Computation Improves Interactive Symbolic Execution

Josiah Dodds

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew Appel

November 2015

Abstract

As it becomes more prevalent throughout our lives, correct software is more important than it has ever been before. Verifiable C is an expressive Hoare logic (higher-order impredicative concurrent separation logic) for proving functional correctness of C programs. The program logic is foundational—it is proved sound in Coq w.r.t. the operational semantics of CompCert Clight. Users apply the program logic to C programs using semiautomated tactics in Coq, but these tactics are very slow.

This thesis shows how to make an efficient (yet still foundational) symbolic executor based on this separation logic by using computational reflection in several different ways. Our execution engine is able to interact gracefully with the user by reflecting application-specific proof goals back to the user for interactive proof—necessary in functional correctness proofs where there is almost always domain-specific reasoning to be done. We use our "mostly sound" type system, computationally efficient finite-map data structures, and the MirrorCore framework for computationally reflected logics. Measurements show a $40\times$ performance improvement.

Acknowledgements

Thank you to my adviser Andrew Appel for teaching me how to be a researcher and a graduate student. I will always aspire to be like Andrew as a mentor and as a leader.

My advisers at KSU, John Hatcliff and Robby, for showing me what can be done in this field, and inspiring to continue my education, as well as collaborating on my early work at Princeton.

My collaborators, Jesper Bengtson and Gregory Malecha, for making difficult concepts much easier and working hard to make all of our research successful together.

All of the friends I made at Princeton, for making light of the sillier parts of being a graduate student.

My parents, for being great parents

Kjersti and Jaxon Dodds, for always providing happiness and perspective

To Kjersti, Jaxon, Mom, Dad and Hannah.

Contents

Chapter 1

Introduction

The C programming language is one of the most commonly used languages for writing critical software. C is used for operating systems, embedded systems, and cryptography, as well as countless libraries that need to have high performance while being compatible with as many languages and systems as possible. The world depends on more systems every day, and many of those systems have C code at their core.

In general, the code we depend on works well, and there is plenty of noncritical code that can fail without causing harm. But when critical code fails, it can be catastrophic. Space projects that cost hundreds of millions of dollars have failed due to problems with software [19, 26], radiation device software has caused overdoses in cancer patients [40], and millions of cars have been recalled to fix software bugs [1]. Those scenarios are only caused by bugs; far more systems can fail when a malicious user attacks them. As an example, a brand of refrigerators was found to have been taken over to send spam e-mail [41].

To believe that a C program will run correctly we must first believe a number of other things. There are a number of components that we will need to be able to *trust* in order to trust our C program. We need to fully understand the program itself, and believe that it will do what we want it to, as well as believing that our model of the programming language reflects what C programs actually do. We also need to believe that when the C code is translated into machine language, the generated machine language has the same behavior as the original program. Finally we need to believe that the machine executing the machine language is doing the correct thing. We call the collection of things we must trust the *trusted computing base*. In general, the larger the trusted computing base is, the harder it is to be confident that the program is correct.

The need for trust follows a chain. It starts with the C program at the top, the most human understandable part of the chain, and moves down to the actual execution of the machine. In order to make the trusted computing base as small as possible, we must examine and minimize the base of every link in the chain.[1].

[1] If we cannot trust the C code or the C compiler, we can review or verify the assembly-language program; this approach is often used

Proof assistants are tools for stating theorems, and then writing and checking proofs of those theorems. If you want to believe a pen and paper mathematical proof you first need to understand the statement of the proof, which is open to ambiguities based on (often unstated) assumptions made by the writer, and then believe every single step of the proof itself, even steps where the proof author says something vague like "clearly" or "obviously". This can be an enormous burden on the person that needs to understand how the proof was done; new proofs can use new techniques that are impenetrable to anyone but experts in the specific domain of the proof. This is a real problem, because often the theorem proved is useful to many people even if the proof itself is difficult for them to understand. If you want to use a pen-and-paper proof that you can't understand, you must trust the process of peer review.

A proof assistant, on the other hand, forces proof statements to be written in well defined logic, making theorem statements completely unambiguous. This is not to say that it is impossible to make mistakes in specifications, it is still possible to write meaningless or incorrect specifications. What it does mean is that once a specification is written down, it has a single meaning with no ambiguity. The proof assistant often provides tools for building proofs. These tools write proofs using well defined inference rules. The tools for building proofs don't need to be trusted. All that needs to be trusted is a part of the proof assistant that checks the proof when it is created. This is analogous to not needing to trust the person writing a pen and paper proof, as long as you can check the proof that they wrote. In summary, if a proof is stated and proved in a proof assistant, to believe the proof we must trust:

1. The statement of the theorem being proved, and
2. the correctness of the checker.

Regarding the correctness of the checker (the kernel of the proof assistant) the best place to look in order to believe the proofs is to peer review. In this case though, it is not a few anonymous reviewers that are looking at an individual proof, it is the entire community that is interested in the correctness of the proof assistant. This is an advantage of concentrating trust. The fewer things that need to be trusted, the more attention they can get.

A verified compiler gives a proof that if it is given a valid C program, it will generate assembly code with the same behavior as the C program. This proof is checked by a proof assistant, meaning that its trusted base is the proof checker and the statement of the theorem. To be able to use the theorem effectively though, we need to be able to show that the C program will always have a valid execution.

The results of the verified compiler tells us that the compiler won't change the behavior of the program, but this is only really useful to us if our program is doing the right thing in the first place. It can be very

in practice, but it is difficult and tedious.

hard to determine what a C program is doing just by examining the source code. The C language is relatively low-level, meaning that a program will need more details about how the computation is done than a higher level language. The high level of control over the details of execution is a large part of why C continues to be popular over 40 years after it was created. It allows programmers to write more efficient programs, but it can make it much harder to understand what the program is actually doing.

To create the most convincing statement possible, we write a mathematical specification of what the program should do. This mathematical model lacks the complexity that a real program needs to work efficiently on a computer. The specification is a new link in the chain above the C source code. A *Program Logic* is used to prove that the result of executing the C program meets the specification. This program logic can be implemented in a theorem prover, and proved correct using the same specification of the C language that the verified compiler uses. This completes the chain of verification from specification all the way to assembly language.

The combination of the tools mentioned here gives proof that the assembly code that the machine executes has the same behavior as an abstract mathematical specification. To believe that the proof is correct, all we need to trust is the mathematical specification itself and the theorem prover's proof checker.

There is still a significant amount of work to do if you wish to prove a program correct. First you must create an accurate specification — which can be difficult, especially for complex programs. Then it can be a very involved process to prove a program meets a specification using a program logic. Program logics implemented in proof assistants are often *interactive*. To build a proof in an interactive program logic, you examine a proof state and then perform an action to manipulate that state. The action will result in a new proof state, which you can repeatedly manipulate until the proof is completed.

Program logics that are proved correct in proof assistants can be used interactively inside of the prover. A sound logic is only useful if it is applied correctly. In a pen-and-paper program logic, correct application of the logic is one more thing that must be trusted. This is not the case in a logic implemented inside of a proof assistant. In a proof assistant, the application of the logic to a program can also be checked, removing it from the trusted computing base.

One of the biggest problems that implementations of complex program logics in proof assistants can run into is that they are slow. This means that the time between giving input and receiving a new proof state can be long. Writing a proof in a program logic is already difficult work requiring intimate knowledge of the program, the programming language, and the logic. Having to wait between each step in a proof multiplies the difficulty of writing a proof. This is because proof development is often experimental in nature. The first step is to make a good guess at a specification. If the proof gets stuck, either the specification or the program needs to change. When those change, the proof will need to start over from the beginning. A slow program logic can drastically limit the number of manipulations of the program or the specification that can be done in any given stretch of time. If the time is long enough, it can become difficult the person doing the proof to remember what they were working on when they last made progress on the proof.

This thesis shows how a proof assistant implementation of a complex program logic in for proving the correctness of C programs can be made significantly faster without any negative impact on the proving experience. This results in a logic that is significantly more intuitive and usable, and is a vital step in making the program logic a widely usable tool.

Many of the parts discussed so far are existing work: The proof assistant we have chosen to use is Coq, Leroy's CompCert [29] is a C compiler implemented and proved sound in Coq, and Appel et al. [5] have created the Verified Software Toolchain (VST) to give a proved-correct chain from specification language to assembly including a specification logic and a program logic to relate the specification to a CompCert C program.

Contributions This thesis discusses the modification of a program logic to improve usability and speed up the application of the logic. Chapter 3 presents a type system that operates within the logic. This typechecker automates many of the steps normally required when using the logic, giving a minimal goal for the user to prove if it can't proceed automatically. Chapter 4 discusses how the logic can be made more efficient by organzing the statements in the logic. This increased organization allows us to use more efficient data structures, improving performance and simplifying the statement of our logic rules. Finally, we completely replace the automation responsible for applying our logic to a program with *reflective* automation. This type of automation is able to take full advantage of Coq's fastest features, giving us a resulting proof system that typically runs at least 40x faster than the previous ones and works on all Verifiable-C non-call assignment statements (`x := e`), meaning it makes progress on a single basic block at a time. The new system is fully compatible with previous tactics, meaning that in places where it is not usable, existing tactics can still make progress on the proof.

Chapter 2

Computation in Coq

There are two main categories for objects in Coq. The first is Prop, which is the sort for propositions in Coq. These propositions can include quantifiers ($\forall$, $\exists$) and implication. New predicates can be defined as inductive relations or built up using other predicates.

The second sort of Coq object is Type which contains data structures that can be computed on by Gallina, Coq's built in functional language. These data structures can be defined inductively and Gallina has the ability to define and match on these inductive data structures. The type boolean, for example can be defined:

```
Inductive boolean : Type :=
| true
| false.
```

So boolean is a type that can be constructed by either true or false; and when given a boolean, a function can decide which of the two it is. This can be contrasted with the definition of True and False, which have the sort Prop instead of Type. Prop is a sort, not an inductively defined type in Coq, so it can't be matched against. Instead the definitions of True and False are

```
Inductive True : Prop :=
 | I : True.

Inductive False : Prop.
```

along with an infinity of other propositions, such as $\forall x : Z, x > 5$. Even though (one might think) this is False, it is not the same proposition.

Gallina programs cannot pattern-match over propositions; they can compute only over inductive data structures, and Prop is not inductive. That doesn't mean that it is impossible to reason about things of type Prop in Coq. Instead of using Gallina to operate on Prop, Coq has tactics, which exist almost exclusively for that reason. The *tactic language*, Ltac, can parse (pattern-match on) propositions. A tactic in Coq is a program that manipulates a proof state while generating a proof object recording its activity. A Coq proof state is a goal (e.g., Prop prop to be proved), along with a number of quantified variables of any type, and a number of hypothesis of type Prop. New tactics can be created by combining other tactics using the **Ltac** language.

This design means that tactics don't need to be sound. If they produce a proof that checks, it doesn't matter how the proof was created. Tactics might do a lot of work to create a small proof term. For example a tactic might perform a backtracking proof search and takes numerous wrong paths before finding a solution. This tactic will have performance that matches the complexity of the proof search. A tactic that continuously makes modifications to a proof state, with each modification recorded to the proof object, is likely to experience performance worse than what you would expect given the number of operations performed. This is because the proof term that the tactic builds grows with every operation applied. In these cases, the overhead of keeping and modifying the proof object (which can take up substantial amounts of memory) can lead to very slow tactic performance.

Logics such as VST's verifiable C program logic are in Prop. Logics in Prop are said to be shallowly embedded. Instead of a "deep embedding" — inductively defined syntax separated from any semantic interpretation–a shallow embedding defines each new operator as a semantic predicate in the language of propositions. Some systems such as Appel's VeriSmall [3] are deeply embedded, meaning they define their own syntax, along with a denotation that gives meaning to that syntax in Coq's logic.

Shallow and deep embeddings have tradeoffs in two areas:

1. interactivity, or how convenient it is for users to interact with the logic, generally in the way they would expect to be able to interact with Coq's logic, and

2. automation, or the ability of the logic writer to provide the user of the logic with tools to efficiently reason about the logic.

A shallow embedding will naturally take advantage of Coq's proof automation. A language for creating and combining tactics called Ltac can be used to write decision procedures about the logic without requiring soundness proofs for those procedures. A deep embedding, however, will be much harder to interact with. To have meaning as a proof goal, a deep embedding will need to be wrapped in its denotation function. Then every operation on the deeply embedded assertion must be proved sound with respect to the denotation function. Take Coq's rewrite tactic as an example. In this tactic if we wish to write a symmetry lemma about an equality named add_symm, we use the **Lemma** keyword and give the type of add_sym:

```
Lemma add_symm : ∀ (a b : nat), a + b = b + a.
```

nat in Coq is the inductively defined natural numbers.

If we are able to give a definition for add_symm with ∀ a b, a + b = b + a, that is a proof of the fact. Because there are generally difficult dependent types involved in constructing the proof term correctly, we use tactics to manipulate a proof state until we have successfully created a proof object.

If we wish to use the tactic we proved to help solve another goal. In Coq's interactive proving mode, goals are presented as hypothesis (things above the line) that you know, and the conclusion that you are trying to prove (below the line).

```
a : nat
b : nat
c : nat
======================
a + (b + c) = a + (c + b)
```

In this case we know that we have Coq variables a, b, and c, and that they are all natural numbers. We are trying to prove a + (b + c) = a + (c + b). we can use the tactic rewrite (add_sym b c) to transform the left side to match the right side:

```
a : nat
b : nat
c : nat
======================
a + (c + b) = a + (c + b)
```

and then use the reflexivity tactic, which checks to see if both sides of an equality are equal, solving the goal if it discovers that they are. In this case they are syntactically equal, so the goal is solved. We supply the arguments b and c to the tactic because it is ambiguous where to rewrite the lemma, and Coq might guess wrong unless we tell it.

If we wish to solve this goal computationally we can create a deep embedding, along with a denotation function for nat, addition, and equality

```
Inductive expr' :=
| num : nat → expr
| add : expr → expr → expr.

Inductive expr :=
| eq : expr' → expr' → expr.

Fixpoint expr'_denote e:=
match e with
| num n ⇒ n
| add e1 e2 ⇒ expr_denote e1 + expr_denote e2
end.

Definition expr_denote e :=
match e with
| eq e1 e2 ⇒ expr'_denote e1 = expr'_denote e2
end.
```

This syntax mirrors the syntax of a Coq goal, it means the same thing but we can compute on it inside of a Gallina program. We define it in two levels (expr and expr') so that every expr has a denotation. This is not a requirement of a deep embedding, but it simplifies the other steps.

When we apply the denotation function to the syntax and get a Prop back, we say that the goal has been reflected. The process of *computational reflection* mirrors a shallow embedding with a deep embedding, does work on the deep embedding, and reflects the result back to a shallow embedding. To perform the work, we can write a new symmetry lemma, this time on the denotation of the deep embedding:

```
Lemma add_sym' : ∀ a b,
expr'_denote (add a b) = expr'_denote (add b a)
```

If the denotation of a deep embedding is equal to a shallow embedding, we say that the deep embedding is a *reification* of the shallow. To use add_sym we need a reified version of our original goal. To reify the goal, we create a tactic called a reifier. This must be done using a tactic because tactics have the ability to match Prop, while Gallina does not. The following is a reification of our original goal:

```
a : expr
b : expr
c : expr
=====================
expr_denote (eq (add a (add b c)) (add a (add c b)))
```

The lemma we wrote doesn't match the syntax of what we need to prove so we can't use the rewrite tactic. We could unfold the definition of expr_denote and expr_denote' in our goal, but then we would lose the deep embedding. Instead, if we want this functionality, we need to write a *Coq function* that does the rewrite and prove that function sound with respect to the denotation function (assuming we have an equality function on our expr, which isn't hard to write):

```
Fixpoint symmetry' e e1 e2 :=
match e with
| add e1' e2' ⇒ if (e1 == e1' && e2 == e2')
                then add e2' e1'
                else add (symmetry e1')
                         (symmetry e2')
| x ⇒ x
end.

Definition symmetry e e1 e2 :=
match e with
| eq ex1 ex2 ⇒ eq (symmetry' ex1 e1 e2) (symmetry ex2 e1 e2)
end.

Lemma symmetry_func_sound :
∀e1 e2 e,
expr_denote (e) = expr_denote (symmetry e e1 e2)
...
```

Now we can rewrite by (symmetry_func_sound b c) giving us:

```
a : expr
b : expr
c : expr
======================
expr_denote (symmetry (eq (add a (add b c)) (add a (add c b))))
```

and simplify. We simplify by using Coq's simpl tactic, which performs $\beta\iota$ reduction. The simpl tactic uses heuristics to try to only reduce goals far enough that they are still easily readable by a proof writer. In this case, simpl will (unseen to the user) first unfold and reduce the symmetry definition:

```
a : expr
b : expr
c : expr
======================
expr_denote ((eq (add a (add c b)) (add a (add c b))))
```

and then the denotation function, reflecting the goal back to Prop:

```
a : nat
b : nat
c : nat
======================
a + (c + b) = a + (c + b)
```

where the goal can be solved by reflexivity.

Why would we ever use a deep embedding when automation is so much work? The main reason is efficiency. An Ltac tactic adds to the size of the proof object every time it does an operation. Proof by computational reflection, on the other hand, can perform as many operations as it wants without creating any proof object at all. Instead, the proof object for computational reflection is the soundness proof for the function that is applied to the deep embedding. Although the example above did a very small amount of work in the symmetry function, in general such a function could be an entire decision procedure for a complex problem. Then we would be able to do a large amount of work on a deeply embedded proof term while hardly generating any proof object at all. In general, this will be substantially more efficient than using Ltac, especially as the size of the deeply-embedded statements grows.

Another advantage to a deeply embedded logic is that Gallina programs can be compiled to byte-code using Coq's vm_compute tactic — or even to machine code by way of OCaml in upcoming versions of Coq

— which allows it to be run significantly more efficiently, at the cost of adding a virtual machine to the trusted computing base.

The technique of computational reflection in Coq combines the interactivity of a shallow embedding with the efficiency of a deep embedding by performing as much automated work as possible on the deep embedding, and returning a readable shallow embedding goal to the user when interaction is required. Proof by reflection is not a new technique, further discussion of proof by reflection can be found in [15, 10].

Proof by reflection is the technique of using proved-sound computation on a deep embedding to make progress in a proof. To use reflection on a shallow embedding, there is first a translation, or *reification* from the shallow embedding into a deep embedding. Then a proved-sound function can be evaluated (preferably using `vm_compute` for the best performance) on the deep embedding, finally the denotation function can be evaluated, or *reflected*, resulting in a shallow embedding again. If this is done all in a single step, the user of a reflective never knows that reflection is being used. That means that the efficient, proved-sound decision procedures that can be used on deep embeddings can be used as tactics, along side other tactics and Ltac automation that might already be built up around a logic.

The process of translating from a shallow embedding to a deep embedding is known as reification. Because Gallina programs can't match on shallow embeddings, reification must be performed by a tactic. Reification leaves us with a deep embedding (or reified expression), which allows us to use proved-sound Gallina functions to progress the proof state without large proof terms. When the deep embedding is at a state that requires human work, the denotation function can be carefully unfolded, or reflected, to return it to a shallow embedding that is easy for the user to view and work with.

Deep embeddings are purely syntactic; but even shallowly embedded languages may have substantial sublanguages that are purely syntactic—defined by inductive data types rather than semantic definitions. In many program logics, for example, the programming language will not be a denotational semantics based on Gallina, but an operational semantics over syntactic program terms. Because the syntax is a deep embedding, it is possible to write Coq functions that reason about them, and then prove those functions sound (usually with respect to the operational semantics). This is a type of reflection that doesn't require reification at all, making it more efficient and allowing it to fit cleanly into the shallowly embedded program logic. One such example in VST is our typechecker which is used to show that expressions in the program successfully evaluate given the state represented by the precondition.

Chapter 3

Typechecking C Expressions

A type system is a formalization that assigns types to subexpressions of a programming language and describes when a program uses those expressions correctly according to their types. It is often possible to build a decision procedure that answers whether or not a program is well typed in a type system. This decision procedure is known as a typechecker. A typechecker is generally meant to be used as part of program compilation, which means it should be reasonably fast and operate with the program as its sole input. It restricts the number of valid programs while guaranteeing certain errors won't occur.

We have designed a typechecker specifically for efficiently and conveniently structuring Hoare-logic proofs. The purpose of the typechecker is to ensure that expressions evaluate in a completely computational manner. It is a Gallina program that either says an expression evaluates, or produces an assertion that will guarantee expression evaluation.

To see why expression evaluation is important to the logic, consider a naive Hoare assignment rule for the C language.

$$\vdash \{P[e/x]\}\, x := e\, \{P\} \qquad \text{(naive-assignment)}$$

This rule is not sound with respect to the operational semantics of C. We need a proof that e evaluates to a value. It could, for example, be a division by zero, in which case the program would crash and no sound Hoare triples could hold. The expression e might typecheck in the C compiler, but can still get stuck in the operational semantics

A better assignment rule requires e to evaluate:

$$\frac{e \Downarrow v}{\vdash \{P[v/x]\}\, x := e\, \{P\}}\ \text{assignment-ex}$$

This rule is inconvenient to apply because we must use the operational semantics to show that v exists. In fact, any time that we wish to talk about the value that results from the evaluation of an expression, we must add an existential quantifier to our assertion. Showing that an expression evaluates can require a number of additional proofs. If our expression is (y / z), we will need to show that our precondition implies: y and z are both initialized, $\mathsf{z} \neq 0$, and ~(y = int_min $\wedge$ z = -1). The latter case causes overflow, which is undefined in the C standard for division. These requirements will become apparent as we apply the semantic rules, and some of them may be trivially discharged. Even so, the proof will be required, when it was likely computationally obvious that it was unnecessary. A type system is the answer to this problem, but unfortunately C has an unsound type system, when we require soundness.

3.1 A Sound C Type System

Instead we change the typechecking game. We don't answer just "type-checks" or "doesn't typechck" because our typechecker can produce verification conditions that say "typechecks if this assertion holds". We create a type checker that computes fully, and gives simple, minimal preconditions to ensure expression evaluation. The preconditions may not be decidable, but they are left to the user of the logic to solve. We define a function typecheck_expr (Section 3.3) to tell us when expressions evaluate in a type context Δ. Now our assignment rules are,

$$\frac{}{\Delta \vdash \{\mathsf{typecheck\_expr}(e, \Delta)\ \wedge\ P[e/x]\}\, x := e\, \{P\}}\ \text{tc-assignment}$$

$$\frac{}{\Delta \vdash \{\mathsf{typecheck\_expr}(e, \Delta)\ \wedge\ P\}\, x := e\, \{\exists v. x = \mathsf{eval}(e[v/x]) \wedge P[v/x]\}}\ \text{tc-floyd-assignment}$$

The typecheck_expr is not a side condition, as it is not simply a proposition (Prop in Coq) but a separation-logic predicate quantified over an environment (Section 3.3.1). This rule will be inconvenient to apply. In general, it is unlikely that the writer of a proof will have their assertion in the form $\mathsf{typecheck\_expr}(e, \Delta) \wedge P$. Instead they will likely have a precondition P from which they can prove $\mathsf{typecheck\_expr}(e, \Delta)$. The Hoare rule of consequence is useful for this:

$$\frac{P \rightarrow P' \quad \Delta \vdash \{P'\}\, c\, \{Q'\} \quad Q' \rightarrow Q}{\Delta \vdash \{P\}\, c\, \{Q\}}$$

It allows us to replace a precondition by one we know is implied by the current precondition. We can use this so that an preconditions doesn't get littered with $\mathsf{typecheck_expr}$ assertions that don't add any information to the precondition. To do this we derive a new rule using the rule of consequence. :

$$\frac{P \longrightarrow \mathsf{typecheck\_expr}(\mathsf{e}, \Delta)}{\Delta \vdash \{P\}\, x := e\, \{\exists v. x = \mathsf{eval}(e[v/x]) \wedge P[v/x]\}}\text{tc-floyd-assignment'}$$

We use the Floyd-style forward assignment rule in our final proof system instead of the Hoare-style weakest-precondition rule. This is not related to type-checking; separation logic with backward verification-condition generation gives us magic wands which are best avoided when possible [9].

This rule asks the user to prove that the typechecking condition holds, without requiring it to be in the precondition. To apply the rule you must show that the precondition implies whatever condition the type checker generates. In this way, our typechecker allows outside knowledge about the program, including assumptions about the preconditions to the function to be used to decide if an expression evaluates. For example, when run on the expression (y/z) the typechecker computes to the assertion $z \neq 0 \wedge \neg(y \neq \mathsf{int\_min} \vee z \neq -1)$ where z and y are not the variables, but the values that result when z and y are evaluated in some environment. The assertions $\mathsf{initialized}(y)$ and $\mathsf{initialized}(z)$ may not be produced as proof obligations if the type-and-initialization context Δ assures that y and z are initialized.

The single expression above exposed a number of things that might go wrong when evaluating a C expression. A type system must exclude all such cases in order to be sound. The following operations are undefined in the C standard, and *stuck* in CompCert C:

- shifting an integer value by more than the word size,
- dividing the minimum int by -1 (overflows),
- subtracting two pointers with different base addresses (i.e., from different malloc'ed blocks or from different addressable local variables),
- casting a float to an int when the float is out of integer range,
- dereferencing a null pointer, and

- using an uninitialized variable.

If an expression doesn't do any of these things and typechecks in C's typechecker, we can expect expression evaluation to succeed. In the tc_floyd_assignment above, we use a function eval which is a function that has a similar purpose to CompCert's eval_expr relation. Defining evaluation as a function in this manner makes proofs more computational—more efficient to build and check. Furthermore, with a relation you need a new variable to describe every value that results from evaluation, because the evaluation might not succeed. Because we know that the execution of an expression will succeed, we were able to simply create a function eval_expr that computes a value from an expression with no new variable needed.

We simplify eval_expr in our program logic—and make it computational—by leveraging the typechecker's guarantee that evaluation will not fail. Our total recursive function eval_expr (e: expr) (rho: environ): in environment ρ, expression e evaluates to the value (eval_expr e ρ). When CompCert.eval_expr fails, our own eval_expr (though it is a total function) can return an arbitrary value. We can do this because the function will be run on a program that typechecks—the failure is unreachable in practice. We then prove the relationship between the two definitions of evaluation on expressions that typecheck (we state the theorem in English and in Coq):

Theorem 1 *For all logical environments ρ that are well typed with respect to a type context Δ, if an expression e typechecks with respect to Δ, the CompCert evaluation relation relates e to the result of the computational expression evaluation of e in ρ.*

Lemma eval_expr_relate :
$\forall$ Δ ρ e m ge ve te, typecheck_environ Δ $\rho \rightarrow$ mkEnviron ge ve te = $\rho \rightarrow$
  denote_tc_assert (typecheck_expr Δ e) $\rho \rightarrow$
  Clight.eval_expr ge ve te m e (eval_expr e ρ)

Expression evaluation requires an environment, but when writing assertions for a Hoare logic, we write assertions that are functions from environments to Prop. So if we wish to say "the expression e evaluates to 5", we write fun $\rho \Rightarrow$ eq (eval_expr e ρ) 5. Because Coq does not match or rewrite under lambda (fun), assertions of this form hinder proof automation. Our solution is to follow Bengtson *et al.* [8] in *lifting* eq over ρ: `eq (eval_expr e) `5. This produces an equivalent assertion, but one that we are able to rewrite and match against. The first backtick lifts eq from val$\rightarrow$ val$\rightarrow$ Prop to (environ$\rightarrow$ val)$\rightarrow$ (environ$\rightarrow$ val)$\rightarrow$ Prop, and the second backtick lifts 5 from val to a constant function in environ$\rightarrow$ val.

3.2 C light

Our program logic is for C, but the C programming language has features that make designing a Hoare logic difficult: *side effects within subexpressions* make it impossible to simply talk about "the value of e" and *taking the address of a local variable* means that one cannot reason straightforwardly about substituting for a program variable (as there might be aliasing).

The first passes of CompCert translate *CompCert C* (a refined and formalized version of C99 [31]) into *C light*. These passes remove side effects from expressions and distinguish *nonaddressable* local variables from *addressable* locals.[1] Even though CompCert does this automatically, we recommend that the user do this in their C code, so that the C light translation will exactly match the original program.

C has pointers and permits pointer dereference in subexpressions:

```
d = p→ head+q→ head
```

Traditional Hoare logic is not well suited for pointer-manipulating programs, so we use a separation logic (Chapter 4, with assertions such as $(p{\rightarrow}\mathrm{head} \mapsto x) * (q{\rightarrow}\mathrm{head} \mapsto y)$. Separation logic does not permit pointer-dereference in subexpressions, so to reason about d = p→ head+q→ head the programmer should factor into: t = p→ head; u = q→ head; d=t+u; where dereferences occur only at top-level in assignment commands. Adding these restrictions to C light gives us *Verifiable C*, which is not a different semantics but a proper sublanguage, enforced by our typechecker. This sublanguage still represents most of C light.

The C light language is deeply embedded in Coq. Its defintion starts with types. These types will occur frequently throughout this thesis. When it makes things more clear, they might also be named c_type :

```
Inductive type : Type :=
  | Tvoid: type (**r the [void] type *)
  | Tint: intsize → signedness → attr → type (**r integer types *)
  | Tlong: signedness → attr → type (**r 64-bit integer types *)
  | Tfloat: floatsize → attr → type (**r floating-point types *)
  | Tpointer: type → attr → type (**r pointer types ([*ty]) *)
  | Tarray: type → Z → attr → type (**r array types ([ty[len]]) *)
  | Tfunction: typelist → type → calling_convention → type (**r function types *)
  | Tstruct: ident → fieldlist → attr → type (**r struct types *)
  | Tunion: ident → fieldlist → attr → type (**r union types *)
  | Tcomp_ptr: ident → attr → type (**r pointer to named struct or union *)
```

[1]Xavier Leroy added the SimplLocals pass to CompCert 1.12 at our request, pulling nonaddressable locals out of memory in C light. Prior to 1.12, source-level reasoning about local variables (represented as memory blocks) was much more difficult.

```
with typelist : Type :=
  | Tnil: typelist
  | Tcons: type → typelist → typelist

with fieldlist : Type :=
  | Fnil: fieldlist
  | Fcons: ident → type → fieldlist → fieldlist.
```

Most expressions in C light contain their type. These types are not generated by a verified program, so a correct type label does not guarantee that an expression typechecks, however they do allow for a convenient structure of the typechecker, which first believes the label of subexpressions to check outer expressions, and then continues to confirm that the outer expression is correct:

```
Inductive expr : Type :=
  | Eval (v: val) (ty: type) (**r constant *)
  | Evar (x: ident) (ty: type) (**r variable *)
  | Efield (l: expr) (f: ident) (ty: type)
                                    (**r access to a member of a struct or union *)
  | Evalof (l: expr) (ty: type) (**r l-value used as a r-value *)
  | Ederef (r: expr) (ty: type) (**r pointer dereference (unary [*]) *)
  | Eaddrof (l: expr) (ty: type) (**r address-of operators ([&]) *)
  | Eunop (op: unary_operation) (r: expr) (ty: type)
                                              (**r unary arithmetic operation *)
  | Ebinop (op: binary_operation) (r1 r2: expr) (ty: type)
                                             (**r binary arithmetic operation *)
  | Ecast (r: expr) (ty: type) (**r type cast [(ty)r] *)
  | Eseqand (r1 r2: expr) (ty: type) (**r sequential "and" [r1 && r2] *)
  | Eseqor (r1 r2: expr) (ty: type) (**r sequential "or" [r1 && r2] *)
  | Econdition (r1 r2 r3: expr) (ty: type) (**r conditional [r1 ? r2 : r3] *)
  | Esizeof (ty': type) (ty: type) (**r size of a type *)
  | Ealignof (ty': type) (ty: type) (**r natural alignment of a type *)
  | Eassign (l: expr) (r: expr) (ty: type) (**r assignment [l = r] *)
  | Eassignop (op: binary_operation) (l: expr) (r: expr) (tyres ty: type)
                                      (**r assignment with arithmetic [l op= r] *)
  | Epostincr (id: incr_or_decr) (l: expr) (ty: type)
                                (**r post-increment [l++] and post-decrement [l--] *)
  | Ecomma (r1 r2: expr) (ty: type) (**r sequence expression [r1, r2] *)
  | Ecall (r1: expr) (rargs: exprlist) (ty: type)
                                               (**r function call [r1(rargs)] *)
  | Ebuiltin (ef: external_function) (tyargs: typelist) (rargs: exprlist) (ty: type)
                                                   (**r builtin function call *)
  | Eloc (b: block) (ofs: int) (ty: type)
                             (**r memory location, result of evaluating a l-value *)
  | Eparen (r: expr) (ty: type) (**r marked subexpression *)
```

```
with exprlist : Type :=
  | Enil
  | Econs (r1: expr) (rl: exprlist).
```

The most useful of these expressions are allowed by the typechecker. We currently do not allow the following: Eseqand, Eseqor, Econdition, Esizeof, Ealignof, Eassignop, Epostincr, Ecomma, Eloc, and Eparen. While this seems like a large chunk of the language, many of them are currently not supported because they are relatively new to CompCert. Many of them can be easily be replaced by statement level constructs that are currently part of Verifiable C, using only local transofrmations.

Some operations, like overflow on integer addition, are undefined in the C standard but defined in CompCert. The typechecker permits these cases.

Finally we have the statements:

```
Inductive statement : Type :=
  | Sskip : statement (**r do nothing *)
  | Sdo : expr → statement (**r evaluate expression for side effects *)
  | Ssequence : statement → statement → statement (**r sequence *)
  | Sifthenelse : expr → statement → statement → statement (**r conditional *)
  | Swhile : expr → statement → statement (**r [while] loop *)
  | Sdowhile : expr → statement → statement (**r [do] loop *)
  | Sfor: statement → expr → statement → statement → statement (**r [for] loop *)
  | Sbreak : statement (**r [break] statement *)
  | Scontinue : statement (**r [continue] statement *)
  | Sreturn : option expr → statement (**r [return] statement *)
  | Sswitch : expr → labeled_statements → statement (**r [switch] statement *)
  | Slabel : label → statement → statement
  | Sgoto : label → statement

with labeled_statements : Type := (**r cases of a [switch] *)
  | LSnil: labeled_statements
  | LScons: option Z → statement → labeled_statements →
  labeled_statements.
```

The only one of these statements that can't be used in Verifiable C is the Sgoto statement. This is because there is no proof rule for Sgoto so a proof of a program with Sgoto in it will simply be stuck when that point is reached.

In summary, every Clight program is a CompCert C program, and every CompCert C program can be translated to Clight with only local transformations. Our typechecker restricts code to a subset of Clight that is friendly to verification.

3.3 Typechecker

The typechecker produces assertions that, if satisfied, prove that an expression will always evaluate to a value.

In the C light abstract syntax produced by CompCert from C source code, every subexpression is syntactically annotated with a C-language type, accessed by (typeof e). Thus our typing judgment does not need to be of the form $\Delta \vdash e : \tau$, it can be $\Delta \vdash e$, meaning that e typechecks according to its own annotation.

We define a function to typecheck expressions with respect to a type context:

```
Fixpoint typecheck_expr (Δ : tycontext) (e: expr) : tc_assert :=
    | Econst_int _(Tint _ _ _) ⇒ tc_TT
    | Eunop op a ty ⇒ tc_andp
        (tc_bool (isUnOpResultType op a ty) (op_result_type e))
        (typecheck_expr Δ a)
    | Ebinop op a1 a2 ty ⇒ tc_andp
        (tc_andp (isBinOpResultType op a1 a2 ty)
          (typecheck_expr Δ a1))
        (typecheck_expr Δ a2)
 ... end.
```

This function traverses expressions emitting *reified* conditions that ensure that the expressions will evaluate to a value in a correctly typed environment. These reified functions belong to the type tc_assert:

```
Inductive tc_assert :=
| tc_FF: tc_error → tc_assert (* False *)
| tc_andp': tc_assert → tc_assert → tc_assert
| tc_isptr: expr → tc_assert (* the expression is a pointer *)
...
```

On top of this we define assertions that are programs that automatically simplfy once they are applied to their subexpressions:

```
Definition tc_andp (a1: tc_assert) (a2 : tc_assert) : tc_assert :=
match a1 with
| tc_TT ⇒ a2
| tc_FF e ⇒ tc_FF e
| _⇒ match a2 with
      | tc_TT ⇒ a1
      | tc_FF e ⇒ tc_FF e
      | _⇒ tc_andp' a1 a2
      end
end.
```

The function tc_andp, for example, will automatically simplify if it is applied to either tc_TT or tc_FF.

A number of facts can be determined computationally. For example we can always determine if a value is an integer by simply examining its type in the program:

```
Definition is_int_type (ty : type) : bool :=
match ty with
| Tint _ _ _⇒ true
| _⇒ false
end.
```

These boolean typechecking functions can be injected into tc_assert using tc_bool:

```
Definition tc_bool (b : bool) (e: tc_error) :=
if b then tc_TT else tc_FF e.
```

Because we are always typechecking ground terms that came directly from a compiled program, using tc_bool in conjunction with tc_andp will always result in a function whose result is either False or where the boolean fact has been eliminated.

Although CompCert's operational semantics are written as an inductive relation in Coq, it also has computational parts. For example, CompCert has functions to help typecheck mathematical operations by determining the type of the result. These functions are called *classification* functions because they classify the types of operators. The following function leverages the classification function to typecheck binary operations:

```
Definition isBinOpResultType op a1 a2 ty : tc_assert :=
match op with
  | Oadd ⇒ match classify_add (typeof a1) (typeof a2) with
              | add_default ⇒ tc_FF
              | add_case_ii _ ⇒ tc_bool (is_int_type ty)
              | add_case_pi _ _⇒ tc_andp (tc_isptr a1) (tc_bool (is_pointer_type ty))
                    ... end
  ... end.
```

Most operators in C are overloaded. For example the addition operator works on both integers and pointers, behaving differently if given two integers, two pointers, or one of each. Classification functions determine which of these overloaded semantics of operators should be used. These semantics are always determined by the types of the operands. The C light operational semantics uses the constructors (add_case_ii, add_case_pi, (and so on)) to choose whether to apply integer-integer add (ii), pointer-integer add (pi), and so on. The type-checker uses the same constructors add_case_ii, add_case_pi, to choose type-checking guards.

Despite the reuse of CompCert code on operations, most of the typechecker checks binary operations.

This is because of the operator overloading on almost every operator in C. The typechecker looks at eight types of operations (shifts, boolean operators, and comparisons can be grouped together as they have the exact same semantics with respect to the type returned). Each of these has approximately four behaviors in the semantics giving a total of around thirty cases that need to be handled individually for binary operations.

The code above is a good representation of how the typechecker is implemented. The first step is to match on the syntax. Next, if the expression is an operation, we use CompCert's classify function to decide which overloaded behavior to use. From there, we generate the appropriate assertion.

3.3.1 Type Context

Expression evaluation requires an expression and an environment. The result of expression evaluation depends on the environment:

```
(* Global-variable environment *)
(* A mapping from variables to memory locations *)
Definition genviron := Map.t block.

(* Adressable local-variable environment *)
(* A mapping from variables to memory locations and c types *)
Definition venviron := Map.t (block * type).

(* Temporary (nonadressable local variable) environment *)
(* A mapping from variables to C values *)
Definition tenviron := Map.t val.

Inductive environ : Type :=
 mkEnviron: ∀(ge: genviron) (ve: venviron) (te: tenviron), environ.
```

We use a simple linear map Map.t (which is equivalent to (ident → option val)) for this environment because in practice the environment will be a quantified variable. It will simply be a variable with no underlying implementation. This means that performance will be unaffected by the implementation of the environment. The type block refers to a block of memory, CompCert's address into the beginning of a malloc'ed block.

To guarantee that certain expressions will evaluate, we control what values can appear in environments. A static type context characterizes the dynamic environments in which we can prove that expressions will evaluate to defined values.

```
Definition tycontext: Type :=
  (PTree.t (type * bool) * (PTree.t type) * type * (PTree.t global_spec)).
```

PTree.t(τ) (Section 6.1.1) is CompCert's efficient computational mapping data-structure (from identifiers to τ) implemented and proved correct in Coq. The type type is not Coq's Type, but the deep embedding of C types that appear in Clight programs.

The elements of the type context are

- a mapping from temp-var names to type and initialization information (temp-vars are simply nonaddressable—ordinary—local variables),
- a mapping from (addressable) local variable names to types,
- a return type for the current function, and
- a mapping from global variable names to types
- a mapping from global variable names to function specifications

The first, second, and fourth items correspond to the three parts of an environment (environ), which is made up of temporary variable mappings, local variable mappings, and global variable mappings.

The fifth item is specific to proofs and is separated from the fourth because function specifications can't be computed on in a reasonable amount of time. Separating these two allows us to do the computations we discuss in Chapter 5.

A temporary variable is a (local) variable whose address is not taken anywhere in the procedure. A variable is taken if the C address of (&) operator is used. Unlike local and global variables, these variables can not alias—so we can statically determine exactly when their values are modified. If the typechecker sees that a temporary variable is initialized, it knows that it will stay initialized. The initialization analysis is conservative, but if the typechecker is unsure, it can emit an assertion guard for the user to prove initialization. In the case of initialization, the typechecker will never give a result of "does not typecheck". It will only say "yes" or "maybe". It would be possible to occasionally simply say "no", but this would require extra information from the type context, and wouldn't save the user much information.

Calculating initialization automatically is a significant convenience for the user; proofs in the previous generation of our program logic were littered with definedness assertions in invariants.

The initialization information is a Boolean that tells us if a temporary variable has certainly been initialized. The rules for this are simple, if a variable is assigned to, that variable will always be initialized in code executed after it. The initialization status on leaving an **if-then-else** is the greatest lower bound of the two branches. Loops have similar rules.

typecheck_environ checks an environ with respect to a tycontext. It does not generate assertions as typecheck_expr does, it simply returns a Boolean that if true guarantees all of the following:

- If the type context contains type information for a temporary variable, the environment contains a (possibly undefined) value for that variable. If the variable is claimed to be initialized, that value must belong to the type claimed in the type environment (never the undefined value).
- If the type context contains type information for a (addressable) local variable, the local variable environment contains a local variable of matching type. The local variable gives a defined address in memory. The contents is not characterized by the type system; for that we use separation-logic (spatial) predicates.
- If the type context contains type information for a global variable, the global environment contains a global variable of matching type.
- If the type context contains type information for a global variable, either
  - the local variable environment does not have a value for that variable or
  - the type context has type information for that variable as a local variable.

The fourth point is required because local variables shadow global variables.

Consider the program

```
float *x;

void* bad_function( ) {
  int x;
  return (void *) x;
}
```

If we allow the environment to only contain information about the global (**float** *) variable x and not the local (**int**) variable x, this program will typecheck in a typecontext that doesn't specify the local variable. It will typecheck because the type context is allowed to be smaller than the dynamic environment and we believe the **return** expression is casting between pointers. We need to disallow casts from int to pointer in our type system, however, so we don't want this function to typecheck. The fourth point above prevents this situation by requiring a variable that is shadowed in a dynamic environment to also be shadowed in the type context.

Initialization information is changed by statements. We only know a variable is initialized once we see that it is assigned to. The typechecker operates at the level of expressions, so the logic rules which work at the statement level must maintain the type context. We will now give some of these rules and explain how they work to keep the type context correct.

We provide a function

```
Definition func_tycontext (func: function) (V: varspecs) (G: funspecs): tycontext
```

that automatically builds a correct type context (for the beginning of the function body) given the function, local, and global specifications. The resulting context contains every variable declared in the function matched with its correct type. We have proved that the environment created by the operational semantics when entering a function body typechecks with respect to the context generated by this function (i.e., via typecheck_environ). Once the environment is created, the Hoare rules use the function updatetycon to maintain the type context across statements.

```
Fixpoint update_tycon (Delta: tycontext) (c: Clight.statement) {struct c} : tycontext :=
 match c with
 | Sset id e2 ⇒ (initialized id Delta)
 | Ssequence s1 s2 ⇒ let Delta' := update_tycon Delta s1 in
                     update_tycon Delta' s2
 | Sifthenelse b s1 s2 ⇒ join_tycon (update_tycon Delta s1) (update_tycon Delta s2)
 | Sswitch e ls ⇒ join_tycon_labeled ls Delta
 | Scall (Some id) _ _ ⇒ (initialized id Delta)
 | _ ⇒ Delta (* et cetera *)
end
```

$$\frac{\Delta \vdash \{P\}\, c\, \{Q\} \qquad \Delta' = \mathsf{updatetycon}(\Delta, \mathsf{c}) \qquad \Delta' \vdash \{Q\}\, d\, \{R\}}{\Delta \vdash \{P\}\, c; d\, \{R\}}\text{seq}$$

updatetycon tells us that variables are known to be initialized after they are assigned to. It also says that variables are initialized if they were initialized before the execution of any statement, and that a variable is initialized if we knew it was initialized at the end of both branches of a preceding **if** statement. When we say initialized, we mean *unambiguously* initialized, meaning that it will be initialized during all possible executions of the program.

The type context is deeply integrated with the logic rules. We write our Hoare judgment as $\Delta \vdash \{P\}\, c\, \{Q\}$. It contains the type context Δ because instead of quantifying over all environments as a normal Hoare triple does, we quantify only over environments that are well typed with respect to Δ. This has a benefit to the users of the rules: it guarantees that all sound Hoare rules update the type context correctly. Because only the type checker will ever need to examine the type context, the user does not need to worry about the contents of Δ, show that the environment typechecks or mention Δ explicitly in preconditions. Our special *rule of consequence* illustrates what we always know about Δ:

$$\frac{(\mathsf{typecheck\_environ}\ \Delta \wedge P) \vdash P' \qquad \Delta \vdash \{P'\}\, c\, \{R\}}{\Delta \vdash \{P\}\, c\, \{R\}}$$

The conjunct $\mathsf{typecheck\_environ}\Delta$ gives the user more information to work with in proving the goal. Without this, the user would need to explicitly strengthen assertions and loop invariants before applying the rule of consequence to keep track of the initialization status of variables and the types of values contained therein.

Although we've mentioned the environment ρ, and that we are typechecking Δ against it, ρ hasn't appeared in our formulas yet. This is because in our logic, we hide the existence of ρ from the user, since it is strictly symbolic and never updated. In fact if we say

$$\mathsf{typecheck\_environ}\ \Delta \wedge P \vdash P'$$

it is equivalent to:

$$\forall \rho.\ \mathsf{typecheck\_environ}\ \Delta\ \rho \wedge P \rho \vdash P' \rho$$

this means that we can hide the fact that the environment always typechecks away in the statement of the triple, and show it to the user only when it might be useful in proving an entailment.

With func_tycontext and updatetycon the rules can guarantee that the type context is sound throughout every proof. To keep the type context updated, the user must simply apply the normal Hoare rules, with our special Hoare rule for statement sequencing shown above.

3.3.2 Soundness

The soundness statement for the typechecker is:

If the dynamic environment ρ is well typed with respect to the static type context Δ (Section 3.3.1), and the expression e typechecks with respect to Δ producing an assertion that in turn is satisfied in ρ, then the value we get from evaluating e in ρ will match the type that e is labeled with.

```
typecheck_environ Δ = true →
denote_tc_assert (typecheck_expr Δ e) →
`typecheck_val (eval_expr e) `(typeof e) = true.
```

The typecheck_val function guarantees that an expression will evaluate to the right kind of value: integer, or float, or pointer. It also gives a range on the value that can be expected from an integer as opposed to a short or a boolean. As a corollary we guarantee the absence of Vundef (the undefined value in CompCert C), which has no type.

We also prove that if our environment ρ matches a CompCert Clight environment, a typecontext Δ typechecks in ρ, and an expression e typechecks in Δ and ρ then the CompCert operational semantics will give the same result as our eval_expr function when evaluating e in ρ.

```
rho = construct_rho (filter_genv ge) ve te →
typecheck_environ Delta rho →
(denote_tc_assert (typecheck_expr Delta e) rho →
Clight.eval_expr ge ve te m e (eval_expr e rho))
```

This fact is required in order to prove a Hoare triple sound with respect to CompCert's operational semantics. It allows us to use our computational eval_expr and typecheck_expr to generate preconditions and postconditions while showing in our proofs that these functions are actually doing the same thing that CompCert's eval_expr (operational semantics for expressions) does.

The proofs proceed by induction on the expression e. One of the most difficult parts of the soundness proof is the proofs about binary operations. We need to prove that when a binary operation typechecks, it evaluates to a value, as a case for the main soundness proof. The proof is difficult because of the number of cases. When all integers and floats of different sizes and signedness are taken into account, there are seventeen different CompCert types. This means that there are 289 combinations of two types. A proof needs to be completed for each combination of types for all seventeen C light operators, leading to a total of 4913 cases that need to be proved. The amount of memory taken by the proof becomes a problem. We use lemmas to group some of the cases together to keep the proof time reasonable.

These cases are not all written by hand: we automate using Ltac. Still, the proofs are large, and Coq takes almost 4 minutes to process the file containing the binary operation proofs.

3.4 A Tactical Proof

This section gives an example of applying the Verifiable C program logic to verify a simple C program interactively in Coq, verifying the C program:

```
int assigns (int a) { int b, c; c = a*a; b = c/3; return b; }
```

The first step is to pass the program through the CompCert clightgen tool to create a Coq .v file. The next step is to specify the program:

$$\Delta \vdash \{\mathsf{Vint}(v) = \mathrm{eval}\,\mathsf{a}\,\rho\}\ \mathsf{body\ of\ assigns}\ldots\ \{\mathsf{retval}\,\rho = \mathsf{Vint}((v * v)/3)\}$$

Barring any unexpected evaluation errors, this specification should hold. The specification states that in an arbitrary initial state, the program will either infinite loop or terminate in a state in which retval = (a*a)/3. This example focuses on proving the specification of the function body. The following shows how automated the entire proof is:

```
Lemma body_test : semax_body Vprog Gtot f_assigns assign_spec.
Proof.
  start_function.forward. forward. entailer!. forward. entailer.
Qed.
```

In fact, the entire proof could easily be completed by one tactic:

```
repeat (forward; try solve[entailer!])
```

The function semax_body creates a Hoare triple, called semax in our logic, for a function body given a list of global variable specifications (Vprog, the empty list), a list of global function specifications (Gtot, list of this function and main), The syntactic function declaration (with parameters and function body) (f_assigns, from program .v file), and a specification (assign_spec, the Coq version of the triple shown above). The tactic start_function unfolds semax_body and ensures that the precondition is in a usable form.

In the following examples, C light AST is shown C-like syntax in the lines marked *(*pseudocode *)*. Assertions are in a canonical form, separated into PROP (propositions that don't require state), LOCAL (assertions lifted over the local environment), and SEP (separation assertions over memory). A discussion of this form can be found in Chapter 4. Empty assertions for PROP and LOCAL mean True and an empty SEP means emp, or the empty heap.

```
Δ := initialized _c (func_tycontext f_assigns Vprog Gtot) : tycontext
============================
 semax Δ
   (PROP ()
    LOCAL(`eq (eval_id _c) (eval_expr(_a * _a)); (`eq (eval_id _a) v))  SEP())
   (_b = _c / 3; return _b;) (* pseudocode standing for C-light AST *)
   (function_body_ret_assert tint (_a * _a / 3) = retval)
```

Above is the state after applying forward for the first time. This tactic performs forward symbolic execution using Coq tactic programs, as various authors have demonstrated [6, 16, 8, 34]. In effect, forward applies the appropriate Hoare rule for the next command, using the sequence rule if necessary. The backtick (`) is the "lifting" coercion of Bengtson *et al.* [8]. function_body_ret_assert tells us that if the command wishes to execute a return statement, then the return value and state must satisfy the given predicate; and any other exit from the command (break, continue, fall-through) is forbidden.

The forward tactic chooses a Hoare-logic inference rule to apply, based on the precondition and the command. It can make most of its decisions using only the command, but occasionally it takes the precondition into account as well. For example, in general, forward uses the Floyd assignment rule that existentially quantifies the "old" value of the variable being assigned to (in this case c). It needs to do this because otherwise we would lose information from our precondition by losing the old value of c. If the variable doesn't appear in the precondition, however, the existential can be removed because it will never be used. The forward tactic checks to see if it needs to record the old value of the variable or not. In this case, it sees that c is not in the precondition and does not record its old value (That is, forward applies a specialized version of the assignment rule that loses information about the old value of the assigned variable.).

The symbolic executor applies the Floyd assignment rule on command c=a*a with type context Δ in which a maps to (type=int, initialized=true). The typechecker, given Δ and the expression a*a, computes the precondition necessary to guarantee that a*a evaluates to a value of type int; that precondition is True. It comes to this conclusion by doing a number of computations:

- Look up a in Δ, make sure that it is initialized (does this once for each subexpression)
- Classify the multiplication operator in a * a, see that it returns an int, check this against the type given for a * a.

- Simplify the resulting reified-assertion of tc_andp tc_TT tc_TT to tc_TT
- Simplify denote_tc_assert tc_TT to True

The symbolic executor must solve the entailment P–True—, where P is the precondition of the whole command. But the entailment P–True— is trivial, and is never shown to the user.

Finally we notice that Δ has been updated with initialized _c. This was done by the sequence rule as discussed in Section 3.3.1.

Applying forward again gives the following separation-logic side condition:

```
a : name _a
b : name _b
c : name _c
Δ := initialized _b (initialized _c (func_tycontext f_assigns Vprog Gtot) : tycontext
============================
 PROP() LOCAL(tc_environ Δ; `eq (eval_id _c) (eval_expr (_a * _a)); (`eq (eval_id _a) v))
 SEP(`TT) ⊢ local (tc_expr Δ (_c / 3)) && local (tc_temp_id _b tint Δ)
```

This is an entailment, asking to prove the right hand side given the left hand side. This entailment is asking us to show that the expression on the right hand side of the assignment typechecks, and that the id on the left side typechecks. We would expect to see that: c is initialized, $3 \neq 0$ and $\neg(c = \mathsf{min\_int} \wedge 3 = -1)$. The local operator injects a pure proposition into our separation logic.

Why is it useful to have tc_environ Δ? This entailment is *lifted* (and quantified) over an abstract environ ρ; if we were to intro ρ and make it explicit, then we would have conditions about eval_id _c ρ, and so on. To prove these entailments, we need to know that (eval_id _a ρ) and (eval_id _c ρ) are defined and well-typed.

In a paper proof it is convenient to think of an integer *variable* _a as if it were the same as the *value* obtained when looking _a up in environment ρ—we write this (eval_id _a ρ). In general, we can not think this way about C programs because in an arbitrary environment, _a may be of the incorrect type or uninitialized – in these cases, the value eval_id _a ρ will not have the type we expect. Or _a may not be in scope; in this case, (eval_id _a ρ) won't even exist. In an environment ρ that typechecks with respect to some context Δ, however, we can bring this way of thinking back to the user.

In the case of an int, the go_lower tactic does one step more: knowing that the value (eval_id _a ρ) typechecks implies it must be Vint x for some x, so it introduces a as that value x. (Again, the name a is chosen from the hint, a: name _a.) Thus, the user can think about values, not about evaluation, just as in a paper proof. Our go_lower tactic, followed by normalize for simplification converts the entailment into

```
c : int
a : int
H0 : Vint c = Vint (Int.mul a a) (*simplified*)
============================
 denote_tc_nodivover (Vint c) (Vint (Int.repr 3))
```

where denote_tc_nodivover stands for "no division overflow". The typechecker has discharged most of our obligations automatically (that _a is in scope and is an integer, that $3 \neq 0$), and calculates the one verification condition that remains to be solved outside the typechecker: that $c/3$ does not overflow. We can no longer see the variables _c and _a.

The defined predicate (denote-tc-nodivover (Vint c) (Vint (Int.repr 3))) simplifies to (¬(c = min_int ∧ 3 = -1)), which solves easily, using the solve_tc tactic. Not all typechecker-introduced assertions will be so easy to solve, of course; in place of solve_tc the user might have to do some real work, or discover that they need to adjust an invariant in order to show that the program typechecks.

The rest of the proof advances through the return statement, then proves that the postcondition after the return matches the postcondition for the specification. In this case it is easy, just a few unfolds and rewrites.

3.5 Related Work

Frama-C is a framework for verifying C programs [20]. It presents a unified assertion language to enable static analyses to work together by sharing information about the program. The assertion language allows users to specify only first-order properties about programs, and does not include separation logic. This makes it too weak to express invariants that are often necessary for full functional verification of C programs. In fact, safety proofs can also require higher order reasoning. It is impossible to specify this type of program at all in Frama-C. The Value analysis [14] uses abstract interpretation to determine possible values, giving errors for programs that might have runtime errors. The WP plugin uses weakest precondition [35] calculus to verify triples. For WP to be sound it must be run with the RTE plugin[25], which is an annotation generator for Frama-C. With the correct options, RTE can be used to generate assertions (i.e., overflow assertions) that allow a mapping from C's machine integers to logical integers. Frama-C does not seem to have any soundness proof, but it is likely to be sound, or very close to sound. A proof in Frama-C gives good evidence for program correctness, but the evidence is not as strong as a proof in a foundational system like Verifiable C. The decreased assurance and logic strength has an advantage though, it is much easier to use quickly on the programs that it is able to reason about it. There are many cases where it might be advantageous to first use Frama-C as a first pass to make good progress towards correctness, and next to use VST to give more complete assurance.

VCC is a verifier for concurrent C programs [17]. It works by translating C programs to Boogie [7], which is a combination of an intermediate language, a VC generator, and an interface to pass VCs off to SMT solvers such as Z3[21]. VCC adds verification conditions that ensure that expressions evaluate. Like the Value analysis, there is no link between the conditions and an operational semantics; that is, perhaps VCC is sound, but there's no formalization of that claim.

Greenaway *et al.* [24] show a verified conversion from C into a high-level specification that is easy for users to read. They do this by representing the high-level specification in a monadic language. They add guards during their translation out of C in order to ensure expression evaluation (this is done by Norrish's C parser [38]). Many of these guards will eventually be removed automatically. Their tool is proved correct with respect to the semantics of an intermediate language, not the semantics of C. The expression evaluation guards are there to ensure that expressions always evaluate in the translated program, because there is no concept of undefined operations in the intermediate language. Without formalized C semantics, however, the insertion of guards must be trusted to actually do this. This differs from our approach where the typechecker is proved sound with respect to a C operational semantics; so we have more certainty that we have found all the necessary side conditions. Another difference is that they produce all the additional guards and then solve most of them automatically, while we avoid creating most such assertions. Expression evaluation is not the main focus of Greenaway's work, however, and the ideas presented for simplifying C programs could be useful in conjunction with our work, possibly to automatically generate parts of specifications that can be used as a starting point for VST users to create program specifications.

Bengtson *et al.* [8] provide a framework for verifying correctness of Java-like programs with a higher-order separation logic similar to the one we use. They use a number of Coq tactics to greatly simplify interactive proofs. Chlipala's Bedrock project [16] also aims to decrease the tedium of separation logic proofs in Coq, with a focus on tactical- and reflection-based automation of proofs about low level programs. Bengtson operates on a Java-like language and Chlipala uses a simple but expressive low-level continuation-based language. Appel [2] used a number of tactics to automate proofs as well. In this system, the user was left with the burden of completing proofs of expression evaluation.

The proof rules we use in this thesis are also used in the implementation of a verified symbolic execution called VeriSmall [3]. VeriSmall does efficient, completely automatic shape analysis.

Tuerk's HolFoot [42] is a tactical system in HOL for separation logic proofs in an imperative language. Tuerk uses an idealized language "designed to resemble C," so he did not have to address many of the issues that our typechecker resolves.

One of Tuerk's significant claims for HolFoot is that his tactics solve purely shape-analysis proofs without any user assistance, and as a program specification is strengthened from shape properties to correctness properties, the system smoothly "degrades" leaving more proof obligations for the interactive user. This is a good thing. As we improve our typechecker to include more static analysis, we hope to achieve the same property, with the important improvement that the static analysis will run much faster (as it is fully reified), and only the user's proof goals will need the tactic system.

Our implementation of computational evaluation is similar to work on executable C semantics by Ellison and Rosu [22] or Campbell [13]. Their goals are different, however. Campbell, for example, used his implementation to find bugs in the specification of the CompCert semantics. We, on the other hand, are accepting the CompCert semantics as the specification of the language we are operating on. Ellison and Rosu have the goal of showing program correctness, which is a similar goal to ours. They show program correctness by using their semantics as a debugger or an engine for symbolic execution.

Chapter 4

Canonical Forms for Assertions

VST uses assertions to reason about programs, but what does it require about the form of these assertions? This chapter discusses how, by increasing the level of organization of assertions, we can use more efficient data structures. This simplifies the statement of some of the Hoare logic rules at the same time as it increases their performance.

All assertions in VST's logic take an environ as an argument. The environment will remain fully abstract during the symbolic execution performed using the assertions. The symbolic execution proceeds not by inserting symbolic values into an environment, but by writing assertions about the values. To do this, we simply add an assertion about the value of a variable when it is looked up in the environment. The function for looking up an nonadressable variable in an environment is eval_id. We can write an assertion of type environ $\rightarrow$ Prop about variable x in environment rho:

(fun $\rho \Rightarrow$ eval_id x ρ = 4)

Or equivalently, hiding the environment by using the lifting operator:

\`eq (eval_id $x$) \`4

The second form is advantageous, because many Coq tactics are unable to perform operation inside of binders, making the first form very difficult to work with in proofs. Although the second form is equivalent to the first, Coq automation has a much easier time dealing with the second.

Simply referring to the environment is not enough to specify a C program because C programs can also refer to the heap, which is not specified by the environment. An assertion in VST's logic can be any semantic predicate of type environ $\rightarrow$ mpred. Memory predicate, or mpred is the type of separation-logic assertions over C memory. Separation logic has operators that make it particularly convenient to reason about the absence of pointer aliasing in programs. It does this by providing the *separating conjunction* (*) operator, which means that if there are two pointers p_1 and p_2 that point at values v_1 and v_2 (using the *mapsto* operator $\mapsto$) the assertion $p_1 \mapsto v_1 * p_2 \mapsto v_2$ means that v_1 and v_2 must be located at different positions in the heap. That is, p_1 and p_2 are not aliases for each other. More formally, a heap that satisfies $P * Q$ can be partitioned into two sub-heaps, one satisfying P and the other satisfying Q.

This is very useful in C programs because the following triple holds

$\{p_1 \mapsto v_1 * p_2 \mapsto v_2\}\, p_1 := v_3\, \{p_1 \mapsto v_3 * p_2 \mapsto v_2\}$

We only had to update one part of the assertion between the precondition and the postcondition because we were certain that p_1 and p_2 point to different locations in memory. If we had used the standard conjunction we would need the fact $p_1 \neq p_2$. That fact isn't too significant in a small examples, but the number of such inequalities required is quadratic with the number of references to the heap in the assertion.

Of course, not all parts of an assertion need to be about memory. Some parts might just refer to the environment, or be pure, in which case we have the following operators:

```
Definition !! (P : Prop) → (environ → mpred) ⇒
fun _~_⇒ P.

Definition local (Q : environ → Prop) : (environ → mpred) :=
fun ρ ⇒ prop (Q ρ).
```

The first, prop takes a pure fact, and lifts it to the level of environ → mpred by taking (and ignoring) the environment and memory arguments. The second, local, makes an environ → prop into an environ → mpred of Q by giving the environment to Q but using the prop function to throw away the memory.

To combine pure facts with separation facts we can use the && operator (which is simply the and operator on mpred) in conjunction with the (!!) operator. Putting those together we can write an assertion such as

```
(* Pure comparison between the Coq variable x and the C value 3 *)
!!(x = 3) &&
(* Tests equality of C local variable _x to the Coq variable for the C
   value X *)
local (`eq (eval_id _x) `x) &&
(p1 ↦ x * p2 ↦ 5)
```

Appel et al. [5] organized these assertions into a PROP/LOCAL/SEP form to make them easier to manipulate in proofs. I will describe this *semicanonical form* in section 4.1. Then I will improve on that form, which makes proof easier to automate than they were in semicanonical form. This *canonical form* is described in Section 4.2.

4.1 Semicanonical Form

We say that assertions written PROP P LOCAL Q SEP R are in semicanonical form. Here each item in P:list prop is a pure assertion that doesn't refer to the program state. Q: list (environ $\rightarrow$ prop) contains assertions that can reason about local variables. and R: list (environ $\rightarrow$ mpred) has spatial assertions that can reason about both local variables and memory. PROP folds conjunction over the elements of P, LOCAL folds lifted conjunction (`and : (environ $\rightarrow$ Prop)$\rightarrow$ (environ $\rightarrow$ Prop) $\rightarrow$ (environ $\rightarrow$ Prop)), and SEP folds the separation logic $\star$, or separating conjunction operator. P, Q, and R, are represented as lists because, particularly in the case of SEP, it is convenient to refer to the nth conjunct. This is much easier to implement when our assertion is restricted to a flat sequence of conjunctions. The precise definition of PROP P LOCAL Q SEP R is:

```
!!(fold_right ( ∧ ) True P) &&
local(fold_right ( `∧ ) `True Q) &&
(fold_right ( ⋆ ) emp R)
```

We also provide functions for representing common local assertions in a convenient manner:

```
Definition temp (i: ident) (v:val) : environ → Prop :=
  `(eq v) (eval_id i)


Definition var (i: ident) (t: type) (v: val) : environ → Prop :=
  `(eq v) (eval_var i t).
```

These functions are strictly aesthetic. In particular, they hide most of the lifting operators that will occur in assertions. We use the temp assertion to say that the lookup of a variable i in some environment is equal to value v. Local is the same but for addressable variables.

The lowest-level Verifiable C logic rules (the rules that are directly proved sound wrt. the CompCert operational semantics) are agnostic to any structure that the assertion might have. The rules refer to assertions as single variables, using entailments to constrain them rather than imposing syntactic requirements on them. The Floyd automation system has higher-level lemmas that require assertions to be in semicanonical form, but also guarantee that the side conditions that result from using the rules will be in semicanonical form.

4.1.1 Substitution in semicanonical form

At the Verifiable C level, there is no choice but to use a semantic notion of substitution:

```
Definition subst {A} (x: ident) (v: val) (P: environ → A) : environ → A :=
fun ρ ⇒ P (env_set ρ x v).
```

This is because we know nothing about the assertion at all, only that it takes an environment and returns an mpred. The following example shows how subst is used in our forward assignment rule:

$$\text{semax\_set\_forward}\ \frac{}{\Delta \vdash \{P\}\ x := e\ \{\exists v.\, x = (e[v/x]) \wedge P[v/x]\}}$$

```
Axiom semax_set_forward:  ∀ Δ (P: environ→ mpred) (x: ident) (e: expr),
  semax Δ
    (▷ (local (tc_expr Δ e) && local (tc_temp_id id (typeof e) Δ e) && P))
    (Sset x e)
    (normal_ret_assert
      (EX old:val, local (`eq (eval_id x) (subst x (`old) (eval_expr e)))
                   && subst x (`old) P)).
```

There are two substitutions here, used to replace any occurrences of the variable x that might have occurred in either the precondition or the expression being assigned into x. The following example of what happens without the substitution shows why this is necessary:

```
{eval_id x = 3}
   x = 4;
{eval_id x = 3 ∧ eval_id x = 4}
```

This is clearly the wrong Hoare triple; the postcondition is unsatisfiable even though the command is quite reasonable. Instead we do the substitution and get:

```
{eval_id x = 3}
   x = 4;
{∃ x_old, x_old = 3 ∧ eval_id x = 4}
```

Although subst is a function, in practice it is never computed during Hoare-logic proofs. This is because it works by updating the environment that P refers to. During symbolic execution, however, the environment is always abstract, which means there is no data structure to perform updates in. So instead of eliminating subst by computation we eliminate it using sound rewrites. To deal with this the Floyd system has an autorewrite [1] database that lets it push subst through functions that won't be affected by the substitution. For example

[1] autorewrite is a tactic that takes a database as an argument and repeatedly attempts to rewrite by the lemmas given in the database. It stops when there is not further rewriting that can be done

```
Lemma subst_sepcon: ∀ i v (P Q: environ→ mpred),
  subst i v (P * Q) = (subst i v P * subst i v Q).
```

Fortunately, we don't need a lemma for every function that might appear in assertions. Lifted functions (for example `eq) can't do anything with the environment, they can only pass it on to their arguments, so by creating a general rule for lifted functions we cover most of the functions that we use, and also most functions that a user might want to write.

Semantic substitution is still inconvenient for a few reasons. First, the rewrite rules aren't complete. This means that in some cases, after applying a logic rule, the user will see subst in a resulting condition. This can stop the automated entailment solvers from working correctly and make the assertion harder to read. The next problem is an issue with autorewrite in general. Autorewrite in Coq is slow. Rewrites aren't known for their performance, and autorewrite can do a large number of rewrites (in the case of trying to simplify a singlesubst the number of rewrites is linear in the size of the assertion being rewritten).

There is a situation when a substitution subst $x\ v\ P$ can be avoided completely. That is when P is *closed* with respect to x, also a semantic notion:

```
Definition closed_wrt_vars {B} (S: ident → Prop) (F: environ → B) : Prop :=
  ∀ rho te',
    (∀ i, S i ∨ Map.get (te_of rho) i = Map.get te' i) →
    F rho = F (mkEnviron (ge_of rho) (ve_of rho) te').
```

Typically we instantiate S with eq x for some identifier x. What closed_wrt_vars means, then, is that if F is supplied an environment that is the same at all locations but the identifier(s) i that satisfy S, the result of F will be the same. That means that if we know closed_wrt_vars (eq x) (e), we can easily prove subst x _e = e. More intuitively, if an expression doesn't contain a variable, a substitution on that variable won't change the expression.

The Floyd system has a version of the assignment rule that takes advantage of this, stating that if the precondition and the expression in the assignment are closed wrt. the variable being assigned into, no substitutions are needed, but there are numerous reasonable cases where this rule doesn't apply and the substitution will still appear.

4.2 Canonical Form

The reason that substitutions are difficult in semicanonical form, and that they need to be semantic, is because there is no *syntactic* restriction on where any individual identifier can appear within an assertion. Our new canonical form imposes such a restriction, and in doing so, eliminates the need for semantic substitution, replacing it with a more efficient and convenient computational syntactic substitution.

In place of Appel's "semicanonical form" PROP/LOCAL/SEP, we introduce a "canonical form", assertD P (localD Q1 Q2)R . The assertD function is a restriction on PROP/LOCAL/SEP, so assertD can be defined using PROP/LOCAL/SEP:

```
Definition assertD (P : list Prop) (Q : list (environ → Prop))
                   (R : list mpred) :=
PROPP (LOCAL Q (SEP (map (liftx) R))).
```

All that assertD does is change the type of R from list (environ → mpred) to list mpred. So if we have a semi-canonical assertion:

```
PROP∼P
LOCAL()
SEP(`data_at 3 _x)
```

we can make it more canonical by instead using the equivalent:

```
assertD P [temp _x x] [data_at 3 x]
```

The other part of canonical form is the localD function, which must create the Q argument to assertD for the assertion to be in canonical form. The purpose of the localD function is to restrict the LOCAL part of the assertion to be made up of two computational maps from C identifiers to values. Let T_1: PTree val be an efficient computational map from C program identifiers to C values, representing the current values of the temporary local variables of the current program state. Let T_2: PTree (type∗val) be a map from identifiers to type∗val representing the addresses of addressable local variables. Then localD T_1 T_2: list(environ→ Prop) is a list of assertions about the contents of the environ, the non-memory portion of the program state; we do not need *arbitrary* assertions of type list(environ→ Prop) because the two tables allow all of the manipulations needed on the environ. Any other assertions can then be moved into PROP.

localD is a *denotation function*, reflecting the syntactic (computationally oriented) T_1 and T_2 back into our semantic world:

Definition localD (T1: PTree.t val) (T2: PTree.t (type * val)) := PTree.fold (fun Q i v =¿ temp i v :: Q) T1 (PTree.fold (fun Q i tv =¿ var i (fst tv) (snd tv) :: Q) T2 nil).

The PTree.fold function behaves the same way as a fold on a list works, but the function is given both the key and the value. All this denotation function says is that every key value pair in the first map represents an equality between the evaluation of a temporary variable (the key) and a value (the value in the mapping), and the second map says the same, but for adressable variables.

In symbolic execution and efficient entailment solving, we operate directly on T_1 and T_2, reflecting the results back only when the less efficient (but easier to understand) semantic view is needed by the user. Now a full assertion is:

assertD P (localD T_1 T_2) R : environ→ mpred

P : list prop T_1 : PTree val T_2 : PTree (type * val) R : list mpred

So if we have an assertion in semicanonical form:

```
PROP P
LOCAL  [temp i p; temp j q; var k r]
SEP  [`R1; `R2]
```

It can be represented in canonical form (and translated automatically by a tactic) as:

assertD P (localD [i ↦ p; j ↦ q;] [k ↦ r]) [R1; R2]

To translate back, you can simply unfold the definitions of assertD and localD.

4.2.1 Substitution in Canonical Form

Substitution in this assertion is as simple as adding/replacing a mapping in T_1 or T_2. To see why, imagine that we are doing a substitution on a temporary variable x. P : list prop and R : list mpred don't refer to an environment, so they are trivially closed wrt. x. This leaves T_1 and T_2. The variable x is a temporary variable, so we know T_2 is closed wrt. x. The map T_1, however, might have a reference to x, meaning we actually need to do a substitution, making sure to replace every reference to that variable. One of the axioms about PTrees is:

```
Theorem PTree.gss
    : ∀(A : Type) (i : positive) (x : A) (m : PTree.t A),
      PTree.get (PTree.set i x m) i = Some x
```

This means that if we update x in some PTree, the old mapping of x will no longer exist, which is the exact definition we want from a substitution (And it takes only logN time, where N is the positive number representing the identifier i.)

The semicanonical Floyd Hoare-rule for forward assignment is:

```
...
@semax Espec Delta (PROPx P (LOCALx Q (SEPx R))) (Sset id e)
      (normal_ret_assert (PROPx P (LOCALx (`eq (eval_id id) (eval_expr e)::Q) (SEPx R)))).
```

The equality `eq (eval_id id) (eval_expr e) is difficult to reproduce in canonical form because it uses (eval_expr e : environ → val). We run into trouble beause canonical form only allows type val to be mapped to by variables in the locals, but we are unable to evaluate a C expression (convert it from expr to val) without an environment. Canonical form is a mapping to val because that allows for syntactic substitutions. We can take advantage of the syntactic expressions of the program to write a standard, easily computable syntactic substitution that implies the substitution fact we need when applied to canonical expressions. We use this technique to create msubst_eval_expr, which looks a lot like eval_expr but operates over symbolic PTrees instead of an environment:

```
Fixpoint msubst_eval_expr (T1: PTree.t val) (T2: PTree.t (val * type)) (e: Clight.expr) : option val :=
  match e with
  | Econst_int i ty ⇒ Some (Vint i)
  | Etempvar id ty ⇒ PTree.get id T1
  | Eaddrof a ty ⇒ msubst_eval_lvalue T1 T2 a
  | Eunop op a ty ⇒ option_map (eval_unop op (typeof a)) (msubst_eval_expr T1 T2 a)
  | Ecast a ty ⇒ option_map (eval_cast (typeof a) ty) (msubst_eval_expr T1 T2 a)
  | Evar id ty ⇒ option_map (deref_noload ty) (eval_vardesc ty (PTree.get id T2))
  | Ebinop op a1 a2 ty ⇒
      match (msubst_eval_expr T1 T2 a1), (msubst_eval_expr T1 T2 a2) with
      | Some v1, Some v2 ⇒ Some (eval_binop op (typeof a1) (typeof a2) v1 v2)
      | _, _⇒ None
      end
  ...
```

One significant difference is that msubst_eval_expr returns an option. This is because it will fail if it is unable to find a mapping for a variable in the symbolic environment that it is given. Instead of (first) performing a syntactic substitution and (then) symbolic-evaluating the expression as we needed to do before evaluating eval_expr, msubst_eval_expr does both in one pass, enabled by the deeply embedded form our local assertions. The symbolic evaluation performed by msubst_eval_expr is partial because there might not be any information about a variable in the assertion.

Here is an example evaluation of how we might call msubst_eval_expr, we use a different font to distinguish specifications from programs, and an underscore to show that a variable is a C identifier (e.g. $\_x$) as opposed to a Coq variable (e.g. y).

msubst_eval_expr [$\_x \mapsto y + 3$] [] (5 + _x)

The symbolic evaluation progresses by matching the expression, so the first thing it sees is the addition operator (Ebinop case above). It recursively evaluates 5 to 5, and x to Some $(y + 3)$ by performing a lookup in the map $[x \mapsto (y + 3)]$. Then since both of the subexpressions evaluated successfully, they can be added together using the same eval_binop that is used by eval_expr giving us a final result:

Some $(5 + (y + 3))$

We write a lemma that requires msubst_eval_expr to succeed (by returning Some):

```
Lemma semax_PTree_set:  ∀ Δ id P T1 T2 R e v,
  msubst_eval_expr T1 T2 e = Some v →
  semax Δ
    (▷ local (tc_expr Δ e) && local (tc_temp_id id (typeof e) Δ e)
            && (assertD P (localD T1 T2) R))
    (Sset id e)
    (normal_ret_assert (assertD P (localD (PTree.set id v T1) T2) R)).
```

msubst_eval_expr must succeed on e in the current precondition, giving a value v. Then, in order to apply this triple, we will need to use the rule of consequence to show two typechecking facts:

- local (tc_expr Δ e): The expression e typechecks in Δ, and
- local (tc_temp_id id (typeof e) Δ e): The type of the type of the identifier we are assigning into matches the type of the expression e.

If we have these two facts and know that msubst_eval_expr succeeds, this lemma calculate the postcondition by modifying T1 with the new mapping.

Although this rule appears complex, the user will never see it. The Floyd automation will choose and apply it automatically, using a derived form of the lemma that builds in the rule of consequence, presenting the typechecking assertions as entailments in their own subgoals.

There is a downside to stating the lemma in this way. The precondition must have mappings for every variable that appears in e. The semicanonical lemma doesn't require this because it uses eval_expr in the local assertion without requiring a complete, successful, symbolic execution. This still works for proofs because `eval_expr $e$ might eventually simplify to `eval_id x, which could appear in other places in the assertion. This is only an inconvenience though, as any triple that was provable in semicanonical form can be proved in canonical form by correctly transforming the assertions.

Floyd's forward assignment rule has an existential to bind the 'old' value of the variable. Our semax_PTree_set avoids this existential by making sure that the only direct reference to the old variable is replaced when the PTree is updated. This simplifies the proofs, as it can be inconvenient and inefficient to deal with those existentials.

In comparison with canonical form, semicanonical form is more convenient for the user when *writing* assertions. PTrees should not be constructed by hand, so we provide a tactic for converting from semicanonical form to canonical form. Semicanonical form allows the LOCAL to remain smaller, because if there is a variable the user knows nothing about, there is no need to add it to the locals.

For example the semicanonical precondition for a program that has a linked list might look like:

```
PROP() LOCAL() SEP(`(lseg LS sh (map Vint contents)) (eval_id _p) `nullval)
```

but in canonical form it would be:

```
assertD nil (localD [_p ↦ p] []) [lseg LS sh (map Vint contents) p nullval]
```

In canonical form, any variable that will be used during the execution of a program must be in the LOCAL because msubst_eval_expr will fail without it. Canonical form is more convenient when moving through a proof of a program. It avoids existentials and substitutions that are slow to simplify, or sometimes don't simplify at all.

Even the current form is not completely canonical. It could be restricted further which would improve performance in some ways, but also inconvenience the user in others. Finding a way to sort the SEP and keep it sorted as new conjunctions are added could lead to more efficient entailment solving. This is because one of the main steps in entailment solving is *cancellation*, which attempts to match each of the r separating conjuncts on the right hand side of an entailment with one of the l conjuncts on the left hand side. If the canceller finds a match, it can remove the matching conjuncts from both sides of the entailment. In an unsorted SEP this takes $l * r$ steps in the worst case. In a sorted SEP it would be decreased to $l + r$, at the cost of making additions to the SEP slightly slower.

This is a harder problem than sorting the LOCAL though, because we want SEP to contain a variety of predicates, including predicates that are created by the user. The other problem is that Coq variables can appear as arguments to the predicates, making it impossible to establish an ordering over them. Even so, any amount of canonicalization of the SEP could help with entailment solving efficiency, so it is worth a look in the future.

The assertions might also be made more canonical by examining the types of pure assertions that often occur and giving them a special location. As an example, propositions about the range of integers commonly occur, especially doing proofs about programs that operate on arrays. Creating a special form for specific parts of pure operations might benefit the automation by making it simple to look for assertions about the range of a particular variable. It might also help future decision procedures by gathering information that is relevant to a particular type of goal. Again, the variety of expressions that we might want to reason about makes organizing this type of assertion more difficult than what we did in the LOCAL.

Chapter 5

Reified Forms in a Reflective Framework

Our proof (semi)automation has been implemented by a program that we call "floyd", written in Coq's Ltac language. Ltac allows for the implementation of such a system, but because it records all of its operations in proof objects, gets very slow for large developments like the ones in VST proofs of C programs.

We improve performance by instead writing our proof automation in Gallina: this is a compilable, statically typed, non-backtracking language with simpler (ML-like) pattern matching on inductive data types, that produces no trace of its execution.

The techniques discussed in previous chapters organize proofs and improve the performance of automation. Initially, these techniques were implemented in the Ltac floyd automation for VST. The floyd tactics can suffer from serious performance issues, leaving frustrating delays between tactic applications. To improve the performance, we create a reflective automation tactic, capable of advancing through most basic blocks in a given C program. The tactic was created with Malecha et al.'s s *Mirror Core* framework [32].

There are a number of operations that need to be implemented to create a complete reflective tactic:

1. a reification tactic to transform proof states into a syntax that can be computed on,
2. functions that compute on the reified syntax, transforming reified syntax into other reified syntax,
3. soundness proofs that the functions make valid transformations on reified proof states wrt Coq,
4. denotation functions to reflect reified syntax back into Coq proof goals, and
5. a tactic to combine these steps into a single, fast, proof state transformation.

This chapter focuses on items 1 and 4. With these complete, we should be able to take the following goal (simplified for readability). We previously discussed the forms of our assertions as they appear in Coq goals.

semax is our name for Hoare triple. We discussed the type context Δ in Chapter 3 and the PROP/LOCAL/SEP form in Chapter 4:

```
=================
semax Δ
  (PROP () LOCAL(temp _x 2; temp _y y) SEP(data_at _z 3))
     w = 3;
     y = 2 / y;
     x = *z;
     if (y)
     { q = 0; }
     else
     { q = 1; }
  (POST)
```

and with the application of a single tactic, called rforward, or reified forward, we quickly get two proof goals back. The first one is the verification condition for the successful evaluation of 2/y:

```
=================
PROP() LOCAL(temp _x 2; temp _y y; temp w 3) SEP(data_at _z 3)  ⊢  !!(y <> 0)
```

The second proof goal is the remainder of the command sequence, the triple that rforward was unable to progress through:

```
=================
semax Δ′
  (PROP () LOCAL(temp _x 3; temp _y (2 / y); temp w 3) SEP(data_at _z 3))
     if (y)
     { q = 0; }
     else
     { q = 1; }
  (POST)
```

currently rforward stopped here because (currently) it can only progress through basic blocks.

5.1 Reification and Denotation

At the heart of Mirror Core is a syntax that is capable of representing any Coq term that is not dependently typed:

```
Inductive expr typ func: Type :=
| Var : var → expr (* an identifier using de Bruijn indices. *)
| Inj : func → expr (* injection of a function into the lambda calculus *)
| App : expr → expr → expr (* left expr applied to right expr *)
| Abs : typ → expr → expr (* abstraction. λ(x : t).e *)
| UVar : uvar → expr. (* unification variable, index into a mapping
                         environment *)
```

The expr type is essentially a lambda calculus where we can inject reified types (using the typ parameter, and functions (using the func parameter). For a thorough discussion of this expr, along with its denotation function, see [32, section 3.1]

The first step in using Mirror Core is to create a tactic for reification. Mirror Core includes a reification plugin written in OCaml[1]. The plugin is modular, meaning it can be updated with rules for translating various languages. The first step is to create a reified syntax for our logic. This syntax is made up of two parts:

1. An inductive data structure with definitions for each type and
2. an inductive data structure with syntax for *interpreted* functions.

The functions are considered interpreted because they have meaning even without examining the denotations. Crucially, Mirror Core can extend a symbol table with *uninterpreted* functions if it must reify a function that it doesn't have a rule for. This allows it to work in the general case, where all symbols may not be known at the time the tactic is created. The denotation of each uninterpreted function is available in a symbol table, but the index in that table is all that can be reasoned about reflectively, since it is impossible to match a function in Gallina.

One of the design decisions to be made is how to reify things that we think of a constants. For example, imagine we want to reify the natural number 3. The actual representation of 3 is S (S (S O)). There are two ways we could represent this in reified form. We could create reified constructors fS (when the denotation of fS is taken, it will become S) and fO. An framework with an extensible reification language will need

[1]One can write a reifier purely in the Ltac language, but it will have quadratic running time. An Ltac that recursively builds a term (A (B (C))) will typecheck C, then B, then B C, then A, then (A (B C)) and so on. A plugin written in Ocaml can reify in linear time. We would like to avoid plugins, because they seem to be less portable than plugin-free developments, but quadratic-time reification is unacceptable

another constructor to allow the extensible part (fS) to fit into the framework's language. In Mirror Core this constructor is called Inj (injection). We also need a reified constructor for function application App. We will discuss Mirror Core's reified syntax more later in the section We reify 3 as

```
App (Inj fS) (App (Inj fS) (App (Inj fS) (Inj fO)))
```

It is easy to see why this reification isn't desirable. It is 3 times as big as the original definition. Instead, if we know that what we are reifying is a constant, which we do for 3. We can create a reified constructor that is parameterized by a natural number fNat. Then we can simply reify 3 as:

```
Inj (fNat 3)
```

Not only is this representation smaller, any functions we have that work on natural numbers can still be used on natural numbers as part of proof automation. We discuss this interaction further in Section 6.1.2.

The reified type definition (the deep embedding of a subset of Coq types) is straightforward, the only particular requirement (guaranteed to Mirror Core by a typeclass instance) is that an arrow type exists:

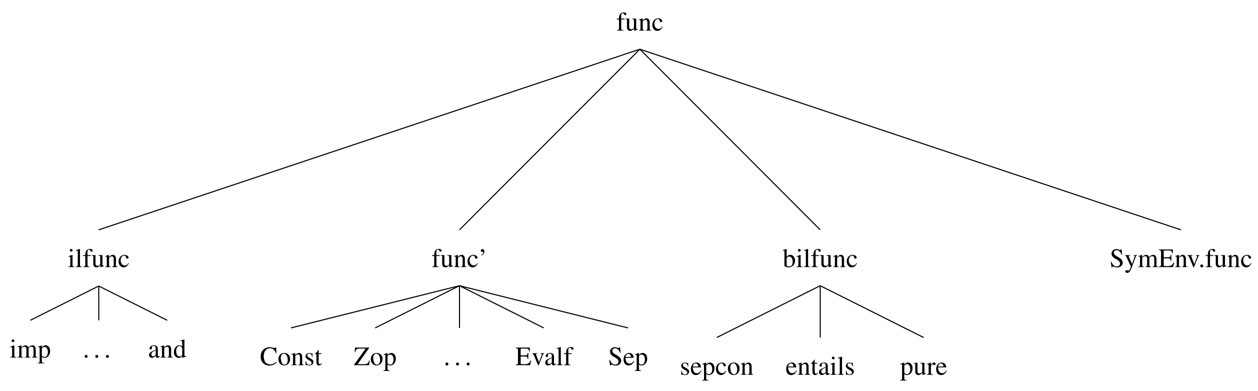
```
Inductive typ :=
| tyArr : typ → typ → typ
| tytycontext : typ
| tylist : typ → typ
| ...
```

There is also a denotation function that maps typ (the reified type) to Type (Coq's type):

```
Fixpoint typD (t : typ) : Type :=
    match t with
        | tyArr a b ⇒ typD a → typD b
        | tytycontext ⇒ tycontext
        | tylist t ⇒ list (typD t )
        | ...
    end.
```

This encoding allows for polymorphic types, as demonstrated by the tylist and tyArr rules, which recursively take the denotation of their type arguments to determine their own type.

We then define an inductive syntax for interpreted functions, which we divide into subgroups to make organization and matching easier. The following tree describes the layout:



This structure is defined as:

Definition func := (SymEnv.func + @ilfunc typ + @bilfunc typ + func')%type.

The first item is a symbolic environment where we can put uninterpreted functions. The second and third are modular logics, a pure and separation logic respectively, that might eventually allow us to take advantage of generic decision procedures such as cancellers. As of now, they are simply used as more constructors, but in future work (discussed in Section 8.1), they will be able to easily plug into decision procedures written by others. Finally, func' is our syntax definition. We show the definition of func' from the bottom of the tree up, as it is defined in Coq.

```
Inductive const := (* Literals for basic data types *)
| fN : nat → const
| fZ : Z → const
| fint : int → const
| fint64 : int64 → const
| ...

Inductive eval := (* C light expression evaluation *)
| feval_cast : type → type → eval
| fderef_noload : type → eval
| feval_field : type → ident → eval
| feval_binop : binary_operation → type → type → eval
| feval_unop : unary_operation → type → eval
| feval_id : ident → eval.
```

```
Inductive other :=
| feq : typ → other
| fnone : typ → other
| fsome : typ → other
| ...

...

Inductive func' :=
| Const : const → func' (* Literals for basic data types *)
| Zop : z_op → func' (* Operations on mathematical integers *)
| Intop : int_op → func' (* Operations over 32 bit integers *)
| Value : values → func' (* C values *)
| Eval_f : eval → func' (* C light expression evaluation *)
| Other : other → func' (* Uncategorized things *)
| Sep : sep → func' (* Separation logic predicates *)
| Data : data → func' (* data types *)
| Smx : smx → func' (* Predicates relating to our triple *).
```

There are two types of functions to note here. The first is functions that take typ as an argument. These are polymorphic functions that operate over the polymorphic types we have already defined. The constructors feq, fnone and fsome above are examples of this. The other interesting function definitions are constructors with arguments of other types. These functions are said to take *constants* as arguments. This is very convenient, because as discussed in chapter 2, our shallowly embedded logic has deeply embedded sub languages that should not need to be reified in order to be computed on. These include things like the types of C variables and all parts of C expressions, as well as Z, nat, and any other Coq data type that might appear and be made up completely of constructors.

Next we need a function that defines the reified type of the functions represented by each constructor:

```
Definition typeof_const (c : const) : typ :=
 match c with
| fN _ ⇒ tynat
| fZ _ ⇒ tyZ
| fint _ ⇒ tyint
| fint64 _ ⇒ tyint64
| ...
```

```
Definition typeof_eval (e : eval) : typ :=
 match e with
| feval_cast _ _ ⇒ (tyArr tyval tyval)
| fderef_noload _ ⇒ (tyArr tyval tyval)
| feval_field _ _ ⇒ (tyArr tyval tyval)
| feval_binop _ _ _ ⇒ (tyArr tyval (tyArr tyval tyval))
| feval_unop _ _ ⇒ (tyArr tyval tyval)
| feval_id _ ⇒ (tyArr tyenviron tyval)
end.

Definition typeof_other (o : other) : typ :=
match o with
| feq t   ⇒ tyArr t (tyArr t typrop)
| fnone t   ⇒ tyoption t
| fsome t   ⇒ tyArr t (tyoption t)
| ...
end.

Definition typeof_func' (f: func') : typ :=
match f with
| Const c   ⇒ typeof_const c
| Zop z   ⇒ typeof_z_op z
| Intop i   ⇒ typeof_int_op i
| Value v   ⇒ typeof_value v
| Eval_f e   ⇒ typeof_eval e
| Other o   ⇒ typeof_other o
| Sep s   ⇒ typeof_sep s
| Data l   ⇒ typeof_data l
| Smx t   ⇒ typeof_smx t
end.
```

And then define the denotation function, which is dependently typed on the application of the typeof function:

```
Definition constD (c : const)
: typD (typeof_const c) :=
match c with
| fN c | fZ c | fPos c | fident c | fCtype c | fCexpr c | fComparison c | fbool c | fint c
| fint64 c | ffloat c | ffloat32 c | fenv c | fllrr c
                                                    ⇒ c
end.

Definition otherD (o : other) : typD (typeof_other o) :=
match o with
| feq t ⇒ @eq (typD t)
| fsome t ⇒ @Some (typD t)
| fnone t ⇒ @None (typD t)
| ...
end.
```

```
Definition evalD (e : eval) : typD (typeof_eval e) :=
match e with
| feval_id id ⇒ eval_id id
| feval_cast t1 t2 ⇒ eval_cast t1 t2
| fderef_noload t ⇒ deref_noload t
| feval_field t id ⇒ eval_field t id
| feval_binop op t1 t2 ⇒ eval_binop op t1 t2
| feval_unop op t ⇒ eval_unop op t
end.

Definition funcD (f : func') : typD (typeof_func f) :=
match f with
| Const c ⇒ constD c
| Zop z ⇒ z_opD z
| Intop i ⇒ int_opD i
| Value v ⇒ valueD v
| Eval_f e ⇒ evalD e
| Other o ⇒ otherD o
| Sep s ⇒ sepD s
| Data l ⇒ dataD l
| Smx t ⇒ smxD t
end.
```

Now we must tell Mirror Core's reification tactic [32, section 4.4.1] how to use the syntax that we created. This is done by creating tables of rules that map parts of proof goals to reified syntax. Mirror Core's reification tactic is a Coq plugin which provides syntax for defining these rules:

```
Reify Pattern patterns_vst_typ += (!!bool) ⇒ tybool.


Reify Pattern patterns_vst_hastype +=
    (RHasType bool (?0)) ⇒ (fun (a : id bool)
                                        ⇒ (@Inj typ func (inr (Const (fbool a))))).


Reify Pattern patterns_vst +=
    (!!@eq @ ?0) ⇒ (fun (a : function reify_vst_typ)
                        ⇒ @Inj typ func (inr (Other (feq a)))).
```

This selection of rules shows most of the features that can be used. The identifiers patterns_vst_typ, patterns_vst_hastype, and patterns_vst tell the plugin which table the rule belongs to. Elsewhere in the file, the tables are defined and given an order in which they will be tried by the reifier. The first pattern defines the reification of the bool type. It simply matches the type !!bool where !! is notation that tells the plugin this will be an exact match by equality. When the type is matched, it tells the reification tactic that it should emit tybool.

The second rule doesn't make an exact match, instead matching the type of the expression with bool. The RHasType pattern takes an argument, denoted by ?0. The 0 means that the variable is given to the first argument (variable a) of the function on the right hand side of the rule. The type id bool of the argument means that it will be passed a Coq bool as opposed to a reified bool. id is just the identity function, but its presence tells the reification tactic how to process the value that is supplied to the function, in this case, id says that there is no further processing required.

The next rule demonstrates reifying a polymorphic equation. Of note is the type of the argument a. It is function reify_vst_typ which means that it will take a Coq type and reify it into a Mirror Core type.

The features shown here are sufficient to define reification for all of the symbols in VST[2]. Mirror Core comes with built-in representations of Coq proof states and lemmas, as well as tactics that reify into those representations. There is no further customization needed, those tactics are automatically created given the customization we have presented here.

As an example, the following Coq term:

```
(fun (x : val) ⇒ plus x (Vint 3))
```

could be reified as

```
Abs tyval (App (App (Inj fplus) (Var 0)) (App (fvint (Inj (fint 3)))))
```

where constructors that start with the letter "f" are constructors in func, similar to feq above.

Although Mirror Core can automatically generate a tactic for reification, it can't generate an equivalent tactic for taking the denotation of a reified expression. The denotation is simply represented by a function that we have already defined. It isn't the functionality that is difficult, but how the denotation is unfolded. Coq has a variety of tactics that perform $\beta\eta$-reduction. An example is vm_compute which unfolds every definition in an expression, applies functions, and simplifies match statements. The vm_compute tactic is notable because it has the best performance of the Coq reduction strategy. It achieves this performance at the cost of increasing the trusted computing base by compiling the goal to bytecode and executing in a virtual machine. Many of Coq's reduction tactics behave badly in the presence of opaque variables. For example, consider the following function, lemma, and resulting proof state:

[2]We currently use 184 "Reify" commands in a 500 line file to define reification for our logic

```
Fixpoint do_n (f : nat → nat) n d :=
match n with
| S n' ⇒ f (do_n f n' d)
| _⇒ d
end.

Goal ∀n, do_n (fun n ⇒ if (ltb 1 n) then n else n) 2 n = n.
vm_compute.
```

```
  n : nat
  ============================
   (if match
         (if match n with
              | 0 ⇒ false
              | 1 ⇒ false
              | S (S _) ⇒ true
              end
          then n
          else n)
       with
       | 0 ⇒ false
       | 1 ⇒ false
       | S (S _) ⇒ true
       end
    then
     if match n with
         | 0 ⇒ false
         | 1 ⇒ false
         | S (S _) ⇒ true
         end
     then n
     else n
    else
     if match n with
         | 0 ⇒ false
         | 1 ⇒ false
         | S (S _) ⇒ true
         end
     then n
     else n) = n
```

Because n is a Coq variable, it can't be evaluated in a match, and the if statement can't be resolved.

This means that the recursive function will appear three times for each recursive call. So if we instead evaluate the following:

```
Goal ∀n, do_n (fun n ⇒ if (ltb 1 n) then n else n) 2 n = n.
vm_compute.
```

we get 3^{50} or approximately 7×10^{23} unfoldings of the function. The same problem arises with many quantified predicates that are unable to unfold completely.

There is no problem in reifying a complicated term that has this behavior, giving us a term that looks something like exprD (complex_thing_reified). If we run vm_compute on this term to try to calculate the denotation, however, the complex thing will also be unfolded once the denotation is taken, resulting in a likely massive term that can take a very long time to compute. Even if it didn't take a long time to compute, it is possible that the user had definitions that they didn't want unfolded, and unfolding those definitions could make the resulting triple difficult to understand. The solution to keeping these definitions folded is to carefully unfold all of the definitions that are part of the denotation function, while leaving everything that belongs to the original term alone. Coq has a tactic called cbv that is perfect for just this purpose. The cbv tactic can take a whitelist of definitions that it should efficiently unfold, considering all other definitions opaque.

We can also supply cbv with a blacklist. This might be more convenient if we knew all of the functions that will appear in our proof goals, because the blacklist would likely be shorter than the whitelist. Since we allow the user to supply functions and predicates to our logic, though, specifying a whitelist is the only viable option.

The problem with cbv is that whitelists can't be defined modularly. That is if I define a tactic

```
cbv [ def1 def2 def3 ]
```

There is no way that I can append another white list onto the one I have defined. At best, a predefined number of names can be supplied as arguments:

```
Ltac do_cbv arg1 arg2 :=
cbv [ def1 def2 def3 arg1 arg2 ]
```

Our cbv whitelist is composed of over 700 constructors, some copied from Mirror Core's whitelist, some copied from Charge!, and some of our own. A very convenient feature for cbv would be a method for composing two whitelists.

5.1.1 Representing Dependent Types

In (pencil-and-paper) separation logic, since Reynolds [39], we abbreviate (p.head $\mapsto$ h $*$ p.tail $\mapsto$ t) with (p $\mapsto$ (h,t)), where (implicitly) there is some record type τ with components (head,tail). We can view this "composite maps-to" as a dependently typed operator, p $\mapsto_\tau$ v, where τ is the (deeply embedded syntactic) description of a C-language data type, and the type of v depends on the value of τ. In our separation logic, we name this dependently typed composite maps-to data_at. Qinxiang Cao in our research group is responsible for the definition of data_at and the operators, theory, and proof automation related to it.

```
reptype (t: type) : Type
data_at : Share.t → ∀ t : type, reptype t → val → mpred

...
| fdata_at : type → sep (*reified data_at)*
...
```

This predicate is a key factor that makes VST usable, but it is difficult to reify because it is dependently typed. Because data_at specifies the behavior of a program instead of just the program's memory organization, data_at relates C values to Coq values. This requires the use of the reptype function, which is simply a mapping from C types to Coq types. Unfortunately, Mirror Core doesn't support dependently typed functions. We are able to avoid this restriction, however, because we know that reptype will always be applied to a C type that comes from a *real* program. C types from real programs are always made completely from constructors with no Coq variables or opaque symbols. When τ is a ground term, then (reptype τ) can be computed to a concrete Coq type before reification. The data_at function with reptype fully evaluated on an argument is no longer a dependently typed function so it can be reified. The tricky part is that we can't use reptype as the type of the reified function because the reified function needs to have reified types. So we define a function reptyp (no e) with the same implementation as reptype except for every Coq type returned in reptype is exchanged for a reified type in reptyp. Here are examples from both functions to illustrate:

```
Fixpoint reptyp (ty: type) : typ :=
  match ty with
  | Tvoid ⇒ tyunit
  | Tarray t1 sz a ⇒ tylist (reptyp t1)
  ... end.

Fixpoint reptype (ty: type) : Type :=
  match ty with
  | Tvoid ⇒ unit
  | Tarray t1 sz a ⇒ list (reptype t1)
  ... end.
```

Then we say the reified type of the data_at function is:

```
Definition typeof_sep (s : sep) : typ :=
match s with
  | fdata_at t ⇒ tyArr tyshare (tyArr (reptyp t) (tyArr tyval tympred))
... end
```

But then to define the denotation function we need a definition:

```
Definition typD_reptyp_reptype: ∀ t, typD (reptyp t) = reptype t.
```

This definition is needed for the denotation function over fdata_at to typecheck. We left out the body of the definition because it is built by Coq tactics to deal with the dependent types involved. We call it a definition rather than a lemma because we will need to be able to compute it to take a denotation. This means that when we complete the construction of the function using tactics, we will conclude with **Defined**, which like **Qed**, checks that the constructed term typechecks, but unlike **Qed**, leaves the term transparent so that it can be computed on.

This is a significant amount of work to be put into reifying a dependently typed function. It doesn't work in the general case either; this technique only works if you are sure that the dependent type will compute away when the function is applied to some number of its first arguments. For the foreseeable future of VST, this dependent typing capability is sufficient, but it might make Mirror Core impossible to use in some other projects with more difficult dependent types.

Chapter 6

Operations on Reified Terms

Once we have the ability to move from Prop to reified syntax (by reification) and back (by reflection, or unfolding the denotation function), the next step is to create Gallina functions that operate over the reified terms we can create. These functions can then be proved sound with respect to Coq's logic, enabling a single Ltac to encapsulate all of the reflective behavior.

6.1 Reified Tactics

A Mirror Core tactic, called an *Rtac* [32, chapter 6]), is a Gallina program that transforms a reified proof state into a reified result. This result can be one of three things:

1. Solved means that the goal can soundly be discharged,
2. Fail means that the tactic didn't make any progress, and the goal should not be changed, and
3. More e' means that the tactic made progress but didn't solve the goal. The new goal should be e'.

To be used as a tactic, these programs from expressions to results must be proved sound with respect to the Coq implication. That is, for tactic tac (greatly simplified):

```
Definition rtac_sound tac :=
∀ reified_state,
goalD reified_state → goalD (tac reified_state)
```

Where goalD is a denotation function for reified proof goals. We have simplified this presentation by leaving out the variables and unification variables that might be created or instantiated by the execution of the tactic. These variables significantly increase the complexity of rtac_sound, but the meaning is approximately the same. Throughout this chapter, we have removed references to these variables and environments for the sake of readability. All of the tactics that were written for VST ignore these variables, simply passing their contents on so that they can be used by built-in Mirror Core tactics that do modify them.

Mirror Core provides a number of tactics that are already proved sound, assuming the arguments they are applied to are also sound. The list below details a core set of Rtac connectives. We use *tac* to represent Mirror Core tactics and *lem* to represent reified lemmas.

INTRO	Pulls a universally quantified variable or the left hand side of an implication into the context (like intro). If the head of the goal is an existential quantifier, a unification variable is created in its place (like eexists).
THEN *tac1 tac2*	Run *tac1* followed by running *tac2* on any resulting subgoals (like the ";" tactic). The tactic fails if either *tac1* or *tac2* fails.
REPEAT *n tac*	Runs the tactic *tac n* times, or until it fails.
TRY *tac*	Runs the tactic tac but succeeds and does nothing when tac fails.
APPLY *lem*	Applies *lem* to the current goal. One subgoal is created for each hypothesis of *lem*. All variables of *lem* must be unified.
EAPPLY *lem*	Similar to APPLY but generates unification variables for any unmatched variable of *lem* instead of failing.
FAIL	Ignore the proof state and fail.

The combination of these tactics is sufficient to create the main structure of a tactic to progress through a C basic block. We start by creating a tactic that will apply to a triple with a single statement:

```
Definition APPLY_SKIP := (APPLY typ func skip_lemma).
```

where skip_lemma is the automatic reification of semax_skip, the Hoare logic rule for the C light ”skip” statement.:

```
Axiom semax_skip:
  ∀{Espec: OracleKind},
   ∀Delta P, @semax Espec Delta P Sskip (normal_ret_assert P).
```

Other lemmas are more complicated. For example the assignment lemma that we reify is:

```
1   Lemma semax_set_localD:
2     ∀temp var ret gt
3              id (e: Clight.expr) ty gs P T1 T2 R Post v,
4      ∀{Espec: OracleKind},
5           typeof_temp (mk_tycontext temp var ret gt gs) id = Some ty →
6           is_neutral_cast (implicit_deref (typeof e)) ty = true →
7           msubst_eval_LR T1 T2 e RRRR = Some v →
8           tc_expr_b_norho (mk_tycontext temp var ret gt gs) e = true →
9           assertD P (localD (PTree.set id v T1) T2) R = Post →
10          semax (mk_tycontext temp var ret gt gs)
11                 (▷ (assertD P (localD T1 T2) R))
12                 (Sset id e)
13                 (normal_ret_assert Post).
```

The core of this rule is the standard Hoare assignment rule $\{P\}\, id := e\, \{\exists v, x = e[v/id] \&\& P[v/id]\}$. We have added side conditions to make the rule sound for C and optimized the rule for canonical form by removing the existential and symbolically evaluating e (Section 4.2.1). Now we briefly discuss each line:

- Line 5: id is the name of a C light program variable; looking up this identifier in the type context yields the C light type "ty".
- Line 6: The type of the expression e is compatible with the type ty
- Line 7: e symbolically evaluates in the precondition to v (see Section 4.2.1)
- Line 8: e type checks (See Chapter 3)
- Line 9: This is simply the postcondition. It is in a separate equality as a side condition due to a limitation with Mirror Core's variable unification

To apply the reified lemma automatically, first the conclusion of the (reified) lemma must match the proof goal, then we must solve each of the subgoals that remain from the application. That means we must create Mirror Core tactics that can solve each of the goals that will arise. The Mirror Core tactic application is:

```
Definition FORWARD_SET
...
  THEN (EAPPLY typ func (set_lemma temp var ret gt i e0 ty)
       (TRY (FIRST [REFLEXIVITY_OP_CTYPE tbl;
                    REFLEXIVITY_MSUBST tbl;
                    REFLEXIVITY_BOOL tbl;
                    AFTER_SET_LOAD tbl;
                    REFLEXIVITY tbl]))
...
```

where the arguments to set_lemma are constants provided to the lemma as discussed in section 5.1.1. First we do the application of the lemma with EAPPLY and then we attempt to apply a number of tactics in order, to deal with the reified subgoals that result from the application. In this case, the subgoals correspond to the hypothesis of semax_set_localD. We list the tactic that applies to the subgoal on each line and then discuss how the tactics work.

- Line 5: REFLEXIVITY_OP_CTYPE
- Line 6: REFLEXIVITY_BOOL
- Line 7: REFLEXIVITY_MSUBST
- Line 8: REFLEXIVITY_BOOL
- Line 9: REFLEXIVITY

AFTER_SET_LOAD cleans up the single remaining subgoal. Although we know exactly which tactic is used to solve which subgoal, Mirror Core does not currently have a tactic to allow for applications of tactics to specific subgoals so we use the FIRST tactic instead. Because most of the tactics start with a basic syntactic match to check that they are being called on a subgoal they can operate on, we don't expect that this is a significant performance problem.

REFLEXIVITY_OP_CTYPE and REFLEXIVITY_BOOL are actually monomorphizations of a single tactic named REFLEXIVITY_DENOTE:

```
Definition REFLEXIVITY_DENOTE (rtype : typ) {H: @RelDec.RelDec (typD rtype) eq}
{H0: RelDec.RelDec_Correct H} : rtac typ (expr typ func) :=
   fun (s : subst) (e : expr) ⇒ (
match e with
| (App (App (Inj (inr (Other (feq _)))) l) r) ⇒
  match func_defs.reflect l rtype, func_defs.reflect r rtype with
  | Some v1, Some v2 ⇒ if @RelDec.rel_dec _eq H v1 v2 then Solved s else Fail
  | _, _⇒ Fail
  end
| _⇒ Fail
end).
```

The arguments to the tactic are rtype, or the type of the equality we are performing reflexivity on, as well as an equality function H over rtype and a correctness proof H0 for that function. The final argument tbl is the table of uninterpreted functions that is generated by reification. The tactic works by first checking that it is being applied to an equality, and extracting the reified expressions (l, r) on each side of the equality. Then it simply takes the denotation of these expressions and runs the provided equality function on the denotations. Then when we want a tactic specialized to a particular type we just apply it to its type argument. If we have declared our type class instances correctly, that should be all we need.

```
Definition REFLEXIVITY_BOOL := REFLEXIVITY_DENOTE tybool.
```

This tactic is very useful and simple, so why is it not provided by Mirror Core? The answer is that it is also unsafe in terms of computation time. Taking the denotation of an expression within a function is only a good idea when the expression is guaranteed to simplify to a small term. If it contains Coq variables, the resulting term could easily be large enough that the computation will take practically forever. This is the case in the majority of situations, but because we have so much computational content already in our logic, this tactic is extremely useful for us. The table argument is the other reason that this tactic is not included in Mirror Core. The type and equality arguments can be provided at the time of tactic creation, but there is no way to provide the table because it will be different for different goals. The inclusion of the table in the arguments is necessary to be able to take the denotation of expressions with uninterpreted functions, but it also means that any tactic that wishes to make use of one of these tactics needs to receive the table as an argument. In practice, this ends up being nearly all of our automation tactics, as well as the top level tactic.

We also use the REFLEXIVITY tactic, which solves syntactic equality, possibly unifying unification variables at the same time. The definition of this tactic is:

```
Definition REFLEXIVITY : rtac typ (expr typ func) :=
fun c s e ⇒
  match e with
| (App (App (Inj (inr (Other (feq ty)))) l) r) ⇒
  match @exprUnify (ctx_subst c) typ func _ _ _ _ _ 3
                                             l r ty s with
    | Some su ⇒ RTac.Core.Solved su
    | None ⇒ RTac.Core.Fail
  end
| _⇒ RTac.Core.Fail
end.
```

It extracts the left and right argument, and then runs Mirror Core's unification function, exprUnify. Returning Solved with the new subst su tells Mirror Core what unification variables now map to, once the tactic is completed. We generally use this for subgoals of the form

```
?23 = expression
```

where the entire tactic unifies a single unification variable with an expression.

Once we have defined the equivalent of FORWARD_SET for each statement we are interested in stepping through, we can combine them in a single tactic we first present an idealized tactic (for clarity, some details are removed that improve the performance):

```
Definition SYMEXE_STEP Struct_env
: rtac typ (expr typ func) :=
  AT_GOAL
    (fun (e : expr typ func) ⇒
          match (get_arguments e) with
          | (Some Delta, Some Pre, Some s) ⇒
            match compute_forward_rule s with
            | Some ForwardSkip ⇒ APPLY_SKIP
            | Some (ForwardSeq s1 s2) ⇒ EAPPLY_SEQ s1 s2
            | Some ForwardSet ⇒ FORWARD_SET Delta Pre s
            | Some ForwardLoad ⇒ FORWARD_LOAD Struct_env Delta Pre s
            | Some ForwardStore ⇒ FORWARD_STORE Struct_env Delta Pre s
            | _⇒ FAIL
            end
          | _⇒ FAIL
          end)
```

This function's type means that it is an Rtac (essentially a function from expr to expr) where the type it operates on is typ and the expression type it operates on is expr over our typ and our func (Section 5.1).

The get_arguments function extracts the type context, precondition, and statement constants from a reified triple. This tactic operates by examining the statement and choosing the correct Rtac to apply to the triple.

There is one tactic here that we haven't discussed before, AT_GOAL, which is used much like **match** in Ltac. It allows you to examine the goal in order to decide which tactic you would like to apply. As long as each of the tactics are sound regardless of what you found when you examined the goal, the entire AT_GOAL tactic will be sound.

The EAPPLY_SEQ tactic is a tactic we defined to apply our reified sequence lemma. It is named EAPPLY because (like the Ltac eapply tactic, in which the "e" stands for "existential"), it has the ability to create a unification ("existential") variable. In the case of the sequence lemma, this unification variable will be P':

$$\text{semax\_seq}\frac{\{P\}\,c\,\{1\}P' \;\; \{P'\}\,c\,\{2\}Q}{\{P\}\,c\,\{1\};c2Q}$$

because P' appears above the line but not below it, it can't be discovered simply by applying the rule. Instead we must prove that some P' exists, which we can do by unifying the unification variable in the process of discharging the subgoals. The other tactics are similar to FORWARD_SET. They eapply a lemma, and then use the same tactics to discharge the resulting subgoals. The SYMEXE_STEP RTac shown above works, but it is inefficient in MirrorCore in two ways: in the course of many steps it accumulates unification variables that (even after they are instantiated) slow down the MirrorCore system. Therefore we add two lines to advise Mirror Core that it should clean up its internal state. This is the real tactic, with much of the middle omitted:

```
Definition SYMEXE_STEP Struct_env
: rtac typ (expr typ func) :=
  THEN (INSTANTIATE typ func)
  (THEN (AT_GOAL
    ...)
    (@RTac.Minify.MINIFY typ (expr typ func) _)).
```

Both of the added lines deal with the mapping from unification variables to optional values. These are carried around in Rtac inside of a computational map. If a unification variable n maps to Some value, that means that the variable has already been unified with value, meaning that it can be substituted in for n wherever n occurs in the expression. This process is performed by the INSTANTIATE tactic, and needs to be done before attempting to unify a lemma using APPLY or EAPPLY. If n doesn't appear in the expression and it was created by Mirror Core, it can be deleted from the table using the MINIFY tactic. This is an important operation, because a large number of unification variables are created and then instantly filled during each lemma application. Without cleaning up the data structure, it can rapidly grow, hurting performance. Variables that were not created by Mirror Core can't be deleted because they exist in a Coq proof goal. Coq requires that all unification variables be instantiated for the proof to check — allowing uninstantiated variables would be unsound in the case that those variables have an uninhabited type. Ideally this cleanup shoul be built into the operations of Mirror Core. Coq automatically performs both of these operations during Ltac applications, and it is unnecessary bookkeeping to require these tactics to be part of user tactics.

6.1.1 Computational finite maps in Coq

It is common in programming languages to want efficient computational mappings from keys to values. The data structure should allow for getting and setting of mappings:

```
Variable get: ∀ (A: Type), elt → t A → option A.
Variable set: ∀ (A: Type), elt → A → t A → t A.
```

These are defined as variables because they are an abstract specification of a generic map, rather than an implementation. Here A is the type of the values and elt is the type of the keys. A map also allows for removal, but we don't discuss that here, as it is not a feature that we make use of. Notice that the get function is partial, so we can always tell if a mapping exists when using the get function.

The following specification describes the interaction between the get and set functions:

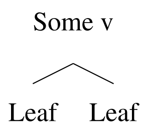
```
Hypothesis gsspec:
  ∀(A: Type) (i j: elt) (x: A) (m: t A),
  get i (set j x m) = if elt_eq i j then Some x else get i m.
```

Again, this is a hypothesis because it is an abstract specification. The lemma gsspec says that if we get variable i after it has been set, we will get the value it was just set with, otherwise, we will get the value from the remaining tree. This also tells us that if we set a mapping twice, a get will return the most recent set.

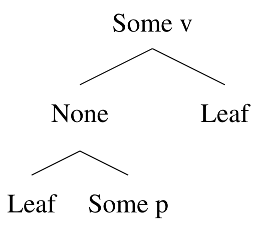
Coq has an efficient implementation of a finite map called a PTree. In a PTree, elt is instantiated with positive, meaning that positive is the type of the keys. The positive type represents binary positive integers and is defined as follows:

```
Inductive positive : Set :=
   xI : positive → positive
 | xO : positive → positive
 | xH : positive
```

The base case is xH which represents the most significant 1 digit, subsequent xO and xI constructors represent 1 and 0 moving from left to right. As an example, the integer 5, or 110 in binary is represented as xO (xI xH) (from here on, we will generally refer to positives in their decimal form). A PTree is a trie over these binary integers. The empty PTree, empty is simply Leaf, and the PTree that represents set 1 v empty is Node Leaf v Leaf.



If we then perform set 5 p, we get the tree



by starting at the root, then matching the value of 5 (x0 (xI xH)) and going left when we see an x0 and right when we see xI when we see xH we are done, and we place the value in that spot. get progresses in the same way, returning the value of the node it stops on when it sees xH, or None when it hits a Leaf.

The worst case complexity of operations on PTrees grows with the largest key in the Tree rather than the number of items in the tree. CompCert also uses these trees, so the largest key tends to be close to the number of items in the tree when the tree is a mapping over program identifiers.

6.1.2 Modular Reflection

We just discussed the general purpose tactics we created to support our proof automation. The remaining tactics we created address a specific challenge that we came across. That challenge is managing the way that computation built into the logic interacts with the reflective framework. In some cases, it is a very smooth interaction, requiring little extra work, but other cases required the bulk of the effort we put into creating a reflective tactic.

We have previously discussed the typechecker (Chapter 3), and a canonical form for assertions (Chapter 4) which allows us to do substitutions computationally. Both of these techniques are computational, in that the proof proceeds by call-by-value β-reduction instead of by the application of inference rules and rewriting. The MirrorCore framework is also computational. This section discusses how these techniques interact. There is no interesting interaction between the substitution and the typechecker because they occur in different places. When we try to apply a reflective framework to a logic that uses these reflective techniques, though, we run into some interesting challenges. This section discusses those challenges and questions if they are avoidable in a differently designed reflective framework.

There are two problems that can arise when attempting modular reflection:

- A function that returns Prop or Type must (in it's reified version) return a reified result if we wish to reason about the result.
- A function that might operate on Coq variables must (in it's reified version) take reified expressions as arguments and will likely return a reified result.

It is possible to have both, neither, or just one of these problems. Having neither problem is convenient, but unfortunately fairly few functions that we use fall into this category.

An example of a function with only the first problem is the reptype function discussed in section 5.1.1. We need to make use of the reptype function to compute the type signature of data_at but if we compute to Type, Mirror Core will be unable to use the result of our computation. Instead we compute to reified type (or typ), which can be examined and compared for equality in Gallina functions. Another option is to convert a function that returns Prop into a function that returns bool such that

```
func_bool args = true → func args
```

If the above lemma is satisfied, a new lemma can be created that uses func_bool in place of func. This new lemma should be trivially provable from the old lemma using the lemma above. The function func_bool will now always return a constructor, so it can be used fully modularly. The cost here is that if func would have returned a new Prop to for the user to solve, func_bool will be forced to simply return false. This is because func_bool should be sound, when it says true func should say True but if the result of func can be anything but True or False, it is impossible for func_bool to be complete, because it can only possibly return true or false.

The other issue is functions whose inputs might contain Coq variables. An example of where we run into this is the PTree.set and PTree.get operations that are in the postcondition of our assignment rules. A PTree is a computational finite map over identifiers (Section 6.1.1). It is a trie over binary integers with functions get, set, and remove. We use these trees in our type context (Section 3.3.1) to map identifiers to static types, and our canonical form (Section 4.2) to map identifiers to symbolic values. This is fully computational, we can perform repeated applications of the lemma:

```
Axiom semax_PTree_set:  ∀ Δ id P T1 T2 R e v,
  msubst_eval_expr T1 T2 e = Some v →
  semax Δ
    (▷ local (tc_expr Δ e) && local (tc_temp_id id (typeof e) Δ e)
              && (assertD P (localD T1 T2) R))
    (Sset id e)
    (normal_ret_assert (assertD P (localD (PTree.set id v T1) T2) R)).
```

which introduces a PTree.set into each postcondition, while msubst_eval_expr does lookups from T1. This can be done with only β-reduction and no rewriting, even though it is likely that T1 will contain Coq variables. We expect it to contain Coq variables because it represents constraints on a symbolic environment. Any program that takes arguments should have at least one Coq variable in T1. As an example:

```
semax Δ
      (assertD P (localD [_x ↦ x; _y ↦ y] []) Q)
      (_z = _x + _y)
      (assertD P (localD [_z ↦ x + y; _x ↦ x; _y ↦ y] []) Q)
```

Both assertions have a T1 where the values are Coq variables. This is because _x maps to a symbolic, or quantified value. The postcondition shows why the value in the mapping must be arbitrary, and not just a single variable. The problem is that there is no way to represent, as a constant, any data structure that contains a Coq variable. This is because there is no way to computationally compare two Coq variables for equality.

In order to computationally check equality over Coq variables, they must first be reified. Reification is designed so that if it sees the same variable twice, it will use the same constructor to represent both instances. The constructors can be computationally compared, so variables can be compared in reified syntax. We will need to do comparisons on the variables in the local assertion almost any time we solve an entailment, so we will need to reify the variables.

During symbolic execution, we want the function application PTree.get t i, to run the lookup-in-tree algorithm as a functional program, replacing the expression with the value bound to i in the tree t. Because the trees T1 and T2 must be reified constructor What that means is that we must rewrite PTree.set and PTree.get to operate on fully reified PTrees. This can be difficult because of the size of the definition of the reified syntax (particularly func discussed in section 5.1).

For example if we have a tree that consists of just a leaf of type val @leaf val, the equivalent reified tree would be Inj (inr (Data (fleaf val))). It is inconvenient to write this large match expression every time we want to match a tree with leaf. Instead we can define as_tree.

```
Definition as_tree (e : expr typ func) : option
  ((typ * expr typ func * expr typ func * expr typ func) + typ) :=
match e with
  | (App (App (App (Inj (inr (Data (fnode t)))) l) o) r) ⇒
    Some (inl (t, l, o, r))
  | (Inj (inr (Data (fleaf t)))) ⇒
    Some (inr t)
  | _⇒ None
end.
```

and instead of

```
match t with
| (Inj (inr (Data (fleaf t)))) ⇒ true
...
```

we can now write

```
match (as_tree t) with
| Some (inr _) ⇒ true
...
```

This allows the function that operates over reified expressions to look very similar to original expression, and also simplifies the proof. It requires a lemma, however, to discover what the tree was if we know the result of as_tree:

```
Lemma as_tree_r : ∀ e t,
as_tree e = Some (inr t) →
e = (Inj (inr (Data (fleaf t)))).
```

Checking the proof of this lemma with **Qed** takes a surprisingly long time. It can take 2 or more minutes on a modern laptop. Again, this is because of the size of the syntax definition. Our func' is made up of almost 90 constructors.

The function and the proof will now have 3 cases. The first have the same behavior (only reified) as the leaf and node cases of the original function. The third case is the case where as_tree returns None. The following is the definition of the reified get function

```
Fixpoint get_reif (i : positive) (m : expr typ func) ty : expr typ func :=
match (as_tree m) with
  | Some (inl (t,l,o,r)) (* Node l o r *)⇒
    match i with
      | xH ⇒ o
      | xO ii ⇒ get_reif ii l ty
      | xI ii ⇒ get_reif ii r ty
    end
  | Some (inr t) ⇒ none_reif t
  | _⇒ (App (Inj (inr (Data (fget ty i)))) m)
end.
```

This is a complete function, meaning we can prove the following lemma:

```
1 Lemma get_reif_eq2 :
2 ∀typ i tr val,
3 exprD' (typtree typ) tr = Some val →
4 exprD' (tyoption typ) (App (Inj (inr (Data (fget typ i)))) tr) =
5 exprD' (tyoption typ) (get_reif i tr typ) .
```

This means that if (tr : expr) has a valid denotation as a PTree (line 3), the reified application of PTree.get to tr (line 4) is equal to the application of the reified get function (get_reif) to tr (line 5). If we didn't require a complete function we could instead write get_reif', which is easier to prove correct:

```
Fixpoint get_reif' (i : positive) (m : expr typ func) ty : option (expr typ func) :=
match (as_tree m) with
  | Some (inl (t,l,o,r)) (* Node l o r *)⇒
    match i with
      | xH ⇒ Some o
      | xO ii ⇒ get_reif' ii l ty
      | xI ii ⇒ get_reif' ii r ty
    end
  | Some (inr t) ⇒ Some (none_reif t)
  | _⇒ None
end.


Lemma get_reif_sound :
∀typ i tr val,
(get_reif' i tr typ) = Some val →
exprD' (tyoption typ) (App (Inj (inr (Data (fget typ i)))) tr)=
exprD' (tyoption typ) val
```

The key difference is that the second function is partial, meaning that it fails when it runs into a PTree with a form that it doesn't expect. There is a fundamental difference in how the two functions can be used and the type of tactics they are built into.

If we have the first function we can use a tactic that essentially does a rewrite, replacing any instance of fget with get_reif. Mirror Core provides a tactic to do this, called SIMPLIFY, which converts any function that maintains semantic meaning into a tactic. Mirror Core provides a function that will replace any application of a reified function with another reified expression. If we use this to create a function that replaces fget with get_reif, we can prove that it maintains semantic meaning by applying the lemma get_reif_eq2. That gives us a proved sound tactic that essentially does a rewrite from a reified function that can't reduce into one that can.

One effect of this approach is that although the rewrite will never fail, it won't guarantee that the PTree is in a form that can be computed on efficiently. It could, for example, create a PTree that is a number of set operations, instead of one that is made up of nodes. In this case, computational efficiency is no better than if we had used a list, and likely worse. Furthermore, the proof about the equality of the semantic meaning of the two types of reified functions requires a lot of effort.

If we use get_reif', which is not complete, we can write a different sort of tactic that might fail. This tactic will have the form:

```
...
match get_reif' i m ty with
| Some ... (* some sound operation assuming that the get succeeded *)
| None ⇒ FAIL (*Rtac that does nothing, and is always sound *)
```

In practice, it is unlikely that the user of the proof system will ever know the difference. We only expect get_reif' to return a result of None if there was a problem with reification, or if a tactic introduced the wrong syntax. Because both of these things are completely under our control, we can be confident that that the user will see the same behavior regardless of which tactic we use.

Another way we leverage incomplete tactics is to solve an equality where there is a unification variable on one side, and the other side needs simplification. This tactic can "simplify" the side that needs it by running a reified function. If the reified function succeeds, it can unify the variable with the result of running the function. This method is easier to prove than the complete method, and likely slightly more efficient, but it may not succeed in all of the cases that the other approach did.

These reified functions are an inconvenience and one of the largest barriers to easily applying a reflective framework in a setting like VST. It is an important open question weather this is avoidable. Can a reflective framework reuse computations that might operate over Coq variables without requiring reified functions?

6.2 The rforward tactic

None of the other steps are useful without a single Ltac tactic that combines them all seamlessly, completely hiding the fact that it is reflective from the user of the tactic. We call the tactic we create to do this rforward or reified forward, where forward is the name of our Ltac tactic that does the same thing. The rforward tactic is heavily based on work by Malecha [32, chapter 6], but I will discuss it here to give an understanding of the modifications made to support our tactics, as well as the interesting relevant parts of the evaluation.

```
Ltac run_rtac reify term_table tac_sound reduce :=
  lazymatch type of tac_sound with
    | ∀ t, @rtac_sound _ _ _ _ _ _ (?tac _) ⇒
          let namee := fresh "e" in
          match goal with
            | ⊢ ?P ⇒
              reify_aux reify term_table P namee;
              let tbl := get_tbl in
              let t := constr:(Some typrop) in
              let goal := namee in
              match t with
                | Some ?t ⇒
                  let goal_result := constr:(run_tac' (tac tbl) (GGoal namee)) in
                  let result := eval vm_compute in goal_result in
                  match result with
                    | More_ ?s ?g ⇒
                      set (g' := g);
                      set (sv := s);
                      let gd_prop :=
                          constr:(goalD_Prop tbl nil nil g') in
                      let gd' :=
                        reduce g' gd_prop in
                      cut (gd'); [
                          change (gd_prop →
                                  exprD_Prop tbl nil nil namee);
                          cut (goal_result = More_ sv g');
                          [ exact_no_check
                              (@run_rtac_More tbl (tac tbl)
                                sv g' namee (tac_sound tbl))
                           | vm_cast_no_check
                              (@eq_refl _(More_ sv g'))
                             ]
                        | clear sv ]
                    | Solved ?s ⇒
                      exact_no_check (@run_rtac_Solved tbl (tac tbl) s namee (tac_sound tbl)
                        (@eq_refl (Result (CTop nil nil)) (Solved s) <:
                          run_tac' (tac tbl) (GGoal goal) = Solved s))
                    | Fail ⇒ idtac "Tactic" tac "failed."
                    | _⇒ idtac "Error: run_rtac could not resolve the result from the tactic :" tac
                  end
                | None ⇒ idtac "expression " goal "is ill typed" t
              end
              end; try (clear namee; clear_tbl)
        | _⇒ idtac tac_sound "is not a soundness theorem."
  end
```

Many choices in the creation of this tactic are made for the sake of efficiency, even at the cost of clarity. For example, we cut the solutions to subgoals and then use the exact_no_check tactic where we would otherwise simply use the apply tactic and then solve subgoals individually. This is because exact_no_check does not typecheck or unify anything, it simply fills in the proof with the provided term, not checking that the term matches the proof goal. This means that exact_no_check is an *unsafe* tactic. It will always succeed when it is run, but when the proof is checked using **Qed**, the checking might fail. This is in contrast to most of Coq's tactics, which do enough typechecking that we can be reasonably confident that if the tactic succeeds **Qed** will also succeed. We are confident using this tactic because we know that we are giving it the exact term that will match the proof goal. Therefore, although the rforward tactic uses an unsafe tactic, it uses it in such a way that rforward can be considered safe. If rforward succeeds, the **Qed** will succeed as well.

The tactic here is not just for performing symbolic execution on verifiable C programs — it is an Ltac for running any RTac that takes a reification table as an argument (in the case of rforward it is SYMEXE_sound). This tactic applied to our symbolic execution RTac is an Ltac that performs fast symbolic execution of verifiable C

```
Ltac rforward := run_rtac reify_vst term_table (SYMEXE_sound 1000) cbv_denote.
```

The run_rtac takes an rtac soundness proof as an argument. The first part of the tactic (line 3), simply extracts the tactic from the soundness proof. This means that if we have a soundness proof rtac_sound solve_g, line 3 will find the tactic itself, solve_g.

The next step is to reify (line 7). The only thing to note here is that we pass the reification tactic the name (namee) that it should give the reified term once it has been created. In this step, if we have a goal:

```
=======
g
```

after reification, we will have two new things above the line:

```
namee := reified_g : expr
tbl := g_tbl : SymEnv
=======
g
```

We don't actually modify the goal here, we just create a new coq variable whose value is the reification of g (we will unfold namee as the example continues). Next we find the table by matching its type in the current proof goal (line 8). This simply looks for something of type SymEnv and stores its name.

The next step is to create the term that will be used to compute the result of symbolic execution (line 13). This **let** declaration creates a variable in Ltac and does not change the proof state. The function run_tac' is a Gallina function we defined that runs an RTac on a *reified* goal. We put the reified term inside of the GGoal constructor to create a complete reified goal, and apply run_rtac' to the tactic and the reified goal. This step only creates the term. It doesn't do any reduction. The reason we do this separately is that we will later need the term in both reduced and unreduced forms.

Next we call vm_compute on the term we just created (line 14). Coq's vm_compute compiles a Gallina term to call-by-value byte-code and then runs it, efficiently computing a $\beta\eta\delta\iota$-normal form. This results in a reified term that is the result of running the RTac. It can be either Fail, Solved s, or More g where (s :subst) represents assignments to unification variables, and (g : goal) is a reified goal that is implied by the original goal. If the result is Fail the Ltac tactic does nothing. If it is Solved, we apply a soundness lemma for RTac that says a goal is solved if an RTac is sound and produced a solved result.

The interesting part of applying the Solved lemma (line 37) is that we need to perform the computation of the tactic function using a vm cast (<:). VM Cast is an operation that does the same thing as a normal cast (:), but it compiles to bytecode and uses a virtual machine. In fact, a vm cast is added to the proof term every time that vm_compute is run. In this case, vm cast is simply a way to automatically build a proof by vm_compute (which is a tactic) and provide it as an argument to run_rtac_solved. Here we are proving that the evaluation that we performed on line 14 is equal to the tactic (solve_g) from our example applied to the reified term (reified_g) from our example. The solved case is less efficient but more readable because it uses (<:) for illustration. Using this operator checks that the cast holds. Even though it does this check using vm_compute, it is rerunning a computation that we already performed on line 14. The vm_cast_no_check tactic that we use in the More case avoids this.

The most work remains to be done if the result is More g, meaning that the tactic might have made progress, but was unable to completely solve the goal completely. This will almost always be the case used in rforward, which will usually step through only part of a program, leaving subgoals for the rest of the program, as well as some entailments that it was unable to solve. In this case, we need to include the (possibly quite large) result of running the tactic in the application of the lemma, and simplify the denotation function to leave a result that is readable by the user. The first step here is to create a goal denotation of the result g. Then we use a tactic reduce that was supplied to the tactic. This is generally a cbv tactic with a whitelist, but it might want to do other sorts of cleanup as well. For example, in the future our reduce tactic will attempt to re-fold identifiers. Identifiers in CompCert are idents where

```
Definition ident := positive
```

Because we use idents as constants and run vm_compute on our reified term, once we have done the computation all aliases for identifiers (like _x or _y) are unfolded, and we see only the positive number that they are represented by. In the future, the reduce tactic will need a way to find all of the names that have been unfolded and fold them again, so the user can see variable names instead of numbers.

The lemma for an RTac that returns More is:

```
Lemma run_rtac_More tac s goal e
  (Hsound : rtac_sound tac)
  (Hres : run_tac' tac (GGoal e) = More_ s goal) :
  goalD_Prop tbl nil nil goal → exprD_Prop tbl nil nil e.
```

In this lemma, two of the hypothesis are given as arguments. This is the same as if they were part of the chain as implications, the only difference is that they can be referred to by name later on, and when the proof starts, they will already be above the line, with the name they are given as arguments.

At the time the tactic is run, the goal is still unchanged. The reification tactic has created reified terms, but they are above the line where the exact tactic can't manipulate them directly. The tactic needs to make the proof state match the lemma exactly while leaving the reduced term as the only remaining subgoal.

The first step in the More case is to cut the denotation of the reduced result term that we computed on line 14. This should be a human-readable Coq term that we will call g'. The cut tactic has two subgoals. The first is:

```
=======
g' → g
```

and the second is

```
=======
g'
```

The second goal is exactly what we want to leave for the user, so we will leave it alone. Now the tactic needs to discharge the first subgoal using the soundness proof of the RTac. It uses the change tactic to unify the first subgoal with the form needed to match run_rtac_more. This tactic replaces the goal with another term that unifies with the original goal (Coq can automatically discover that they are equal). In this case we are using change to tell Coq that the denotation of the reified terms of g and g' are equal to g and g'. This leaves us with a new goal:

```
=======
gd_prop reified_g' → exprD_Prop reified_g
```

We ran the RTac that relates reified_g' to reified_g on line 14, so we can use cut again to get two subgoals:

```
=======
goal_result = More_ sv g' → gd_prop reified_g' → exprD_Prop reified_g
```

```
=======
goal_result = More_ sv g'
```

The first goal now matches our soundness lemma exactly, but still needs the soundness proof for the tactic. We already have this in rforward so we apply it using exact_no_check (line 28) which installs it into the proof (without bothering to check that it has the right type, which of course it will).

Similarly, to discharge the second goal, we use vm_cast_no_check (line 31) which will insert a vm cast (<:) but again, not check that the cast is valid, or that it matches the proof goal. This means that when we use vm_cast_no_check we don't need to recompute the (possibly lengthy) tactic run that we already computed on line 14. Instead, we can insert a term that simply asserts that it needs to be computed by virtual machine when the proof is checked.

This solves all of the subgoals but the reduced denotation of what remained when the RTac finished running — that is the reflection of goal g — meaning that all the user of the tactic will ever see is this result. The reflection of the goal g might also contain (or be exclusively made of) equalities representing unification variables that existed in g and were instantiated during the execution of the tactic. So if we had a goal

```
=======
f ?12 = 1 + 2
```

where ?12 is a unification variable. If we used run_rtac with a tactic that solves the goal above, we would actually end up in the More case, because the goal isn't solved unless there is a value given for ?12. Instantiation of unification variables can only be done by Ltac, so while the tactic can solve the goal, it needs to return some information to the proof state. It does this in the form of a goal:

```
=======
?12 = 3
```

where 3 is the value that the tactic discovered that ?12 should be unified with.

6.3 Related work

Computational reflection in Coq has certainly been used before—it is hard to imagine doing the very computational 4-color proof [23] without the use of reflection. But here we will focus on applications to program logics and static analysis.

The CompCert compiler [30] and the VeriSmall static analyzer [3] are both implemented as Gallina programs and both run by extraction to Ocaml, not by computational reflection in Coq. Both of these have decidable entailments, so they don't need to interact with a human. This makes the performance gains from extraction to Ocaml worthwhile. *Verifiable C*, on the other hand, is not just a static analyzer, it's a fully expressive program logic in which users can write program invariants in higher-order logic (actually CiC). No prover for Verifiable C can hope to be complete, so there will always be a need for interaction.

Verifiable C is not the first expressive foundational program logic: Tuerk built one in Isabelle/HOL for a simple C-like language [42], with significant proof automation programmed in Isabelle's tactic language; no performance numbers are quoted. Bengtson *et al.*'s Charge! [8] is a tactical proof automation for a separation logic for the Java programming language, embedded in Coq, with tactical proof automation similar to the Ltac-based forward tactic we have described here. Bengtson *et al.* do not cite performance numbers, but they are terrible. Chlipala [16] reports on a separation logic for a compositional low-level programming language, with Ltac-based proof automation; build-times for proofs are reported in minutes (median = 3 minutes, for an AssociationList microbenchmark). In all of these systems, tactical proof automation is quite slow, and computational reflection would help.

Malecha, Chlipala, and Braibant [33] apply a predecessor of MirrorCore to achieve symbolic execution and entailment solving in the Bedrock assembly language, and report speedup of $4.6\times$ over Ltac for realistic examples.

Bengtson is currently in the process of applying MirrorCore to the Charge! program logic, as we have done here for Verifiable C.

Chapter 7

Evaluation

This chapter presents two tests of the rforward tactic. The first test is the symbolic execution of a program with n statements that assigns 0 into n different variables. This test is actually a best case for the Ltac version because it is able to notice that the precondition is always closed w.r.t the variable we are assigning into and avoid introducing an existential at each step. The difference between this test and one that repeatedly assigns to the same variable is constant rather than asymptotic. We show the performance of Ltac and Rtac as n increases in Figure 7.1.

Each data point is the average of 10 consecutive runs on a modern laptop. The timing information was obtained by Braibant's Timing plugin [11] The amount of memory on the laptop is over 2 gigabytes, which is the maximum amount of memory that 32-bit Coq can use.

Although this is a synthetic test, we consider it a lower bound on the performance increase we will see in the process of a real proof. Real proofs have assertions (preconditions) that might more complex, and are probably smaller than our synthetic preconditions. Numerous other tests indicate that the running time of our symbolic executors are affected primarily by the size of the assertions, rather than the complexity. Our computational speed ups, as described in previous chapters, scale well with assertion size.

The computation that we need to do in both real and synthetic cases is almost all getting and setting from efficient tree maps, where get and set take $\log(n)$ time (where n is the largest index in the domain of the map). Our program input will be compiled by CompCert, and CompCert uses small indices. The remainder of the operations that RTac does are linear (or slower) unifications between lemmas and proof goals, as well as linear (or slower) rewrites by proved equalities. Repeated linear operations are what lead to the $n^{2.5}$ operation we observe in the chart above, not repeated $\log(n)$ gets and sets.

The chart shows that there is a speedup of (typically) $40\times$ (e.g., running $n = 8$ consecutive C-program

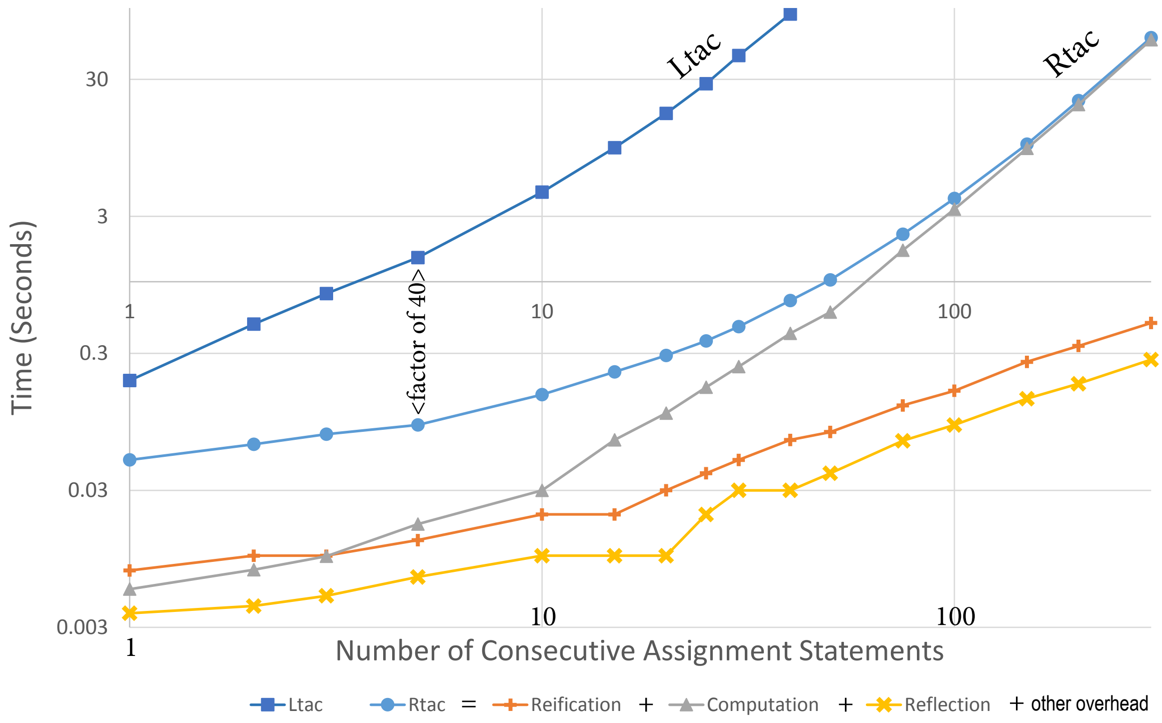


Figure 7.1: Run time of Ltac vs Rtac, for n consecutive C statements (log-log scale).

statements). Speedup improves with scale, because the cost of reification/reflection is paid only once, no matter how many Hoare-logic steps the reified symbolic execution computes: $n = 50$ Hoare logic steps in Ltac takes over 2 minutes, while the same number of steps in Rtac takes only .9 seconds, running almost 150× faster! Even for $n = 1$ there is a 4× speedup. This greatly improves interactivity of the logic. In general it seems that growth of time relative to the number of program steps is quadratic. Our benchmark that uses more complex preconditions scales even better (not shown in a graph), with 74× speedup at $n = 10$.

The *computation* curve, at the upper end, clearly shows a time complexity of $n^{2.5}$. This time complexity is roughly what we expect, because (in this example) the precondition grows linearly with the number of steps, and each step has proof operations at least proportional to the size of the precondition.

On a real-life example, a loop-body from OpenSSL's SHA function[1] takes 336 seconds to verify in our Ltac forward tactic; rforward takes only 12 seconds—a 28× speedup. The sequence includes 13 local-variable assignments, 5 loads, and 1 store, several of which contain huge mathematical expressions (resulting from macro-expansion in the C source code). The assertions in that example are large, with many local-variable and spatial conjuncts.

Unfortunately, after Rtac has efficiently computed a proof, Coq's **Qed** blows up, taking minutes in some cases. There are two things that might cause this. The first possibility is that we may have made a mistake in the construction of our LTac for running reified tactics. In the creation of his tactic, Malecha did numerous

[1]The second loop body in the sha256_block_data_order function of the cited paper [4]; there are 850 nodes in the loop body's abstract syntax tree.

tests and carefully studied the behavior of **Qed** [32, section 4.3]. Our work attempts to duplicate this, but this was only a temporary measure until Mirror Core includes a general purpose tactic to do the same thing. It is likely that switching to a general-purpose Mirror Core tactic will improve both the performance and the **Qed** time of our tactics.

The second possible problem is that Qed blowups tend to occur when Coq cannot find an efficient $\beta\eta$-conversion sequence to prove an equality (even if the tactic script demonstrated one). This problem can be improved in future versions of Coq by allowing the cast operator to carry information about how a reduction should be performed. For example, when we use cbv with a whitelist to show that a reified term is equivalent to a term in Prop, **Qed** will still use a naive reduction, which could start by unfolding the term in Prop and take a very long time. Including this sort of hint doesn't affect the correctness of Coq at all, in fact, if someone doesn't trust it, they can ignore it completely. All it should affect is speed, as it is simply a heuristic for evaluation order.

Chapter 8

Conclusion

Building a sound, foundational, program logic for C is difficult, but even harder is making that logic usable. Along with completeness problems, and the amount of background needed to understand such a logic, performance problems can render a logic nearly unusable. This thesis addresses the problem of making an efficient (yet still foundational) symbolic executor by using computational reflection in several different ways.

The previous chapters described the techniques that lead to a significant speedup in the Verified Software Toolchain (VST's) proof automation. Chapter 3 discusses a fully computational typechecker, capable of guaranteeing expression evaluation for many expressions, while producing assertions for those expressions it can't guarantee. These assertions are given to the user to solve, giving them control where it is needed most, while dealing with as many of the easier cases as possible.

Chapter 4 introduces a new canonical form for assertions. This canonical form allows substitutions to move from an inefficient semantic implementation, to an efficient syntactic one. It does this using fast, computational data structures to increase the organization of assertions, leading to an increase in syntactic information about those assertions. In conjunction with functions that can perform symbolic evaluation of an expression in a precondition instead of a state, this canonical form simplifies and speeds up many of the basic lemmas in our logic.

Chapter 5 discusses making the entire logic computational by creating efficient functional programs to use in place of the slower tactical programs that previously drove the logic. Like the tactical programs, the functional programs are tailored to our application domain. We created the reflective tactics using the Mirror Core reflective framework, which gave a great start down the path of a computational logic. We ran into trouble, however, with some of the more advanced features of our logic. The chapter discusses advances we made to allow us to use dependently typed predicates in a reification system that isn't written to allow dependent types. It also discusses how we were able to fit the computational nature of the typechecker and the canonical form into place inside of Mirror Core's computation.

Using these techniques, we created a tactic rforward that is capable of quickly advancing through C basic blocks. Chapter 7 shows a performance increase of around 40$\times$ for our tests, and discusses why we expect this to be close to a lower bound on the performance increase to be observed in general.

8.1 Possible future work

Program verification using forward Hoare-logic (or separation logic) rules intersperses the application of Hoare rules with entailment solving. Entailments can arise from interfaces between user-written and machine-generated assertions, as well as from side conditions from the application of some of the Hoare-logic rules.

This thesis has focused on symbolic execution via the application of Hoare rules. Our reified tactics solve some of the most trivial entailments (which is a significant portion of the entailments), but still reflects many back to the user to solve. Currently, a user proving a C program correct will spend most of their time proving these entailments, and rewriting their assertions to ease in this solving. Entailments can be solved (or partially solved) by the Ltac entailer tactic [5, Chapter 26], but it is slow, and not as complete as a user might hope. It should be possible to use the facilities of Coq and Mirror Core to implement entailment-solving (or entailment simplification) much faster in a reified form.

Now that we have tactics for reification and symbolic execution of goals, we have the ability to generate reified entailments. With entailments already in reified form, they are ripe for use with other computational techniques. The entailments can be split into pure and separation entailments by extracting pure facts from separation facts. The resulting pure entailments look much like the verification conditions that the verification community has been solving for years. SMT solvers have been carefully tuned to solving this type of problem and now research such as SMTCoq [27] is showing promising steps towards leveraging SMT solvers to get fast, foundational proofs in Coq. Ideally, this research could be made general enough that it would not only be able to solve our goals, but many of the recurring mathematical goals that Coq users regularly run into. Research in this direction could have a large impact on the Coq community.

Efficient procedures also exist for solving separation logic assertions [37, 12, 36, 18] although many of them apply to specific data structures like linked lists, and mostly focus on shape analysis, while our assertions contain information about contents as well. The more automation we can use to try to solve separation logic goals, the more immediately useful our program logic will become, particularly if the automation is implemented as efficient Coq computation.

Looking beyond entailment solving, the completeness of the reflective tactic could be improved. It currently needs to stop for interaction when the tactical system needs to be supplied with invariants or witnesses for function calls. Ideally, the code could be annotated with assertions in the style of Dafny [28], allowing the reified tactics to proceed through while loops and function calls without stopping. Challenges here include picking a good specification language, translating that language into a form compatible with our logic, and developing the new tactics that would be needed for applying new reified lemmas.

As the level of non interactive automation in our logic increases, the confusion of the user is likely to increase as well. In the current system, when you are presented with an entailment you have an understanding of where it came from. That is because you have been stepping through the program one statement at a time, examining the precondition at each step. As automation starts stepping through basic blocks (or more) at a time, the conditions that are presented to the user might be harder to contextualize. It is important that either automation advance to the point it can solve most of the entailments that will ever arise, or that research goes into how to best present entailments to a user so that their meaning in the context of the program can be understood. Using proof automation in an interactive system such as Coq provides a unique opportunity for allowing proof writers to understand the direction of the verification. If that interaction is lost, it might be better to use an entirely deeply embedded logic.

This thesis has taken important steps towards making the VST logic fast enough to use on a wide range of real programs. The use of the reified tactics as part of the verification of SHA256 show that they can make a strong contribution on the road to verification of libraries that are essential to the security and safety of the world.

Bibliography

[1] Toyota to recall 1.9 mln Prius cars for software defect in hybrid system. *Reuters*, 2014.

[2] Andrew W. Appel. Tactics for separation logic, 2006.

[3] Andrew W. Appel. VeriSmall: Verified Smallfoot shape analysis. In *First International Conf. on Certified Programs and Proofs*, December 2011.

[4] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems*, to appear, 2015.

[5] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.

[6] Andrew W. Appel and Xavier Leroy. A list-machine benchmark for mechanized metatheory. Technical Report RR-5914, INRIA, May 2006.

[7] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin Heidelberg, 2006.

[8] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in Coq. In *ITP*, pages 315–331, 2012.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05: Third Asian Symposium on Programming Languages and Systems, LNCS 3780*, pages 52–68, 2005.

[10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[11] Thomas Braibant. Timing-plugin. `https://github.com/braibant/Timing-plugin`, 2014.

[12] James Brotherston, Dino Distefano, and Rasmus L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, pages 131–146, 2011.

[13] Brian Campbell. An executable semantics for CompCert C. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 60–75. Springer Berlin Heidelberg, 2012.

[14] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *Ninth Source Code Analysis and Manipulation, 2009*, pages 123–124. IEEE, 2009.

[15] Adam Chlipala. *Certified programming with dependent types*, volume 20. MIT Press, 2011.

[16] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.

[17] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[18] Byron Cook, Christoph Hasse, Joel Ouaknine, Matthew Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, 2011.

[19] Keith Cowing. NASA reveals probable cause of Mars polar lander and Deep Space-2 mission failures., 2000.

[20] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[21] Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[22] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

[23] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[24] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *Third International Conference on Interactive Theorem Proving, LNCS 7406*, pages 99–115. Springer, August 2012.

[25] Philippe Herrmann and Julien Signoles. Frama-Cs annotation generator plug-in. `http://frama-c.com/download/rte-manual-Sodium-20150201.pdf`, 2013.

[26] Douglas Isbell, Mary Hardin, and Joan Underwood. Mars climate orbiter team finds likely cause of loss. *NASA Release*, 1999.

[27] Chantal Keller. SMTCoq. `https://github.com/smtcoq/smtcoq`, 2015.

[28] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[29] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[30] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[31] Xavier Leroy. CompCert, 2015.

[32] Gregory Malecha. *Extensible Proof Engineering in Intensional Type Theory*. PhD thesis, Harvard, 2014.

[33] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional computational reflection. In *Interactive Theorem Proving*, pages 374–389. Springer, 2014.

[34] Andrew McCreight. Practical tactics for separation logic. In *TPHOL: International Conference on Theorem Proving in Higher Order Logics*, pages 343–358, 2009.

[35] Jeff Mull. Maldives controversy, September 2010. [Online; posted 13-September-2012].

[36] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.

[37] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[38] Michael Norrish. C-to-Isabel parser. http://www.ssrg.nicta.com.au/software/TS/c-parser/, 2013.

[39] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

[40] Barbara Wade Rose. Fatal dose: Radiation deaths linked to AECL computer errors. *Canadian Coalition for Nuclear Responsibility*, 1994.

[41] Michelle Starr. Fridge caught sending spam emails in botnet attack. *Cnet*, 2015.

[42] Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*, pages 469–484, 2009.