Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic

Adam Chlipala

Harvard University
Cambridge, MA, USA
adamc@cs.harvard.edu

Abstract

Several recent projects have shown the feasibility of verifying low-level systems software. Verifications based on automated theorem-proving have omitted reasoning about *first-class code pointers*, which is critical for tasks like certifying implementations of threads and processes. Conversely, verifications that deal with first-class code pointers have featured long, complex, manual proofs. In this paper, we introduce the Bedrock framework, which supports mostly-automated proofs about programs with the full range of features needed to implement, e.g., language runtime systems.

The heart of our approach is in mostly-automated discharge of verification conditions inspired by separation logic. Our take on separation logic is *computational*, in the sense that function specifications are usually written in terms of *reference implementations in a purely functional language*. Logical quantifiers are the most challenging feature for most automated verifiers; by relying on functional programs (written in the expressive language of the Coq proof assistant), we are able to avoid quantifiers almost entirely. This leads to some dramatic improvements compared to both past work in classical verification, which we compare against with implementations of data structures like binary search trees and hash tables; and past work in verified programming with code pointers, which we compare against with examples like function memoization and a cooperative threading library.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods

General Terms Languages, Verification

Keywords interactive proof assistants, separation logic, low-level programming languages, functional programming

1. Introduction

The desirability of verifying systems infrastructure software has long been recognized. If our operating systems and runtime systems are not correct, then we can hope for little guarantee about the behavior of our applications. Thus, the pay-off of formal correctness verification of systems software is large, but the human cost of verification has been considered to be so great as to offset the benefit. Several recent projects have given reason to reconsider that position. The L4.verified project [18] has finished a complete functional correctness verification for a realistic operating system microkernel, based on a quite non-trivial amount of proof code for the Isabelle/HOL proof assistant [27]. The Verve project [30] has completed a mostly-automated Boogie [1] verification of the core of an operating system which is designed to serve as a kind of run-time system for code written in C#.

The productivity advantage of mostly-automated verification over manual tactic-based proving seems clear. Unfortunately, the well-known automated verification systems restrict specifications to first-order logic, which creates a dramatic restriction on which specifications may be expressed compactly, or even expressed at all. For instance, it seems unlikely that these first-order systems will ever be able to handle a proof of type safety for a non-trivial programming language, but exactly that kind of proof is required to complete the Verve project. It is possible to use different tools for different parts of a verification, but the programmer's task is certainly made simpler by sticking to a single environment. This choice should also help to minimize the trusted code base, which seems quite relevant, given the security consequences of operating system bugs.

There has been a significant amount of work on low-level verification systems that are higher-order in at least two distinct senses: they allow higher-order logic in specifications, and they allow reasoning about code pointers as data. For instance, the Certified Assembly Programming project has suggested several such program logics, including XCAP [24] and SCAP [13]. These systems have been used to complete complex verifications of threading libraries and other examples involving first-class code pointers. Unfortunately, the Coq [8] proofs involved are very long, complex, detailed, and brittle to changes of specification.

In this paper, we introduce Bedrock, a Coq library in which we have attempted to reconcile the above concerns. In particular, Bedrock is:

- **Low-level**: The framework supports verification of programs that, for performance reasons or otherwise, cannot tolerate any abstraction beyond that associated with assembly language.
- **Foundational**: The output of a Bedrock verification is a theorem whose statement depends only on the predicates chosen for the key specifications and on the operational semantics of some machine language. That is, there is no need to trust that the verification framework is bug-free; rather, one need only trust the usual Coq proof-checker and the formalization of the machine language semantics.
- **Higher-order**: Bedrock facilitates quite pleasant reasoning about code pointers as data.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

- **Computational**: Many useful functions are specified most effectively by comparing with "reference implementations" in a pure functional language. Bedrock supports that model, backed by the full expressive power of Coq's usual programming language.
- **Structured**: Bedrock is an *extensible programming language*: any client program may add new control flow constructs by providing their (sound) proof rules. For example, adding high-level syntax for your own calling convention or exception handling construct is relatively straightforward and does not require tweaking the core library code. Unfortunately, for reasons of space, we will not discuss this aspect of Bedrock more in the present paper.
- **Mostly-automated**: Tactics (proof procedures) automate verification condition generation (in a form inspired by separation logic [28]) and most of the process of discharging those conditions. Manual proof effort is generally confined to, first, annotations specifying which simplification rules for abstract impure predicates should be applied at which program points; and, second, hint lemmas about mathematical objects like sets, maps, and lists. Crucially, neither kind of manual work deals explicitly with program syntax, memories, or program states.

The next section of the paper introduces Bedrock with the complete code for two simple examples: swapping two values in memory and incrementing the integer value in every node of a linked list. After walking through the examples, we explain how the reasoning we applied may be generalized into a mechanical procedure for reducing verification conditions in separation logic to simpler conditions that deal only with normal mathematical objects. This forms the basis of a very effective verification framework. To back up that claim, we perform two main sorts of comparison.

First, we compare against some examples implemented with Jahob [32], a representative of recent developments in mostly-automated functional correctness verification of data structure implementations. While Jahob deals with high-level Java programs, in Bedrock we must implement a `malloc` and `free` library before getting started, and all programs must contain explicit memory management code for both the heap and the stack. Despite that handicap, our annotation burden is comparable to that documented for Jahob, and we even achieve much more compact proofs for some examples, including hash tables, thanks to the *computational* specification approach we adopt.

Second, we implement a set of interesting programs that use code pointers in ways beyond usual function calling conventions. These examples cannot even be specified, let alone verified, in classical verifiers. In Bedrock, each of our higher-order examples requires only about a page of annotation, compared to thousands of lines in related work. The examples are an abstract memoization module, operating on imperative functions that compute pure functions over machine words, using arbitrary persistent local state; destructive concatenation of linked lists, written in continuation-passing style using explicit closures, which is the largest example from the first XCAP paper [24]; and a simple cooperative threading library, similar to an earlier XCAP implementation [25, 26]. For the examples where we compare against past XCAP implementations, we reduce annotation overhead by a factor of about 100, and we also pass the qualitative threshold of giving mostly-automated proofs that often adapt automatically to specification changes.

The complete code for the framework and the examples is available online at:

`http://adam.chlipala.net/bedrock/`

2. Bedrock by Example

The C programming language is often described as a "macro assembly language." Bedrock fits that description taken literally. The Bedrock Coq library is parametrized over a machine language with some concept of a basic block. In this paper, we deal with the framework instantiated to a simple idealized machine language. The language is idealized in that it has infinite-sized words and an infinite memory. In other respects, it is a realistic machine language, with a finite supply of global registers, a memory that can be modeled as an infinite array of words, and so on. Further details of this language will not be critical for what follows, and our code examples will provide further demonstrations. We expect that the Bedrock framework is applicable both to more realistic machine languages and, for slightly higher-level programs that do not need to manipulate machine contexts, to common compiler intermediate languages.

Bedrock generalizes the XCAP program logic [24], in the sense that, in place of deduction rules for particular instructions of a fixed machine language, Bedrock can be thought of as having a single rule that operates a basic block at a time. The rule (and the other deductive parts of Bedrock) is parametrized over a standard operational semantics for the machine language in use. Every basic block is assigned a logical precondition, and the basic block rule requires that, if execution enters a block in a state satisfying its precondition, then execution must proceed safely and reach a jump to some block whose precondition is then satisfied. In our idealized language, execution only "goes wrong" when a jump is made to a label that does not exist in the (unverified) program. Therefore, the block preconditions are the main component of program correctness theorems.

XCAP is based on an assertion logic (the language in which preconditions are written) whose design involves some subtleties which we will mostly ignore in this presentation, referring the reader to the first XCAP paper [24] for further details. It is a reasonable approximation to say that the assertion language is mostly a standard second-order logic, in the sense that quantification is allowed over both normal mathematical objects and over specifications themselves, but not over some more exotic domains like functions over specifications. So far, this summary describes a sublanguage of the logic built into Coq. All of the differences stem from an unusual connective of the form $i@@p$, which says that there is a basic block at program counter i whose precondition is implied by specification p. This connective is the source of support for reasoning about sophisticated uses of code pointers, based on the possibility for quantified variables to appear inside p. We may even use second-order variables that stand for other specifications.

Prior work with XCAP has involved coding assembly programs directly. Bedrock makes the programmer's job easier by supporting a structured programming notation, such that programs look much like C code where atomic statements are literal assembly instructions. The details of Bedrock's low-level deductive system do not appear in verifications based on structured programming. Rather, we apply the standard technique of verification condition generation, so that the conditions to be proved do not refer to program syntax. As is usual, verification condition generation depends on invariant annotations in the program code. Structured syntax and its associated proofs are automatically "compiled" to proofs about normal machine code programs.

The rest of the details of Bedrock verification are best introduced through examples that demonstrate the core of our proof methodology, a procedure for simplifying verification conditions that are based on separation logic [28]. Figure 1 provides a quick informal reference for the programming and specification features that we will use.

Program syntax

R_i	Register
`$[E]`	Memory dereference
`L <- E`	Assignment command
`C;; C`	Command sequencing
`Goto E`	Computed jump
`[p] While (E) { C }`	Loop (with invariant)
`Use [lemmaName]`	Proof hint

Program states `st`

`st#`R_i	Project value of register
`st.[E]`	Project value stored at memory address
`st[L <- E]`	Update based on assignment command

Assertions `p`

`st ~> p`	State predicate (`st` bound in `p`)
`p /\ p`	Conjunction
`Ex x. p`	Existential quantification
`Ex x : T. p`	Existential with type annotation
`i @@ A`	Assertion about precondition of a code pointer
`![ P ] st`	Separation logic assertion

Separation logic assertions `P`

`[< p >]`	Lift normal assertion
`u ==> v`	"Pointer `u` points to value `v`"
`P * P`	Separating conjunction
`Ex x. P`	Existential quantification
`Ex x : T. P`	Existential with type annotation
`![ x ]`	Second-order specification variable
`!{ f` v_1 `...` v_n `}`	Abstract predicate

Figure 1. Bedrock syntax reference

```
Definition swap := bmodule {{
  bfunction "swap" [st ~> Ex fr : hprop,
    Ex a : nat, Ex b : nat,
    ![ st#R0 ==> a * st#R1 ==> b * ![fr] ] st
    /\ st#Rret @@ (st' ~>
      ![ st#R1 ==> a * st#R0 ==> b
         * ![fr] ] st') ] {
    R2 <- $[R0];;
    $[R0] <- $[R1];;
    $[R1] <- R2;;
    Goto Rret
  }
}}.

Theorem swapOk : moduleOk swap.
  structured; sep.
Qed.
```

Figure 2. A Bedrock function implementing pointer swapping

2.1 Swapping the Values at Two Memory Locations

Figure 2 gives the complete code to implement and verify a Bedrock function for swapping the values at two memory addresses. This code is processed by the normal, unmodified Coq interpreter, thanks to the use of Coq's syntax extension mechanism (or "macro system"). We start by defining a code module with the `bmodule` keyword. Inside the module is a single function `"swap"`, introduced with the `bfunction` keyword. After the function name appears the function precondition, which we will turn to shortly. First, we note that the function body here is a list of assembly instructions. The function inputs are in registers `R0` and `R1`, and the first three instructions use register `R2` as a temporary in swapping the contents of the memory cells pointed to by the inputs. The notation `<-` is for assignment, while `$[E]` stands for the memory cell pointed to by expression `E`. The last instruction returns from the function by jumping to the return pointer stored in register `Rret`.

The function precondition may appear daunting at first. It uses a few standard concepts with perhaps unusual ASCII syntax, along with a few less usual concepts. First, the notation `st ~> p` is a special "lambda" form that triggers the use of a special parsing nonterminal for `p`, so that `p` is parsed as an XCAP-style assertion. The variable `st` is bound in `p`; it is the function argument, standing for a machine state.

The body of `swap`'s precondition begins with three existential quantifiers, written `Ex`. The simplest two of the three bound variables are `a` and `b`, which stand for the initial contents of the memory cells pointed to by `R0` and `R1`, respectively. In this idealized machine language, memory cells contain natural numbers, so we annotate `a` and `b` with the Coq type `nat`. We will return shortly to the remaining variable `fr`.

After the quantifiers, we have an assertion of the form `![ P ] st`, where `P` is an assertion of separation logic that we are requiring must hold in machine state `st`. Within these brackets, we may write `u ==> v` to assert that the memory cell at address `u` holds word `v`. The separating conjunction `P * Q`, which has lower parsing precedence than `==>`, asserts that the machine memory may be broken into two disjoint pieces, such that `P` satisfies one and `Q` the other. In our example, we calculate two memory cell addresses for "points-to" facts using the `#` operator, which projects a particular register value out of a machine state.

A third assertion is added to this memory precondition with the separating conjunction: we see the variable `fr` used with the syntax `![fr]`. Inside a separation logic assertion, the `![_]` notation indicates a *specification variable*. That is, in this example, we are quantifying over a memory specification `fr` and then using it to describe one part of the initial memory. Concretely, `fr` has type `hprop`, the type of predicates over partial heaps. The name `fr` is meant to be suggestive of *frame condition*, a condition that describes all parts of memory that are irrelevant to the present function. Standard separation logic contains the *frame rule* for the statement partial correctness judgment $\{P\}s\{Q\}$:

$$\frac{\{P\}s\{Q\}}{\{P * R\}s\{Q * R\}}$$

That is, when a statement satisfies a particular precondition and postcondition pair, the statement is also correct with respect to the conjunction of an arbitrary specification R to both precondition and postcondition. R stands for some additional part of the memory that s will not be allowed to touch. It is more natural to describe assembly programs with preconditions alone, especially to facilitate unusual control patterns that do not fit the stack-based function convention that normal separation logic assumes. One consequence is that the usual frame rule is inapplicable. Instead, we may represent frame conditions explicitly with second-order quantification, as we do in this example.

To give us the usual flexibility of the frame rule for function calls, we must refer to the frame condition in one more place. This is part of a use of the $i@@p$ connective that we introduced earlier. We state that register `Rret` points to a code block that is safe to jump to if a particular condition holds. We have a nested use of the `~>` notation, this time defining a specification over a new state `st'`, which effectively stands for the state upon *returning from* the function, while `st` stands for the state upon *calling* the function.

The return-time invariant is identical to the initial separation logic assertion, except that we have swapped the values found at the two distinguished memory locations, as one would expect from this function's informal specification. Crucially, the same frame variable appears in the two snapshots of memory. Since we impose no further conditions on fr, it will be impossible to prove that any memory slice satisfies fr, with the sole exception of the memory on entry to the function. Therefore, if the function manages to satisfy the condition attached to the return pointer, all memory but the two distinguished cells will have been preserved.

One further subtle point in this specification style deserves some explanation. Most classical verification tools support *ghost variables*, which are additional program variables used solely for specification and verification. In more standard specifications for our swap example, we would probably see ghost variables used in place of existential quantification for the initial values a and b of the two distinguished memory cells. In classical first-order tools, ghost variables increase expressiveness beyond simple existential quantification, but only because *function return pointers are not made explicit*. Ghost variables provide the ability to share variables between function preconditions and postconditions. When we are able to talk about return pointers in a first-class way, as we have in this example, we need only preconditions, and so there is no further need for sharing of variables between specs.

Once the function has a specification, we can prove that the specification is met. A Coq Theorem command begins our proof of that fact. The proof is given with a Coq *proof script*, a program in a domain-specific language for proof search. In this case, our script says that the proof proceeds in two steps. First, we call the Bedrock tactic structured to reduce the theorem to the truth of a set of automatically-generated verification conditions. We use the semicolon operator to chain on a second tactic, which should be run on every verification condition. This second tactic is the Bedrock tactic sep, a generic simplifier for conditions involving separation logic.

In our particular example, there is one condition to prove, which amounts to showing the following implication, where we use the notation st[i] to denote the effect of executing instruction i in state st:

```
State 1: st
Pred. 1: R0 ==> a * R1 ==> b * ![fr]
State 2: st[R2 <- R0][$[R0] <- $[R1]][$[R1] <- R2]
Pred. 2: R1 ==> a * R0 ==> b * ![fr]
```

We want to show that the truth of Predicate 1 in State 1 implies the truth of Predicate 2 in State 2. A good first step is to inline the state information into the two predicates, so that both are expressed in terms of the same variable st. We write st.[E] for the value found at memory cell E in state st.

```
Pred. 1: st#R0 ==> a * st#R1 ==> b * ![fr]
Changes: R0 <- st.[st#R1]; R1 <- st.[st#R0]
Pred. 2: st#R1 ==> a * st#R0 ==> b * ![fr]
```

Next, we can use Predicate 1 to simplify the list of changes. Any read st.[u] may be replaced by v, whenever Predicate 1 contains a conjunct u ==> v. Our new implication is:

```
Pred. 1: st#R0 ==> a * st#R1 ==> b * ![fr]
Changes: R0 <- b; R1 <- a
Pred. 2: st#R1 ==> a * st#R0 ==> b * ![fr]
```

To finish the proof, we want to eliminate the need to take into account the set of memory changes. Our strategy to do so is to *execute the changes symbolically in Predicate 1*. That is, for each write of value v to memory at address u, we find a fact u ==> v' in Predicate 1 and replace v' with v. That algorithm reduces the implication to:

```
Pred. 1: st#R0 ==> b * st#R1 ==> a * ![fr]
Pred. 2: st#R1 ==> a * st#R0 ==> b * ![fr]
```

This implication is *almost* trivially true, modulo the fact of *'s commutativity. Applying such facts manually can be quite a hassle. Our separation logic tactic takes commutativity and associativity into account and automatically finishes proofs like this by *cancellation*. That is, we iterate through finding a conjunct that appears on both sides of the implication and "crossing it out." If the verification has been set up properly, cancellation eventually gives us an implication between identical formulas, which we can dispatch trivially.

We have completed our first Bedrock verification, and we would like now to stress one key property of our approach that distinguishes it from the mostly-automated verification methods embodied in tools like ESC [14], Boogie [1], and Jahob [32]. Solvers for boolean satisfiability (SAT) and satisfiability modulo theories (SMT) have become increasingly practical in program verification, and more and more projects work by reducing verification conditions to domains that such solvers understand. The solver works by a mechanical process with too many steps for humans to follow closely. Completeness guarantees for decision procedures sometimes make this loss of simplicity acceptable. Unfortunately, the constraint-solver approach to verification tends to lead to relatively inexpressive specification languages, as these solvers do not support techniques like local universal quantification over specification variables.

Bedrock relies on a very different approach. Users of SMT-based verification tools often describe separation logic as too hard to automate, but we think of that statement as only true in the context of normal SMT solvers. A simple *syntactic* algorithm can be very effective at discharging separation logic implications. The informal procedure we just demonstrated for swap scales up to much more interesting verifications, as we will demonstrate with our next example and in our further case studies.

When the proof state is set up properly beforehand, our sep tactic reduces separation implications to facts about normal mathematical objects like numbers, sets, maps, and lists. These simpler facts are usually straightforward to discharge with traditional solvers, ideally those giving completeness guarantees. The key win is that the reduction removes all need to reason about machine states or memories. We manage to handle that part of the reasoning in a *quantifier-free* way, avoiding one of the biggest headaches for SMT solvers. The syntactic simplification procedure is much simpler for a programmer to keep track of than the usual SMT-based alternatives, which have to do with predicting the action of quantifier-instantiation triggers or adding manual instantiation annotations.

At this point, the reader may be willing to believe that such a syntactic approach works for trivial examples like swap, while maintaining skepticism that we can scale to more complex examples like typical imperative data structures. Our next example shows how to accomplish that scaling, based on *computational abstract predicates* and modest use of *unfolding hints* that simplify uses of those predicates.

2.2 Incrementing All of a Linked List

Figure 3 shows our next example, which walks a singly-linked list of words, incrementing the value of every word. A list node is a pair of adjacent bytes, the first storing the data value and the second storing the next pointer, which is 0 in the final node of a list. The definition of the function linc introduces some structured programming constructs. We have a standard "while" loop, which, as usual, must be prefaced by a loop invariant (here placed inside square brackets). For this example, the loop invariant is the same as the function precondition, so we assigned that shared condition the name lincS. There are also a few Use statements here; we will explain shortly how they are used to guide automated proving.

The specification lincS follows the basic form of our previous example's precondition. The variables a and b have been replaced

```
Definition lincS : state -> PropX pc state := st ~>
  Ex fr, Ex ls, ![ !{llist ls st#R0} * ![fr] ] st
  /\ st#Rret @@ (st' ~>
    ![ !{llist (map S ls) st#R0} * ![fr] ] st').

Definition linkedList := bmodule {{
  bfunction "linc" [lincS] {
    [lincS]
    While (R0 != 0) {
      Use [llist_nonempty_fwd];;
      Use [llist_nonempty_bwd];;
      $[R0] <- $[R0] + 1;;
      R0 <- $[R0+1]
    };;
    Use [llist_empty_fwd];;
    Use [llist_empty_bwd];;
    Goto Rret
  }
}}.

Theorem linkedListOk : moduleOk linkedList.
  structured; sep.
Qed.
```

Figure 3. A Bedrock function to increment all of a linked list

by `ls`, which is a normal, purely-functional Coq list. This list serves as a *functional model* of the imperative list that is to be manipulated, and the action of the `linc` function can be modeled with a *purely functional reference implementation*. In particular, the pre-state contains the assertion `!{llist ls st#R0}`, where the notation `!{_}` denotes the use of an abstract predicate. In this case, we are requiring that a linked list is present in memory, rooted at `st#R0` and containing data elements matching those in `ls`. We model the action of the function in our choice of alternate arguments to `llist` in the post state. We write `!{llist (map S ls) st#R0}`, applying the usual higher-order function `map` to replace every element of `ls` with the result of applying `S` (the increment-by-one function) to it.

We should emphasize that, while we try to aid intuition about the computational specification approach by writing that it relies on "purely functional reference implementations," we are not literally writing a library in both functional and imperative versions and proving equivalence between them. Non-trivial functional programs may still deserve quite non-trivial verifications of their own, as ample research on the subject can attest to. Instead, we rely only locally on functional programs as alternatives to traditional mathematical notation. For example, the specification discussed in the previous paragraph is structured mostly as normal second-order logic, with one localized use of a functional program that applies `map` to express how the function should mutate a list.

Contrast this with an approach that might be used with SMT solvers, where a list is modeled as an array, and we might characterize `linc`'s behavior like this in terms of pre- and post- versions a and a' of an array:

$$\forall i.\ a'[i] = a[i] + 1$$

By using a quantifier, we forfeit completeness guarantees, as most first-order theories with quantifiers are undecidable. With the computational approach in Coq, we usually avoid the need to consider decidability questions. We know that any Coq term has a single well-defined answer which may be determined by normalization under appropriate conditions. As Coq executes terms automatically throughout its proof infrastructure, a well-chosen computational abstraction can give us many proof steps for free.

Some verification tools such as Jahob [32] allow the use of sets and set theory in specifications, which removes some need for quantifiers. However, universally-quantified invariants still appear in most data structure implementations. In all of our case studies so far, there have been only four functions that we verified with invariants that use quantifiers in any way besides strings of existential quantifiers at the beginnings of preconditions, as seen in our examples so far. All of our data structure invariants stick to this restricted use of quantifiers, too. This pattern is much closer to the well-understood *ghost variables* of classical verification than to the more involved use of quantifiers that causes trouble for SMT solvers. Quantifiers are hard to reason about, and our encoding approach allows us to delegate almost all reasoning work to the simple syntactic procedure that we began describing with the last example.

In our proof of correctness for the `linkedList` module, the `structured` tactic will hand us three verification conditions: we must show that the function precondition implies the loop invariant (which is trivial for this example), that the loop body preserves the loop invariant when we assume that the loop test succeeded, and that the loop invariant implies the postcondition when we assume that the loop test failed.

We focus on the first of the non-trivial implications, dealing with the effect of going once through the loop. Skipping to the second stage of the procedure we used for the last example, we have the following implication goal, which includes a pure fact implied by the success of the loop test. Since we are not returning from the function immediately, the existential variables of the post-state are up to us to instantiate. In effect, the loop body contains an "assignment" for each of these "ghost variables," and we will choose the righthand side of each such assignment as we complete the proof. We will write the new variable values as `ls'` and `fr'` until we determine what we want them to be; these are *unification variables*, part of Coq's standard proof search support.

```
   Pure: st#R0 <> 0
Pred. 1: !{llist st#R0 ls} * ![fr]
Changes: R0 <- st.[st#R0]+1
Pred. 2: !{llist (st.[st#R0+1]) ls'} * ![fr']
```

We would like to simplify the memory projection from the changes list, but clearly our previous simple procedure is inadequate. That procedure involves consulting the set of points-to facts in Predicate 1, and we have no such facts here. However, given our knowledge of how linked lists are represented, we know that the list must not be empty because `R0` is nonzero, so Predicate 1 is *equivalent to* some predicate with a points-to fact for `R0`. To make this more formal, we should start with the formal definition of the abstract predicate `llist`. We use Coq's standard facilities for recursive definitions and pattern-matching, along with a new separation assertion notation `[< p >]`, which lifts the pure predicate `p` into an impure predicate that asserts the truth of `p` and applies only to empty heaps.

```
Fixpoint llist (ls : list nat) (hd : nat) : sprop :=
  match ls with
    | nil => [< hd = 0 >]
    | x :: ls' => Ex u, [< hd <> 0 >]
      * hd ==> x * (hd+1) ==> u * !{llist ls' u}
  end.
```

This definition says: An empty list is represented by an empty memory and a head pointer with value 0. A nonempty list is represented based on a local existentially-quantified "ghost variable" `u`, standing for the next pointer. We assert that the head pointer is nonzero, that it points to the head `x` of `ls`, that the next memory cell contains u, and that u is the root of a linked list representing the tail ls' of ls.

From this definition, it is clear intuitively that we can "materialize" a points-to fact for the beginning of a nonempty list. The following *unfolding lemma* makes that fact explicit, in terms of a separation logic implication operator ===>, which uses three = characters, in contrast to the points-to operator which uses two.

```
Theorem llist_nonempty_fwd : forall ls hd, hd <> 0
  -> llist ls hd ===> Ex x, Ex ls', Ex u,
    [< ls = x :: ls' >] * hd ==> x * (hd+1) ==> u
    * !{llist ls' u}.
  destruct ls; sepLemma.
Qed.
```

The theorem is proved trivially, using destruct to ask for a case analysis on the list ls, and then calling a variant of our separation logic simplifier to do the rest of the work. We are now ready to learn the purpose of the Use statements in the program code. The first such statement references our new unfolding lemma. By including this annotation statement, we are asking the separation simplifier to use our new lemma to replace an instance of the ===> lefthand side with the corresponding righthand side. Coq uses unification to discover the values of the lemma variables ls and hd, and Coq uses its extensible proof hint mechanism to discharge any extra hypotheses. In this example, that hypothesis is hd <> 0, which is discharged using the pure fact from the proof state. Coq introduces new variables to stand for the values x, ls', and u that we assert to exist. This brings our state to:

```
   Pure: st#R0 <> 0 /\ ls = x :: ls''
Pred. 1: st#R0 ==> x * (st#R0+1) ==> u
         * !{llist ls'' u} * ![fr]
Changes: R0 <- st.[st#R0]+1
Pred. 2: !{llist ls' (st.[st#R0+1])} * ![fr']
```

Now that a points-to fact for R0 is exposed, we can simplify our explicit memory accesses.

```
   Pure: st#R0 <> 0 /\ ls = x :: ls''
Pred. 1: st#R0 ==> x * (st#R0+1) ==> u
         * !{llist ls'' u} * ![fr]
Changes: R0 <- x+1
Pred. 2: !{llist ls' u} * ![fr']
```

It is now easy to run symbolic evaluation of the memory changes in Predicate 1.

```
   Pure: st#R0 <> 0 /\ ls = x :: ls''
Pred. 1: st#R0 ==> x+1 * (st#R0+1) ==> u
         * !{llist ls'' u} * ![fr]
Pred. 2: !{llist ls' u} * ![fr']
```

At this point, cancellation will allow us to finish this case of the proof, in a subtle method of *automatic application of a frame rule*. The two abstract predicates may be canceled if we set unification variable ls' to ls''. That leaves the frame condition unification variable fr' for us to determine. By setting it equal to all of Predicate 1 that remains, we can finish the case by reflexivity. The part of Predicate 1 that has been absorbed into fr' constitutes *the parameter R to the standard frame rule*, and it has been determined using a very generic unification technique that applies just as well to non-standard control structures that need not follow a usual function call convention.

The loop invariant is a conjunction, and we have finished with the first conjunct. The second conjunct is an assertion about the conditions under which it is safe to jump to the function return pointer. In particular, we need to show:

```
Pred. 1: st#Rret @@ (st' ~>
  ![ !{llist (map S ls) st#R0} * ![fr] ] st')
Pred. 2: st#Rret @@ (st' ~>
  ![ !{llist (map S ls') st#R0} * ![fr'] ] st')
```

It turns out that one @@ assertion about a code pointer implies another about the same pointer if their predicate operands imply each other in the reverse order. We apply that rule and also substitute the values of ls' and fr' that we learned in proving the last conjunct.

```
Pred. 1: !{llist (map S ls'') u}
         * st#R0 ==> x+1 * (st#R0+1) ==> u * ![fr]
Pred. 2: !{llist (map S (x :: ls'')) st#R0} * ![fr]
```

Coq automatically uses computation to replace map S (x :: ls'') by S x :: map S ls''. To make further progress, we would like to unfold the abstract predicate in Predicate 2, similarly to the way we did for Predicate 1 in an earlier step. It is easy to prove another lemma that is like llist_nonempty_fwd but runs the implication in the other direction. This is exactly the lemma that we suggest with the second Use statement within the loop body. Based on that hint, our sep tactic will first introduce new unification variables x', ls''', and u', standing for the existential quantifiers in the theorem statement. Then, the pure goals st#R0 <> 0 and map S (x :: ls'') = x' :: ls''' are queued for solving by normal mathematical means. By stating the first of these facts outside of the separation implication in the theorem statement, we ask that it be proved before proceeding, and this is easily done, based on the first pure hypothesis in the proof state. The second pure fact is queued to be reconsidered after we finish the impure part of the proof. Hopefully the unification variables will have been determined by then, simplifying the job of our pure solvers, many of which are unable to handle unification variables.

Before we can get there, we must finish with this new modified proof state based on the lemma statement:

```
Pred. 1: !{llist (map S ls'') u}
         * st#R0 ==> x+1 * (st#R0+1) ==> u * ![fr]
Pred. 2: st#R0 ==> x' * (st#R0+1) ==> u'
         * !{llist ls''' u'} * ![fr]
```

The canceler finishes this proof by unifying x' with x+1, u' with u, and ls''' with map S ls''. Notice that this is very simple, eager syntactic unification that occurs as we try to cross predicates off from both sides of the implication. We need none of the complexity of "E-graph matching" as pioneered in solvers like Simplify [10]. The particular unification that we discover leaves the earlier queued fact map S (x :: ls'') = x' :: ls''' solvable trivially by computational normalization, and we have finished proving the correctness of this verification condition.

The final verification condition has a similar but simpler proof, based on analogous unfolding lemmas for empty lists that are invoked in the last two Use statements of the program. Each of our unfolding lemmas has a trivial one-line proof. The Use statement also supports partial instantiation of a lemma's quantifiers, when unification is not sufficient to discover instantiations correctly; and there is a further form to allow references to the current machine state in computing the instantiations. We find this kind of quantifier instantiation to be easier to keep track of than in the case of universally-quantified program invariants, as the quantifier reasoning may be kept local and is also completed solely by unification in a majority of cases.

3. A Simplification Procedure for Separation Assertions

Our walk-throughs of examples have demonstrated the basic five-step procedure that we use in discharging all Bedrock verification conditions. The procedure is easily formalized as an algorithm, broken into discrete steps that are reasonably easy for humans to keep track of. Here is the procedure, including three "extra" steps that are not specific to separation logic.

1. **Standard first-order logic simplification**: Before beginning, it is useful to put standard Coq simplification to work. For instance, our last example involved proving an implication between two conjunctions. This can be reduced to separate proofs of the two conclusion conjuncts, in each case assuming the truth of both hypothesis conjuncts. This stage also involves using each hypothesis disjunction to split the proof into two cases, replacing each existentially-quantified hypothesis with a version of its body that refers to a freshly-introduced variable, and so on.
2. **Forward unfolding of abstract predicates**: We apply the hints suggested with `Use` statements, along with some hints that have been registered globally, so that they should be used wherever they apply. The separation logic hypothesis ("Predicate 1" in our examples) is easily normalized to an iterated separating conjunction of impure facts. We walk through the conjuncts, looking for a hint proving an implication whose lefthand side unifies with the current conjunct. When we find a match, we first check that the premises of the lemma can be proved immediately, by calling all solvers that have been registered as hints. If this process succeeds, we introduce new variables for any existential quantifiers in the implication conclusion, new pure hypotheses for any pure conclusions, and replace the original conjunct with the new impure conclusions from the lemma. This process is iterated until no more hints match.
3. **Simplification of memory accesses**: For every points-to fact `u ==> v` in the separation hypothesis (referring to state `st`), we replace every occurrence of the memory access `st.[u]` with `v`, anywhere the former appears in the proof state.
4. **Symbolic execution of memory writes**: For every write of value `v` to address `u` implied by the current basic block's straightline instructions, find a fact `u ==> v'` in the separation hypothesis and replace `v'` with `v`.
5. **Backward unfolding of abstract predicates**: This phase is very similar to the forward unfolding phase, but with consideration of hints that apply to the separation conclusion ("Predicate 2" in our examples). We replace the conclusion of a lemma with its premises. New existential variables are instantiated as *unification variables*, whose values should be determined later through syntactic unification. Rather than adding pure premises to our proof state, we queue them as proof obligations to return to later. These obligations often contain some of our new unification variables. With proper foresight in `Use` annotations, the cancellation step will determine the values of almost all of these unification variables, so that, when we return to the obligations, they are in forms not so different from what SMT solvers are designed to handle.
6. **More aggressive first-order logic simplification**: Here we repeat all the simplifications of the first step, with one addition: When the goal begins with an existential quantifier, we generate a fresh unification variable, substitute it for the quantifier's variable, and make this substitution result the new goal. This quickly reduces our goal to a set of goals, some using separation logic and some using more standard mathematical theories. We will try to finish with the separation goals before proceeding, in hopes of determining most unification variable values.
7. **Cancellation**: We iterate through all conjuncts in the separation hypothesis, trying to find a unifiable conjunct in the conclusion. Every matching pair is eliminated. By reflexivity of implication, we are finished when the full hypothesis and conclusion may be unified.
8. **Proof of remaining pure facts**: If cancellation worked properly, we should now be faced only with a collection of pure goals that deal only with standard mathematical theories. These can be discharged by various solvers that are registered as hints. For instance, we have used congruence closure, a solver for linear arithmetic, brute force search via Prolog-style logic programming, and other techniques.

We also allow the programmer to register *rewrite rules*, designed to simplify proof states by simple syntactic replacement of one pattern by another. We apply these rules wherever possible, between every pair of steps above.

The whole process is implemented in Coq's Ltac language [9], which is a Turing-complete domain-specific language for proof search. By construction, every Ltac program generates a *proof term* to explain why it concluded that a fact is true. These proof terms use a relatively simple, generic logical language that relies on a small number of axioms. Thus, it is possible to implement a small, trustworthy standalone checker for these proof terms. To trust a proof, one need only trust the checker, not the more heuristic process by which proof terms are found. Therefore, since we implement our procedure as the Ltac program `sep`, we arrive at correctness by construction: whenever the procedure succeeds, it has made only valid deductions.

The `sep` procedure is the workhorse for Bedrock verification, but it is important that programmers can build on it with further problem-specific Ltac code. For instance, most reasoning about code pointers must be done through Ltac, rather than through traditional solvers, which are unable to cope with the second-order quantification that is usually involved. Each variety of control transfer generally demands its own modest piece of Ltac code. For instance, our last example program involved reasoning about the way the `@@` operator is used to model return pointers. Our case studies use about 10 lines of Ltac code to guide the automatic application of the appropriate reasoning principle. Other constructs like computed function calls, exception handling, and so on will generally require similar up-front investments. Once this support code is written, no extra work is required in the verification of individual programs.

Though `sep` is designed to automate most of the proof process, we often break it into its individual steps while verifying a program, so that we can watch to make sure that each step goes the way we expect. Compared to SMT solvers and related techniques, we think it is an advantage of our high-level approach that humans can keep track of the steps of the algorithm and glean useful information from watching its progress. That is, at each stage of the proof, the human user is shown intermediate states that look much like those given for the examples in Section 2. Even the full `sep` tactic is not an all-or-nothing solver; when it fails to prove a goal, it returns the unproved obligations in the same human-understandable format, with the possibility for exploratory proving to determine what went wrong.

For each domain of uses of code pointers, we usually write a single automation tactic, but we also break the proof process into a series of calls to simpler named tactics. On approaching a new verification condition, we step through these sub-stages to get a sense for which `Use` annotations and hints will be needed. Any step may be traced in more detail, using Coq's usual proof debugging commands. When the user figures out that a particular `Use` annotation is useful, a tactic may be called to add it from within the exploratory proving process; these hints can later be migrated into the program source, once all of the cases work. At that point, we switch to the mostly-automated approach. We want to stress that these lower-level tactics are still more like states in a small, fixed finite state machine than like the more common, very manual Coq proof scripts. Also, in switching to mostly-automated proofs, we often find that our proofs continue working even after making modest changes to a program's specification. The human prover need only find the key reasons why a program is correct, and then the automation can take care of the details.

4. Evaluation

The Bedrock framework implementation consists of about 5000 lines of Coq code. We have also implemented a suite of case study programs, chosen to exercise both traditional first-order reasoning and the more unusual higher-order reasoning about code pointers. None of the case studies require manual proof about specific program states, despite the fact that we are always proving full functional correctness of our libraries, not just more shallow properties like memory safety. Figure 4 gives some statistics about the case study programs.

We explain the column meanings in left-to-right order: First, we give each module's total lines-of-code count (including both implementation and proof), followed by its number of structured assembly functions and the number of lines of code devoted to these functions, minus proof-related annotations. After that, we have, respectively, lines-of-code counts for data structure representation invariants, pure lemmas (which do not deal with memories or machine states), impure lemmas (which are phrased as implications in separation logic), invariants included in programs (e.g., loop invariants), `Use` annotations, proof hints (which guide automatic solving of pure goals), and the main correctness proof scripts for modules. In the cases of impure and pure lemmas, we differentiate between the lines devoted to theorem statements and the lines devoted to their proofs (with the latter inside parentheses).

The last column of the table shows the time needed to compile each module, proofs included, on a 3.16 GHz Intel Core 2 Duo processor running Linux. Among the examples with the most code, two take about a half and a quarter hour, with the rest finishing in 6 minutes or less. To suggest a fair comparison with other verification tools, we note that our case studies spend over half their time in generation and checking of proof terms, which is a kind of "bonus" on which most other tools spend no time. Bedrock is also implemented on top of the normal Coq engine, which uses naive interpretation to execute proof search programs; we would expect to see a constant-factor speedup of at least several times with a custom implementation in a general-purpose language. Ideally, a future Coq version will include an improved optimizing engine that brings these benefits to our existing code. Even as the framework stands today, we find that the programmer experience is quite reasonable, with just a few seconds wait time required when focusing on and stepping through the proof of a single verification condition. As is usual, the different conditions may be tackled in parallel; though Coq does not support this kind of parallelism yet, we expect that simple optimizations in Coq to enable multi-core execution would also bring large speed-ups.

To give a more qualitative account of the verification experience, we turn to descriptions of the case studies.

First, we have the `Malloc` module, a heap memory management library that all of the other modules rely on. Its interface is independent of the implementation strategy; for now, we are using a simple unsorted free list with no coalescing. This data structure is an interesting take on the familiar linked list: we have variable-sized free list nodes that must store both their own sizes and their "next" pointers, rather than uniformly-sized list nodes that point to data allocated elsewhere. Verifying `Malloc` requires proving impure lemmas covering maneuvers like splitting a free list block into one piece to return to client code and another to keep in the list, based on the size of the memory chunk requested by the client.

The remaining case studies fall into two categories. First, we compare against recent developments in data structure verification by implementing our own versions of five case studies used with the Jahob tool [32]. Second, we demonstrate Bedrock's support for reasoning about code pointers, via examples mostly inspired by past work with the XCAP logic [24].

4.1 Data Structures

Our data structure examples can be split into three groups, based on the mathematical domain used to model the data structures. Each of our implementations provides the same functionality as the "public methods" in a particular example from a recent paper on Jahob [32]. We prove full functional correctness of each method, showing that the imperative code realizes the same behavior as with the "obvious" implementation of each operation via a small functional program. Our job is inherently harder than in Java-specific Jahob, since every module is at the assembly level of abstraction and must do explicit management of heap and stack memory, but our results show that the effective burden is not so great.

First, we have arrays, modeled using functional lists. One module develops the program-independent aspects of this theory, including impure lemmas covering different ways of isolating cells within arrays. For instance, here is a theorem for unfolding an array in a way that exposes a points-to fact for the cell at index `n`.

```
Theorem array_mid_fwd : forall n a ls,
  n < length ls
  -> array ls a ===> !{array (firstn n ls) a}
    * (a+n) ==> nth n ls 0
    * !{array (skipn (n+1) ls) (a+n+1)}.
```

Notice the key use of the *computational* approach we have mentioned a few times already. Usual mostly-automated program verification deals with array cell isolation mostly through universally-quantified invariants. Instead, we take advantage of the rich possibilities for computation with lists in Coq, describing a three-way split of an array in terms of recursive functions for keeping only the first `n` elements of a list, extracting the `n`th element, or dropping the first `n` elements, respectively. A few rewriting hints about the interactions of these recursive functions enable very effective proof automation about array operations, after we add a few `Use` statements that suggest lemmas like `array_mid_fwd`. We, in effect, reduce quantifier-heavy reasoning about arrays to quantifier-free reasoning about functional lists. The `ArrayList` module applies this approach in the implementation of 19 common operations on an abstract datatype of growable arrays. As, compared to linked data structures, arrays offer relatively little opportunity for simple computational abstraction, our Bedrock `ArrayList` involves about 40% more annotation than the Jahob version.

Next, we have a theory of sets, modeled as Coq values of type `nat -> bool`. That is, a set is a mathematical function from machine words (natural numbers) to booleans, telling us whether each word belongs to the set. We also define a common interface for assembly implementations of imperative finite sets, providing operations for membership checking and addition and removal from a set. We give two implementations of this interface, with unsorted singly-linked lists and with binary search trees. The latter module additionally provides functions for extracting the minimum or maximum element from a nonempty set. The computational approach helps us give dramatically simpler binary search tree invariants than in the Jahob code, where even the basic data structure invariants include 10 universal quantifiers. We rely instead on the ability to filter sets by computable predicates; for instance, in considering an internal node of the tree, to come up with the set that its left child

Module	Total	#Funcs.	Impl.	Data	Pure (pfs)	Imp. (pfs)	Inv.	`Use`	Hints	Main	Build (min)
`Malloc`	267	3	71	8	3(1)	50(14)	16	24	2	1	5
Theory of arrays/lists	111	-	-	5	2(2)	35(32)	-	-	7	-	<1
`ArrayList`	771	19	272	3	79(37)	29(15)	168	83	20	1	35
Theory of sets	159	-	-	-	61(43)	-	15	-	7	-	<1
`SinglyLinkedList`	157	4	54	6	-	14(6)	20	14	2	6	2
`BinarySearchTree`	355	6	127	10	-	32(7)	43	40	2	28	13
Theory of maps	69	-	-	-	7(23)	-	20	-	1	-	<1
`AssociationList`	200	6	85	6	-	12(6)	28	27	2	1	3
`Hashtable`	374	6	90	6	36(13)	38(13)	56	39	18	1	4
`Memoize`	191	2	45	7	21(6)	6(3)	36	5	5	18	2
`AppendCPS`	155	2	56	5	-	9(3)	44	12	-	1	4
`ThreadLib`	225	4	67	9	6(2)	22(7)	34	20	1	7	6

Figure 4. Statistics on case studies, mostly in terms of number of lines of code of each kind

represents, we filter the original set by a function that keeps only those elements less than the current data value.

Finally, we define a similar theory of maps, modeled with the Coq type `nat -> option nat`, where an "`option nat`" is either `None` or `Some n`, for number `n`. Our two finite map implementations use unsorted singly-linked lists and hash tables. Unlike its Jahob counterpart, our hash table implementation uses the linked list implementation, treated abstractly through its interface, within hash buckets. The computational approach again brings some significant simplifications for hash tables. A crucial function is "`only`", which restricts a map to just those keys that hash to a particular value, implemented using the natural number equality test `eq_nat_dec`.

```
Definition only (m : map) (hmax n : nat) : map :=
  fun k => if eq_nat_dec (hash hmax k) n
           then m k else None.
```

We use `only` to state impure lemmas like the following, which isolates a particular bucket within a hash table.

```
Theorem htableOk_mid_fwd : forall n m hmax
  curHash len a,
  n < len -> htableOk m hmax curHash len a
  ===> Ex a', !{htableOk m hmax curHash n a}
  * (a+n) ==> a'
  * !{alist (only m hmax (curHash+n)) a'}
  * !{htableOk m hmax (curHash+n+1)
      (len-n-1) (a+n+1)}.
```

Without focusing on the details, we point out that a single case of the hash table representation predicate `htableOk` is expanded to two other uses that together skip one bucket, where the missing bucket is described in terms of a points-to fact to a pointer `a'`. This pointer itself represents an association list, which we express using the abstract predicate `alist` exported by the `AssociationList` module. We say that the association list represents the restriction of the overall map to just those keys that belong in the bucket we have isolated.

Summing up all of the different kinds of proof and annotation from Figure 4, we arrive at a total of about 200 for `Hashtable`. The Jahob version uses over 250 lines of annotation just within the methods used to implement the remove-from-map operation. The bulk of these annotations are qualitatively different from those required in Bedrock, as they involve explicitly nested proofs based on the quantifier structure of invariants. Our removal function is self-contained, including one 5-line invariant and 5 `Use` statements. The remaining lines are normal executable code.

Our data structure case studies deal only with keys and values that are uninterpreted machine words. However, since we are working with low-level programs, machine words are sufficient as a representation of arbitrary data structures. These words may be treated as pointers by client code, and we may write invariants about the layout of such nested data structures solely in terms of the functional model of a set or map, avoiding any coupling to the details of the data structure implementation. It is still true that our implementations of sets and maps may not have the intended semantics when used to store nested data structures via pointers, since our fixed comparison and hashing operations compare pointers and ignore further structure. Variations on the case studies from this section should support the use of user-provided comparison and hashing functions, when accompanied by proofs of key facts about them (e.g., a comparator implements a total order in terms of some functional model of the nested data structure).

4.2 Code Pointers as Data

Our remaining case studies go beyond the domain of most previous mostly-automated verification tools, where most assertions about callable code pointers may not even be expressed, let alone verified. Here we are still proving full functional correctness, though it is often less obvious what precisely that means for libraries that work with code pointers. We choose our specifications to formalize the intuitive notion of the abstraction that the library is meant to provide. For instance, our threads library will include a yield function whose specification guarantees that, when the yielding thread is rescheduled, its private memory has not been changed.

Our first example is a generic function memoizer, backed by hash tables, which are themselves backed by association lists, creating a non-trivial tower of uses of abstract datatypes. Any function may be memoized into values of a particular abstract datatype of memo tables. Each function is associated with a pure specification, a mathematical relation between input and output words. Additionally, a memoized function may use *local state*, represented by a completely unconstrained separation logic predicate, which might stand for, e.g., a local cache data structure. Memo tables use hash tables internally, along with an extra invariant that refers only to the functional map that models the hash table contents. We require that every mapping is consistent with the pure specification. This allows us to expose the memo table as a function with almost the same specification as the original. (We must expand the specification to require that the memo table is loaded in memory before the call and guarantee it is loaded afterward.)

Our final two case studies are based on examples from papers about the XCAP program logic. First, we reimplemented and reverified the main example from the original XCAP paper [24]. This is an implementation of in-place concatenation of two linked lists, written in a form that might be output by an ML compiler. Rather than following a C-style calling convention, the "append" function's return pointer is represented as an explicit closure, which pairs a function pointer with an "environment" argument that the function expects to be passed when called. The old version of this example was coded in an assembly language even more idealized than the one we use in this paper, where `malloc` and `free` are instructions, rather than library functions that must be implemented. Nonetheless, despite this disadvantage, our Bedrock version is substantially simpler. The XCAP version involves about 1500 lines of very manual proof. Our new version relies on just three of the four unfolding lemmas used for lists in Section 2.2, 12 `Use` statements, and a one-line main proof. Our invariants correspond to code that already appeared explicitly as basic block preconditions in the XCAP version, so it seems fair to say that we have reduced the proof burden by two orders of magnitude.

We also verified a cooperative threads library, similar to one implemented in XCAP [25, 26]. That is, the library relies on threads to yield to other threads, in place of the more common interrupt-based approach of preemptive threading. Here, the setting of past work is more challenging, as they verified real x86 code, but the basic approach is similar. We implement a fair round-robin thread scheduler, supporting dynamic thread creation, yielding, and exiting. There is a shared global invariant, characterizing the part of memory that threads may use to communicate with each other. Each blocked thread also has its own local invariant, which is indexed by the thread's saved stack pointer. Here is an example specification that uses the abstract data type of thread schedulers; in particular, this is the precondition for the yield function.

```
Definition yieldS := st ~> Ex fr, Ex ginv,
  Ex invs, Ex root,
  susp ginv (fun sp => sep ([< sp = st#Rsp >]
                       * ![fr])%Sep) st#Rret
  /\ codesOk ginv invs
  /\ ![ !{mallocHeap 0} * st#Rsp ==> root
      * !{threads invs root} * ![ginv] * ![fr] ] st.
```

We see explicit quantification over a global invariant `ginv`. The `susp` predicate characterizes when a code pointer (the return pointer, in this invariant) is safe to *susp*end, in terms of the global and local invariants. The local invariant we choose here combines the frame condition with the requirement that the stack pointer upon resumption equals the original. The `codesOk` invariant asserts similar properties for all of the suspended threads, as represented by the functional model `invs`. Both `susp` and `codesOk` are implemented using second-order quantification. The precondition ends with a more normal separation logic formula, requiring a valid `malloc` heap, a pointer to the scheduler data structure in the first local stack slot (found at an address based on stack pointer `Rsp`), the scheduler data structure itself, the global invariant, and the frame condition.

The old XCAP implementation used about 8000 lines of code just for the proofs of correctness for three basic blocks that implement key operations on saved thread contexts [26]. A comparable amount of additional complexity was associated with the thread scheduler and its data structure. The complete Bedrock implementation, including assembly code and verification, totals just 250 lines. We again see a reduction of about two orders of magnitude in the amount of program-state-specific proof.

Of course, counting lines of annotation and proof gives only an approximation of the human cost of verifying a program. We would rather know the time spent by a programmer in coming up with these lines, since some shorter proofs may be harder to find in practice. Unfortunately, we have not found much data of this kind in the literature. We can give one piece of anecdotal evidence for the effectiveness of the Bedrock approach: The authors of the XCAP cooperative thread scheduling library mention that they spent six person-months completing it [25]. This figure includes time spent building and tweaking the verification framework, but we hope it still gives a good sense of the scale of this sort of effort. In contrast, starting from no prior code dealing with threads, we coded and verified our threads library in the course of two days of work.

5. Related Work

We have already discussed the CAP project [15] extensively. That line of work has been very successful at designing program logics suitable for foundational verification of low-level code, leading to successful case studies in dynamic thread creation [11], embedded code pointers [24], garbage collection [21], self-modifying code [3], and hardware interrupts [12]. The Bedrock project takes these results as its base and adds infrastructure for higher-level coding and automated verification, demonstrating that foundational guarantees can be had, for code that manipulates code pointers as data, without giving up the signature benefits of classical verification tools.

Classical verification tools like ESC [14], Boogie [1], and Jahob [31, 32] have been shown to be effective in a variety of verification tasks that can be completed using first-order solvers (or additionally, in the case of Jahob, using an integrated proof language for a higher-order logic). Hawblitzel and collaborators have done automated classical verification of garbage collectors [16] and the core of an operating system kernel [30] with Boogie. Bedrock aims to take this style of highly productive verification and add three main benefits. First, we apply the *computational* approach to specification to reduce the need for logical quantifiers and simplify proofs. Second, we support callable code pointers with polymorphic specifications. Third, we reduce the trusted code base to a foundational level, where only the standard Coq proof checker must be trusted to believe a verification. Foundational verification also opens the door for integration with proofs that would be hard to write in a classical verifier but are tractable in general-purpose proof assistants. For example, the Verve project [30] aims to integrate a soundness proof for a typed assembly language; with such a proof and a Bedrock verification both carried out in Coq, the combined system can have a simple foundational correctness theorem.

Separation logic has been applied successfully in program analysis tools, including Smallfoot [2]. Follow-on work on abductive inference [4] infers procedure specifications that can be used to prove memory safety, without requiring whole-program analysis. Alternative shape analysis techniques have been applied automatically in tools like TVLA [29] and XISA [5]. Compared to Bedrock, these tools have applied to languages at higher levels of abstraction than assembly, they have not been verified formally, and they do not support reasoning about code pointers. All of these features contribute to some desirable properties of these other tools: superior performance and greater applicability to mainstream programming languages.

McCreight's Coq tactics for separation logic [20] address similar concerns to the tactic support presented in this paper, raising the level of abstraction in proof about imperative low-level programs. This alternate verification framework satisfies all of our desiderata from the introduction, with the exception that it is not "mostly-automated." Proofs using McCreight's tactics still involve a significant number of manual proof steps, applying such operations as explicit rearrangement of separating conjunctions using associativity and commutativity. For example, a useful comparison comes from a simple in-place linked list reversal function implemented with both systems. McCreight's proof includes about 80 atomic tactic calls, while the Bedrock proof includes about 10 atomic tactic calls and 4 Use statements. About half of McCreight's atomic tactic calls mention variable or hypothesis names that are bound within the proof; none of the tactics in the Bedrock proof do.

The computational approach to data structure specification has been adopted by many projects working within higher-order logics. Mehta and Nipkow [22] used computational specification without separation logic, and the combination with separation logic has appeared in work on XCAP [24], Ynot [23], CFML [6], and VeriFast [17].

The Ynot library [7] for Coq supports automated verification of monadically impure Haskell-like programs annotated with specifications, using separation logic. Every Coq program is also a valid Ynot program, so implicit memory management is baked into the system. Bedrock is designed to apply to languages that are more realistic for low-level infrastructure. While the Ynot language and its verification rules are specified with Coq axioms, Bedrock supports languages defined foundationally with operational semantics. Bedrock has a few more high-level advantages over Ynot: it applies frame rules automatically, supports the inlining of quantifier instantiation hints (`Use` statements) in programs, and deals with the compilation of more convenient, higher-level code into lower-level formats.

The VeriFast [17] verifier has been under development in parallel with Ynot and Bedrock and involves many similar design decisions. Program verification is mostly-automatic, based on computational separation logic specifications with support for higher-order reasoning. VeriFast is a standalone tool, with no formal proof of correctness, which enables better performance and easier integration of convenience features for programmers. VeriFast also targets C and Java rather than assembly-level programs. Lack of integration with a traditional proof assistant may make it harder to carry out some verifications that rely on reasoning outside the domain of the standard automation.

The Bedrock approach usually reduces goal facts about program behavior into sets of quantifier-free facts about normal mathematical objects like numbers, sets, maps, lists, and trees. So far, we have proved many of these facts with some amount of manual proof, since this total burden is not significant compared to the costs of reasoning concretely about data structures and pointer aliasing. We might also try taking advantage of recent progress on complete decision procedures [19] for very rich theories that encompass much of our pure proof obligations. Any such decision procedure should be relatively straightforward to integrate with Coq, if it can be made to generate proof witnesses.

6. Conclusion

We have introduced Bedrock, a framework for implementation and verification of low-level programs in Coq. Bedrock's key productivity features center on automatic proof of invariants expressed with the concepts of separation logic. Unlike other systems providing similar levels of automation, Bedrock supports reasoning about first-class code pointers in terms of second-order variables that stand for unknown invariants. This facility is critical for handling of runtime and operating systems. To the best of our knowledge, ours is the first verified thread library with less than a few thousand lines of proof. Our implementation fits well within that mark, with a total of 250 lines, including program code and proofs.

The framework is parametrized over machine languages and their operational semantics. To date, we have only experimented with Bedrock instantiated to an idealized language with infinite-width words. We expect that the approach will continue to work well for the machine languages of today's common microarchitectures, though future work is needed to be sure. Effective decision procedures for bitvector arithmetic are likely to be the main new requirement.

Bedrock follows a very *computational* approach to separation logic, where most imperative functions are specified in terms of purely-functional "reference implementations." This allows us to replace much formal reasoning with execution of programs in the specification language. Quantified invariants are one of the greatest challenges for automated verification, and the computational approach allows us to replace almost all uses of quantifiers with calls to recursive functions in a functional language.

Most program verification today is done by direct reduction to domains with effective decision procedures. Even complete procedures for complex theories like (restricted forms of) transitive closure can have poor enough performance that the human verifier must do extra work to guide the procedures. With Bedrock, we follow an alternate approach, where the heart of verification condition proving is a *symbolic* procedure for separation logic simplification. This procedure has a few simple steps that humans can follow without much trouble, and our Coq library provides good support for stepping through the procedure and animating the moves it makes. Understanding this process requires no reasoning about "semantic" theories, just about the syntactic form of separation assertions.

When a program is annotated with some relatively simple invariants and invocations of simplification lemmas for abstract predicates, the simplification procedure reduces program correctness to the truth of quantifier-free facts that refer only to very basic mathematical theories. These facts are much easier to reason about than the original program, and we can hope that most will eventually be automatable with a few decision procedures. Thus, Bedrock helps programmers do the "hard parts" of verification with the familiar activity of programming and solve the remaining problems with traditional mathematical methods.

Acknowledgments

We would like to thank Gregory Malecha and Ryan Wisnesky for helpful comments on drafts of this paper. This work was funded within the DHOSA project through AFOSR MURI grant FA9550-09-01-0539.

References

[1] 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 *Proc. FMCO*, 2006.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. FMCO*, 2005.

[3] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proc. PLDI*, 2007.

[4] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proc. POPL*, 2009.

[5] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proc. POPL*, 2008.

[6] Arthur Charguéraud. Program verification through characteristic formulae. In *Proc. ICFP*, 2010.

[7] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proc. ICFP*, 2009.

[8] Coq Development Team. The Coq proof assistant reference manual, version 8.3. 2010.

[9] David Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, 2000.

[10] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[11] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP*, 2005.

[12] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. PLDI*, 2008.

[13] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. PLDI*, 2006.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, 2002.

[15] Nadeem Abdul Hamid and Zhong Shao. Interfacing Hoare logic and type systems for foundational proof-carrying code. In *Proc. TPHOLs*, 2004.

[16] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *Proc. POPL*, 2009.

[17] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Proc. APLAS*, 2010.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, 2009.

[19] Viktor Kuncak, Ruzica Piskac, Philippe Suter, and Thomas Wies. Building a calculus of data structures (invited paper). In *Proc. VMCAI*, 2010.

[20] Andrew McCreight. Practical tactics for separation logic. In *Proc. TPHOLs*, 2009.

[21] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI*, 2007.

[22] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In *Proc. CADE*, 2003.

[23] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proc. ICFP*, 2008.

[24] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL*, 2006.

[25] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Modular verification of machine-level thread implementation. Technical report, 2006.

[26] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic system code: Machine context management. In *Proc. TPHOLs*, 2007.

[27] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Journal of Automated Reasoning*, 5, 1994.

[28] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, 2002.

[29] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24, 2002.

[30] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. PLDI*, 2010.

[31] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proc. PLDI*, 2008.

[32] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *Proc. PLDI*, 2009.