The Bedrock Tutorial

Adam Chlipala

March 28, 2013

Contents

Bedrock is a Coq library that supports implementation, specification, and verification of low-level programs. Low-level means roughly "at the level of C or assembly," and the idea of "systems programming" is closely related, as some of the primary target domains for Bedrock are operating systems and runtime systems.

Bedrock is *foundational*, meaning that one needs to trust very little code to believe that a program has really been verified. Bedrock supports *higher-order* programs and specifications. That is, programs may pass code pointers around as data, and specifications are allowed to quantify over other specifications. Bedrock supports *mostly automated* proofs, to save programmers from the tedium of step-by-step formal derivations; and Bedrock is also an *extensible low-level programming language*, where programmers can add new features by justifying their logical soundness.

This advertising pitch can be a bit of a mouthful. To make things more concrete, we'll start with three small examples. Some knowledge of Coq will be helpful in what follows, but especially our first pass through the examples should be accessible to a broad audience with a basic level of "POPL literacy." Readers interested in applying Bedrock, but who *don't* have backgrounds in core Coq concepts, should consult some other source. Naturally, the author is partial to his *Certified Programming with Dependent Types*. Other popular choices are *Software Foundations* by Pierce et al. and *Coq'Art* by Bertot and Casteran.

This document is generated from a literate Coq source file (`doc/Tutorial.v` in the Bedrock distribution) using `coqdoc`.

1 Three Verified Bedrock Programs

1.1 A Trivial Example: The Addition Function

To begin, we issue the following magic incantation to turn a normal Coq session into a Bedrock session, where we have run Coq with command-line arguments `-R BEDROCK/src Bedrock -I BEDROCK/examples`, with `BEDROCK` replaced by the directory for a Bedrock installation, where we have already run `make` successfully, at least up through the point where it finishes building `AutoSep.vo`.

```
Require Import AutoSep.
```

Importing a library module may not seem like magic, but this module, like any other module in Coq may choose to do, exports syntax extensions and tactics that allow a very different sort of coding than one sees in most Coq developments. We demonstrate by implementing a function for adding together two machine integers. Bedrock is an environment for *verified* programming, so we should start by writing a *specification* for our function.

```
Definition addS := SPEC("n", "m") reserving 0
  PRE[V] [ True ]
  POST[R] [ R = V "n" ^+ V "m" ].
```

Up through the :=, this is normal Coq syntax for associating an identifier with a definition. Past that point, we use a special Bedrock notation. The *SPEC*("*n*", "*m*") part

declares this as a spec for a function of two arguments with the given formal parameter names, and *reserving* 0 declares that this function will require no more stack space than is needed to store its parameters. (As Bedrock is targeted at operating systems and similar lowest-level code, we opt for static tracking of stack space usage, rather than forcing use of a fixed dynamic regime for avoiding stack overflows. Furthermore, handling of the stack is not built into the underlying program logic, and it is possible to implement alternate regimes without changing the Bedrock library.)

A specification includes a *precondition* and a *postcondition*. The notation $PRE[V]$ introduces a precondition, binding a local variable V that can be used to refer to the function argument values. In this example, we impose no conditions on the arguments, so the precondition is merely *True*. Actually, Bedrock uses a fancier domain of logical assertions than Coq's usual `Prop`, so we need to use the $\lceil$ ... $\rceil$ operator to *lift* a normal proposition as an assertion. More later on what assertions really are. Note that the rendering here uses pretty LaTeX symbols; see some of the files in the `examples` directory for the concrete ASCII syntax.

A postcondition begins with the notation $POST[R]$, which introduces a local variable R to stand for the function return value. In our postcondition above, we require that the return value equals the sum of the two function arguments, where we write addition with the $\hat{+}$ operator, which applies to machine words.

Now that we know *what* our function is meant to do, we can show *how* to do it with an implementation. This will be a Bedrock *module*, which in general might contain several functions, but which will only contain one function here.

```
Definition addM := bmodule "add" {{
  bfunction "add"("n", "m") [addS]
    Return "n" + "m"
  end
}}.
```

The syntax should be mostly self-explanatory, for readers familiar with the C programming language. Two points are nonstandard, beyond just the concrete syntax. First, we refer to variable names with string literals. These are *not* string literals in the Bedrock programming language, but merely a trick to get Coq's lexer to accept C-like programs. Second, the function header ends in a Coq term between square brackets. This is the position where each function *must* have a specification.

It doesn't seem surprising that *addM* should be a correct implementation of an addition function, but, just to be sure, let's *prove it*.

```
Theorem addMOk : moduleOk addM.
Proof.
  vcgen; sep_auto.
Qed.
```

The predicate *moduleOk* captures the usual notion from Hoare Logic, etc., of when a program satisfies a specification. Here we prove correctness by chaining invocations of two

tactics: *vcgen*, which performs *verification condition generation*, reducing program correctness to a set of proof obligations that only refer directly to straightline code, not structured code; and *sep_auto*, a simplification procedure based on *separation logic* that is quite a bit of overkill for this example, but gets the job done. (There actually *is* some quite nontrivial reasoning going on behind the scenes here, dealing with complexity hidden by our nice notations; more on that later.)

1.2 Pointers and Memory: A Swap Function

A crucial component of low-level programming is mutable state, which we introduce with a simple example: a function that takes two pointers as arguments and swaps the values in the memory cells that they point to. Here is its spec.

```
Definition swapS := SPEC("x", "y") reserving 2
  ∀ v, ∀ w,
  PRE[V] V "x" ↦ v * V "y" ↦ w
  POST[_] V "x" ↦ w * V "y" ↦ v.
```

We see several important changes from the last spec. First, this time we reserve 2 stack slots, to use for local variable temporaries. Second, the spec is *universally quantified*. The function may be called whenever the precondition can be satisfied *for some values of v and w*. Note that the same quantified variables appear in precondition and postcondition, giving us a way to connect the initial and final states of a function call.

Both precondition and postcondition use notation inspired by *separation logic*. The syntax $p \mapsto v$ indicates that pointer p points to a memory cell holding value v. The $*$ operator combines facts about smaller memories into facts about larger composite memories. The concrete precondition above says that the function will be aware of only two memory cells, whose addresses come from the values of parameters "x" and "y". These cells start out holding v and w, respectively. The postcondition says that the function swaps these values.

Here is an implementation.

```
Definition swap := bmodule "swap" {{
  bfunction "swap"("x", "y", "v", "w") [swapS]
    "v" ← * "x";;
    "w" ← * "y";;
    "x" * ← "w";;
    "y" * ← "v";;
    Return 0
  end
}}.
```

We write private local variables as extra function formal parameters. The operator ;; sequences commands, the operator $\leftarrow *$ is a memory read, and $* \leftarrow$ is memory write.

Our function is not very complex, but there are still opportunities for mistakes. A quick verification establishes that we implemented it right after all.

```
Theorem swapOk : moduleOk swap.
Proof.
  vcgen; sep_auto.
Qed.
```

1.3 An Abstract Predicate: In-Place List Reverse

Bedrock also supports highly automated verifications that involve *data structures*, formalized in a way similar to *abstract predicates* in separation logic. As an example, consider the following recursive definition of an abstract predicate for singly linked lists.

```
Fixpoint sll (ls : list W) (p : W) : HProp :=
  match ls with
    | nil ⇒ [| p = 0 |]
    | x :: ls' ⇒ [| p ≠ 0 |] * ∃ p', (p ↦ x, p') * sll ls' p'
  end%Sep.
```

The type W is for machine words, and the $\%Sep$ at the end of the definition asks to parse the function body using the rules for separation logic-style assertions.

The predicate *sll ls p* captures the idea that mathematical list *ls* is encoded in memory, starting from root pointer p. The codomain *HProp* is the domain of predicates over memories.

We define *sll* by recursion on the structure of *ls*. If the list is empty, then we merely assert the lifted fact $p = 0$, forcing p to be null. Note that a lifted fact takes up no memory, so we implicitly assert emptiness of whatever memory this *HProp* is later applied to.

If the list is nonempty, we split it into head x and tail *ls'*. Next, we assert that p is not null, and that there exists some pointer p' such that p points in memory to the two values x and p', such that p' is the root of a list encoding *ls'*. By using $*$, we implicitly require that all of the memory regions that we are describing are *disjoint* from each other.

To avoid depending on Coq's usual axiom of functional extensionality, Bedrock requires that we prove administrative lemmas like the following for each new separation logic-style predicate we define.

```
Theorem sll_extensional : ∀ ls (p : W), HProp_extensional (sll ls p).
Proof.
  destruct ls; reflexivity.
Qed.
```

We add the lemma as a hint, so that appropriate machinery within Bedrock knows about it.

```
Hint Immediate sll_extensional.
```

We want to treat the predicate *sll* abstractly, relying only on a set of rules for simplifying its uses. For instance, here is an implication in separation logic, establishing the consequences of a list with a null root pointer.

```
Theorem nil_fwd : ∀ ls (p : W), p = 0
  → sll ls p ⟹ [| ls = nil |].
```

```
Proof.
  destruct ls; sepLemma.
Qed.
```

The proof only needs to request a case analysis on *ls* and then hand off the rest of the work to *sepLemma*, a relative of *sep_auto*. Staying at more or less this same level of automation, we also prove 3 more useful facts about *sll*.

```
Theorem nil_bwd : ∀ ls (p : W), p = 0
  → ⌜ ls = nil ⌝ ⟹ sll ls p.
Proof.
  destruct ls; sepLemma.
Qed.

Theorem cons_fwd : ∀ ls (p : W), p ≠ 0
  → sll ls p ⟹ ∃ x, ∃ ls', ⌜ ls = x :: ls' ⌝ * ∃ p', (p ↦ x, p') * sll ls' p'.
Proof.
  destruct ls; sepLemma.
Qed.

Theorem cons_bwd : ∀ ls (p : W), p ≠ 0
  → (∃ x, ∃ ls', ⌜ ls = x :: ls' ⌝ * ∃ p', (p ↦ x, p') * sll ls' p') ⟹ sll ls p.
Proof.
  destruct ls; sepLemma;
    match goal with
      | [ H : _ :: _ = _ :: _ ⊢ _ ] ⇒ injection H; sepLemma
    end.
Qed.
```

So that Bedrock knows to use these rules where applicable, we combine them into a *hint package*, using a Bedrock tactic *prepare*.

```
Definition hints : TacPackage.
  prepare (nil_fwd, cons_fwd) (nil_bwd, cons_bwd).
Defined.
```

Now that we have our general "theory of lists" in place, we can specify and verify in-placed reversal for lists.

```
Definition revS := SPEC("x") reserving 3
  ∀ ls,
  PRE[V] sll ls (V "x")
  POST[R] sll (rev ls) R.

Definition revM := bmodule "rev" {{
  bfunction "rev"("x", "acc", "tmp1", "tmp2") [revS]
    "acc" ← 0;;
    [∀ ls, ∀ accLs,
      PRE[V] sll ls (V "x") * sll accLs (V "acc")
```

```
      POST[R] sll (rev_append ls accLs) R ]
    While ("x" ≠ 0) {
      "tmp2" ← "x";;
      "tmp1" ← "x" + 4;;
      "x" ← ∗ "tmp1";;
      "tmp1" ∗ ← "acc";;
      "acc" ← "tmp2"
    };;
    Return "acc"
  end
}}.
```

Note that the function implementation contains a *While* loop with a *loop invariant* before it. As for all instances of invariants appearing within Bedrock programs, we put the loop invariant within square brackets. We must be slightly clever in stating what is essentially a strengthened induction hypothesis. Where the overall function is specified in terms of the function *rev*, reasoning about intermediate loop states requires use of the *rev_append* function. (There is also something else quite interesting going on in our choice of invariant. We reveal exactly what in discussing a simpler example in a later section.)

Tactics like *sep_auto* take care of most reasoning about programs and memories. A finished Bedrock proof generally consists of little more than the right hints to finish the rest of the process. The *hints* package we created above supplies rules for reasoning about memories and abstract predicates, and we can use Coq's normal hint mechanism to help with goals that remain, which will generally be about more standard mathematical domains. Our example here uses Coq's *list* type family, and the only help Bedrock needs to verify "*rev*" will be a lemma from the standard library that relates *rev* and *rev_append*, added to a hint database that Bedrock uses in simplifying separation logic-style formulas.

```
Hint Rewrite ← rev_alt : sepFormula.
```

Now the proof script is almost the same as before, except we call Bedrock tactic *sep* instead of *sep_auto*. The former takes a hint package as argument.

```
Theorem revMOk : moduleOk revM.
Proof.
  vcgen; sep hints.
Qed.
```

2 Foundational Guarantees

What does *moduleOk* really mean? The Bedrock library defines it in a way that can be used to generate theorems about behavioral properties of programs in an assembly-like language (the **Bedrock IL**), such that the theorem statement only depends on a conventional operational semantics for this language. This means we can apply Coq's normal *proof checker* to validate our verification results, without trusting anything about the process whereby the

32-bit machine words	w	$\in$	$\mathbb{W}$
Code labels	ℓ	$\in$	$\mathbb{L}$
32-bit registers	r	::=	$\mathsf{Sp} \mid \mathsf{Rp} \mid \mathsf{Rv}$
Locations	l	::=	$r \mid w \mid r + w$
Lvalues	L	::=	$r \mid l$
Rvalues	R	::=	$L \mid w \mid \ell$
Binops	o	::=	$+ \mid - \mid \times$
Instructions	i	::=	$L \leftarrow R \mid L \leftarrow R\ o\ R$
Tests	t	::=	$= \mid \neq \mid < \mid \leq$
Jumps	j	::=	$\mathsf{goto}\ R \mid \mathsf{if}\ R\ t\ R\ \mathsf{then\ goto}\ \ell\ \mathsf{else\ goto}\ \ell$
Blocks	B	::=	$i^*; j$
Programs	P	::=	B^*

Figure 1: Syntax of the Bedrock IL, where * denotes zero-or-more repetition

proofs were constructed. When the *trusted code base* of a verification system is so small, we call the system (or the theorems it produces) *foundational*.

This section of the tutorial explains exactly what is the final product of a Bedrock verification. Here and throughout the tutorial, we omit fully detailed formalizations, since the Coq source code already does a more thorough job of that than we could hope to do here.

2.1 The Bedrock IL

Figure 1 gives the complete syntax of the Bedrock IL, which is meant to be a cross between an assembly language and a compiler intermediate language. Like an assembly language, there are a fixed word size, a small set of registers, and direct access to an array-like finite memory. Like a compiler intermediate language, the Bedrock IL is designed to be compiled to a variety of popular assembly languages, though this compilation process is more straightforward than usual. There is no built-in notion of local variables or calling conventions, but code labels are maintained with special syntactic treatment, to allow compilation to perform certain jump-related optimizations soundly.

The IL has a standard operational semantics, implemented in Coq. A global parameter of a program execution maps code labels to machine words, so that memory and register values may be treated uniformly as words, even with stored code pointers. The semantics *gets stuck* if a program tries to jump to a word not associated with any code label. Further, another piece of global state gives a *memory access policy*, identifying a set of addresses that the program may read from or write to. Execution gets stuck on any memory access outside this set. One consequence of verifying a whole-program Bedrock module is that we are guaranteed lack of stuckness during execution, starting in states related appropriately to the module's specs.

Second-order variables	α		
Coq propositions	P	$\in$	Prop
Machine states	s	$\in$	$\mathbb{S}$
State assertions	f	$\in$	$\mathbb{S} \to \mathsf{PropX}$
Coq terms	v		
PropX	ϕ	$::=$	$\lceil P \rceil \mid \mathsf{Cptr}\ w\ f \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \supset \phi \mid \forall x, \phi \mid \exists x, \phi$ $\mid \alpha(v) \mid \forall \alpha, \phi \mid \exists \alpha, \phi$

Figure 2: Syntax of the XCAP assertion language

2.2 The XCAP PropX Assertion Logic

It is one of the surprising facts of formal semantics that merely stating an operational semantics for a programming language is not enough to enable *modular* program verification. That is, we want to verify libraries separately and then compose their correctness theorems to yield a theorem about the final program. We must fix some *theorem format* that enables easy composition, and a *program logic* may be thought of as such a format.

Bedrock adopts an adapted version of the XCAP program logic by Ni and Shao. The central novel feature of XCAP is support for *first-class code pointers* in a logic expressive enough to verify *functional correctness*, not just traditional notions of type safety. However, XCAP's insight is to apply the *syntactic approach to type soundness* in this richer setting. We no longer think in terms of assigning types to the variables of a program, but we retain the key idea of establishing a global *invariant* that all reachable program states must satisfy, where the invariant is expressed in terms of a *syntactic language of assertions*. In Coq, this notion of *syntactic* means *deep embedding* of a type of formulas.

Figure 2 gives the syntax of PropX, XCAP's language of formulas, otherwise known as an *assertion logic* when taken together with the associated proof rules. The standard connectives $\wedge$, $\vee$, $\supset$, $\forall$, and $\exists$ are present, but a few other cases imbue the logic with a richer structure. First, the lifting operator $\lceil \_ \rceil$ allows injection of *any Coq proposition*.

One may wonder what is the point of defining a layer of syntax like this one, instead of just using normal Coq propositions. The surprising answer is that it is difficult to support *modular theorems about first-class code pointers* without some extra layer of complication, and for XCAP, that layer is associated with formulas *Cptr w f*. Such a formula asserts that word w points to a valid code block, whose specification is f, a function from machine states to formulas. The idea is that $f(s)$ is true iff it is safe to jump to w in state s.

It may be unclear how this logic connects to our earlier examples. We only have a way to say when it is *safe* to jump to a code block, which does not directly yield the discipline of functions, preconditions, and postconditions. The explanation is that we *encode* such disciplines using *higher-order* features. Bedrock IL programs, like assembly programs, are inherently in *continuation-passing style*, and it is possible to lower *direct style* programs to this format and reason about them in a logic that only builds in primitives for continuations,

not functions. The freedom to work with continuations when needed will be invaluable in implementing and verifying systems components like thread schedulers.

The second line of the grammar for PropX gives some more interesting cases: those associated with *impredicative quantifiers*, which may range over assertions themselves. With these quantifiers, we can get around an apparent deficiency of *Cptr*, which is that its arguments must give the *exact* spec of a code block, whereas we will generally want to require only that the spec of the code block be *implied* by some other spec. We define an infix operator @@ for this laxer version of *Cptr*.

```
Notation "w @@ f" := (∃, Cptr w #0 ∧ ∀ s, f s ⊃ #0 s)%PropX.
```

This syntax is complicated by the fact that we represent impredicative quantifiers with *de Bruijn indices*. Unraveling that detail, we can rephrase the above definition as: program counter w may be treated as having spec f if there exists α such that (1) α is the literal spec of w and (2) any state s satisfying f also satisfies α.

A PropX ϕ is translated to a `Prop` using the *interp* function, applied like *interp specs* ϕ, where *specs* is a partial function from code addresses to specs. Under the hood, *interp* is implemented via an explicit natural deduction system for PropX. This system is unusual in that *the impredicative quantifiers have introduction rules but no elimination rules.* As a result, we may really only reason non-trivially about those quantifiers at the level of the meta-logic, which is Coq. One consequence is that we cannot transparently and automatically translate uses of *interp* into normal-looking Coq propositions. However, this can be done for formulas that do not use implication. A Bedrock tactic *propxFo* handles that automated simplification, where it applies.

The most commonly used Bedrock tactics are designed to hide the use of PropX where feasible, though sometimes details creep through. It is important that we have this machinery around, to allow modular reasoning about programs with first-class code pointers.

One further foundational point is worth making: while most separation logics outside of Coq build into their assertion languages such constructs as $\mapsto$ and $*$, with XCAP and related systems, we instead define these as derived operators, with definitions in terms of the basic PropX connectives.

2.2.1 A Note on the Format of Invariants

Recall the beginning of the *While* loop from our last example:

$$
\begin{array}{l}
[\forall\ ls, \forall\ accLs, \\
\quad PRE[V]\ sll\ ls\ (V\ \text{"x"}) * sll\ accLs\ (V\ \text{"acc"}) \\
\quad POST[R]\ sll\ (rev\_append\ ls\ accLs)\ R\ ] \\
While\ (\text{"x"} \neq 0)\ \{
\end{array}
$$

The loop invariant is strange, since it includes both a *precondition* and a *postcondition*. Standard Hoare-logic loop invariants only represent assertions over single program states. How should an invariant like the above be interpreted?

One answer is to consider it a new notation in separation logic, as in a VSTTE 2010 paper

by Thomas Tuerk. One way to think of it is: an invariant's *precondition* describes what the machine state looks like upon entering a loop iteration, and the *postcondition* describes what the state must be transformed into before it is legal to *return* from the current function.

It also turns out that this idea of loop invariant is actually *more natural* for assembly language than the more common notation is. It all has to do with the idea that assembly programs are naturally thought of as in *continuation-passing style*, since call stacks and return pointers are represented explicitly via memory and registers. Thus, the natural idea of specification for a function is just a precondition, not a precondition plus a postcondition.

Using a more informal notation, the surface syntax for loop invariants could be written like:

$\forall \vec{x}.\ \{P\}\ \{Q\}$

There are both a precondition and a postcondition, potentially with both mentioning some variables quantified at the top level. We desugar such invariants to underlying Bedrock IL specifications (preconditions) like so:

$\{\exists \vec{x}.\ P \wedge \{Q\}\mathsf{Rp}\}$

We use a *nested Hoare double*, writing $\{Q\}\mathsf{Rp}$ to assert that *register* Rp *holds a pointer to a code block whose precondition is compatible with* Q. That is, we mention the return pointer explicitly, rather than keeping it as implicit in our use of a postcondition.

Actually, to support separation logic reasoning, the desugaring is a bit more complex. It is more accurate to write as:

$\{\exists \vec{x}, \alpha.\ P * \alpha \wedge \{Q * \alpha\}\mathsf{Rp}\}$

We build the *frame rule* into the desugaring scheme. Some piece of memory is described by an unknown predicate α on entry and must still be described by α on exit.

So there is a first-principles explanation of what Bedrock loop invariants mean.

This treatment of invariants is natural in continuation-passing style; implementing the conventional notation would actually require more work for the Bedrock authors. However, the alternate notation is actually quite useful in concert with separation logic. Invariants in this style can provide very effective hints on where the *frame rule* ought to be applied.

Specifically, suppose that the state upon entry to a loop is described by an invariant with precondition $P * R$ and postcondition $Q * R$. In other words, the loop will transform some state from satisfying P into satisfying Q, and there is some additional state satisfying R that need not be touched. In such a state, we can write a loop invariant with precondition based only on P and postcondition based only on Q. We forget about R for the rest of this function's verification.

Such a technique is very helpful in traversals of linked data structures. For instance, consider a loop over the cells of a linked list. Traditional separation logic requires a loop invariant that splits the full list into a prefix list segment that has already been visited and a suffix list that has not yet been visited. In Bedrock, we can instead use a loop invariant that only mentions the unvisited suffix. Automatic application of the frame rule lets us gradually hide list elements from the invariant as we visit them.

For a lengthier explanation of this pattern, see again Tuerk's VSTTE 2010 paper.

The next section gives an example program containing one invariant, which triggers a

similar use of the frame rule, but for a function call rather than a loop. That is, after the call, we no longer need to know about certain parts of memory, so we need not mention those parts in the invariant. It may be a useful exercise to consider the invariant in light of the desugaring to continuation-passing style.

2.3 The XCAP Program Logic

Now we are finally ready to describe the end product of a Bedrock verification (though, as forewarned, we will stay fairly sketchy, since details abound in the Coq code). A verified program is nothing more than a normal Bedrock IL program, where *each basic block is annotated with a* PropX *assertion*. For the program to be truly verified, two conditions must be proved for each block b with spec f. First, a *progress* condition says: for any initial state satisfying f, if execution starts at the beginning of b, then execution continues safely without getting stuck, at least until after the jump that ends b. Second, a *preservation* condition says: for any state satisfying f, if execution starts at the beginning of b and makes it safely to another block b', then b' has some spec that is satisfied by the machine state at this point.

The terms *progress* and *preservation* are chosen to evoke the *syntactic approach to type soundness*, which is based around a small-step operational semantics and an inductive invariant on reachable states: each state (program term) is well-typed, according to an inductively defined typing judgment. In XCAP, we follow much the same approach, where a single small-step transition is *one basic block execution*, and the inductive invariant is that *the current machine state satisfies the spec of the current basic block*.

Thus, adapting the almost trivial syntactic type soundness proof method, we arrive at some theorems about verified Bedrock programs. First, if execution begins in a block whose spec is satisfied, then *execution continues forever without getting stuck*. Second, if execution begins in a block whose spec is satisfied, then *every basic block's spec is satisfied whenever control enters that block*. The first condition is a sort of *memory safety*, while the second is a kind of *functional correctness*.

The *moduleOk* theorems we established in the last section are actually about a higher-level notion, that of *structured programs* and what it means for them to be correct. We defer details of structured programs to a later section. For now, what matters is that structured programs can be *compiled* into verified Bedrock IL programs, at which point their code and the associated guarantees can be understood as in this section.

3 Interactive Program Verification

A central design point of Bedrock is to provide tactics to make the verification of individual programs be as automatic as possible. Realistic programs involve many details, few of which are interesting, so why ask the programmer to account for them manually? The programmer helps Bedrock by providing *hints* that can come in many varieties. There are the normal `auto` and `autorewrite` hints familiar to most Coq users, but there are also new notions like *unfolding rules* and *symbolic evaluators*. Bedrock verifications of serious programs will tend

to use all of these notions in concert.

Final proofs may be automated, but, during development, it is usually helpful to step through proofs incrementally. Of course, this is a mode that Coq supports very well. However, with Bedrock, there is no need to work at such fine granularity as is found in the average Coq proof script. Instead, there is a small vocabulary of automation procedures that modify proof states in ways that a human can follow.

The workhorse *sep* tactic is essentially defined as the following:

```
Ltac sep hints := post; evaluate hints; descend; repeat (step hints; descend).
```

The tactic is parameterized on *hints*, which provides a set of abstract predicate unfolding rules. There is a default hints package *auto_ext*, and *sep_auto* is defined as *sep auto_ext*.

A few basic steps make up the *sep* procedure. First, *post* does post-processing on the results of *vcgen*'s verification condition generation, trying to eliminate as much explicit PropX notation as possible. Next, *evaluate* implements *symbolic execution*, modifying a block's precondition to reflect the effects of a piece of straightline code, possibly including the equivalents of assume statements to record the results of conditionals. The *descend* tactic is a generic simplifier; among other steps, it descends under existential quantifiers and conjunctions in the conclusion, introducing new unification variables for the quantifiers. The rest of the procedure is a loop over *step* and *descend*, where the former implements a variety of basic steps in a Bedrock proof.

It is probably easiest to illustrate the basic steps by example. To make things interesting, consider this function which calls our earlier swap function. In filling out the new *reserving* clause, we keep in mind that the new function will call "*swap*", and we plan not to use any private local variables. Therefore, the only reserved stack space that we need is that for the function call, computed by this recipe: one slot for each function argument, plus one for a saved return pointer, plus the callee's number of reserved stack slots. The notation $p \stackrel{?}{\mapsto} n$ indicates an allocated memory region of unknown contents, beginning at p and spanning n words.

```
Definition sillyS := SPEC("p", "q") reserving 5
  PRE[V] V "p" ?↦ 1 * V "q" ?↦ 1
  POST[R] ⌜ R = 3 ⌝ * V "p" ?↦ 1 * V "q" ?↦ 1.

Definition sillyM := bimport [[ "swap"!"swap" @ [swapS] ]]
   bmodule "silly" {{
    bfunction "main"("p", "q") [sillyS]
      "p" *← 3;;
      "q" *← 8;;

      Call "swap"!"swap"("p", "q")
      [∀ v,
        PRE[V] V "q" ↦ v
        POST[R] ⌜ R = v ⌝ * V "q" ↦ v ];;
```

```
    "q" ← * "q";;
    Return "q"
  end
}}.
```

We use the *Call* notation, which always requires an invariant afterward. That invariant position is often a convenient place to simplify the state that we are tracking. The invariant above is an example: we *forget about the pointer p*, remembering only q, since the rest of the function only touches q. Bedrock's tactics automatically justify this state reduction, with a reasoning pattern reminiscent of separation logic's *frame rule*. That pattern and many others are encapsulated in the definition of *step*.

To demonstrate the recommended interactive verification approach, we will step through a more manual proof of the most challenging verification condition, the one associated with the *Call* command. Interested readers may step through this proof script in Coq, so we will not dump the gory details of subgoal structure into this tutorial. Instead, we give a high-level account of what each subgoal means and how it is dealt with.

```
Theorem sillyMOk : moduleOk sillyM.
Proof.
```

vcgen.

Focus 5.

This is the subgoal for the function call. We always begin with post-processing the verification condition.

post.

Next, we need to execute the instructions of the prior basic block symbolically, to reflect their effects in the predicate that characterizes the current machine state.

evaluate auto_ext.

At this point, we are staring at the spec of "*swap*", which begins with some existential quantifiers and conjunctions. One of the conjuncts comes from a use of the @@ derived PropX operator, to express the postcondition via a fact about the return pointer we pass to "*swap*". We call *descend* to peel away the quantifiers and conjunctions, leaving the conjuncts as distinct subgoals.

descend.

This subgoal is the precondition we gave for "*swap*", with an extra fact added to characterize the stack contents in terms of values of local variables. There are unification variables in positions that were previously existentially quantified. This sort of goal is just what *step* is designed for.

step auto_ext.

We are thrown back another goal, this time stated as an implication between separation logic assertions. One of the unification variables has been replaced with a known substitution for local variable values, which will enable *step* to discharge the subgoal completely this time.

step auto_ext.

Here is an easy subgoal. It asks us to find a spec for the return pointer we pass in the function call, and exactly such a fact was given to us by *vcgen*.

step auto_ext.

We have finished proving the precondition of "*swap*". Now we must prove that its postcondition implies the invariant we wrote after the function call. The form of the obligation is an implication within PropX, where the antecedent is the postcondition of "*swap*" and the consequent is the invariant we wrote after the call. Recall that simplifying PropX implications into normal-looking Coq formulas is difficult. However, we can rely on *step* to simplify the implication into some more basic subgoals, some of which will still be PropX implications.

descend; *step* auto_ext.

The first resulting subgoal is an implication between the postcondition of "*swap*" and the *PRE* clause from the post-call invariant. Again, this is exactly the sort of separation logic simplification that *step* handles predictably.

step auto_ext.

Now we are asked to find a specification for the original return pointer passed to "*main*". Again, *vcgen* left us a matching hypothesis.

step auto_ext.

We are in the home stretch now! The single subgoal asks us to prove an implication $P \supset Q \supset R$, where P is the *POST* clause of the post-call invariant, Q is the postcondition of "*swap*", and R is the literal specification of the original return pointer for "*main*". In fact, R is an application of a second-order variable to the current machine state. We also have a hypothesis telling us that R is implied by the postcondition we originally ascribed to "*main*" in its spec.

Our first step is to reduce the implication to just $P \supset R$, augmented with extra *pure* (memory-independent) hypotheses that we glean from Q. The intuition behind this step is that we already incorporated in P any facts about memory that we will need.

descend; *step* auto_ext.

Now we prove $P \supset R$ using the hypothesis mentioned above, which can be thought of as a quantified version of $U \supset R$. That means *step* can help us by reducing the subgoal to $P \supset U$.

step auto_ext.

Here is another PropX implication, which we want to simplify to convert as much structure as possible into normal Coq propositions.

step auto_ext.

The first of two new subgoals is an equality between the current stack pointer and the value that the spec of "*main*" says it should have. We call a library tactic for proving equalities between bitvector expressions.

words.

The last subgoal is an implication between the *POST* clause of the post-call invariant and the overall postcondition of "*main*". This is just a separation logic implication, so we make short work of it.

step auto_ext.

This concludes our proof of the most interesting verification condition. Let's back up to the high level and prove the whole theorem automatically.

```
Abort.
```

A manual exploration like the above is about learning which hints will be important in proving the theorem. One might even do this exploration using more usual manual Coq proofs. In the end, we distill what we've learned into hint commands. In the script above, we saw only one place where *sep* wouldn't be sufficient, and that was an equality between machine words. Therefore, we register a hint for such cases.

```
Hint Extern 1 (@eq W _ _) ⇒ words.
```

Now it is easy to prove the theorem automatically.

```
Theorem sillyMOk : moduleOk sillyM.
Proof.
  vcgen; (sep_auto; auto).
Qed.
```

We might even make small changes to the program specification or implementation, and often a proof script like the above will continue working.

4 More

This tutorial will likely grow some more sections later. One topic worth adding is Bedrock's *structured programming system*, which includes support for extending the visible programming language with new control flow constructs, when they are accompanied by appropriate proofs.