The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier

Citation
Adam Chlipala. 2013. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. SIGPLAN Not. 48, 9 (September 2013), 391-402.

As Published
http://dx.doi.org/10.1145/2500365.2500592

Publisher
Association for Computing Machinery (ACM)

Version
Author's final manuscript

Accessed
Fri Nov 23 03:07:52 EST 2018

Citable Link
http://hdl.handle.net/1721.1/86042

Terms of Use
Creative Commons Attribution-Noncommercial-Share Alike

Detailed Terms
http://creativecommons.org/licenses/by-nc-sa/4.0/
The Bedrock Structured Programming System
Combining Generative Metaprogramming and Hoare Logic in an Extensible Program
Verifier
Adam Chlipala
MIT CSAIL
adamic@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
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.

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 that are within the ray 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 $\ell_1$ in a state satisfying the spec of $\ell_1$, then if execution ever reaches code address $\ell_2$, the spec of $\ell_2$ 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 [23], 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

```plaintext
appendS : spec := SPEC("x", "y") reserving 2
   A1 ls1, A1 ls2.
   PRE[V] s1 ls1(V "x") = s1 ls2(V "y")
   POST[R] s1(1 ls1 ++ ls2).

bfunction "append"("x", "y", "z", "tmp") {appendS}
   if ("x" = 0) {
      Return "y"
   } else {
      "z"
      "tmp" ← "x" + 4;
       PRE[V] [ V "x" \not= S0 ] * [ V "tmp" = p1 ] * V "x" → x
       * (V "x" + ∆4) → p1 * s1 ls1 p1 * s1 ls2 (V "y")
       POST[R] [ R = V "x" ] * s1 (x :: ls1 ++ ls2)(V "x")
      While("tmp" ≠ 0) {
         "x" ← "tmp";
         "tmp" ← "x" + 4
      };
      "x" + 4 ← "y";
      Return "z"
   }
```

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.

**Input**
- Exit label (jump here when done, points to a previously written block)

**Side Effect**
- Write some new blocks to end of program.

**Output**
- Entry label (jump here to begin, points within newly written blocks)

**Macro**
- (standing for a statement)

---

**Figure 3.** The interface of a low-level macro

---

**Input**
- Exit label

**Side Effect**
- Straightline Code

**Parameter**
- Instruction i

**Entry label**

---

**Figure 4.** Sketch of macro for single straightline instructions

---

**Input**
- Entry labels (outputs)

**Side Effect**
- Then

**Output**
- Else

**Entry label**

---

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

---

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.

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. 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 verifica-
Consider a macro definition. At a high level, the verification condition generator 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 predicate transformer of a macro answers the question which uses of this macro are safe and functionally correct? 

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

\[
\text{VerifCond} : \lambda \text{cin} \to \text{Postcondition} \land \text{Precondition} \land \text{Blocks} = \text{Exit} \land \text{Exit} \land \text{Postcondition}.
\]

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 \(T \in \mathcal{T}\) and the verification condition generator be \(C \in \mathcal{C}\). Client code sees the following:

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

Preconditions and postconditions are predicates, or functions from machine states to logical propositions, so we often notate them as anonymous \(\lambda\) functions. The judgment \(s \xrightarrow{e} s'\) asserts that running instruction \(e\) 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 \(e\). The postcondition is actually the strongest postcondition for this particular macro, denoting exactly the set of states that could be reached by running \(e\) 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:** \(\left( \lambda b \in \{0,1\} \right) C_b(\lambda s. \text{P}(s) \land s \xrightarrow{e} b) \land (\forall s. \text{P}(s) \Rightarrow \exists b, s \xrightarrow{e} b)\)
- **Postcondition:** \(\lambda s'. \left( \bigvee b \in \{0,1\} T_b(\lambda s. \text{P}(s) \land s \xrightarrow{e} b)(s') \right)\)

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 independently 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 \text{Prop} for the type of logical propositions, or standalone facts; and \text{assert} for the type of predicates over machine states, or functions from states to \text{Prop}.

\[
(\text{assert}) \equiv \text{Prop} \\
(\text{asserts}) \equiv \text{Prop} \\
(\text{Precondition}) \equiv \text{Prop} \\
(\text{Postcondition}) \equiv \text{Prop} \\
(\text{Blocks}) \equiv \text{Prop} \\
(\text{Exit}) \equiv \text{Prop} \\
(\text{PostconditionOk}) \equiv \text{Prop} \\
(\text{BlocksOk}) \equiv \text{Prop} \\
(\text{Record}) \equiv \text{Prop} \\
\]

\[
\text{codeGen Precondition Base Exit Postcondition VerifCond} \\
\text{codeOut cin} \\
\]

\[
\text{cmd} := \text{codeOut cin} \\
\]

\[
\text{Postcondition} \equiv \text{Prop} \\
\text{VerifCond} \equiv \text{Prop} \\
\text{Generate} \equiv \text{Prop} \\
\text{cmd} := \text{codeOut cin} \\
\]
We will explain these definitions in reverse. A macro, of type \texttt{cmd}, has a \textit{dependent function type}, where the return type of the function is allowed to mention the \textit{value} of the actual parameter. In this case, the formal parameter \texttt{cin} is the input precondition, and \texttt{cmd} contains the output of code generation. Where our earlier 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 \texttt{codeOut} record contains the computed postcondition and verification condition, as well as \texttt{Generate}, a function that we call if we wish to do code generation, too.

A \texttt{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 \texttt{Generate} function is passed not only the \texttt{Exit} label for a statement, but also \texttt{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 \texttt{Base} + n. It is important to fix the labeling convention to allow the macro to create internal jumps in its code blocks.

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

This latter statement uses a predicate \texttt{BlocksOk} that can be thought of as a judgment \( \Gamma \vdash B \), asserting the correctness of basic block \( B \) under the assumptions in \( \Gamma \), a finite map from code labels to assumed preconditions for their associated blocks. Here we build \( \Gamma \) with a function \texttt{imps}, which constructs a finite map containing (a) the \texttt{Exit} label with spec \texttt{Postcondition}, (b) the macro’s \texttt{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 \texttt{BlocksOk} takes as premises (a) the truth of the verification condition and (b) an assertion that the \texttt{Exit} label is less than the \texttt{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 end 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 \textit{invariants}. For instance, consider a macro for a while loop, with conditional test \( e \), \textit{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**: \( C(\lambda s. I(s) \land s \xrightarrow{e} 1) \land (\forall s. P(s) \Rightarrow I(s)) \land (\forall s. I(s) \Rightarrow \exists b. s \xrightarrow{b} b) \land (\forall s. T(\lambda s'. I(s') \land s' \xrightarrow{e} 1)(s) \Rightarrow I(s)) \)

- **Postcondition**: \( \lambda s. I(s) \land s \xrightarrow{e} 0 \)

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 \( \xrightarrow{e} \) 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 \textit{Hoare doubles} (preconditions only) rather than the more common \textit{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**: \( \forall s, p, s', P(s) \land s \xrightarrow{e} s' \land \{I\}_p \Rightarrow Q(s') \)

- **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 \texttt{all} 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.

\begin{verbatim}
SPECT("1") reserving S
A1 L
PREV \{ all L \ (V "1") \}
PGST[R] \{ all (rev L) \ R
\end{verbatim}

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:

\[
\forall L. \{ \lambda V. \ sll(L, V(\ell)) \} f(l, 5) \{ \lambda V, R. \ sll(\text{rev}(L), R) \}
\]

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 \( R_p \) holds the return pointer and \( R_v \) the function return value.

\[
\exists L. \ V, \ sll(L, V(\ell)) \land \ sll(\text{rev}(L), R_v) \{ R_p \} f(l, 5)
\]

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 \( S_p \) holds the stack pointer.

\[
\exists L, V. \ \text{locals}([l], V, 5, S_p) \land \ sll(L, V(\ell)) \land \ \text{locals}([l], V', 5, S_p) \land \ sll(\text{rev}(L), R_v) \{ R_p \} f
\]

Here locals is just another abstract predicate similar to \( \text{sll} \). A fact 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 \( \land \) 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 \( R_p \). 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 \( \forall \ T. \ P(f(\bar{y}, n)) \{ Q \} \), 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:

\[
\forall y, V. \ \text{locals}(\bar{y}, V, n, S_p) \land P(V) \land \ \text{locals}(\bar{y}, V', n, S_p) \land Q(V, R_v) \{ R_p \} f
\]

To do code generation and computation of verification conditions and postconditions, a macro now needs to know the local variable set \( \bar{y} \) 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 \( \{ P \} \subseteq \{ Q \} \) from \( \{ P \} \subseteq \{ Q' \} \), given \( P \Rightarrow P' \) and \( Q' \Rightarrow Q \).

We have implemented a simple \textit{wrapper} combinator for macros, embodying the rule of consequence. A programmer uses pre-existing 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 \textit{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 left-hand column of the figure gives the program implementation. Our program declares a module \( n \), which imports function \texttt{malloc} from module \texttt{malloc}, with \textit{specification} \texttt{malloc5}. Next, we define function \texttt{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 \texttt{cmd}. We take advantage of two high-level macros to make our implementation more declarative.

First, we use a \textit{parsing} syntax similar to ML-style pattern matching, via the \texttt{Match} macro. Its arguments name an array, its length (\texttt{Size} clause), and a local variable storing our current index within the array (\texttt{Position} clause). A \texttt{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 \textit{declarative querying} of arrays, chosen as a simple example illustrating many of the same challenges applicable to more expressive SQL-style querying. The \texttt{For} macro takes as arguments a loop variable to store an array index, another variable to store the value found at that index (\texttt{Holding} clause), the array to query (\texttt{in} clause), the length of that array (\texttt{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]]]

begin
  "output" ← 0;
  "position" ← 0;

  While ("position" < "cmdLen") {
    Match "cmd" Size "cmdLen" Position "position"
      Case (0 ++ (1 ++ (2 ++
        (* CollectBelow *)
      Case (1 ++ "lower" ++ "upper")
  (* MaxInRange *)
      Case (1 ++ "lower" ++ "upper")
  (* CollectBelow *)
      Case (2 ++ "lower" ++ "upper")
  (* Inductive *)
  Theorem ok :
    \forall m. moduleOk m.
    Proof.
      vgen: abstract (parse1; 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., ValueInRange) and specified with pure mathematical functions in lowercase (e.g., valueInRange).

The specification main2 applies to our main function, giving its list of formal arguments, the arrays cmd and data plus their lengths. Two universal quantifiers (“∀”) 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 star-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 [ ], 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 goodness 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 vecgen and sep, come from the Bedrock library. Others, like parseo and foro, 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 5.5

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:

\[
\text{Match array Size size Position pos} \{ \text{Case pattern bodyStmt end} \} \text{ Default { defaultStmt}}
\]

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:

\[
\text{assume}(a = 0); \text{posn} \leftarrow b; \text{lower} \leftarrow c
\]

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 4 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 [81], 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:

\[
\text{For index \texttt{PRE[i]} invariant } \\
\text{For index Holding value in array Size len Where condition } \\
\{ \texttt{bodyStmt} \}
\]

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 \texttt{prefix} for the list of array cell values that have already been visited. In the course of a naive complete loop through the array, \texttt{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 precondition 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. 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 Caml4 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 ocamlc opt 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
Of course, there are natural 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 2, 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. [22–25] 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]. Other related work with mechanized separation logic has gone on at the level of C programs, as in Apple 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 MetaHaskell [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 [29] 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 C/Clone [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.

<table>
<thead>
<tr>
<th>File</th>
<th>Program</th>
<th>Invar.</th>
<th>Tactics</th>
<th>Other</th>
<th>Overh.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>70</td>
<td>45</td>
<td>25</td>
<td>30</td>
<td>2.2</td>
</tr>
<tr>
<td>LinkedList</td>
<td>50</td>
<td>35</td>
<td>20</td>
<td>35</td>
<td>3.2</td>
</tr>
<tr>
<td>Malloc</td>
<td>60</td>
<td>40</td>
<td>20</td>
<td>40</td>
<td>2.5</td>
</tr>
<tr>
<td>ListSet</td>
<td>70</td>
<td>45</td>
<td>25</td>
<td>45</td>
<td>1.5</td>
</tr>
<tr>
<td>TreeSet</td>
<td>80</td>
<td>50</td>
<td>25</td>
<td>50</td>
<td>2.2</td>
</tr>
<tr>
<td>Queue</td>
<td>90</td>
<td>55</td>
<td>30</td>
<td>55</td>
<td>1.5</td>
</tr>
<tr>
<td>Memoize</td>
<td>60</td>
<td>40</td>
<td>20</td>
<td>40</td>
<td>2.0</td>
</tr>
</tbody>
</table>

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