The Bedrock Structured Programming System

Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Abstract

We report on the design and implementation of an extensible programming language and its intrinsic support for formal verification. Our language is targeted at low-level programming of infrastructure like operating systems and runtime systems. It is based on a cross-platform core combining characteristics of assembly languages and compiler intermediate languages. From this foundation, we take literally the saying that C is a "macro assembly language": we introduce an expressive notion of *certified low-level macros*, sufficient to build up the usual features of C and beyond as macros with no special support in the core. Furthermore, our macros have integrated support for strongest postcondition calculation and verification condition generation, so that we can provide a high-productivity formal verification environment within Coq for programs composed from any combination of macros. Our macro interface is expressive enough to support features that low-level programs usually only access through external tools with no formal guarantees, such as declarative parsing or SQL-inspired querying. The abstraction level of these macros only imposes a *compile-time* cost, via the execution of functional Coq programs that compute programs in our intermediate language; but the *run-time* cost is not substantially greater than for more conventional C code. We describe our experiences constructing a full C-like language stack using macros, with some experiments on the verifiability and performance of individual programs running on that stack.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.3.4 [*Programming Languages*]: Compilers

Keywords generative metaprogramming; interactive proof assistants; low-level programming languages; functional programming

1. Introduction

A fundamental tension in programming language design is between performance and abstraction. Closer to the high-performance end of the spectrum, we have languages like C, which are often used to implement low-level infrastructure that many applications depend on. Examples include operating systems and runtime systems, which today are almost exclusively written in C and its close relatives.

Need we always choose between performance and abstraction? One approach to retaining both is *metaprogramming*, from basic cases of textual macro substitution as in the C preprocessor, to sophisticated code generators like Yacc. Here we employ abstractions that in a sense are *compile-time only*, running programs that generate programs in C or other low-level languages. As a result, we incur no run-time cost to abstraction.

Unfortunately, code generation is hard to get right. The C preprocessor's textual substitution mechanism allows for all sorts of scoping errors in macro definitions. There exist widely used alternatives, like the macro systems of Lisp/Scheme, OCaml (Camlp4), and Haskell (Template Haskell), which can enforce "hygiene" requirements on lexical scoping in macro definitions.

The functional programming community knows well the advantages of code generation with languages like Haskell and ML as *metalanguages* for *embedded domain-specific languages (DSLs)*. That is, one represents the programs of some object language, like C or CUDA, as data within the metalanguage, making it possible to compute programs of the object language *at compile time* using all of the abstraction features of the metalanguage.

There is a tower of static assurance levels for this kind of generative metaprogramming. Tools like Yacc receive little static assurance from their implementation languages, as they output source code as strings that the implementation language does not analyze. Conventional macro systems, from primitive token substitution in C to hygienic Scheme macros, can provide stronger *lexical* guarantees. Next in the tower are languages that allow for *static typing* of macros, guaranteeing that any of their applications produce both scope-safe and type-safe object-language code. Languages in this category include MetaML [30] and MacroML [13], for *homogeneous* metaprogramming where the meta and object languages are the same; and MetaHaskell [22], for *heterogeneous* metaprogramming where meta and object languages are different (e.g., Haskell and CUDA). Our contribution in this paper is to continue higher up this tower of static guarantees: we support separate static checking of macros to guarantee that *they always produce functionally correct object-language code*.

The programming language and associated verification tools will need to provide a *proof rule* for each macro. A naive proof rule just expands macro applications using their definitions, providing no abstraction benefit. We could ask instead that *macro definitions include proof rules*, and that the verification system requires programmers to *prove that their macro definitions respect the associated proof rules*. In effect, we have a macro system supporting modularity in the style of well-designed functions, classes, or

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2326-0/13/09.
http://dx.doi.org/10.1145/2500365.2500592

libraries, with clear separation between implementation and interface.

In this paper, we report on our experiences building such a macro system for a low-level language, intended as a replacement for C in the systems programming domain. This macro system is part of the Bedrock library for the Coq proof assistant, which provides an integrated environment for program implementation, specification, and verification. A prior publication [8] introduced Bedrock's support for proof automation, implementing example programs using an earlier version of our macro system but not describing it in detail. Since that paper was written, the macro system has evolved, and we have used it to implement more interesting examples. We now present its design and implementation in detail, along with the higher-level language design principle that motivates it.

Our metalanguage is Gallina, the functional programming language in Coq's logic. Our object language is a tiny compiler intermediate language. We build up the usual features of C via unprivileged macros, and then we go further and implement features like *declarative parsing* and *declarative querying of data structures* as macros rather than ad-hoc external tools like Yacc and MySQL.

To support productive formal verification, we tag each macro with a *proof rule*. To be more precise, each macro contains a *predicate transformer*, which evolves a logical assertion to reflect the effects of a program statement; and a *verification condition generator*, which outputs proof obligations associated with a use of the macro. These are some of the standard tools of highly automated program verification, and requiring them of all macros lets us build a generic verification environment that supports automated proofs about programs built with macros drawn from diverse sources. We call the central abstraction **certified low-level macros**.

Our implementations and proofs have much in common with compiler verification. For instance, CompCert [21] is a C compiler implemented in Coq with a proof that it preserves program semantics. Each of our macro implementations looks like one case of the CompCert compiler, dealing with a specific source language syntax construct, paired with the corresponding correctness proof cases. A specific program may draw upon several macros implemented by different programmers, who never considered how their macros might interact. The proof approach from CompCert does not extend straightforwardly to this setting, since each CompCert subproof refers to one or more *simulation* relations between the programs of different fixed languages. It is not apparent how such an approach could be extended to verify compilation rules for extensible languages. For instance, in that conventional setting, function pointers may be specified informally as "points to the code our compiler would generate from the corresponding source-level function body," an explanation that does not adapt naturally to the case of an extensible compiler. We would rather not verify handling of function pointers relative to a particular set of available macros.

The key selling point of compile-time metaprogramming is *performance*. Programmers will come up with different code constructs that mimic those available in higher-level programming languages, but implemented in a way that facilitates case-specific optimization and in general promotes performance. We are able to attach specifications to such macros in a way that makes reasoning about use of a macro more like reasoning about a function call than reasoning directly about low-level code after macro expansion.

At a high level, then, our contribution can be viewed alternatively as *an extensible programming language* or *an extensible Hoare-logic program verifier*. We present **the first low-level language with a notion of macros that cleanly separates interface and implementation, to such an extent as to enable mostly automated correctness verification without macro expansion**. We prove correctness theorems within a framework for *modular verification*, where we can verify libraries separately and then link together their correctness theorems into whole-program correctness theorems, without revisiting details of code implementation or proof. This modularity holds even in the presence of a mutable heap that may hold callable code pointers, thanks to the XCAP [26] program logic that sits at the base of our framework.

$$\begin{array}{rrcl}
\text{Constants} & c & ::= & \text{[width-32 bitvectors]} \\
\text{Code labels} & \ell & ::= & \ldots \\
\text{Registers} & r & ::= & \mathsf{Sp} \mid \mathsf{Rp} \mid \mathsf{Rv} \\
\text{Addresses} & a & ::= & r \mid c \mid r + c \\
\text{Lvalues} & L & ::= & r \mid [a]_8 \mid [a]_{32} \\
\text{Rvalues} & R & ::= & L \mid c \mid \ell \\
\text{Binops} & o & ::= & + \mid - \mid \times \\
\text{Tests} & t & ::= & = \mid \neq \mid < \mid \leq \\
\text{Instructions} & i & ::= & L \leftarrow R \mid L \leftarrow R\ o\ R \\
\text{Jumps} & j & ::= & \mathsf{goto}\ R \mid \mathsf{if}\ (R\ t\ R)\ \mathsf{then}\ \ell\ \mathsf{else}\ \ell \\
\text{Specs} & S & ::= & \ldots \\
\text{Blocks} & B & ::= & \ell : \{S\}\ i^*; j \\
\text{Modules} & M & ::= & B^*
\end{array}$$

Figure 1. Syntax of the Bedrock IL

The remaining sections of this paper will walk through the steps of building our stack of macros. We begin with a tiny assembly-like language and proceed to introduce our notion of certified low-level macros for it, build macros for basic C-like constructs, add support for local variables and function calls, and culminate in the high-level macros for declarative parsing and querying. Next we evaluate the effectiveness of our platform in terms of program verifiability and run-time performance, and we finish with a comparison to related work.

Source code to both the Bedrock library and our example programs is available in the latest Bedrock source distribution at:

`http://plv.csail.mit.edu/bedrock/`

2. The Bedrock IL

The lowest layer of Bedrock is a simple intermediate language inspired by common assembly languages. This Bedrock IL is easy to compile to those "real" languages, using only localized replacement of IL concepts with assembly language constructs. A single verification result applies to all target architectures. So far, we have built a compiler to AMD64 assembly (currently unverified), and we expect it will be easy to implement sound translations to other popular architectures like ARM and 32-bit x86.

Figure 1 gives the full syntax of the Bedrock IL. There are small finite sets of registers, addressing modes (for both 8-bit and 32-bit memory accesses), and forms of lvalues and rvalues (borrowing C terminology). Straightline instructions do assignment or arithmetic, and jump instructions do unconditional branching or branching based on the result of an arithmetic comparison. A standalone code module is a set of basic blocks, each with a code label and a specification.

Bedrock is based on the XCAP program logic [26] of Ni and Shao, and block specifications use the XCAP assertion language, which supports a limited form of higher-order logic tailored to reasoning about first-class code pointers. A module is *correct* when every one of its blocks satisfies two conditions:

- *Progress*: When control enters the block in a machine state satisfying the spec, control continues safely at least until the end of the block.

- *Preservation*: When control enters the block in a machine state satisfying the spec and exits the block by jumping to some other block, the spec of the new block is satisfied.

In this way, we can construct an inductive proof of infinite safe execution, giving us *memory safety*. Further, we may also treat each block spec as an *assertion*, so that we verify absence of assertion failures, giving us *functional correctness*. XCAP supports separate verification and linking of modules, via mechanisms that we will not go into here, but we do want to emphasize that XCAP supports *modular verification*, where libraries can be verified separately in a way that permits composition of their correctness theorems to conclude whole-program theorems. It is not trivial to design a framework that permits such modular proof rules in the presence of a mutable higher-order heap, where code from one module might find in the heap a pointer to code from another module written and verified independently. Building on XCAP enables us to realize this level of modularity without doing much explicit extra work.

What does it mean for a Bedrock IL program to be *safe*? There are two sorts of bad instruction executions that violate safety. First, a jump to a nonexistent code label is illegal. Second, a Bedrock IL program is restricted to read or write only memory addresses allowed by a configurable *memory policy*.

These informal notions are codified in a conventional Coq operational semantics. We support sound compilation to a variety of assembly languages by considering machine states to contain a read-only component describing the byte ordering of words in memory, with enough expressiveness to describe little-endian, big-endian, and other orders. As a result, we can work with a trivial "array of bytes" memory model, rather than the more complex memory model used in, e.g., the informal ANSI C semantics and the closely related formal semantics of CompCert [21]. The simplistic memory model we adopt makes it difficult to reason about many standard program transformations and optimizations, but we still achieve reasonable performance with an optimizer-free implementation (see Section 6).

There is a great benefit to starting from a language with such a simple semantics. Our final platform is *foundational*, in the sense that we produce formal proofs that can be checked and audited with relatively small trusted code bases. We need to trust the Coq proof checker and its dependencies, which can be made small compared to program verifiers in general. Beyond that, we must make sure that the statement of the theorem we prove matches our informal notion of program safety or correctness. The theorems that come out of our system are stated only in terms of the Bedrock IL operational semantics, which is quite small and easy to audit. Most importantly in the context of this paper, the final theorem statements are independent of any details of our macro system.

A final foundational theorem will be of the form "if execution begins at code address ℓ_1 in a state satisfying the spec of ℓ_1, then if execution ever reaches code address ℓ_2, the spec of ℓ_2 will be satisfied." In the sections that follow, we will introduce convenient notations not just for programs, but also for specifications; for instance, see the notation for specifying functions with local variables in Section 4. If such a notation appears in a final correctness theorem, then its definition must be considered part of the trusted base. However, we are able to prove final theorems that reduce our notations to simpler concepts. For instance, while we may verify the `main()` function of a program against a precondition-postcondition-style specification inspired by separation logic [28], the final theorem statement need only say that `main()` requires "all memory addresses between 0 and N are allowed by the memory access policy." We prove that the latter implies the former, and so a final audit of a verified program need only ensure that the latter is accurate.

```
Definition appendS : spec := SPEC("x", "y") reserving 2
  Al ls1, Al ls2,
  PRE[V] sll ls1 (V "x") ∗ sll ls2 (V "y")
  POST[R] sll (ls1 ++ ls2) R.

bfunction "append"("x", "y", "r", "tmp") [appendS]
  If ("x" = 0) {
    Return "y"
  } else {
    "r" ← "x";;
    "tmp" ←∗ "x" + 4;;
    [Al p1, Al x, Al ls1, Al ls2,
      PRE[V] ⌈ V "x" ≠ $0 ⌉ ∗ ⌈ V "tmp" = p1 ⌉ ∗ V "x" ↦ x
        ∗ (V "x" +̂ $4) ↦ p1 ∗ sll ls1 p1 ∗ sll ls2 (V "y")
      POST[R] ⌈ R = V "r" ⌉ ∗ sll (x :: ls1 ++ ls2) (V "x") ]
    While ("tmp" ≠ 0) {
      "x" ← "tmp";;
      "tmp" ←∗ "x" + 4
    };;

    "x" + 4 ∗← "y";;
    Return "r"
  }
end
```

Figure 2. Bedrock implementation of destructive list append

3. Certified Low-Level Macros

Bedrock's structured programming system uses macros to present a C-like view of the Bedrock IL. Figure 2 shows an example, a destructive linked-list append function, beginning with its formal specification, whose explanation we put off until later. The body of the implementation uses standard control flow constructs like `if` and `while` to group together sequences of operations on local variables, which are named with string literals to appease Coq's lexer. (In this paper, we never use a string literal for its more conventional purpose in C-like code, so it is safe to assume that literals indicate object-language identifiers.) A token $\leftarrow *$ indicates a memory read operation, while $*\leftarrow$ indicates a write. A *loop invariant* appears within square brackets. We will have a bit more to say later about the language of specifications and invariants, though it is not our focus in this paper, as our architecture applies to other specification languages, too.

For now, the take-away message from Figure 2 is that it denotes *a Coq program for computing an assembly program*, in the style of heterogeneous generative metaprogramming. None of the control flow constructs are built into the language, but rather they use macros that any Bedrock programmer can define without modifying the library. We also want to prove that the computed assembly program is correct. Programming, compilation, and proving are all done completely within the normal Coq environment.

We will introduce our central concept of macro in two stages. First we introduce the aspects needed to support compilation of programs, and then we add the features to support verification.

3.1 The Code Generation Part of a Macro

In our setting, a macro stands for a *statement* in the sense of C-like languages. To make sense of this concept, we must impose a notion of *structured programs* on top of the freeform control flow of the Bedrock IL.

Figure 3 summarizes such an interface. The fundamental task of a macro is to *mutate a program by appending new basic blocks to it*. Successive blocks are assigned successive numeric labels. To run a macro, it must be passed one input: the *exit label* of the block its statement should jump to when finished. This label will have

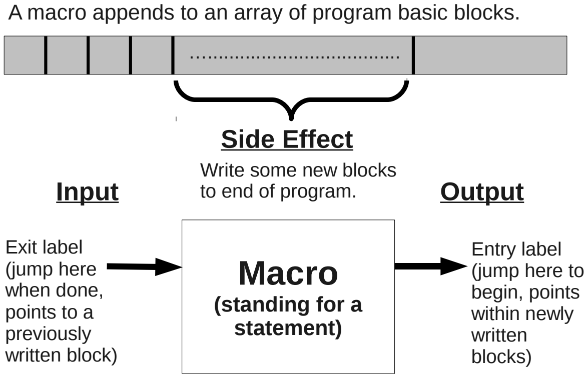
A macro appends to an array of program basic blocks.
....
Side Effect
Write some new blocks to end of program.
Input
Output
Exit label (jump here when done, points to a previously written block)
Macro
(standing for a statement)
Entry label (jump here to begin, points within newly written blocks)

Figure 3. The interface of a low-level macro

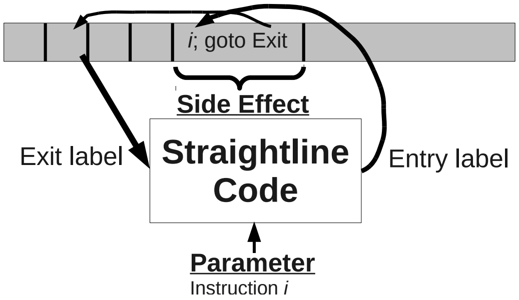

Figure 4. Sketch of macro for single straightline instructions

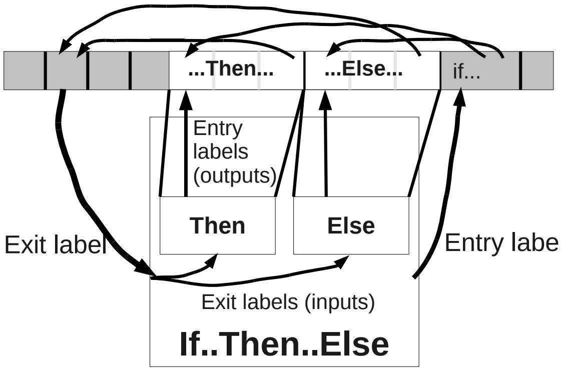

Figure 5. Sketch of macro for if..then..else

been allocated previously by other macros. A macro runs by not only creating new basic blocks, but also returning one more value: an *entry label* within the freshly generated code blocks. A macro's statement is run by jumping to the entry label.

Consider the very basic example of a macro for converting a straightline IL instruction into a statement. Figure 4 sketches a simple implementation. Macros are implemented as functional programs in Coq's logic, so it is natural to think of them as *statement combinators*. The straightline code combinator takes an instruction i as its parameter. When run, the macro allocates a single block, consisting of i followed by a direct jump to the exit label. The output entry label points to the new block.

A more involved example is the if..then..else macro, as sketched in Figure 5. While Bedrock IL includes a standard conditional jump instruction, we would rather program in terms of a higher-level statement combinator that hides details of code block label allocation. The if..then..else combinator takes three arguments: a *test expression*, suitable as an argument to Bedrock IL's low-level conditional jump; a *then statement*, to run when the test passes; and an *else statement*, to run when the test fails. The two statement parameters will generally be constructed with further macro applications, and the if..then..else combinator should work correctly regardless of the details of the two statements.

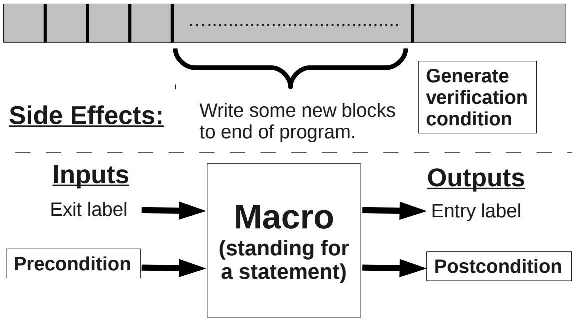

Figure 6. The interface of a certified low-level macro

An instance of the if..then..else combinator works as follows:

- The macro is passed an *exit label* as input. The same label is passed along unchanged as the *exit label* inputs to the *then* and *else* statements. That is, regardless of which way the test expression evaluates, the statement we jump to should exit the if..then..else upon completion.
- Each of the *then* and *else* macros generates some code blocks to represent itself, outputting an *entry label* for each. The if..then..else macro adds one more code block, a conditional jump using the test expression, jumping to the *entry label* of either the *then* or *else* case, depending on how the test evaluates.
- Finally, the overall macro outputs the address of the conditional test as its own *entry label*.

The working of the macro is broadly similar to standard compilation strategies. To support an expressive macro system, it is crucial to choose a macro interface compatible with a wide range of compilation rules. Many other such architectures are possible. For instance, Benton et al. [2] use an intermediate language with explicitly scoped local labels. The verification support of our approach ought to adapt straightforwardly to many such compilation schemes, since in a sense verification considers only a macro's *interface* and not its *implementation*.

It is worth mentioning one mundane aspect of macro implementation, which is registering concrete parsing notations for macros. We rely on Coq's extensible parser for this purpose. For instance, here is the notation we introduce for calls to the if..then..else combinator `If_`:

```
Notation "'If' c { b1 } 'else' { b2 }" := (If_ c b1 b2)
  (no associativity, at level 95, c at level 0) : SP_scope.
```

All of the standard control-flow constructs of C-like languages are straightforward to encode in our macro system, but, before we consider more examples, we back up to extend the macro interface for verification support.

3.2 Verification Support

Figure 6 gives the expanded interface of *certified* low-level macros, where we add the ingredients needed to build a formal verification environment that supports automated proofs about programs that use macros drawn from diverse sources. At a high level, the addition is this: We previously defined a macro as a **compilation rule**. Verification support involves also including a **predicate transformer** (closely related to the idea of a *strongest postcondition* calculator) and a **verification condition generator**.

The **predicate transformer** of a macro maps a *precondition* to a *postcondition*. Both sorts of conditions are logical predicates over Bedrock IL machine states; that is, they can be represented as functions typed like `state` → `Prop`, returning logical propositions given machine states as input. The precondition describes the machine state on entry to a statement, and the postcondition describes the machine state when control exits the statement. The postcondition is computed as a function, or *transformation*, of the precondition, leading to the name *predicate transformer*. At a high level, the predicate transformer of a macro answers the question *what does this macro do?*

A macro also includes a **verification condition generator**, which given a precondition outputs a logical formula (a verification condition) that the programmer must prove. If the verification condition is false, the statement may not work as expected. At a high level, the verification condition generator answers the question *which uses of this macro are safe and functionally correct?*

More formally, we define the verification part of a macro with:

$$\begin{array}{rrcl}
\text{Predicates} & \mathbb{P} & = & \mathsf{state} \to \mathsf{Prop} \\
\text{Predicate transformers} & \mathbb{T} & = & \mathbb{P} \to \mathbb{P} \\
\text{Verification condition generators} & \mathbb{C} & = & \mathbb{P} \to \mathsf{Prop} \\
\text{Verification parts of macros} & \mathbb{V} & = & \mathbb{T} \times \mathbb{C}
\end{array}$$

At a high level, the intent of these definitions is as follows: Consider a macro definition $M(x_1, \ldots, x_n) = E$, expanding an M invocation into some more primitive statement E. Let the associated predicate transformer be $T \in \mathbb{T}$ and the verification condition generator be $C \in \mathbb{C}$. Client code sees the following Hoare-logic proof rule for a macro invocation with a particular precondition $P \in \mathbb{P}$:

$$\{P \land C(P)\}M(e_1, \ldots, e_n)\{T(P)\}$$

The formal version of this notation is complicated by the need to deal with *sets of IL basic blocks*, rather than the usual structured statements. To give an intuition for those details, we return to extended versions of our previous example macros. First, consider the straightline code macro with instruction parameter i. When called with precondition P, the macro produces:

- **Verification condition:** $\forall s.\ P(s) \Rightarrow \exists s'.\ s \xrightarrow{i} s'$
- **Postcondition:** $\lambda s.\ \exists s'.\ P(s') \land s' \xrightarrow{i} s$

Preconditions and postconditions are predicates, or functions from machine states to logical propositions, so we often notate them as anonymous λ functions. The judgment $s \xrightarrow{i} s'$ asserts that running instruction i transforms state s into state s'. Instruction executions that would violate Bedrock IL safety do not belong to this relation.

Thus, the verification condition expresses the idea that the precondition implies safety of executing the instruction i. The postcondition is actually the *strongest postcondition* for this particular macro, denoting exactly the set of states that could be reached by running i in a state satisfying P. In general, macros need to expose *sound* postconditions, which include all possible states upon statement exit, but macros need not always return *strongest* postconditions. Indeed, the chance to return weaker postconditions is essential for supporting separation of interface and implementation in macros, where some details of macro implementation are hidden in specifications that are easier to reason about than strongest postconditions.

Consider now the if..then..else macro, where the parameters are a conditional test e, a *then* statement with predicate transformer T_1 and verification condition generator C_1, and an *else* statement with predicate transformer T_0 and verification condition generator C_0. If the overall precondition is P, the macro produces:

- **Verification condition:** $\left(\bigwedge_{b \in \{0,1\}} C_b(\lambda s.\ P(s) \land s \xrightarrow{e} b)\right) \land (\forall s.\ P(s) \Rightarrow \exists b.\ s \xrightarrow{e} b)$
- **Postcondition:** $\lambda s'.\ \bigvee_{b \in \{0,1\}} T_b(\lambda s.\ P(s) \land s \xrightarrow{e} b)(s')$

We write $s \xrightarrow{e} b$ to indicate that conditional test e evaluates to Boolean value $b \in \{0, 1\}$ in state s. The overall verification condition is the conjunction of the two sub-statements' verification conditions, plus the requirement that the precondition imply safety of evaluating the conditional test. The overall postcondition is a disjunction of the sub-statements' postconditions. In each case, we pass off extended preconditions that combine the original precondition P with information on the outcome of the test e.

We have decided that a macro constitutes a *parsing rule*, a *compilation rule*, a *predicate transformer*, and a *verification condition generator*. The last crucial ingredient is a *proof that all of the above agree with each other in an appropriately sound way*. That is, we package macros with *dependently typed records* that contain not just functions but also machine-checked proofs about those functions. Such a thing is easy to do in Coq, as shown by our definition of macros below. (These definitions actually occur in a Coq *section* that introduces parameters describing the code's assumptions about its own module and others, but we omit such details here to simplify the presentation.)

Note that this code incorporates a few optimizations, including representing verification conditions as *lists of propositions* instead of single propositions, to allow conjunction of verification conditions without creating arbitrarily bushy conjunction trees; and *staging* of macros in a manner similar to currying, so that programs may be verified without the need to generate code for them. We write `Prop` for the type of logical propositions, or standalone facts; and `assert` for the type of predicates over machine states, or functions from states to `Prop`.

```
(* Delayed code generation part of macro output *)
Record codeGen (Precondition : assert) (Base Exit : N)
      (Postcondition : assert) (VerifCond : list Prop) := {
  Entry : N;
  Blocks : list (assert * block);

  PreconditionOk : ∃ bl, nth_error Blocks (nat_of_N Entry)
    = Some (Precondition, bl);

  BlocksOk : vcs VerifCond
    → Exit < Base
    → List.Forall (fun p ⇒ blockOk
      (imps Blocks Base Exit Postcondition) (fst p) (snd p))
     Blocks
}.

(* Immediate verification part of macro output *)
Record codeOut (Precondition : assert) := {
  Postcondition : assert;
  VerifCond : list Prop;
  Generate : ∀ Base Exit : N,
    codeGen Precondition Base Exit Postcondition VerifCond
}.

(* Overall type of a macro, a dependent function type *)
Definition cmd := ∀ cin : assert, codeOut cin.
```

We will explain these definitions in reverse. A macro, of type `cmd`, has a *dependent function type*, where the return type of the function is allowed to mention the *value* of the actual parameter. In this case, the formal parameter `cin` is the input precondition, and we return a value of a type `codeOut cin` specialized to the precondition. Where our earlier less formal definitions treated predicate transformers and verification condition generators as independent functions over preconditions, here we instead make the whole macro a function on preconditions.

A `codeOut` record contains the computed postcondition and verification condition, as well as `Generate`, a function that we call if we wish to do code generation, too.

A `codeGen` record contains the output of code generation. While our earlier sketches refer to generation of new code blocks as an imperative process, we implement generation in Coq via returning lists of *new* blocks only. Furthermore, a `Generate` function is passed not only the `Exit` label for a statement, but also `Base`, the address that will be given to the first new code block returned by this macro. In general, the nth block in the output will have label `Base` $+\ n$. It is important to fix the labeling convention to allow the macro to create internal jumps in its code blocks.

Specifically, the `codeGen` record includes the `Entry` label of the statement and the list of its new `Blocks`. Additionally, proofs of two *theorems* must be packaged in the same dependent record. The `PreconditionOk` theorem asserts that the requested precondition really has been assigned as the spec of the statement's entry block. The `BlocksOk` theorem asserts that every block in `Blocks` is correct according to the rules of XCAP.

This latter statement uses a predicate `blockOk` that can be thought of as a judgment $\Gamma \vdash B$, asserting the correctness of basic block B under the assumptions in Γ, a finite map from code labels to assumed preconditions for their associated blocks. Here we build Γ with a function `imps`, which constructs a finite map containing (a) the `Exit` label with spec `Postcondition`, (b) the macro's `Blocks` themselves with their associated specs, and (c) any label-spec pairs for other modules that are listed explicitly as imports. We also note that `BlocksOk` takes as premises (a) the truth of the verification condition and (b) an assertion that the `Exit` label is less than the `Base` label, so that the macro knows it will not accidentally generate a block with the same address as the exit block it should jump to.

The new components of certified macros are very similar to conventional logical manipulations in program verification tools and verified compilers. The power of the idea comes from formalizing the requirements on statements with a precise, open interface. We have proved a Coq theorem that any statement satisfying the `cmd` interface is compiled into a valid XCAP module, when its verification condition holds and it is started in a state satisfying the claimed precondition. This notion of module validity follows XCAP's modular verification approach, so that it is possible to link such theorems together to produce theorems about larger composite modules, without revisiting module internals.

We close out this section with more examples of combinators for conventional C-like control constructs, before turning in the following sections to local variables, calling conventions, and higher-level notations not commonly supported in C-like languages.

3.3 More Examples

Certified low-level macros are compatible with classic Hoare-logic proof rules that require the programmer to provide *invariants*. For instance, consider a macro for a while loop, with conditional test e, *loop invariant* I, and a body statement that has predicate transformer T and verification condition generator C. The macro's logical outputs then mirror standard proof rules, where P again stands for the precondition passed into the macro.

- **Verification condition:** $C(\lambda s.\ I(s) \wedge s \xrightarrow{e} 1)$
  $\wedge\ (\forall s.\ P(s) \Rightarrow I(s))$
  $\wedge\ (\forall s.\ I(s) \Rightarrow \exists b.\ s \xrightarrow{e} b)$
  $\wedge\ (\forall s.\ T(\lambda s'.\ I(s') \wedge s' \xrightarrow{e} 1)(s) \Rightarrow I(s))$
- **Postcondition:** $\lambda s.\ I(s) \wedge s \xrightarrow{e} 0$

The respective verification conditions are the conditions of the loop body, an implication from loop precondition to loop invariant, an implication from loop invariant to safety of conditional test, and an implication from loop body postcondition to loop invariant. The overall loop postcondition is the conjunction of the loop invariant with the failure of the conditional test.

It is also possible to write macros for function calls, where the crucial challenge is to reason about the passing of a return pointer as first-class data. The macro must allocate a new block whose address will be passed as return pointer. We follow the convention that a function receives its return pointer in Bedrock IL register Rp. The compilation part of a function-call macro may simply use its own exit label as the return pointer.

Since there is no succinct logical counterpart to $\xrightarrow{i}$ for expressing the effect of a whole function call, we ask the programmer to provide an invariant I, characterizing the machine state after the function call returns. To state the logical outputs of the function-call macro, we use the XCAP notation $\{P\}p$ to indicate that machine word p is the address of a basic block whose spec is implied by condition P. As is usual at the assembly level, our programs are effectively in continuation-passing style, so it is natural to reason about first-class return pointers with *Hoare doubles* (preconditions only) rather than the more common *Hoare triples* (preconditions and postconditions).

As usual, below let P stand for the precondition passed into the function-call macro. Additionally, let Q be the spec of the function being called, which is in precondition-only form just as for return pointers.

- **Verification condition:**
  $\forall s, p, s'.\ P(s) \wedge s \xrightarrow{\mathsf{Rp} \leftarrow p} s' \wedge \{I\}p \Rightarrow Q(s')$
- **Postcondition:** I

The verification condition is stated in terms of the assignment of an arbitrary code pointer p to the return pointer register Rp, subject to the constraint that invariant I is a valid spec for p's basic block. The invariant I itself is used as the call's postcondition.

A complete calling convention also deals with function arguments and local variables. Further, the reader may be worried about the readability of complex specs in continuation-passing style with Hoare doubles. The next section addresses both concerns, describing how we use more macros and other notational conventions to present a more normal perspective on C-like functions.

4. Local Variables and Calling Conventions

Figure 2 contained a specification in a conventional precondition-and-postcondition style, hiding the complexities of continuation-passing-style reasoning. As another example, here is Coq code for a spec we might write for an in-place linked list reversal function. The predicate `sll` stands for "singly linked list," and its parameters are a functional list and a pointer to the root of a matching linked list in memory.

```
SPEC("l") reserving 5
  Al L,
  PRE[V] sll L (V "l")
  POST[R] sll (rev L) R
```

In order, the four lines of such a spec may be thought of as:

1. A *header* giving the function's formal arguments, plus how many free additional stack slots it requires (`reserving` clause)
2. Universal quantifiers for *specification variables* (similar to *ghost variables*) that may be mentioned in both precondition and postcondition
3. A *precondition*, introducing a local name `V` for a function assigning values to actual arguments
4. A *postcondition*, introducing a local name `R` for the function return value

A more compact notation, closer to usual Hoare logic conventions but with explicit use of functions to handle binding of `V` and `R`, might be:

$$\forall L.\ \{\lambda V.\ \mathsf{sll}(L, V(l))\}f(l, 5)\{\lambda V, R.\ \mathsf{sll}(\mathsf{rev}(L), R)\}$$

A mechanical translation produces a continuation-passing style version of this spec, where we only need to assign a precondition. As is usual with this sort of translation, *universal* quantifiers scoped over both precondition and postcondition turn into *existential* quantifiers appearing within the precondition; and calling-convention registers appear explicitly, where Rp holds the return pointer and Rv the function return value.

$$\{\exists L, V.\ \mathsf{sll}(L, V(l)) \land \{\mathsf{sll}(\mathsf{rev}(L), \mathsf{Rv})\}\mathsf{Rp}\}f(l, 5)$$

In this specification, the call stack is still treated implicitly. We must eventually be explicit about *call stack as data structure* to bridge the semantic gap to the Bedrock IL. Here is the final version of the specification, using a predicate locals to represent a call stack frame. Register Sp holds the stack pointer.

$$\begin{aligned}&\{\exists L, V.\ \mathsf{locals}(\{l\}, V, 5, \mathsf{Sp}) * \mathsf{sll}(L, V(l))\\ &\quad\land \{\exists V'.\ \mathsf{locals}(\{l\}, V', 5, \mathsf{Sp}) * \mathsf{sll}(\mathsf{rev}(L), \mathsf{Rv})\}\mathsf{Rp}\}f\end{aligned}$$

Here locals is just another abstract predicate [27] like sll. A fact $\mathsf{locals}(D, V, n, v)$ says that a stack frame with domain D and variable values V is located at address v, and it has at least n more free stack slots remaining. The overall list-reverse spec above indicates that the stack frame must be present with some variable values V at call time, and the separating conjunction $*$ expresses that the call stack and the input linked list must occupy disjoint memory. A nested Hoare double ascribes a precondition to the return pointer Rp. This "postcondition" is the same as the function "precondition" except that (1) local variable values are allowed to change to some arbitrary V' and (2) the linked list has been reversed.

In general, a traditional function spec looks like $\forall \vec{x}.\ \{P\}f(\vec{y}, n)\{Q\}$, where P is a function over V, and Q is a function over V and R. The desugaring of this spec into a low-level XCAP precondition is:

$$\begin{aligned}&\{\exists \vec{x}, V.\ \mathsf{locals}(\vec{y}, V, n, \mathsf{Sp}) * P(V)\\ &\quad\land \{\exists V'.\ \mathsf{locals}(\vec{y}, V', n, \mathsf{Sp}) * Q(V, \mathsf{Rv})\}\mathsf{Rp}\}f\end{aligned}$$

To do code generation and computation of verification conditions and postconditions, a macro now needs to know the local variable set $\vec{y}$ and the reserved stack slot count n. Therefore, we add these two values as additional parameters to each macro. We do not update any of the logical components of macros; local variables are instead handled in a more traditional macro expansion style, without any new hiding of implementation from interface. For instance, a statement that reads a local variable will be compiled to refer directly to the concrete offset of that variable from the stack pointer, with verification conditions that reveal the compilation details directly. We have extended Bedrock's separation logic proof automation to reason properly about the interaction of such stack pointer indirection and the locals predicate.

5. Higher-Level Notations

Certified low-level macros do not merely encode strongest postcondition calculators, to declare the effects of statements. Instead, macro authors have the freedom to expose weaker postconditions, just as the author of a conventional function may choose a weaker postcondition to enforce data abstraction or another form of information hiding. Conventional Hoare logics have a *rule of consequence* facilitating this sort of hiding in a straightforward way, where one may conclude $\{P\}c\{Q\}$ from $\{P'\}c\{Q'\}$, given $P \Rightarrow P'$ and $Q' \Rightarrow Q$.

We have implemented a simple *wrapper* combinator for macros, embodying the rule of consequence. A programmer uses preexisting macros to implement a chunk of code, taking advantage of the associated proof automation to simplify verification of that chunk. Then the chunk may be *wrapped* with a new predicate transformer and verification condition generator, if appropriate conditions are proved. In particular, say that the original statement has transformer T' and generator C', and the wrapper is introducing new transformer T and generator C. The conditions that make the wrapping sound are:

- $\forall P.\ C(P) \Rightarrow C'(P)$
- $\forall P, s.\ C(P) \land T'(P)(s) \Rightarrow T(P)(s)$

These conditions are a direct reinterpretation of the rule of consequence, and they are feasible to prove for a variety of higher-level notations that construct the underlying statements programmatically, via recursive functions over macro parameters. In the rest of this section, we give a motivating example for two higher-level macros and then describe their implementations.

5.1 A Motivating Example

Figure 7 shows an example program that uses higher-level macros. The program is a simple idealization of a network server processing queries over a database. We deal not with network IO, but rather just with decoding and processing requests and encoding the results in memory. We say that the database is an array of machine integers, as is the packet of encoded query requests. The output of the server is stored in a linked list.

The lefthand column of the figure gives the program implementation. Our program declares a module `m`, which imports function `malloc` from module `malloc`, with *specification* `mallocS`. Next, we define function `main`, beginning with its list of local variables, including formal arguments.

The main body of the function is a loop over all requests contained in the request array `cmd`. We take advantage of two high-level macros to make our implementation more declarative.

First, we use a *parsing* syntax similar to ML-style pattern matching, via the `Match` macro. Its arguments name an array, its length (`Size` clause), and a local variable storing our current index within the array (`Position` clause). A `Match` has a sequence of cases consisting of patterns and bodies. A pattern is a sequence of items that are either constants, to indicate that a specific machine word value must be found in the corresponding array position; or a variable name (string literal), to indicate that any machine word may appear in the position. All variables are assigned the appropriate values from the array before executing the body of a matching pattern.

The other high-level macro is for *declarative querying* of arrays, chosen as a simple example illustrating many of the same challenges applicable to more expressive SQL-style querying. The `For` macro takes as arguments a loop variable to store an array index, another variable to store the value found at that index (`Holding` clause), the array to query (`in` clause), the length of that array (`Size` clause), and a filter condition choosing which array cells to

Figure 7. Implementation, specification, and verification of our main case study program

```
(* Program implementation *)
Definition m := bimport [[ "malloc"!"malloc" @ [mallocS] ]]
  bmodule "m" {{
    bfunction "main"("cmd", "cmdLen", "data", "dataLen",
                     "output", "position", "posn", "lower", "upper",
                     "index", "value", "res", "node") [mainS]
      "output" ← 0;;
      "position" ← 0;;
      [(* ... invariant omitted ... *)]
      While ("position" < "cmdLen") {
        Match "cmd" Size "cmdLen" Position "position" {
          (* ValueIsGe *)
          Case (0 ++ "posn" ++ "lower")
            "res" ← 0;;
            [(* ... invariant omitted ... *)]
            For "index" Holding "value" in "data"
              Size "dataLen"
              Where ((Index = "posn") && (Value ≥ "lower")) {
              "res" ← 1
            };;
            "node" ← Call "malloc"!"malloc"(0)
            [(* ... invariant omitted ... *)];;
            "node" ∗← "res";;
            "node" + 4 ∗← "output";;
            "output" ← "node"
          end;;

          (* MaxInRange *)
          Case (1 ++ "lower" ++ "upper")
            "res" ← 0;;
            [(* ... invariant omitted ... *)]
            For "index" Holding "value" in "data"
              Size "dataLen"
              Where (("lower" ≤ Value) && (Value ≤ "upper")
                     && (Value ≥ "res")) {
              "res" ← "value"
            };;
            "node" ← Call "malloc"!"malloc"(0)
            [(* ... invariant omitted ... *)];;
            "node" ∗← "res";;
            "node" + 4 ∗← "output";;
            "output" ← "node"
          end;;

          (* CollectBelow *)
          Case (2 ++ "lower" ++ "upper")
            [(* ... invariant omitted ... *)]
            For "index" Holding "value" in "data"
              Size "dataLen"
              Where ((Index ≥ "lower")
                     && (Value ≤ "upper")) {
              "node" ← Call "malloc"!"malloc"(0)
              [(* ... invariant omitted ... *)];;
              "node" ∗← "value";;
              "node" + 4 ∗← "output";;
              "output" ← "node"
            }
          end
        } Default {
          Fail (* Impossible: the match was exhaustive. *)
        }
      };;

      Return "output"
    end
  }}.
```

```
(* Requests that the server may receive.
 * (W is the type of machine words.) *)
Inductive req :=
| ValueIsGe (index valueLowerBound : W)
(* Test if the value stored at this array index is
 * at least as large as this value. *)

| MaxInRange (lowerBound upperBound : W)
(* Find the maximum value within the given
 * range of values. *)

| CollectBelow (indexLowerBound valueUpperBound : W)
(* Collect a list of all values satisfying
 * the given bounds on index and value. *).

(* Representation of requests as lists of words. *)
Definition encode (r : req) : list W :=
  match r with
    | ValueIsGe a b ⇒ $0 :: a :: b :: nil
    | MaxInRange a b ⇒ $1 :: a :: b :: nil
    | CollectBelow a b ⇒ $2 :: a :: b :: nil
  end.

(* Representation of sequences of requests *)
Fixpoint encodeAll (rs : list req) : list W :=
  match rs with
    | nil ⇒ nil
    | r :: rs' ⇒ encode r ++ encodeAll rs'
  end.

(* Omitted here: some helper functions *)

(* Compute the proper response to a request,
 * as transformation on a list of output values. *)
Definition response (data acc : list W) (r : req) :  list W :=
  match r with
    | ValueIsGe index lower ⇒
      valueIsGe data index lower :: acc
    | MaxInRange lower upper ⇒
      maxInRange lower upper data :: acc
    | CollectBelow lower upper ⇒
      collectBelow upper (skipn (wordToNat lower) data) acc
  end.

(* Proper response to a request sequence *)
Definition responseAll (data : list W) (rs : list req)
                       (acc :  list W) :  list W :=
  fold_left (response data) rs acc.

(* Specification of the server main() function *)
Definition mainS := SPEC("cmd", "cmdLen", "data", "dataLen")
  reserving 15 Al r, Al d,
  PRE[V] array (encodeAll r) (V "cmd") ∗ array d (V "data")
    ∗ mallocHeap
    ∗ ⌈ V "cmdLen" = length (encodeAll r) ⌉
    ∗ ⌈ V "dataLen" = length d ⌉
    ∗ ⌈ goodSize (length (encodeAll r) + 3) ⌉
  POST[R] array (encodeAll r) (V "cmd") ∗ array d (V "data")
    ∗ mallocHeap ∗ sll (responseAll d r nil) R.

(* Omitted: lemmas and tactics about lists *)

(* Correctness theorem *)
Theorem ok : moduleOk m.
Proof.
  vcgen; abstract (parse0; for0; post; evaluate hints;
    repeat (parse1 finish; use_match);
    multi_ex; sep hints; finish).
Qed.
```

visit in the loop (Where clause). The key property of For is that *it performs compile-time analysis of the* Where *condition to optimize query execution*. If the Where condition implies a known value for the array index, the "loop" only visits that cell. If the condition implies a lower bound on array indices, the loop skips the earlier indices. *The proof rule of the* For *macro hides the details of which optimizations are used*, allowing verification to proceed only in terms of high-level logical concepts.

The righthand column of Figure 7 excerpts the specification and correctness proof for the program. The specification follows the Bedrock style of *using pure functional programs as specifications for low-level programs*. In particular, we define functional datatypes to represent requests to the server, and we write functional programs to map requests to their network encodings and their appropriate responses. The three different kinds of queries are represented with capitalized datatype constructors (e.g., ValueIsGe) and specified with pure mathematical functions in lowercase (e.g., valueIsGe).

The specification mainS applies to our main function, giving its list of formal arguments, the arrays cmd and data plus their lengths. Two universal quantifiers ("Al") bind names to the purely functional versions of cmd and data, respectively.

The precondition (PRE) clause binds a local function V that may be used to access the actual parameter values (e.g., V "cmd" as the value of cmd). The first three $*$-separated formulas here say, respectively, that there are regions with an array pointed to by parameter cmd and storing an encoding of the queries, an array pointed to by parameter data and storing an encoding of the database, and finally the internal data structures of the malloc library. Further subformulas use the lifting operator $\lceil\ \rceil$, for lifting normal Coq propositions into separation logic formulas. We require that cmdLen and dataLen store the lengths of the proper arrays, and we require that adding 3 to the query array length does not overflow a 32-bit word (which the goodSize predicate formalizes).

A postcondition (POST) binds the local variable R to stand for the return value. Here we assert that the same two arrays remain in memory unchanged, the malloc library state is intact, and the return value is the root of a linked list.

Finally, the program is proved to meet its spec with a short sequences of Coq tactics. Some of them, like vcgen and sep, come from the Bedrock library. Others, like parse0 and for0, come from the libraries providing the Match and For macros. In this way, macro authors can also provide reusable procedures for discharging the sorts of verification conditions that their macros generate, while leaving the programmer full flexibility to apply other proof techniques instead.

Some details are omitted in the figure. The main implementation includes 7 invariants, predicates over intermediate program states that guide verification condition generation. Together these invariants take up 72 lines of code. We also omit 350 lines of lemma statements and proof script, setting up the key arguments behind the program's correctness. Crucially, *these lemmas are only about properties of lists*, our functional models of arrays and imperative linked lists; we need do no manual proving work specific to program syntax, program states, or memories.

This program uses at least 7 different macros, at varying levels of abstraction from statement sequencing to declarative querying, but the verification is independent of macro implementation details. The level of proof automation used here also means that small changes to the program often require little manual adaptation of proofs. Nonetheless, in the end we get a *foundational* theorem, stated in terms of a simple assembly-level operational semantics and with a proof checked by the normal Coq proof checker. Furthermore, the example program runs only about 25% more slowly than a conventional C program for the same task, as compared to an OCaml implementation we built at a similar level of abstraction, which runs in about 400% of the time of the C program. We return to performance details in Section 6.

5.2 Parsing Arrays of Machine Words

Consider now the general interface of the parsing macro from Figure 7. Its simplest form is as follows:

```
Match array Size size Position pos {
  Case pattern bodyStmt end
} Default { defaultStmt }
```

Here the challenge is giving the macro a strong enough interface without revealing its internal workings. We intentionally set out to hide the details of array access, but programmers should still be able to prove reasonable correctness theorems about programs that use this macro. One concrete challenge is which precondition we choose as the input to *bodyStmt*, a sub-statement that is run when pattern-matching succeeds. We do not want to mention the exact sequence of instructions executed between the start of the Match and that point. If we did, the programmer would need to reason about memory safety and semantics of a sequence of array accesses, on *every* invocation of the macro.

The alternate approach that we take here is to construct a *bodyStmt* precondition *in terms of a simpler instruction sequence, different from the one actually executed, but with the same effect*. The instruction sequence is written in terms of the *functional model* of the array, a Coq list.

For instance, the pattern 0 ++ "posn" ++ "lower" from Figure 7 matches an array span starting with constant 0 and followed by any two other values, such that those two values are bound to names posn and lower before running *bodyStmt*. Let L be a mathematical list asserted to describe the contents of the array. The semantics of the example match may now be expressed with, first, an assertion that $L = a, b, c, \ldots$ for fresh variables a, b, and c; and second, this instruction sequence:

$$\mathsf{assume}(a = 0); \mathtt{posn} \leftarrow b; \mathtt{lower} \leftarrow c$$

We express exactly the intent of the pattern, without exposing any details of array access. Instruction sequences like the above are handled trivially by Bedrock's proof automation, without imposing memory safety proof obligations on the programmer. It is easy to code a recursive Coq function to translate a pattern into such a sequence of instructions.

The postcondition of the Match macro is just the disjunction of the postconditions for *bodyStmt* and *defaultStmt*, when they are passed preconditions constructed using the encoding technique above. The verification conditions include some administrative facts about variables: all program variables mentioned are declared as local variables, and the variables *array* and *pos* are not also used within patterns. The verification conditions of *bodyStmt* and *defaultStmt* are also inherited. Finally, one new verification condition asserts that the overall precondition for the Match implies that there exists an array at the address specified by *array*, where *pos* is a valid index into that array, and adding the length of the pattern to *pos* does not overflow a 32-bit word.

5.3 Declarative Querying of Arrays

Figure 7 demonstrated the For macro, which loops over exactly those cells of an array that satisfy a filter condition. The macro implementation analyzes the syntax of the condition to perform optimizations. Such a process is a simplified version of what goes on in an SQL query planner. The interface of the For macro should hide the details of the optimizations that are performed.

Our complete implementation uses a form of loop invariant that is similar to the one introduced by Tuerk [31], where an invariant includes both a precondition and a postcondition to simplify application of the separation logic frame rule. However, we will present here a simplified version with only a conventional loop invariant. There the general form of a `For` invocation is:

```
[After prefix PRE[V] invariant]
For index Holding value in array Size len Where condition
  { bodyStmt }
```

The *condition* may refer both to local variables and to the special variables `Index` and `Value` for the current array index and cell value, respectively. Our current implementation performs two optimizations: When *condition* is a conjunction where one conjunct is `Index` = i, then the "loop" need only visit cell i. When *condition* is a conjunction where one conjunct is `Index` $\geq i$, then the loop may begin at i rather than 0. The optimizer correctly handles cases where i is a local variable.

As in the previous subsection, here we have the challenge of assigning `For` a specification that does not reveal too much about its inner workings. Ideally, programmers who use `For` will only be faced with proof obligations about mathematical lists, not arrays or the details of compiling complex Boolean tests. This time, we use the conventional idea of a *loop invariant*, but modified to expose a list-level view of the action.

The schematic `For` invocation above includes a loop invariant, where an `After` clause introduces a local name *prefix* for the list of array cell values that have already been visited. In the course of a naive complete loop through the array, *prefix* progresses from the empty list to the full value list. However, optimizations may lead to some loop iterations being skipped, though such details should be hidden in the interface of `For`.

We are now ready to sketch what that interface is. The postcondition is simply that there exists an array in memory pointed to by *array*, such that the loop invariant holds for the full array contents. Verification conditions include both administrative requirements on variable names and the inherited verification condition of *bodyStmt*. The more interesting conditions are:

1. The loop invariant is independent of the values of *index* and *value*, which will often be changed without calling the loop body to *fast forward* past indices that the optimizer determines cannot match the filter condition. (Note that the invariant may still depend on the *functional list* of array cells already visited, just not on the values of local variables used incidentally to track progress in visiting cells.)
2. The precondition implies the loop invariant.
3. When the loop invariant holds for an array value prefix L, and when the filter condition will reject the next value if it equals v, then the loop invariant also holds for prefix L, v. This condition is crucial to justify the soundness of fast forwarding.
4. The postcondition of *bodyStmt* implies the loop invariant, where *bodyStmt* is passed a suitably high-level precondition expressed in terms of mathematical lists and the loop invariant.

6. Evaluation

Together, the core macro system definitions and all of the macros we have built on top of them take up about 4000 lines of Coq code within the Bedrock library.

We have built a few tactics to lower the cost of macro implementation, but our primary interest has been in the effort required to implement and verify individual programs, like the example of Figure 7. An important part of evaluation is the run-time performance of programs, since we motivated certified low-level macros with the possibility to combine the high performance of low-level languages with the abstractions of high-level languages. To gather some rough evidence that our implementation succeeds in this respect, we ran an experiment with four different implementations of the same "server" example[1]:

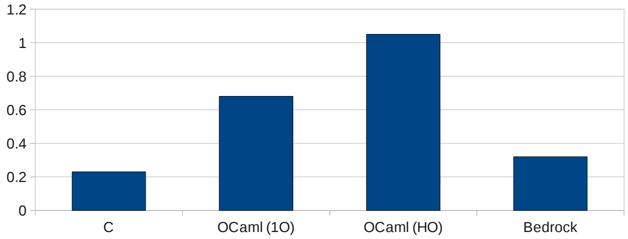

Figure 8. Running time (in seconds) of the four different implementations of the server example, running the same randomly generated workload of 200 queries on an array of 100,000 elements

1. A conventional C program (50 lines) not taking advantage of metaprogramming. This is our baseline of high performance.
2. A first-order (later abbreviated 1O) OCaml program (36 lines), with no variant types (beyond linked lists) or first-class functions. This is our baseline of the cost of switching to a popular high-level language.
3. A higher-order (later abbreviated HO) OCaml program (106 lines, including reusable library code), taking advantage of idioms like parser combinators and embedded interpreters for ASTs. This version illustrates the cost of employing nice abstractions that can be type checked in isolation, which is not the case for abstractions embodied as Camlp4 macros.
4. The Bedrock implementation from Figure 7 (50 lines, compiled to 561 lines of assembly with the Bedrock `malloc` library linked in), which is the only one of the implementations that we verified as correct.

Figure 8 shows the results of running all four compiled programs on a particular random workload chosen to be large enough for performance differences to show. These experiments were run on a 2.4 GHz Intel Xeon processor with 12 GB of RAM, using GCC 4.4.5 and OCaml 3.12.1. They show the first-order OCaml program running in roughly 300% of the time of the C program, with the higher-order OCaml program running in roughly 500% of the C program's time. In contrast, the Bedrock program runs in about 125% of the time of the C program. We replicated the experiment on a 4.2 GHz AMD FX processor with 16 GB of RAM, where the performance difference between C and Bedrock was within the granularity of measurement, and the first- and higher-order OCaml programs ran respectively in about 300% and 350% of the time for the C and Bedrock programs. It is worth pointing out that Bedrock is the only language implementation used here that does not yet have an optimizer, whereas we used the default optimization options of each other compiler, `gcc` and the `ocamlopt` native-code OCaml compiler.

We experimented with verifying programs for several months before generating any executable assembly code. The first program we actually executed was iterative factorial, and the second was the server program in Figure 7. Both worked correctly on the first try, providing at least some small testament to the debugging power of our verification environment.

We have verified a few other substantial case studies, as detailed in Figure 9, breaking down their lines of code into relevant categories. The case studies include some classic functions over imper-

[1] In directory `examples/comparison` of the Bedrock distribution

File	Program	Invar.	Tactics	Other	Overh.
Server	50	79	167	239	9.7
LinkedList	42	26	27	31	2.0
Malloc	43	16	112	94	5.2
ListSet	50	31	23	46	2.0
TreeSet	108	40	25	45	1.0
Queue	53	22	80	93	3.7
Memoize	26	13	56	50	4.6

Figure 9. Case study verifications, with data on annotation burden, in lines of code

ative linked lists, the `malloc` library, implementations of a single finite set abstract data type interface with both unsorted lists and binary search trees, an implementation of the bag abstract data type using a queue, and an implementation of an abstract data type of memoized functions, where a closure stored in memory contains both a function pointer and an associated 1-element memo table. All case studies are built up from certified low-level macros, where our running example Server uses the most advanced macros.

In Figure 9, the *program* column counts lines of executable code; *invar.* refers to function specs, loop invariants, and post-function call invariants; *tactics* refers to literal proofs of theorems as well as reusable tactic functions and hints; and *other* collects the remaining lines, which are mostly helper function definitions and lemma statements. We omit lines of code that are duplicated between Coq modules and their interfaces.

The final column of Figure 9 gives the verification *overhead*, or ratio of verification code to executable code. The more basic examples range from overheads of 1.0 to 5.2, which compare reasonably well to common ratios in Coq developments, such as the overhead of about 7 in the original CompCert paper [21]. The overhead is slightly below 10 for our server example, which involves a fair amount of program-specific correctness reasoning.

We should also point out that the code size statistics from the original Bedrock paper [8] can also be taken as evidence for the effectiveness of the macro system presented in this paper, which is a moderate evolution of the macro system used previously and never presented in detail before.

The main weakness of our present implementation is the running time of automated proof search and proof checking. On the machines used in the experiments, the server example runs for about an hour finding and checking proofs. The other case studies take less time, but still have running times measured in minutes, at best. We do not view this as a huge obstacle from a scientific perspective, as much opportunity remains to take advantage of parallelism. The Server example generates about 100 independent verification conditions, each of which could be discharged on a different processor core, dropping the total verification time for the example to a few minutes. We need only wait for Coq to support forking processes to handle disjoint subgoals of a single proof.

7. Related Work

The original Bedrock paper [8] used a simpler macro system but did not report on its details. Compared to that macro system, our new one from this paper is distinguished by including a calling convention supporting named local variables, as well as the implementation and verification of high-level macros like our parsing and querying examples.

Benton et al. [2] report on a system for formal verification of assembly programs in Coq using higher-order separation logic. Based on an intermediate language with scoped local labels, they define a few macros of moderate complexity, like while loop and function call (the latter handling only storing the return pointer to a register, not stack frame management). Proofs are largely manual with interspersed use of automation tactics, leading to about 10 lines of proof per line of assembly code, as opposed to an about even ratio between the two in our most complicated macro-based example. Their work explores an interesting alternate separation logic featuring a novel higher-order frame rule. We expect that an approach like ours in this paper could be adapted to the formalism of Benton et al. and many others, as our respective projects deal mostly with orthogonal concerns.

Work by Myreen et al. [24, 25] has demonstrated another mechanized separation logic suitable for reasoning about realistic assembly languages. The FLINT project has produced XCAP [26], the assembly-level program logic we adopt in this work, as well as several other logics targeting interesting aspects of assembly program behavior [5, 10–12, 23]. Other related work with mechanized separation logic has gone on at the level of C programs, as in Appel et al.'s Verified Software Toolchain [1]. Mechanized proofs in these logics have been long and manual, and previously no macro system had been demonstrated for them at the level of this paper's most sophisticated examples.

The L4.verified kernel verification project [20] employs a tool [15] for verified translation of C code into functional programs in higher-order logic, applying optimizations that choose simpler types for functions that use fewer varieties of side effect. In a sense, their tool reverses the action of the Bedrock macro system, providing similar foundational guarantees for the fixed C language. In contrast, our work makes it possible for programmers to add their own new language constructs, including higher-level macros like our declarative array querying that are unlikely to be reconstructible automatically from C code.

Macros have been most studied in the context of Lisp and Scheme. Notable in that area is work by Herman and Wand [17], which shows how to assign macros interfaces sufficient to guarantee *hygiene*, or lack of undesirable variable capture, without the need to expand macro definitions to check program validity. Our work also assigns interfaces to macros, for low-level rather than high-level code, and facilitating formal verification of functional correctness rather than just proper use of binding constructs.

In the functional programming world, the use of *embedded domain-specific languages* is already quite popular, for supporting safe code generation to low-level languages via high-level functional programs. For instance, the Harpy Haskell library [14] supports generation of x86 assembly code. Recent tools like Meta-Haskell [22] allow generation of programs in low-level languages like C, CUDA, and assembly, where Haskell-level type checking guarantees that code generators will never output type-incorrect programs. Our work in this paper can be viewed as starting from the same foundation of type-safe metaprogramming and expanding it to support verification of functional correctness.

Extensible C variants like xtc [16] and xoc [9] allow the programmer to specify new statement forms and static checking disciplines. These systems can provide much nicer syntax than what we have available within Coq, and there is room for such features to migrate to proof assistants. The key disadvantage of existing extensible compilers is that they provide no foundational guarantees; custom static analysis disciplines do not have clear semantics, so these languages are not compatible with foundational verification of functional correctness.

Safe "systems languages" include low-level languages like Cyclone [19] and various higher-level languages with features targeted at systems programming. Some are known for their use in particular operating systems projects, such as Modula-3 in SPIN [4] and Sing# in Singularity [18]. None of these languages are extensible, and none give foundational safety or functional correctness guarantees.

Tools like Smallfoot [3] and Space Invader [6, 32] apply separation logic to check memory safety of low-level programs automatically. Other systems, including TVLA [29] and XISA [7], can verify memory safety or even functional correctness using other formalisms for data structure invariants. These tools apply to fixed programming languages with no modular macro systems.

8. Conclusion

We have presented *certified low-level macros*, a technique supporting metaprogramming for low-level software, where programs may be verified formally without looking inside the definitions of macros. Rather, macros export formal interfaces that include predicate transformers and verification condition generators. We have demonstrated how to build up a C-like language stack, starting from a simple intermediate language and culminating in high-level notation for tasks like parsing and declarative querying that are usually supported via ad-hoc external tools. Concrete programs have mostly automated Coq proofs, generating foundational results with statements independent of our macro system; and the performance of our compiled programs is competitive with C programs.

One direction for future research is optimization of the programs that result from macro expansion. As these programs are in a form more like conventional assembly language than conventional compiler intermediate languages, it is not obvious how to do sound optimization. We plan to investigate how we might take advantage of the verified preconditions associated with all basic blocks, to recover semantic information and perhaps allow optimizations that are too difficult to implement with traditional dataflow analysis.

Acknowledgments

The author would like to thank Gregory Malecha and Thomas Braibant, for work on Bedrock proof automation that is used in the case studies from this paper; as well as Thomas Braibant, Ryan Kavanagh, Antonis Stampoulis, and Peng Wang, for their feedback on drafts of this paper.

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] A. W. Appel. Verified software toolchain. In *Proc. ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer-Verlag, 2011.

[2] N. Benton, J. B. Jensen, and A. Kennedy. High-level separation logic for low-level code. In *Proc. POPL*, pages 301–314. ACM, 2013.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2005.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. SOSP*, pages 267–283. ACM, 1995.

[5] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI*, pages 66–77. ACM, 2007.

[6] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proc. POPL*, pages 289–300. ACM, 2009.

[7] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Proc. POPL*, pages 247–260. ACM, 2008.

[8] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. PLDI*, pages 234–245. ACM, 2011.

[9] R. Cox, T. Bergan, A. T. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. ASPLOS*, pages 244–254. ACM, 2008.

[10] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP*, pages 254–267. ACM, 2005.

[11] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. PLDI*, pages 401–414. ACM, 2006.

[12] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. PLDI*, pages 170–182. ACM, 2008.

[13] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *Proc. ICFP*, pages 74–85. ACM, 2001.

[14] M. Grabmüller and D. Kleeblatt. Harpy: Run-time code generation in Haskell. In *Proc. Haskell Workshop*, page 94. ACM, 2007.

[15] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *Proc. ITP*, volume 7406 of *LNCS*, pages 99–115. Springer-Verlag, 2012.

[16] R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51. ACM, 2006.

[17] D. Herman and M. Wand. A theory of hygienic macros. In *Proc. ESOP*, volume 4960 of *LNCS*, pages 48–62. Springer-Verlag, 2008.

[18] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[19] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX ATC*, pages 275–288. USENIX Association, 2002.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, pages 207–220. ACM, 2009.

[21] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. ACM, 2006.

[22] G. Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *Proc. ICFP*, pages 311–322. ACM, 2012.

[23] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI*, pages 468–479. ACM, 2007.

[24] M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. POPL*, pages 107–118. ACM, 2010.

[25] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. TACAS*, volume 4424 of *LNCS*, pages 568–582. Springer-Verlag, 2007.

[26] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL*, pages 320–333. ACM, 2006.

[27] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. POPL*, pages 247–258. ACM, 2005.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, May 2002.

[30] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. PEPM*, pages 203–217. ACM, 1997.

[31] T. Tuerk. Local reasoning about while-loops. In *Proc. VSTTE Theory Workshop*, 2010.

[32] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proc. CAV*, volume 5123 of *LNCS*, pages 385–398. Springer-Verlag, 2008.