating and type-checking such large terms is not performant, and large proof terms can take a significant amount of memory. Some of these performance concerns could be addressed by instead implementing tactics in OCaml\cite{8}; however, OCaml cannot resolve the issue of proof size.

2.2 How Verification Is Different

It is worth emphasizing that formally verifying software - that is, using a proof assistant to prove interesting properties about it - is fundamentally different than trying to better understand software by testing it. Testing can only tell us how software will behave under certain circumstances. It cannot give us universal guarantees about how the software will behave no matter how it is run, and it cannot give guarantees about what the software will never do, no matter how it is run. Formal verification is also fundamentally different from proofs about algorithms: such proofs can make no guarantee about the correctness of a particular implementation of that algorithm. They might offer protection from mistaken theoretical ideas, but they cannot protect against bugs in the actual code implementing the algorithm in question.\footnote{This is surely the kind of proof Knuth was talking about when he famously cautioned readers to “Beware of bugs in the above code; I have only proved it correct, not tried it.” \cite{10}} Formal verification’s power is its ability to close this gap, bringing theoretical assurances down to the level of particular implementations.

2.3 Verification: A High-Level Example

A simple example demonstrating the practical benefits of verification is the following: suppose we have a database in which users can access their health records. Users should only be able to access their own records, or other records they have specifically authorized to access. Given the sensitive nature of the data involved, it is crucial that we ensure that this access policy is not violated. The traditional approach would be to write tests that simulate a variety of authorized and unauthorized queries, and to make sure that the system behaves correctly in each case.

However, particularly in real-time, concurrent systems such as the one in this example, the “surface area” tests need to cover is extremely large. For instance, suppose the database has the following vulnerability: if it loses power without warning and restarts, it will enter a recovery mode\footnote{This is not a very well-designed database, as we are about to see. Perhaps the recovery-mode behavior is a bit improbable from a software-engineering point of view, but hopefully not from a theoretical point of view.} that allows users to make...
limited kinds of queries while the data is being checked for integrity. Suppose this more limited query execution engine has a bug, and does not ensure the user has appropriate permissions before fulfilling the query. This is a scenario that might not occur to developers to test, since it involves a combination of unexpected events (a crash and an unauthorized query) that would appear to be unrelated. On the other hand, an attempt at writing correctness proof of the entire program would show that the specification (the database never hands out data to an unauthorized user) is violated in this particular case.

2.4 Why We Don’t Verify

Formal verification is not widely deployed in practice because of its perceived difficulty. Verification is difficult for many reasons. To begin with, it can be challenging to write good specifications for programs that cover all possible executions of the code being specified while providing a more understandable abstraction than the code itself. Another primary reason is that the semantics of term languages like Gallina are very different from those of many languages commonly used in practice. Gallina is a pure, functional language in which routines cannot have side effects; further, Gallina code must be proven to terminate. The code that could benefit the most from being formally verified – low-level systems code, which is difficult to test but must be highly reliable – tends to be written in stateful, imperative languages. Enabling verification in the context of these paradigms is a difficult problem, and VST represents one approach to addressing it.

3 Verified Software Toolchain

The Verified Software Toolchain (VST) is a project aimed at enabling formal, mechanized reasoning about C code. VST is built on the Coq proof assistant and represents a pragmatic approach to verification, emphasizing formal reasoning about code written in a widely used language. Although C is ubiquitous in systems programming, and hence a logical choice for a framework like VST to operate on, C’s semantics are difficult to formally specify. Additionally, because C is imperative and stateful, its semantics do not map neatly onto the pure-functional semantics of Coq. Despite these challenges, there are examples of VST being put to use on real-world code (for instance, Andrew Appel’s verification of OpenSSL’s SHA256 implementation). A large part of VST’s

the reader agrees that this is not too far-fetched a scenario.

In a sense, writing a correctness proof like the one in this example involves exploring all possible corner cases in the software. This makes proving the correctness of large programs a difficult endeavor, but at least it is within the realm of possibility – unlike finding every possible weakness in a large codebase by hand.

It is true that it is difficult, but in recent years it has become much more feasible than many realize.

Making understandable abstractions of the code’s behavior is the entire point of an abstraction; otherwise, we’re back where we started, with an unintelligible specification replacing unintelligible code.
strength lies in its proof-automation facilities; it provides tools to automate the construction of the complex proofs that arise in formally verifying C programs, particularly on a real-world scale.

3.1 Separation Logic

At the core of VST is a logical system for modeling how C programs can (safely) use heap memory, called separation logic [17]. Expressions in this logic consist of so-called “pure” facts (plain Coq propositions, usually in classical logic), along with statements about how memory is used. Taken together, these form so-called “memory predicates”, or mpreds. Operators for making statements about memory in separation logic include the emp primitive, representing an empty heap; the separating-conjunction (‘star’ or *) operator, which combines two mpreds, and represents the statement that both are true on disjoint regions of memory; and the separating-implication operator (often called the magic wand”, since it is often represented \(\rightarrow\)). Separating implication represents a statement about the result of joining two disjoint heaps; namely, the statement \(P \rightarrow Q\) (for \(P, Q\) mpreds) means that adding a (disjoint) heap that satisfies \(P\) (via separating conjunction) yields a new, combined heap satisfying \(Q\). Separating implication it is less often used than the other operators because it has proved difficult to formalize a proof theory for separation logic that takes it into account. There is also the maps-to operator (\(\mapsto\)), which expresses the idea of “dereferencing” (i.e., the fact that particular data lie at a particular place in memory).

Separation logic represents an extension of Hoare Logic, a system for reasoning formally about imperative code by ‘chaining together’ pre- and post-conditions for individual statements, yielding a proof about pre- and post-conditions for the overall program consisting of such statements. An advantage of Hoare logic is that it yields proofs that are similar in structure to the programs they deal with. Proofs in Hoare logic can be large and difficult to construct by hand, however. Only with the recent rise of efficient and usable proof assistants (such as Coq) has it become feasible to use Hoare Logic and its derivatives on programs of meaningful scale. Adding the ability to state and prove facts about heap-memory usage gives this logic the generality it needs to represent most facts about the correctness of C programs that one would care to prove formally.\(^\text{14}\)

In the process of proving theorems and lemmas about programs in VST, the user will also have to prove subsidiary facts about heap memory. These proof goals come in the form of separation logic entailments; these are usually expressed \(A \vdash B\) for mpreds \(A\) and \(B\). Much like implication in classical logic, entailment in separation logic expresses the idea “any system state satisfying \(A\) also satisfies \(B\)”. The general problem of solving these entailments is undecid-

\(^{14}\text{There are some properties that we would like to prove about C programs that we cannot prove with such a system: for instance, we cannot use separation logic (alone) to reason about running time of C programs.}\)
able\textsuperscript{15} but heuristics can be used to significantly automate this process. One important heuristic is \textit{cancellation}: it rests on the observation that if $B \vdash C$ is true, then $A \ast B \vdash A \ast C$ will be as well; therefore, proving the former is sufficient to prove the latter. Care must be taken with this heuristic, as it can turn a provable goal into an unprovable one. For instance, it may be the case that information from $A$ is needed to derive $C$ from $B$ (that is, $A \ast B \vdash C$, but not $B \vdash C$). Cancelling in this case amounts to throwing away information (the $A$ on the left-hand side) that was necessary for the goal to be provable.

Including powerful proof automation that can automatically solve or simplify many entailments, without overstepping and rendering provable entailments into unprovable ones, is an important part of making a system like VST usable.

### 3.2 What VST Contains

The Verified Software Toolchain is a powerful, general platform for formal reasoning about software written in C: that is, it provides facilities for stating facts about C programs, as well as for proving those facts. It contains a formal model of the semantics of C, drawing on work done for the CompCert C compiler\textsuperscript{12}. It also contains a separation logic-based system for modeling the possible behaviors of C programs when executed (\textit{symbolic execution}). The proofs about C programs that VST enables are expressed in the context of this system. VST also provides tactics (that is, Ltac procedures) that significantly automate the process of building such proofs; this is the part of VST that we have attempted to improve using \textsc{MirrorShard}.

This thesis had a significant software-engineering component to it, as it involved “gluing together” two systems (VST and \textsc{MirrorShard}) not originally designed to work together. Though \textsc{MirrorShard} was designed to be adaptable and to work generally with systems like VST, it was not created with VST in particular in mind. For this reason, this project can be seen as an important test-case for \textsc{MirrorShard}, a way to assess its generality\textsuperscript{16}. The implementation details of this project, and the lessons learned over the course of the implementation, can help to provide a road-map for others attempting similar projects in the future: both those making \textsc{MirrorShard} and other solvers like it more easily compatible with the systems which use them, and in terms of making systems like VST easier to extend with solvers like \textsc{MirrorShard}.

### 3.3 VST’s Proof System

VST takes input that is essentially the abstract syntax-tree of a C program, processed through the front-end of CompCert, the formally-verified C compiler. Once in this form, these programs can be formally reasoned about. As described above, VST enables the user to specify and prove a wide range of

\textsuperscript{15}This is because expressions in separation logic can contain propositions from ordinary higher-order logic, which is itself undecidable.

\textsuperscript{16}\textsc{MirrorShard} was originally built for the Bedrock system (described in [9]), which has similar goals to VST but models its own, custom languages instead of C.
properties about the programs she imports into VST. Proofs of these properties are supported using Coq's interactive theorem-proving paradigm, in which tactics are used to manipulate the objects representing proofs. Properties about programs are generally proved using symbolic execution. In the process, many “side-conditions” are generated: extra goals that need to be proved to show that the desired postconditions follow from the given preconditions. These often take the form of separation logic entailments, as discussed above (Section 3.1). Therefore, an entailment solver is a crucial part of a system like VST.

3.3.1 VST’s Entailer Architecture

VST has several Ltac procedures for manipulating the separation logic entailments that must be proved over the course of proving properties about a program (that is, these entailments are proof goals). These include `cancel`, which applies simple lemmas to cancel out terms on either side of the entailment; `normalize`, which rewrites goals using a database of lemmas that describe how entailments can be rearranged into a normal form; and `entailer!`, which combines these tactics, and others, to attempt to solve (or at least simplify) entailments. `cancel` (and, hence, `entailer!`, which calls it) are unsafe, in that they may turn a provable goal into an unprovable one; therefore, care must be taken in their use. A “safe” version of this tactic, `entailer`, is guaranteed not to do this: it only applies the cancellation lemmas if these lemmas solve the goal successfully, returning the entailment to the user, unmodified, if it cannot.

As mentioned, all of these tactics are written purely in Ltac. The tactics - particularly `normalize`, and hence, `entailer/entailer!`, which call `normalize` - are slow; even modestly complex entailments can take many seconds to solve, a cost which adds up quickly when one is attempting to prove properties of a large program whose proofs involve many such entailments. `normalize` is especially slow because of its use of the `autorewrite` tactic, which attempts to rewrite the current goal using a database of lemmas; it attempts to do so essentially by brute force, repeatedly attempting to rewrite by each lemma in the database until it fails, then moving on to the next one. Each rewrite is somewhat expensive because of the type-checking and syntax-tree manipulation involved; this is compounded by the fact that some lemmas might be applied several times before failing, and the order in which the lemmas are applied cannot be changed (since all calls to `normalize` use the same, ordered, database).

For backwards compatibility, the current entailor tactics will remain part of the VST codebase for the foreseeable future. It should be emphasized that the current entailor system in VST works quite well, and is able to solve or significantly simplify many entailments. It was actually used in this project; for instance, we used these tactics to prove the correctness of the linked list lemmas that we supplied to MIRRORSHARD (Section 5.2). Proving these lemmas would have been extremely difficult without it.
4 MirrorShard

In contrast to the entailment solver described in Section 3.3.1, MirrorShard is a reflective solver. Here we will talk about what this means, give more detail about its workings, and describe how they differ from those of the Ltac entailment solver.

4.1 Computational Reflection

Part of the power of the Coq proof assistant lies in the generality of Gallina, the language in which both proofs and the programs to which they apply are expressed. Gallina represents a powerful combination of formality and flexibility. One example of Gallina’s flexibility is the variety of ways in which it can be used to express proofs. The traditional approach in Gallina (which we will refer to as term manipulation) is to build proofs out of smaller parts, applying different subsidiary lemmas in a syntax-tree structure. This is typically done in a tactic language such as Ltac. An alternate approach, computational reflection, is to express the proof as the application of a single reflection function, which, when executed, will prove the desired fact.

Reflective solvers avoid the problems of tactic-based proof construction (see Section 2.1) by exploiting the generality of Gallina, Coq’s term language, and perform proof-automation within Gallina itself. The primary advantage of this approach is in the size of the proof-terms it produces: the proof yielded by a reflective solver might be a call to a single Gallina function (the solver), rather than the massive tree of subterms Ltac would put together. Another advantage is that the process of constructing the proof itself is easier to reason about. Ltac does not understand Coq’s underlying type system, and this mismatch can lead to many runtime (i.e., proof-construction-time) bugs that are difficult to catch (particularly because Ltac lacks powerful debugging facilities). Though incorrect tactics will not cause Coq to accept an invalid proof (Coq’s type-checker will catch the error), it can be difficult to determine what caused the error in cases where tactics fail to produce a valid proof. In contrast, a reflective solver’s ability to produce a correct proof can itself be formally proven, making its behavior more predictable.

In order to use a reflective solver, the terms the solver is to work on must first be reified[17] literally, “made real”. In this case, by reification we mean putting terms into a form that is amenable to automated reasoning and manipulation in Gallina. This is necessary because Gallina is less flexible than Ltac in its ability to pattern-match. Gallina cannot attempt to match on arbitrary types or propositions: it can only match on terms that have a well-defined set of possible data constructors. It is in this sense that Gallina can only work on “real” terms: Gallina must know that the terms can be constructed in a limited number of possible, concrete ways. Reification is necessary to put terms into a form that has this property, while still retaining all the information the reflective

[17]This approach is also known as deep embedding.
solver needs in order to reconstruct the original terms later. This reconstruction, performed after the solver has finished his work, is called reflection. After reflection, the user can interact with the output of the solver without needing to understand details of the internal (reified) representation.

4.2 **MirrorShard: An Overview**

The **MirrorShard** project is a platform for reflective solving in Coq. **MirrorShard** focuses primarily on solving separation logic entailments. MirrorShard implements cancellation for separation logic entailments, but cancellation alone is not enough for most program verification tasks: we also need to be able to reason about application-specific structures and program values. As an extensible framework, however, **MirrorShard** provides several mechanisms that make it possible for the user to enrich this cancellation algorithm to solve entailments that arise in real-world settings.

To this end, **MirrorShard** includes an extensible plugin system that can take in user-supplied lemmas (in a reified form) and attempt to apply these lemmas to the entailment at hand. In this way, **MirrorShard** can be made aware of the specific data structures being used in the programs at hand; this makes its entailment-solving facilities much “smarter” than they would otherwise be. An entailment solver that is completely agnostic with respect to data structures is much less powerful, as lemmas about data structures are helpful in guiding the solver in a useful direction. For the purposes of this paper we have supplied **MirrorShard** with lemmas describing the behavior of C linked lists, but it is of course possible to describe other structures (such as trees) as well; doing so in the scope of VST is an area for future work.

**MirrorShard** also enables the user to supply *pure provers*. These are routines that make use of an entailment’s hypotheses (facts from the left side of the entailment, or hypotheses to the entire entailment) to simplify or solve the entailment. These provers are pure in the sense that they work on the level of “pure” logical propositions: that is, facts that do not contain information about heap memory. One simple example which we used in our work is the SymmetryProver; it solves goals of the form $a = b$ by looking for hypotheses of the form $b = a$ or $a = b$, and applying them if it finds them. Provers can be called from other provers, enabling them to be combined together. For instance the CompositeProver, provided with **MirrorShard**, executes two other provers (given as arguments) in order; if the first fails to solve the current entailment, the second is called.

---

18To make automated reasoning easier, **MirrorShard** only supports a fragment of separation logic; among other things, it does not support the “magic wand” operator. Fortunately, this fragment is still powerful enough to enable **MirrorShard** to handle most entailments we care about, in practice.
**Inductive expr : Type :—

  | Var : var → expr
  | Func : func → list expr → expr
  | Equal : tvar → expr → expr → expr
  | Not : expr → expr
  | UVar : uvar → expr

Figure 2: Grammar of the expr type.

4.3 Reification

MIRRORSHARD departs from previous work on computational reflection by offering an extensible syntactic representation (what we will refer to below as a reified form). This extensibility enables systems that use MIRRORSHARD for reflective solving to reason about a larger set of symbols. In particular, MIRRORSHARD can be customized after the fact to take into account new, application-specific symbols that were not known to MIRRORSHARD’s authors at the time MIRRORSHARD was written.

One important step in the usage of MIRRORSHARD (or any reflective solver) that does not arise in the Ltac enterailer is reification: that is, putting terms into a form in which they can be manipulated by Gallina. This enables more efficient computation: Gallina, unlike Ltac, is statically typed and is compiled (automatically) to Coq’s bytecode, so it can be executed more quickly. This is not the case with Ltac.

MIRRORSHARD’s reification works as follows. The Coq (Gallina) terms given to MIRRORSHARD to work on must be translated into the form of the expr (“expression”) datatype. Here is the declaration of this datatype, a description of its grammar:

In the rest of this section we will describe the semantics of these constructors.

- **Var**, which enables the representation of universally-quantified variables. This constructor is important in the reification of terms with universally-quantified subterms. To distinguish between different variables, Var uses natural-number indices.

- **UVar**, which enables the representation of unification variables (essentially placeholders in Gallina terms) so that terms containing unification variables can be reified. This is a fairly common use-case; for instance, VST’s

---

19 In fact, writing pattern-matches that are expected to sometimes fail is a common design pattern; Ltac actually “falls over” to the next pattern if one pattern fails.

20 var, func, uvar, and tvar are all natural-number indices used to distinguish between different variables, functions, unification variables, and (reified) types, respectively. They are literally just aliases for the nat type (type of natural numbers in Coq).

21 During the construction of terms, it is often convenient to leave “holes” in terms, filling them in later. This is done frequently in a tactic-oriented approach to proof construction. This happens frequently with existential quantifiers: if we have some predicate B of a, the tactic \texttt{exists} can turn a goal of the form \texttt{exists a, B a} into a goal with a unification variable, which will
load and store lemmas often result in entailments containing unification variables as subterms. Like \texttt{Var}, \texttt{UVar} takes a natural-number argument, tracking which unification-variable is being represented.

- **\texttt{Func},** which is the workhorse of this representation. This represents the application of a Gallina function to a list of \texttt{expr}s representing its arguments. The user defines a list of the functions about which she wants to make \texttt{MirrorShard} aware, including an index for each.\footnote{Among other things \texttt{MirrorShard} takes a functions list the user wishes to inform it about; the index is the location of the desired function in the list.} Any function whose type does not have dependency (i.e., whose type does not have “forall” quantifiers), can be reified into this form. This form is useful for reifying even terms one would not typically think of as being functions, including nullary functions and nullary data-type constructors (such as \texttt{O}, the “zero” object of natural numbers). When defining the functions list, the user must supply a denotation for each function (i.e., the function that that particular application of the \texttt{Func} constructor is to represent); these denotations are used later, when reflecting reified terms back into standard Coq terms.

- **\texttt{Equal},** which represents a statement of the equality of two \texttt{expr}s. A separate constructor is needed for this, as normal equality in Coq involves dependent types, which \texttt{MirrorShard} cannot deal with.

- **\texttt{Not},** which represents negation of another \texttt{expr}.

Some versions of \texttt{MirrorShard} have an additional type of expression - \texttt{Const} - which enables the injection of arbitrary Coq expressions; however, the presence of \texttt{Const}s makes reflection more difficult, as reflection may lead to attempts to compute on terms that should not be computed on (it is desirable, for instance, not to allow for computation on propositions, as doing so is extremely time-consuming and seldom useful). It also makes equality on reified terms undecidable in general, which makes it difficult to write provers that work with reified terms. We determined that it was best to do away with \texttt{Const}s entirely, instead reifying all terms that would have been \texttt{Const}s as \texttt{Func}s, and allowing the user to specify which of these functions can meaningfully be computed on, and which cannot. In \texttt{MirrorCore}, the successor to \texttt{MirrorShard}, the \texttt{Const} constructor will no longer exist.\footnote{\texttt{MirrorShard} is rendered like \texttt{B \#12}. \#12 (the number will vary) is a unification variable representing a “hole” that needs to be filled in. This is useful if we know an appropriate \texttt{a} exists, but don’t yet know which \texttt{a} works. We may want to reify terms containing such “holes”, leading to the need for the \texttt{UVar} constructor.}

In order to support usage of this reified form (and, hence, use of \texttt{MirrorShard}), tactics had to be created to reify Coq terms. While it is possible to perform reification by hand, this would be prohibitively time-consuming for all but the most trivial expressions; Ltac’s more flexible pattern-matching facilities make it possible to write tactics that will do this without requiring manual intervention. Reification tactics had already been implemented for the Bedrock
system, MirrorShard’s original client, but we had to modify them, adding pre- and post-processing steps to adapt them to our purpose. The most notable of our modifications were the following:

- Changing the syntactic form of $TT$ and $FF$ ($\top$ and $\bot$ in separation logic, respectively) to mask an incompatibility due to MirrorShard’s lack of support for typeclasses.

- Rewriting to make sure that all pure facts embedded in the entailment are joined with a spatial component (even if that component is the empty heap, $\text{emp}$). This is necessary because MirrorShard’s cancellation system is unable to process pure facts on their own.

- Various tactics, lemmas, and heuristics for dealing with and and not in embedded pure facts, in such a way as to make conjunctions of embedded pure facts more accessible to MirrorShard as hypotheses during the pure-proving phase of cancellation.

Though the target language of reification (the reified $\text{expr}$ form) is the same across all clients of MirrorShard (modulo perhaps the presence or absence of the $\text{Const}$ constructor), the source language of the translation depends on that system’s representation of the terms that MirrorShard will need to process. In particular, the translation depends on the details of the system’s implementation of separation logic. In order to use MirrorShard with a separation logic system, MirrorShard must be supplied with proofs that the system meets some basic axioms of separation logic. In the case of VST, we have provided these.

In addition to requiring a list of supported functions, MirrorShard also requires a list of supported types, along with the indices that are used to represent those types in their reified form. Function signatures in the functions list are expressed in terms of these reified types. These types are not permitted to be parameterized over other types - another way of stating the condition that dependent types are not permitted. This necessitates workarounds, as even polymorphic types are not supported. In our case, for example, we had to treat lists of different types of elements as different types - so $\text{list int}$ and $\text{list bool}$, for instance, would have to be reified as separate types in order to be supported by MirrorShard. The $\text{map}$ function applied to different types of lists (in Coq, $\text{@map int}$ versus $\text{@map bool}$) would need to be reified in an analogous way. If the user wishes to use many different forms of these polymorphic types (or wishes to use other, more complicated instances of dependent types), there is the potential that a large number of separate reified functions and types would need to be created; if this number were large enough, it might negatively affect performance (as the reflection process involves linear lookups in the lists of types and functions). In the examples on which we have used MirrorShard, however, we find that we do not use more than two or three different forms of these types or functions, making this less of an issue in practice.

Footnote: This is a consequence of its lack of support for dependent types more generally.
The user needs to do some extra work to allow for reification, but not a particularly large amount. We have provided a list of “default” functions and types that we consider to be important (in particular, functions and types that are used in the provers and lemmas we provide by default). These include, for instance, booleans and various representations of numbers (with their corresponding operations); types representing C variables and pointers; and other important utility functions. If these functions cover all of the functions the user wishes to make use of in her code for reasoning about entailments, nothing extra need be added. The same applies to separation logic predicates, which the user must also supply. In particular, she must supply all the predicates that will be required to reify the separation logic entailments that her C code will produce. We have included the basic ones: field_at (a statement about data stored in the field of a data-structure on the heap), lseg (a statement about the contents of a linked list on the heap), and list_cell (a statement about a single linked list cell on the heap, and its contents).

Depending on the program, the functions and predicates we have provided may suffice; or others may need to be declared. Though MirrorShard’s reification tactics have the ability to extend the functions and predicates contexts with new functions and predicates they encounter (new in the sense of not being in the lists supplied to MirrorShard), these supplied lists must include all functions and predicates mentioned by the lemmas the user supplies to MirrorShard; otherwise, MirrorShard will not be able to apply the lemmas.

Here is a schematic diagram showing the process of reflective solving at a high level, with the use of a prover. This diagram is from a presentation by Gregory Malecha; it can be found in context here: [14]. Note the two different “levels” - labeled ‘syntax’ (corresponding to the reified form) and “semantics” (corresponding to the Coq terms represented by the reified form).

This diagram neatly represents the process of proof by reflective solving: we start with a goal, then reify it to get a reified version of that goal, expressed in terms of exprs. We then apply provers to it, and reflect back into the original “semantics” level, getting a simplified version of our goal (if it has not

Figure 3: Schematic of Reflective Solving.
already been solved completely). The soundness lemma for the prover we applied guarantees that the transformations made to the goal while it was in reified form are valid. The diagram leaves out other steps performed by MIRRORSHARD, however, such as lemma application and cancellation.

4.3.1 An Example Entailment

Here is an example of what the reified form of an entailment (and, hence, its constituent terms) actually looks like. Suppose we have the following entailment:

\[ a \ast b \ast \text{field\_at sh T id y x} \vdash b \ast a \ast \text{field\_at sh T id y x} \]

Using the reify derives tactic we have defined, which calls on the MIRRORSHARD reifier, along with some additional processing steps we have added (see Section 4.3), we can reify this entailment. When we do, we get this:

\[
\text{goalD} (\text{all\_types\_r nil}) \text{funcs preds uenv nil}
\]
\[
(\text{lift\_ands}
\quad (\text{Sep Star} (\text{Sep Func} 7\% \text{nat nil}) (\text{Sep Func} 8\% \text{nat nil}))
\quad (\text{Sep Func} 0\% \text{nat}
\quad (\text{Func} 55\% \text{nat nil}
\quad :: \text{Func} 56\% \text{nat nil}
\quad :: \text{Func} 57\% \text{nat nil}
\quad :: \text{Func} 58\% \text{nat nil :: Func} 59\% \text{nat nil :: nil})))
\]

Not all details of this reified form are instructive. It is worth observing, however, that the entire reified entailment takes the form of a single function application: goalD, or “goal denotation”. Note also how all the terms of the entailment have been converted into a predictable form: applications of constructors of the expr datatype (or sexpr, an analogous datatype for describing separation logic expressions). They are now only distinguished by their numerical indices. For example, Sep Func 0%nat is the reified representation of the field\_at separation predicate.\footnote{This is very similar to mapsto.}

Functions and separation predicates not recognized by MIRRORSHARD (i.e., not in the lists of functions and predicates the user provided MIRRORSHARD) are reified as “fresh” functions; they are essentially added onto the end of the user’s
functions list. We can see this in the list argument to \texttt{Sep.Func 0\%nat}, where we have several universally quantified variables reified in this way (\texttt{func 56\%nat nil} through \texttt{func 59\%nat nil}). This is also evidenced in the reification of (universally quantified) predicate \texttt{a} and \texttt{b}, which reify to \texttt{Sep.Func 7\%nat nil} and \texttt{Sep.Func 8\%nat nil}, respectively.\footnote{The indices for the “fresh” functions and predicates depend on the length of the user-supplied functions and separation predicates lists, since the “fresh” ones are added on to the end. The exact numbers each is assigned will thus vary between different environments.}

4.4 Lemmas

The lemmas the user wishes to provide to \texttt{MirrorShard} must also be reified so that \texttt{MirrorShard} will be able to use them in its attempts to solve entailments. These lemmas are each of the form $A \vdash B$, for separation logic predicates $A$ and $B$. Each lemma is provided either as a “left lemma”, in which case \texttt{MirrorShard} uses it to transform goals of the form $Z * A * Y \vdash X$ into goals of the form $Z * B * Y \vdash X$; or as a “right lemma”, in which case \texttt{MirrorShard} uses it to transform goals of the form $X \vdash Z * B * Y$ into goals of the form $X \vdash Z * A * Y$. This distinction is important, as some lemmas are extremely useful if applied in one direction, but might be counterproductive - or even unsound - if applied in the opposite direction.\footnote{Suppose, for instance we have memory predicates $A$ and $B$, and we have as a lemma $A * B \vdash FF$ [where $FF$ is separation logic’s $\bot$; i.e., the unprovable fact from which anything can be proved]. Using this fact as a left lemma could be useful, as it would allow us to rewrite $A * B \vdash C$ into $FF \vdash C$, which is true [trivially]. On the other hand, using this as a right lemma would cause us to rewrite $C \vdash A * B$ into $C \vdash FF$, which is not provable.}

In the context of \texttt{MirrorShard}, these lemmas are also referred to as hints.

Reifying lemmas largely involves reifying the expressions on either side of the entailment (and then composing them into a \texttt{MirrorShard} term representing a reified entailment). \texttt{MirrorShard} provides tactics for this, which depend on the user having declared reified forms for all the terms on either side of the entailment she wishes to use as a lemma.\footnote{The functions that apply in the lemmas must be pre-defined by the user; it will not do for \texttt{MirrorShard} to render them as “fresh” functions as in Section 4.3.1} (We have adapted these tactics slightly to make them applicable to VST). Proofs of these lemmas also need to be provided; having these proofs gives \texttt{MirrorShard} the flexibility of using reified lemmas that have been provided without function- or type-contexts. After reflection, these Coq proofs can be applied to ensure that the rewriting \texttt{MirrorShard} performed was sound. Here, too, \texttt{MirrorShard}’s lack of support for dependent types becomes somewhat problematic: as with dependently-typed functions, lemmas with dependently-typed arguments need to be partially applied, using the same trick described above for the cases of types and functions. In the case of lemmas things become slightly more complicated, as these lemmas then need to be added to the list of lemmas \texttt{MirrorShard} knows about, but this can only be done after they are reified. We discuss how we addressed these challenges in Section 5.2.
4.5 Pure Provers

\textsc{MirrorShard} must also be provided with \textit{pure provers}, Gallina routines that attempt to complete the process of proving goals once they have been rewritten using the hint-lemmas discussed in Section 4.4. In doing so, provers can make use of the hypotheses to the supplied goal. Each prover includes a main proving routine, as well as the code for a preprocessing step run on the hypotheses first, called ‘summarization’, which can improve the performance of the main proving step by distilling the provided hypotheses into a list of hypotheses that will be useful to the prover. Like the lemmas, these provers need to be themselves proven correct, in order to ensure that the (reflective) proof they help to generate is valid.

Here I will describe the provers we use (some of which are provided by default with \textsc{MirrorShard}, others of which we wrote for the purposes of this project).

- The reflexivity prover, which simply discharges goals of the form $A = A$. This checks only for syntactic equality; that is, it essentially checks only to see if the two sides consist of the same functions applied to the same lists of arguments, without making reference to the interpretation (reflection) of functions involved.

- The symmetry prover, which (as discussed above) attempts to prove goals of the form $A = B$ by looking for hypotheses of the form $A = B$ or $B = A$. Again, this uses only of syntactic equality.

- The computation prover, which discharges goals of the form $A = B$, if the computational interpretations of $A$ and $B$ are syntactically equal. This is done by computing on both sides of the (reified) equality, and seeing if two results are decidable equal. Unlike the two provers above, the computation prover goes beyond checking syntactic equality of the given terms; as a result, it can be more powerful, but is also potentially more time-consuming. This is one of the new provers we added as part of this project; in fact, it is one of my primary contributions to the project. This prover is discussed at length in section 5.1.

- The congruence prover, which attempts to implement a subset of the behavior of the Ltac tactic \texttt{congruence}. It makes use of hypotheses, using transitivity of equality. For instance, it can solve goals depending on facts of the form $A = B \rightarrow B = C \rightarrow A = C$). This prover was also added as part of this project.

- The composite prover (which takes other provers as arguments) is used to combine provers together (this is necessary, since \textsc{MirrorShard} takes a single prover as an argument). Being a critical piece of the prover infrastructure, it is provided with \textsc{MirrorShard}. However, it only supports one type of composition: namely, it calls on the provers it is passed as arguments, one at a time, in order. One could imagine other schema for combining provers; for instance, we might want to try to apply all the
hypotheses we can (as in the congruence prover) and then compute. This could be an interesting and potentially useful area for future improvement. Since provers are applied in order, order is important, particularly since the more complicated prover strategies take longer to run. We composed provers in the order they are listed above (reflexivity, symmetry, computation, congruence); essentially, in order from least to most complex/time-consuming.

4.6 Solving

Once **MirrorShard** has been appropriately informed of the types, functions, and lemmas that will be used, making use of **MirrorShard** is quite simple from the user’s point of view. The user simply needs to call on the `rcancel` tactic. This tactic will appropriately extract **MirrorShard**’s environment (the types, functions, and predicates lists) from the modules that contain them, then call **MirrorShard**’s reifier on the goal the user is trying to solve. If this is successful (which, if the user has appropriately set up the environment, it will be), **MirrorShard**’s canceler is then called. The behavior of the canceler is described in more detail above (Section 3.1); in brief, it applies lemmas, simplifies using internal cancellation heuristics, then attempts to simplify or solve the goal using the provided provers, if it has not already been solved. If **MirrorShard** fails to solve a goal but succeeds in simplifying it, it is able to return to the user the simplified goal in its standard Coq representation. This process is called reflection; it consists, more or less, of converting the functions and predicates that make up the lemma into the denotations the user supplied when declaring the types and predicates (making them accessible to **MirrorShard**).

As noted above, for many typical applications the amount of work needed to set up the environment is minimal; it may even be zero if the lemmas, types, and functions we provide by default are sufficient to express all the facts the user needs to prove about her code. However, the system is also quite customizable, should the user’s needs be more unusual. Customizations can range from adding new types and functions to **MirrorShard**’s environment; to adding lemmas to the environment (a slightly more involved step); to even writing a new pure prover plugin for **MirrorShard**, should the user want to support solving strategies beyond the ones we support by default (that is, the ones outlined in Section 4.5). This last customization is significantly more involved, as the user will need to prove the canceler that she writes correct; however, this particular type of customization be necessary less often. We anticipate that users advanced enough to require custom solvers will be willing to learn how to prove those solvers correct; in any event, there is no way around this requirement. The default solvers will suffice for a wide range of applications. This customizability is due to the modular, extensible design of **MirrorShard**, one of its key advantages.
5 My Contributions

In adapting MIRRORSHARD to work well with VST, we had to build some infrastructure around MIRRORSHARD, to allow it to understand the specifics of VST. First, many of the proof goals VST generates are currently phrased in a mostly-reflective style\(^{30}\) which makes use of computation. Computation is simple to implement at the tactic level, but it is surprisingly difficult in the context of computational reflection. In Section 5.1, I discuss the computation prover, which I developed to enable MirrorShard to reason about terms by computation\(^{31}\). The implementation and correctness proof of the computation prover were contributed by me\(^{32}\).

Second, in order to improve MIRRORSHARD’s ability to reason automatically about proof goals involving linked lists, we needed to provide it with a theory of list segments: that is, a collection of lemmas that can be applied to simplify entailments containing facts about linked lists. Fortunately, such a theory already exists \([16]\); I have adapted it to our purposes, and re-proved its constituent lemmas in the context of VST (Section 5.2).

5.1 Computation Pure Prover

The computation prover is crucial to be able to determine equality on terms that may be semantically equal (i.e., compute to the same result) but possibly not syntactically equal. The computation prover represents an important extension to what “vanilla” MIRRORSHARD can provide, and makes its entailment-solving capabilities more powerful for our purposes. The ability to automatically discharge goals by simplification is an important capability that VST’s Ltac-based entailment (indeed, any tactics-based proof system in Coq) can provide, often by a call to simpl or vm_compute. To match the old entailment, we needed to supply this functionality as well.

5.1.1 Computability

Depending on their semantics, some reified MIRRORSHARD terms should be considered computable, while others must not. An example of a non-computable function is one that is implicitly quantified over a term that does not appear explicitly in the reified representation. For instance, eval_id evaluates an identifier with respect to a particular environment, but we never reify the full environment, instead treating it as a single uninterpreted function\(^{33}\). Therefore, eval_id is thus considered not to be computable. Other functions should not

---

\(^{30}\) By “mostly-reflective”, we mean that most terms are amenable to reification, because they could already be considered to be (mostly) in a reified form. Other than the parts that are not, allowing MIRRORSHARD to operate on these terms is just a matter of switching reified representations to MIRRORSHARD’s.

\(^{31}\) This could also be called reasoning by reduction.

\(^{32}\) Josiah Doherty made some minor modifications to the correctness proof in order to enable it to work with changes in the prover’s “computability” system (see Section 5.1.2).

\(^{33}\) This is because, in general, we will not know the contents of the environment until runtime.
be computed on because doing so would be prohibitively complicated; this is the case, for instance, with terms whose denotation involves complex Prop expressions. Due to these expressions' heavy use of dependent types, computation on them can be extremely time-consuming.\footnote{Further, because of proof irrelevance, the semantic equality of two proofs of the same type (i.e., proofs of the same fact) is seldom interesting.}

As a further complication, we wanted the user to be able to supply a list of extra functions (user-defined functions) to use with \texttt{MIRRORSHARD} on top of the default ones. We wanted to give the user the option to have some of these not be computable, for greater flexibility. We therefore require the user to supply a function which maps function indices (that is, the index of their reified signature in \texttt{MIRRORSHARD}'s internal list) to \texttt{bool}, that should return true or false if the given (reified) user-supplied function is or is not to be considered computable (respectively).

5.1.2 \texttt{do_computation} and \texttt{do_computation_equal}

The basic function used by the computation prover, \texttt{do_computation}, makes use of knowledge about computability to compute safely on a term. It is given an entire reified term as an argument, as well as a \texttt{tvar} (reified type) representing the type of the term, once computed.\footnote{This type is required by the expression-denotation function for reflecting reified terms, \texttt{exprD}.} We first call \texttt{is const} on the term in question, which recursively checks to ensure all the constituents of the reified term are computable; if any is not, we return \texttt{None}. If all subterms are computable, we then call \texttt{exprD} on the term, which returns the Gallina terms they stand for. This function is called \texttt{do_computation} because it returns a standard Gallina term that can be efficiently and safely computed on with any of the standard computation tactics, such as \texttt{vm_compute}. \texttt{do_computation} also has an accompanying correctness proof, stating that if it successfully computes on a term, the result will be equal to calling \texttt{exprD} on that term.

\texttt{do_computation}, in turn, is primarily used by \texttt{do_computation_equal}, which attempts to decide (semantic) equality between two reified terms (or, more accurately, on a single reified term built using the \texttt{Equal} constructor), provided those terms can both be computed on and that the type of their result has decidable equality. To do so, it calls \texttt{do_computation} on each of the terms being compared for equality; if both calls succeed, \texttt{do_computation_equal} then attempts to decide equality between the two results using the equality decision procedure provided in the reified representation of the type of the result. These decision procedures can be conservative, in that they are required to return \texttt{false} if (but not only if) the terms in question are not equal. For types without decidable equality, we provide the \texttt{no_eqb_type} constructor as a convenience, which builds a type with a "decision procedure" that always returns \texttt{false}. \texttt{do_computation_equal} returns a boolean value - \texttt{true} if the procedure succeeded and the terms were equal; \texttt{false} if the procedure failed (e.g., a term was not computable), or the terms turn out to be unequal. Obviously, in the case
of terms that do not have decidable equality (terms built with \texttt{no_eqb_type}), \texttt{do_computation_equal} will always return \texttt{false}. \texttt{do_computation_equal} has its own proof of correctness, stating that, if \texttt{do_computation_equal} returns \texttt{true}, then there must exist some term (of the appropriate type) that both terms being compared compute to (under \texttt{do_computation}). For reasons just discussed, the proof is silent on what happens when \texttt{do_computation_equal} returns \texttt{false}.

To aid the reader’s understanding, we provide the code for \texttt{do_computation} and \texttt{do_computation_equal}, as well as helper functions for determining computability (‘const-ness’):

This functionality for safely computing on reified terms and comparing the results for equality is used by \texttt{computation_prover}, which implements semantic equality-checking, enabling the automatic solution of many goals whose proofs depend on equalities that are true, but not "obviously" so based on syntax alone. In this section I will describe the workings of this prover; this section might also be useful to users of \texttt{MIRRORSHARD} who wish to implement their own provers, as it provides some guidance for doing so.

The prover (like \texttt{do_computation} itself) is quantified over user-supplied type and function contexts, as well as a user-supplied function describing which items in the function context correspond to computable functions (as described above \ref{5.1.1}). This quantification is done implicitly, through Coq’s \texttt{Section...Variable} syntax. The mechanics of the prover itself are quite simple: \texttt{computation_prove}, the function that actually performs the “proving”, is essentially a single invocation of \texttt{do_computation_equal} on the term it is attempting to prove. \texttt{computationSummarize} and \texttt{computationLearn}, routines for manipulating the hypothesis list given with the goal to be proven, are even simpler: since the hypotheses are not used in this prover, they are essentially no-ops, passing them through without changing them (this is the case, in fact, for some other provers that do make use of hypotheses; often the original hypothesis list, unchanged, will be what is needed). The correctness proofs for these auxiliary functions are fairly trivial.

The overall correctness proof for \texttt{computation_prover} is somewhat more interesting. A few auxiliary lemmas were used to describe the behavior of \texttt{exprD} when its functions list is extended to account for user-added functions. Though these are not strictly necessary for the final version of the proof, they are described here as they might be useful in similar provers, as they were used in the original version of the proof\footnote{Removing the \texttt{Const} constructor obviated the need for these lemmas.}. The core of these lemmas is \texttt{exprD_functions_extend_special}, which essentially states the following: for two functions lists \texttt{fs} and \texttt{fs’}, if \texttt{fs} and \texttt{fs’} “agree” everywhere that \texttt{fs} has a valid entry (i.e., for all \texttt{n} such that \texttt{fs} has a value at index \texttt{n}, \texttt{fs’} has the same value at the same \texttt{n}), then for any reified expression, the following holds: if the denotation of that expression (\texttt{exprD}) in the context of \texttt{fs} exists and has some value (in Coq terms, is not \texttt{None}), then the denotation of that expression in the context of \texttt{fs’} must have the same value. This more general result is used to show

Variable user_computable : func → bool.

Definition is_const_base (f : func) :=
  orb (NPeano.ltb f (computable_prefix_length all_types))
   (user_computable f).

Fixpoint is_const (e : expr) :=
  let is_const_l (es : list expr) : bool :=
    fold_right andb true (map is_const es)
in
  match e with (* it and its arguments must be const *)
  | Func f es ⇒ andb (is_const_base f) (is_const_l es)
  | _ ⇒ false
  end.

Definition do_computation (e : expr) (t : tvar) :
  option (tvarD all_types t) :=
  if is_const e then match (@exprD all_types all_functions nil nil e t) with
  | Some v ⇒ Some v
  | None ⇒ None (** should be dead code if e is well typed **) end
else None.

Definition do_computation_equal (e : expr) : bool :=
  match e with
  | Equal t l r ⇒
    match do_computation l t with
    | Some l' ⇒
      match do_computation r t with
      | Some r' ⇒ get_Eq all_types t l' r'
      | _ ⇒ false
      end
    | _ ⇒ false
    end
  | _ ⇒ false
end.

Figure 5: Implementation of do_computation and related functions.
that, when a functions list is "extended" by appending onto the end, the denotation of an expression that is valid in the first context will be "preserved". The actual proof in the current version of computation_prover_correct is a rather straightforward application of the correctness theorems for do_computation and do_computation_equal; in its original form, it also required manipulating calls to exprD using the above-described lemmas. The remainder of the code is simply applying this correctness theorem in a fairly cookie-cutter way, similar to the other provers used with MirrorShard.

5.1.3 Usage Example

Here is a simple instance where the computation prover becomes useful. Suppose we have a separation predicate \( X : Z \rightarrow \text{mpred} \) and are trying to prove the following entailment: \( X (1 + 3) \vdash X (2 + 2) \).

MirrorShard lacks an explicit understanding of integer arithmetic. On its own, it cannot determine that, in fact, \( 1 + 3 = 2 + 2 \). This is where the computation prover comes in: after applying lemmas and attempting to cancel, the computation prover is run on the goal. The integer type has been provided in MirrorShard's types list, and the integer arithmetic functions and data constructors have been provided in MirrorShard's functions list. MirrorShard has been instructed to treat all of these functions as computable, so it then interprets and evaluates them, comparing their result. Integer equality is decidable, so this comparison is possible. We end up with (effectively) \( \text{Zeq} \_\text{bool} 4 4 \), the application of the equality decision procedure for integers, which returns a boolean. This function returns true; this fact, coupled with the computation prover's correctness proof, creates a proof of the original entailment.

5.1.4 Discussion

The computation prover is a useful example of the kind of new, nontrivial behavior that can be added to MirrorShard by means of pure-provers. As demonstrated in the example just discussed, the computation prover brings many new goals within the reach of automatic resolvability of MirrorShard. As I also hope I have made clear, implementing a new prover need not be particularly difficult: in the case of this prover, the burden (both in terms of code and in terms of proof obligations) was not particularly great. Provers are an

---

37\( \text{Z} \) is Coq's integer type.
38In this example, one could imagine that we originally had a more complicated entailment, and that the one described here is what remains after these first steps.
39These are part of the "default" functions list we created, containing a range of functions we consider to be widely useful. The same is true of the types involved and the "default" types list.
40This is because they are in the "computable prefix"; the functions at the beginning of the functions list, by convention, are deemed to be computable. The length of this prefix is specified by the user (in this case, we use the one from the "default" list).
41This decision procedure is given in the definition of the reified integer type - again, part of the "default" list.
important feature of MirrorShard's extensibility infrastructure, enabling it to be adapted to various purposes and increasing its generality.

5.2 Lseg Lemmas

5.2.1 Reification Strategy

As mentioned above (Section 4.4), reification of the lemmas pertaining to linked list segments (lsegs) was made more complicated by the existence of dependencies between types in the lemma's arguments.

Since MirrorShard's lemma reification tactics were not designed with this workflow in mind, we needed to implement special tactics of our own to automate the process of partially applying and reifying lemmas. This is in contrast to functions and types, which did not need to be reified via tactics before they could be added to MirrorShard's repertoire. In the current version of our work, this process is significantly automated. This automation was another primary contribution I made to this project.

The default lemmas we provide are laws governing the behavior of linked lists, drawn from Navarro Perez's paper [16] (in particular, Figure 2 in that paper). This paper outlines ten lemmas, expressed as separation logic entailments, that describe the behavior of linked lists on the heap; the paper also contains a proof of their completeness. The first five are called "well-formedness" lemmas in Navarro Perez's paper; these describe situations that cannot occur, such as the separating-conjunction of two linked lists starting at the same memory location (if this is the case, then one of the two must be the empty list). The second five are called "unfolding" lemmas; these mostly have to do with appending linked lists, or with prepending individual cells to the beginning of a linked list. In order to use these lemmas with MirrorShard, we had to restate them. Navarro Perez's formalization dealt only with shape properties about lists (i.e., the lemmas he states say nothing about the lists' contents), and the primitives he deals with are slightly different than ours (his next operator is similar to, but not exactly the same as, our field_at operator, as the former also makes a statement about the particular field being a linked list cell).

There are some caveats to MirrorShard's ability to reason about list segments using these lemmas. First, as the Navarro Perez paper makes clear, the theory is only complete if the lemmas are applied according to a particular algorithm (also outlined in the paper). However, MirrorShard can only apply the lemmas according to its own cancellation algorithm. Additionally, MirrorShard is too constrained in its ability to perform rewrites to make full use of all of the lemmas specified in the paper. In particular, MirrorShard can only rewrite one term into many, and can only rewrite one side at a time (so that
it can use lemmas of the form $A \vdash B * C$ as left lemmas only, and cannot use lemmas of the form $A * B \vdash C * D$ at all). For this reason, MIRRORSHARD in its current form cannot apply most of Navarro Perez's unfolding lemmas that pertain to concatenating lists.

Thus, while I have stated and proved correct all of the lemmas from the Navarro Perez paper, not all are useful to MIRRORSHARD. I have included them in the hope that they will be useful in a future version in which these constraints are relaxed. The "well-formedness" lemmas are given to MIRRORSHARD as left-lemmas; the "unfolding" lemmas are given as right-lemmas. These lemmas are sufficient to enable MIRRORSHARD to perform many kinds of useful reasoning about linked lists, automatically; in this respect, MIRRORSHARD's automation is more powerful than that of the previous entailment system.

5.2.2 Performance of Reification

Reification is a somewhat involved process, since it involves traversing the syntax-trees representing potentially large objects, as well as dealing with quantifiers that may be present inside these terms. Each time the user calls MIRRORSHARD's canceler on an entailment expressed in standard Coq terms, the canceler must reify the term so that it can be evaluated by MIRRORSHARD. The exception to this is lemma reification, which only needs to happen once per lemma, for lemmas without dependent types. Even for lemmas with dependent types, the number of times lemma reification is called is proportional to the number of such lemmas times the number of types to which the user anticipates those lemmas being applied; that is to say, it does not depend on the number of times MIRRORSHARD's canceler is actually called.

As currently written, reification is slow; the majority of the time spent solving entailments with MIRRORSHARD is consumed by reification (see Section 6). However, we have identified changes to the reifier that could make reification much faster. Though we have not implemented them, we consider these optimizations to be an important area of future work. First, the reification procedures could be rewritten in OCaml as opposed to Ltac; this could improve performance greatly, both because OCaml is more efficient than Ltac generally, and because OCaml-based tactics do not cause Coq to type-check terms as often as Ltac ones, eliminating one large source of overhead. We regard modular OCaml tactics as being essential to a production-quality reflective-solver framework; writing such tactics would be a natural extension of the work we have already done in writing tactics and would not be too difficult.

Another improvement would be to alter VST's tactics so that they keep the entailments they deal with in reified form as much as possible: if it is possible to automatically symbolically execute multiple statements without generating any side-conditions that the user must prove, there is no reason to reflect and then re-reify the goals each time VST processes a statement. This technique is known as reified symbolic execution. Implementing this optimization would require redesigning other parts of VST's tactics infrastructure to take advantage of reification, making it possibly a somewhat difficult undertaking. However,
this technique, which eliminates some calls to reification entirely, could be a significant optimization.

One optimization to improve the performance of lemma reification would be to reify each lemma once, and then partially apply it once it is in reified form. This would have the effect of ensuring that the potentially time-consuming lemma reification process need only happen once per lemma. However, because lemma reification only needs to be done once per project (i.e., once per set of lemmas the user wishes to give MIRRORSHARD), we have yet to implement this.

Even without these mitigating techniques, we generally find that the performance of reification is satisfactory, though far from perfect (see Section 6). Though there is certainly room for improvement, in most cases reification does not take too long, and the overhead of reification is made up for by the more efficient entailment-solving that MIRRORSHARD provides, so that the entire process (reification, solving, then reflection) does not take much longer than the Ltac-based entailer.

5.2.3 lseg Entailment Example

The addition of the theory of list segments does improve the generality of the MIRRORSHARD-based solver, compared with the Ltac-based solver. Here is a simple example of an entailment that the lseg lemmas enable MIRRORSHARD to solve, and that the Ltac-based solver cannot (at time of writing) solve.

\[
\forall \text{sh contents } y .
\]
\[
y \neq \text{nullval } \rightarrow
\]
\[
lseg \text{lsh contents nullval } y \vdash (!\text{False}) \&\& \text{emp}.
\]

This corresponds with “well-formedness” lemma number 2 in the Navarro Perez paper. Knowledge of this fact about list segments - essentially, that it is impossible to have a nontrivial (nonempty) list segment that starts at null - renders this lemma solvable. With MIRRORSHARD’s and the Ltac entailer system’s cancellation heuristics alone, this would not be solvable.

6 Results

Here are the results of our testing of the MIRRORSHARD-based entailment solver in the context of a simple but realistic program - reversing a linked list - in VST. The specific separation logic entailments we tested the solvers on can be found in the Appendix (8.4). The measurements involve using the solvers to solve these entailments, or to simplify them as much as possible. For these entailments, both solvers yield the same result (solution, or simplification to essentially the same result).

---

44This might be done, for example, by reifying the lemma partially applied with placeholder variables, then later replacing those variables with the actual arguments to which the lemma will be applied.

45The code for this program can be found in the reverse.c and verif_reverse.v files in the VST subversion repository: [https://svn.princeton.edu/appel/vst/progs/](https://svn.princeton.edu/appel/vst/progs/)
In order to take timing data, we used the Coq `Time` command, taking the mean across three trials. In order to take memory usage data, we used the command `/usr/bin/time -v` on Coq as it compiled a file containing only the entailment and its required imports; we monitored Coq’s peak memory usage, and averaged this across three trials as well. These results were taken on the author’s computer, a quad-(physical)-core Intel i7 notebook computer with 16Gb of RAM. The computer was running 64-bit Ubuntu 14.04 GNU/Linux; the Coq version used was 8.4pl3. For the Ltac entailment results, the `entailer!` tactic was used; for the `MIRRORSHARD` results, our `rcancel` tactic was used; for the `MIRRORSHARD` results excluding reification time, we used the tactics `pose_env`, `reify_derives`, and then `mirror_cancel_default`, timing only the last step.

Because of the inefficiency of our current reifier, our `MIRRORSHARD`-based solver is much slower than it will eventually be. Therefore we have included the “`MIRRORSHARD` Cancel Step” column in our timing data. This column leaves out reification, showing the time for just the cancellation step in `MIRRORSHARD` (including applying provers and lemmas). This statistic should be seen as an upper bound on the maximum performance we might hope to achieve from improved reification; using OCaml tactics for reification, we should be able to get close.

These results should be considered preliminary; an immediate area of future work will be to replicate them in a more controlled setting.

### 6.1 Time

<table>
<thead>
<tr>
<th>Entailment</th>
<th>Ltac Solver</th>
<th><code>MIRRORSHARD</code> Solver</th>
<th><code>MIRRORSHARD</code> Cancel Step</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unfold_entail</code></td>
<td>2.240s</td>
<td>3.430s</td>
<td>0.854s</td>
</tr>
<tr>
<td><code>while_entail2</code></td>
<td>1.568s</td>
<td>2.178s</td>
<td>0.420s</td>
</tr>
</tbody>
</table>

### 6.2 Space

<table>
<thead>
<tr>
<th>Entailment</th>
<th>Ltac Solver</th>
<th><code>MIRRORSHARD</code> Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unfold_entail</code></td>
<td>635,277Kb</td>
<td>748,628Kb</td>
</tr>
<tr>
<td><code>while_entail2</code></td>
<td>627,763Kb</td>
<td>678,071Kb</td>
</tr>
</tbody>
</table>

### 6.3 Discussion

Based on the above results, it seems clear that we are not yet reaping the full performance benefits of reflection. It is worth reiterating that these metrics -

---

46 For this we used the maximum resident set size; since the memory consumed by Coq during these trials never came even close to filling my physical RAM, this should be equal to the maximum amount of memory consumed. This is a crude metric, as Coq keeps much more in memory than just the entailment being solved, but it should give some indication of the two systems’ relative space efficiency.

47 The VST tactic `go_lower0` was used as a preprocessing step to the `MIRRORSHARD` trials. This tactic converts entailments from a high-level form into ordinary separation logic. This tactic runs quickly and should not have a large contribution to the times listed below; it was not included in the timing analysis.
particularly the space metrics - are crude; one future area of research will be to use more rigorous, fine-grained techniques to more accurately assess the time and space consumption of the two entailment systems.

The timing results are not especially surprising, because our reification tactics are currently artificially slow. Once the tactics are reimplemented in OCaml, we may be able to get the time for reification down to ~0.1s \[13\]. For this reason, it is worth focusing on the timing data that leave out the reification step, which are extremely encouraging. Shaving even a second off solving an each entailment leads to a much better experience for the user, who must solve many such entailments over the course of verifying a program. With reified symbolic execution, these savings could be further amplified. The space results were more unexpected. We believed that, because of the smaller proof terms involved in proof by computational reflection, the MirrorShard-based system would use noticeably less memory than the Ltac-based one. However, as these results show, it actually consumes more memory!

The reasons for this will need to be investigated further, but here is a hypothesis: it could be that, for these small entailments, the memory savings of MirrorShard over Ltac, which should be proportional to the size of the proof terms Ltac generates, are outweighed by a constant memory overhead involved in using MirrorShard. That is, if we model the memory usage as some constant overhead involved using the system, plus a cost proportional to the complexity of the proof, \[48\] we have

\[
\text{MemoryConsumed}_{\text{MirrorShard}} = C_{\text{MirrorShard}} + \text{Complexity} \times D_{\text{MirrorShard}},
\]

and

\[
\text{MemoryConsumed}_{\text{Ltac}} = C_{\text{Ltac}} + \text{Complexity} \times D_{\text{Ltac}}.
\]

It may be that \(D_{\text{MirrorShard}} < D_{\text{Ltac}}\), but that \(D_{\text{MirrorShard}} > C_{\text{Ltac}}\). In this case, for relatively simple entailments like these, the \(C\) term predominates, leading to better memory performance for the Ltac entailment. If this is the case, MirrorShard will have better memory performance than Ltac for sufficiently large entailments, in which the \(D\) term will predominate.

In order to test this hypothesis, we will need to run MirrorShard on more complicated entailments whose solution in Ltac generates larger proof terms. It is also possible that future versions of MirrorShard will improve its memory usage, rendering it more efficient even for these small entailments.

### 7 Related Works

The idea of using separation logic to verify the correctness of imperative code is not a new one. This was first shown to be practical with the Smallfoot system \[6\], an implementation of a system like VST, but on a smaller scale and using a new language rather than an established one. Though groundbreaking, Smallfoot had some important limitations. First, it was capable only of shape analysis; it

\[48\] By “complexity” I don’t have any specific metric of proof complexity in mind. This is meant as a vague, speculative hypothesis that will have to be formalized and tested in further research.
could not prove the more detailed specifications of functional correctness that, for instance, VST can. Also, Smallfoot was developed in ML rather than a proof assistant, and is not itself formally verified to be correct, whereas VST is.

Bedrock \cite{9} is a newer project in a similar spirit to Smallfoot: it allows for specifying and proving properties about programs written in its own specialized (imperative, systems-level) language. Like VST, it is written and proved correct in Coq, and can verify functional as well as shape correctness. Bedrock aims to provide a platform on top of which a suite of powerful verification technologies can be built. Bedrock is another system pioneering the use of computational reflection for separation entailment solving; it was the original client of MirrorShard. Bedrock uses MirrorShard somewhat differently, however; the techniques and workarounds we describe above (Section \ref{5}) are unique to VST.

Another instance of using computational reflection for verifying imperative programs is VeriSmall \cite{5}, a reimplementation of a subset of the functionality of Smallfoot in Coq that uses computational reflection as its proof strategy. Some of the insights from VeriSmall could be combined with the techniques in this paper, to build a system for reified symbolic execution in VST.

JSCert \cite{7} is another project in a similar vein to VST and Bedrock. It has more or less the same aim as VST, but uses Javascript instead of C as its language of choice.

VST builds on a model of the C language built for CompCert \cite{12}, a formally verified C compiler written in Coq. CompCert’s translation from source code to machine code is proven correct. Bridging the very different semantics of Coq and C was a significant undertaking, and VST benefits greatly from being able to build on the work in this area done by the CompCert team.

Finally, one important example of VST’s use in verifying realistic C programs is Andrew Appel’s current work in verifying the SHA256 hash function \cite{2}. This work pushed the limits of VST and Coq, and helped to show how important the performance issues we discuss in this paper are, providing important motivation.

8 Conclusion

This paper describes the process of replacing VST’s core entailment subsystem with a new one based on the reflective solver, MirrorShard. We have successfully done this, as well as demonstrated the advantages that the new, reflective entailment has over the previously-existing one. We hope that we have provided a road-map for others seeking to undertake similar projects in the future, particularly in terms of using reflective solvers such as MirrorShard with pre-existing codebases not originally designed with reflection in mind. Additionally we hope that we have shown that such projects can be highly motivated: the potential performance gains from reflection are nontrivial, even if more work is needed in our case to achieve them.

\footnote{This can be found in the sha subdirectory the VST repository: \url{https://svn.princeton.edu/appel/vst/sha}}
Particularly since general, powerful solvers such as MirrorShard have already been written, we believe that the effort it takes to adapt these solvers for use within larger systems is more than justified. It is our hope that papers such as this one, which document real-world experiences with integrating reflective solvers, will reduce that effort by serving as a guide for others who want to undertake similar projects in the future.

8.1 Integration Into VST

How does our work fit into the big picture of VST as a whole? As it turns out, it fits quite neatly, requiring no substantial changes to the overall architecture of VST, beyond the entailer. VST's entailer subsystem is reasonably self-contained, and a "full-stack" call to MirrorShard (that is, refication, then cancellation, then possibly reflection) is intended to perform exactly the same function as the original entailer. As a result, the new MirrorShard-based entailer system is more or less a "drop-in" replacement for the old entailer.

As mentioned above, the old entailer tactic (and other tactics associated with it) has been left in VST's codebase. In part, this is because it is useful to have an entailer system to "bootstrap" MirrorShard's lemmas. We currently use the entailer to prove many of the data-structure lemmas that make MirrorShard more useful.

Currently many higher-level tactics in VST - such as the forward tactic, which moves symbolic execution forward one step - still make use of the old entailer to simplify and solve the entailments they generate. In the long run, replacing these tactics' calls to the old entailer with calls to MirrorShard will yield great practical benefit, as it will enable the user to reap the benefits (in terms of speed and space) of the reflective approach by default. It is also a prerequisite for reified symbolic execution (Section 5.2.2).

8.2 Goals Achieved

In this project, our primary goals were to improve performance (as measured by the time consumed by calls to the solver) and to improve generality (measured by the breadth of goals that the solver can discharge, and the extent to which it is able to simplify the goals that it cannot). As demonstrated by our results, we believe that both of these can be achieved.

We have also demonstrated that these improvements have very real ramifications on the usability of VST on realistic programs, particularly for programs that make use of linked list data structures. We believe that, in most cases, adding lemmas about new data structures to MirrorShard's database of hints is all that will be necessary to expand these benefits to other kinds of programs. Even without these data-structure-specific improvements, the increase in entailment-solving performance that we have been able to achieve should be a boon to users of VST. The tactics for performing symbolic execution in VST are slow, limiting users' ability to experiment with different approaches and iterate quickly. These tactics spend much of their time - and, hence, the user's time -
solving entailments. For this reason, as we have shown, the benefits to the user are very real, and can be substantial.

8.3 Lessons Learned

Primarily we have seen how powerful reflective solvers such as MirrorShard can be. The difference in performance between MirrorShard and VST's Ltac-based entailment, as we have shown and emphasized above, can be quite remarkable. Reflection represents one frontier in terms of thinking about new proof techniques in Coq. We believe, based on our work with MirrorShard, that there is reason to be optimistic about reflection, and that it might in the future lead to far more efficient proof-construction - and to far more efficient proofs, in terms of the space they consume. The importance of this last should not be understated: Andrew Appel's project on verification of SHA256 has frequently run up against memory limitations (most frustratingly, the inability of 32-bit versions of Coq to address more than 2GB of memory); as a workaround, many lemmas have had to be split into smaller parts, not because this made the most sense structurally, but because it was the only way to keep the proofs' memory usage within the limits of what Coq would allow. Having to change abstractions and restructure code solely for performance concerns is undesirable; using a reflective solver might be another way to avoid excessive memory usage, without having to restructure and break apart lemmas.

We have also seen that setting up a system like VST for use with a reflective solver like MirrorShard is nontrivial, in large part because of the need to write code for reification. In the future, we hope that this process could be improved, either through the construction of more general reification tactics that can be used to reify terms from different systems, not just VST; or through the creation of best-practices and code-templates that can serve as a guide to those building reifiers, saving them from having to do so more or less from scratch. There may need to be differences between reifiers for different systems: the kinds of terms that will need to be supported will differ, perhaps significantly, as will the sorts of workarounds that will need to be deployed to overcome whatever limitations the reified form presents. Extensions to Ltac, such as Mtac\textsuperscript{[19]}, which make it easier to write correct tactic code, may have a role to play in easing the construction of reflection procedures. When efficiency is needed, tactics written in OCaml will also have an important role.

The inability to express dependent types in reified form is a particularly troublesome limitation, and it may be a difficult one to surmount, as it might not be possible to specify a type for a reflection procedure for dependently-typed terms in a type-system of the kind that Coq has\textsuperscript{[13]}. It might be feasible, however, to support certain special cases of dependency - such as algebraic datatypes, of which standard Coq lists are an example - in reflection. Since many of the instances of dependent types we encountered in our work fall into such special cases, and do not require all the generality of full dependent types,\textsuperscript{50}

\textsuperscript{50}Doing so, if possible at all, is very difficult and is an open research problem.

35
this alone could be a significant improvement.

In our view, our work in bridging VST and MirrorShard reveals the power of Coq’s approach to proof-construction: namely, the separation of the tactic system for building proofs from the term language in which the proofs themselves are expressed. This enables tactics to be swapped out for each other with relative ease, since a clean, API-like boundary already exists between the proof-construction system and Coq’s internal workings: the constructors and sub-lemmas used to build the proofs themselves. This significantly simplified the process of replacing the original VST entailment with the MirrorShard entailment solver, as a well-defined form for expressing entailments in Coq already existed; all that was necessary was to create a new tactic that could manipulate syntactic objects of this form (in our case, a tactic that reified them and then passed them into MirrorShard’s canceler).

Finally, we have also seen the power of the MirrorShard’s modularity. By taking advantage of MirrorShard’s extension systems (particularly, its ability to work with user-supplied lemmas and solvers), we were able to effectively adapt it to our purposes. In fact, MirrorShard’s extensibility is actually somewhat limited: as discussed previously, user-supplied plugins can only be used and combined in very specific ways, in a specific, predefined order. Nonetheless, for our purposes, its rather rigid structure for making use of the lemmas and provers we supplied it was sufficient (aside from needing our hack to make MirrorShard work with lemmas with dependently-typed arguments, discussed above). This flexibility is important: since the core code of reflective solvers is generally quite complicated (this is certainly the case for MirrorShard), giving the user the ability to extend the solver’s behavior without having to understand the solver’s internals in full detail is almost a necessity.

8.4 Future Work

Of course, much more can be done to improve our integration of VST and MirrorShard. Throughout this paper, we have outlined several potential optimizations to our system for using MirrorShard with VST that could potentially improve performance. Most of these have to do with reification, the most expensive part of the process, and also the most complicated to implement. Writing these tactics in OCaml, or otherwise minimizing the number of checks Coq performs during reification, can be useful in this regard.

As we have also mentioned, adding more lemmas to describe properties of new data structures can also be fruitful. For this paper we were able to draw on Navarro Perez’s work in formalizing linked list segments; adding support for other kinds of structures may be somewhat more difficult, as we may have to formalize them ourselves, including determining what the appropriate lemmas to prove and give MirrorShard are (as well as determining whether these lemmas should be applied on the left or on the right). If we are to expand the lemmas list, we may also need to come to terms with the performance ramifications of adding more lemmas. In particular, if scalability proves to be a problem for the lemmas list, we may need to consider having multiple lists
for different purposes, or order the list more carefully so that the lemmas are attempted in a particularly productive order (the order in which the lemmas are applied depends on their order in the list).

We may, at some point in the near future, need to upgrade to a newer version of MirrorShard, as development of MirrorShard has largely ceased. A new version, MirrorCore, is under development; additionally, custom forks of MirrorCore are being used in the Bedrock project [13]. Among other benefits, MirrorCore will have a more flexible rewriting engine, enabling rewriting many terms to one; this will enable MirrorCore to make use of all of Navarro Perez’s linked list lemmas, enabling it to reason about lists in a more complete way. The reified format of MirrorCore is somewhat different from that of MirrorShard, so some parts of our reification procedures will need to be rewritten. Our lemmas and provers may need to be stated differently as well. MirrorCore’s performance characteristics may also be somewhat different from MirrorShard; though ideally performance will be better in all cases, we may need to deal with corner cases where performance worsens with the upgrade.

It is our hope that this project to integrate the MirrorShard solver into VST is not the last of its kind: VST could benefit from the automated theorem-proving capabilities provided by other solvers. One particularly exciting possibility is that of integrating a Satisfiability Modulo Theories (SMT) solver, capable of automatically deciding problems that can be stated in terms of the satisfiability of conjunctions of equations written in (efficiently decidable) underlying theories. The utility of such solvers for formalizing programs has already been demonstrated – for instance, in platforms like Microsoft’s Boogie/Dafny [11], or in the Liquid Types project (which adds refinement typing to Haskell and ML, backed by SMT) [18]. Integrating an SMT solver into VST would present a different set of challenges from the integration of MirrorShard. Similar to MirrorShard, a reification-like protocol would have to be created for communicating the goals to be solved to the SMT solver; in the process, we would have to determine which types of goals would be useful to apply SMT to (one suggestion has been using SMT to help resolve goals involving array indexing [1]). Ideally this would be an extensible format, able to encompass new types of proof goals if new theory-solvers are added to the SMT backend. The equivalent of reflection (i.e., returning results back as Coq terms) would present its own set of challenges, as the SMT solvers that currently exist are not formally verified. Therefore, the SMT solver used with VST would need to be able to supply Coq with a “proof witness” – data that Coq can then use to verify the correctness of the SMT solver’s solution (since SMT’s theories are generally in NP, this can be done quickly). However, not all SMT solvers have the capacity to produce detailed witnesses of the kind we would need.

Finally, we believe there are many other exciting applications of reflective solvers like MirrorShard/MirrorCore, beyond the context of VST. Though MirrorShard and MirrorCore currently focus on solving separation logic entailments, there are many other potential uses for such reflective solvers – and, indeed, the long-term goals of the MirrorShard project include providing a
more general platform for reflective solving [13]. In many other areas of theorem proving, the same performance issues we encountered in VST can arise, and a platform like MirrorShard might be useful in remedying them. Regardless of the context in which reflection is being used, a similar approach to the one taken in this paper will need to be followed. Though the details will depend on the specifics of the reflection system being used and the problem to which it is being applied, it will still be necessary to define a reification procedure, applicable lemmas, and something similar to solver procedures. We hope, therefore, that this paper might be able to serve as a guide to those working with MirrorShard (and, to a lesser extent, other reflective solvers), even in seemingly unrelated domains.

Reflective solving is, in many respects, a somewhat unexplored territory in theorem proving; though not a recent development, it has not been put to widespread practical use. The advent of MirrorShard can help to lower the bar to using reflection as a solving technique, opening reflection up to many potential applications. We hope that this paper will be just one of many explorations into the possible uses for reflective solving.
References


[3] Andrew W. Appel. *Program logics for certified compilers*. Cambridge University Press, 2014. url: http://books.google.com/books?hl=en&lr=&id=ABkmAWAAQBAJ&oi=fnd&pg=PR10&dq=%22applications%22+should+read+Chapters+1%E2%80%932+&ei=vs1ty7N4M4vZtwfYsIDQ&oq=%22applications%22+should+read+Chapters+1%E2%80%932+&gs_sm=1&ved=0CCkQ6AEwAg


Appendix: Entailments Used to Test the MIRRORSHARD Entailer System in VST

The following were the entailments on which we tested our entailment solver, comparing it to VST's original solver.

**Lemma unfold_entail**

name → name → name → ∀ (sh: share) (contents: list val), writable_share sh → ∀ (cts1 cts2: list val) (w v: val), isptr v → ∃ (a: Share.t) (b: val), PROP (contents = rev cts1 ∪ cts2) LOCAL (tc_environ Delta2; 'eq eval_id_w; 'eq eval_id_v) SEP (lseg LS sh cts1 w nullval); (lseg LS sh cts2 v nullval) ⊢ local (tc_expr Delta2 (Etempvar_v (tptr t_struct_list))) && ('field_at a t_struct_list_tail b) (eval_expr (Etempvar_v (tptr t_struct_list)) * TT).

**Lemma while_entail2**

name → name → name → name → ∀ (sh: share) (contents: list int), PROP () LOCAL (tc_environ Delta; 'eq (eval_id_t) (eval_expr (Etempvar_p (tptr t_struct_list)))); ('eq (eval_id_s) (eval_expr (Econst_int (Int.repr 0) tint)) SEP (lseg LS sh (map Vint contents)) (eval_id_p) 'nullval) ⊢ EX cts: list int, PROP () LOCAL ('eq (Vint (Int.sub (sum_int contents) (sum_int cts))) (eval_id_s)) SEP (TT; (lseg LS sh (map Vint cts)) (eval_id_t) 'nullval).