Live Verification in an Interactive Proof Assistant

SAMUEL GRUETTER, VIKTOR FUKALA, and ADAM CHLIPALA, MIT, USA

We present a prototype for a tool that enables programmers to verify their code as they write it in real-time. After each line of code that the programmer writes, the tool tells the programmer whether it was able to prove absence of undefined behavior so far, and it displays a concise representation of the symbolic state of the program right after the added line. The user can then either write the next line of code, or if needed or desired, write a specially marked comment that provides hints on how to solve side conditions or on how to represent the symbolic state more nicely. Once the programmer has finished writing the program, it is already verified with a mathematical correctness proof. Other tools providing real-time feedback already exist, but ours is the first one that only relies on a small trusted proof checker and that provides a concise summary of the symbolic state at the point in the program currently being edited, as opposed to only indicating whether user-stated assertions or postconditions hold.

Program verification requires loop invariants, which are hard to find and tedious to spell out. We explore a middle ground in the design space between the two extremes of requiring users to spell out loop invariants manually and attempting to infer loop invariants automatically: Since a loop invariant often looks quite similar to the symbolic state right before the loop, our tool asks the user to express the desired loop invariant as a diff from the symbolic state before the loop, which has the potential to lead to shorter, more maintainable proofs.

We prototyped our technique in the interactive proof assistant Coq, so our framework creates machine-checked proofs that the developed functions satisfy their specifications when executed according to the formal semantics of the source language. Using a verified compiler proven against the same source-language semantics, we can ensure that the behavior of the compiled program matches the program's behavior as represented by the framework during the proof. Additionally, since our polyglot source files can be viewed as Coq or C files at the same time, users willing to accept a larger trusted code base can compile them with GCC.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: software verification, symbolic execution, interactive proof assistants

ACM Reference Format:
Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live Verification in an Interactive Proof Assistant. *Proc. ACM Program. Lang.* 8, PLDI, Article 209 (June 2024), 26 pages. https://doi.org/10.1145/3656439

1 INTRODUCTION

Software verification has the potential to cut down significantly on bugs in software. In particular, if one proves that a program implemented in an optimized way in an efficient low-level language behaves according to a specification written in a high-level specification language, a large class of bugs can be excluded that could arise from the optimizations or from delicate, performance-minded design choices of the low-level language.

However, writing proofs about software can be a repetitive task, but fortunately, like many repetitive tasks, it can be automated by writing programs that perform it. But often, it is hard to find the right level of automation: One might think that the more automation, the better, but the more automated a prover is, the more it is at risk of going down a wrong route in its proof search and wasting time on proof steps that a human could easily recognize as useless. The reason is that, for a typical nontrivial program, the programmer has some (potentially domain-specific) insight

ACM 2475-1421/2024/6-ART209
https://doi.org/10.1145/3656439

about its correctness and about what strategies are promising to try, but the verification tool might not have this knowledge. An important question is therefore (a) how users can convey insight to the verifier. Equally important, but often neglected, is the opposite direction, i.e. (b) how the verifier can convey everything it knows to the user. This feature can be useful in two ways: If, while the user is writing a program, the verifier constantly provides a concise summary of everything it knows to be true at the current cursor position (also known as a symbolic state), this summary can help the user decide whether the program is correct up to that point, and it can hint at what the right next command in the program might be. And if the verifier fails to verify that an instruction is safe (for example, that an array access is within bounds), by looking at this summary of everything that the verifier knows, the user can guess more quickly why the verifier failed.

Our answer to (a) is to use Coq's tactic language Ltac both to implement the verifier and as the language in which users express their domain-specific insights, which leads to smooth cooperation between the two; and our answer to (b) is to use Coq's proof-goal display (which consists of a list of hypotheses that can be assumed and a conclusion that has to be proven) to display the current symbolic state of the program that the user is writing.

We start with some "clever tricks" to allow single source files to be accepted as legal code in both Coq and C, where interactive proof scripts appear amidst lines of normal C code. Then we add features that take advantage of the proof assistant, providing snapshots of "just right" complexity, describing what the framework inferred about all possible program states at particular code points. Along the way, we develop ideas that may mitigate some of the classic usability challenges of verification with Hoare logics, like the need to invent loop invariants out of whole cloth.

More specifically, we make the following contributions:

- We present a prototype of a framework that supports symbolic live debugging of (a subset of) C. It runs entirely within the Coq proof assistant and produces ASTs of the functions' source code as objects in Coq, with a correctness lemma for each function. Our tool's correctness need not be trusted, because it produces proofs that are verified by Coq's kernel. The correctness lemmas are expressed in terms of Bedrock2's source-language semantics [Erbsen et al. 2021], so our programs can be compiled with Bedrock2's verified RISC-V compiler.
- Most software-verification tools require users to provide loop invariants, which can become quite long and tedious to write down. We present a way to express a loop invariant as a diff from the inferred symbolic state at the beginning of the loop (§ 3.1.7 and § 4.4). Using some tactics, users can generalize and/or strengthen the symbolic state, and our framework can then use this modified symbolic state as the loop invariant. So the user still needs to provide the insight that leads to a suitable loop invariant, but it is not necessary to spell out the whole loop invariant. This solution potentially leads to an easier, more intuitive, and more enjoyable user experience and to proofs that are more robust against code changes, because diffs (edits) tend to be smaller than whole invariants.
- We argue that proof automation should optimize the user experience for failing proofs (the default case in a proof developer's day-to-day work) rather than for proofs where everything works, and we describe three principles that emerge from this focus (§ 4.8), including centering automation of side-condition solving around the notion of *safe steps* (§ 4.8.3), i.e. proof steps that do not turn provable goals into unprovable goals. We provide users with means to register domain-specific proof steps, enabling proofs that rely on backtracking only very locally and thus are both automated and easy to debug at the same time.
- If one is willing to trust our tool's notation-based parser, our polyglot Coq source files can also be viewed as C files and compiled with GCC, or if one is willing to trust Bedrock2's C pretty-printer, one can pretty-print the ASTs to C and compile with GCC (§ 3.2).
- We developed and verified a small but promising set of functions (§ 6) in our framework.

```
1   (* -*- eval: (load-file "../LiveVerif/live_verif_setup.el"); -*- *)
2   Require Import LiveVerif.LiveVerifLib.
3   Load LiveVerif.
4   #[export] Instance spec_of_memset: fnspec :=                                  .**/
5
6   void memset(uintptr_t a, uintptr_t b, uintptr_t n) /**#
7     ghost_args := bs (R: mem → Prop);
8     requires t m := <{ * array (uint 8) \[n] bs a
9                          * R }> m ∧
10                      \[b] < 2 ^ 8;
11    ensures t' m' := t' = t ∧
12         <{ * array (uint 8) \[n] (List.repeatz \[b] \[n]) a
13            * R }> m' #**/                                                   /**.
14  Derive memset SuchThat (fun_correct! memset) As memset_ok.                    .**/
15  {                                                                          /**. .**/
16    uintptr_t i = 0;                                                         /**.⋯.**/
22    while (i < n) /* decreases (n ^- i) */ {                                 /**. .**/
23      store8(a + i, b);                                                      /**. .**/
24      i = i + 1;                                                             /**. .**/
25    }                                                                        /**. .**/
26  }                                                                          /**. Qed.
27  End LiveVerif. Comments .**/ //.
```

Fig. 1. memset example, as displayed in Emacs, with lines 5 of Ltac (lines 17-21) folded away into ⋯

1.1 A First Glance At an Example

Figure 1 shows an example of a verified memset function. The file is a Coq file, but if we prefix it with an opening C comment /*, it becomes a C file. Lines 15 to 26 look like C code but are in fact just notations for proof tactics that gradually build the abstract syntax tree (AST) of the function, along with its correctness proof. The proof is completely automated, except for 5 lines of tactic code (lines 17-21, shown in Figure 3c) that express the desired loop invariant as a diff from the symbolic state before the loop. We will discuss this example in more detail in § 3.1, but we first provide some background in § 2.

2 BACKGROUND

This section provides some background to make the paper accessible to readers without prior knowledge of proof assistants, Coq, or program verification inside proof assistants. For each subsection, it should be safe to decide whether to skip it based on its title.

2.1 Editing Coq Proofs: Proof Goals and the Proof Cursor

The central notion for interactive proof development in Coq is that of a *proof goal*. On paper, we write proof goals as $\Gamma \vdash P$, where Γ is a list of variables and hypotheses that can be assumed, and P is the conclusion to be proven. In the actual Coq implementation, each variable and hypothesis is printed on a separate line, and the $\vdash$ is printed as a horizontal line (for example, see Figure 3a & b).

ProofGeneral is an Emacs extension for developing Coq proofs. For each Coq file being edited, it shows three windows: a window for the file itself, a window for the current proof goals, and a window to display error messages. In addition to the regular text-editing cursor, the file window also has a *proof cursor* that can be moved forward and backward using separate key bindings (or GUI buttons), and the proof-goal window always displays the proof goals that remain open at the current position of the proof cursor. If a proof contains an error, ProofGeneral ensures that the proof cursor can never be advanced past that error.

We will see in § 3.1 how we can repurpose the proof-goal window to serve as a debugger window displaying the symbolic values of all variables and memory, and how the proof cursor can be seen as the indicator of a debugger pointing to the next instruction to be executed.

2.2 Evars in Coq: Lazily Instantiated Existential Variables

While writing proofs in Coq, it is sometimes desirable to delay choosing some term until a later point where the updated proof goal makes it more obvious what the right choice for that term is. For example, if we have the proof goal $a : \mathbb{Z},\ b : \mathbb{Z},\ c : \mathbb{Z},\ H_1 : a < b,\ H_2 : c > b \vdash a < c$ and want to apply the lemma `Z.lt_trans`, which says $\forall\ n\ m\ p.\ n < m \Rightarrow m < p \Rightarrow n < p$, Coq can infer (by unifying the lemma's conclusion with the goal's conclusion) that n has to be instantiated to a and p to c, but it is not immediately clear what term m should be instantiated with. So either the user has to provide it explicitly by running the tactic `apply Z.lt_trans with (`m` := `b`)`, which results in two subgoals with the same hypotheses as the original goal and conclusions $a < b$ and $b < c$ respectively; or the user can delay the choice of m by running `eapply Z.lt_trans`. The `eapply` tactic is a variant of the `apply` tactic that creates so-called **evars** (short for existential variables) for terms that cannot be determined yet. On this example, it results in two subgoals with conclusions $a < ?m$ and $?m < c$, respectively, where the question mark is used to mark m as an evar, i.e. as **some hole that will be filled in later**. Note that the two occurrences of $?m$ in the two subgoals are linked: As soon as $?m$ is instantiated to some term in one goal, it is also instantiated to the same term in the other goal. To continue the example, one could now run the `eassumption` tactic on the first goal, which applies any assumption from the list of hypotheses that matches the conclusion. The `e` at the beginning of the tactic's name means that it can instantiate evars in order to unify a hypothesis with the conclusion, so it will pick H_1 and instantiate $?m$ to b.

2.3 A Use Case of Evars: Deriving a Definition Based on its Proof

Coq's **`Derive`** command can be used to create a definition and a proof about it at the same time. For example, if we want to define a list `myList` such that it contains (at least) `1` and `2` as its elements, we can start with the the command **`Derive`** `myList` **`SuchThat`** `(In 1 myList ∧ In 2 myList)` **`As`** `myLemma`. It starts the definition of a list named `myList`, along with a lemma named `myLemma`. Note that the definition of `myList` is not yet given at this point and will only be filled in gradually while writing the proof. This command creates an evar `?myList` for the definition being made and opens the proof goal `⊢ In 1 ?myList ∧ In 2 ?myList`. Using the `split` tactic turns it into two goals, `⊢ In 1 ?myList` and `⊢ In 2 ?myList`. Given the lemma `in_eq` which says `∀ (A : Type) (a : A) (l : list A), In a (cons a l)`, we can run `eapply in_eq` on the first subgoal, which unifies the conclusion of that lemma with `In 1 ?myList`. This step *partially instantiates* the evar `?myList`, namely to the term `(cons 1 ?l)`, which in turn contains a new evar `?l`. Therefore, the second subgoal now becomes `⊢ In 2 (cons 1 ?l)`. Then, the proof can be completed by applying `in_cons` which says that `∀ (A : Type) (a b : A) (l : list A), In b l ⇒ In b (cons a l)`, leading to `⊢ In 2 ?l` and then applying `in_singleton : ∀ (A : Type) (x : A), In x (cons x nil)`, which instantiates the remaining evar `?l` to the singleton list containing just `2`.

So, through this series of proof steps, the list `myList` was defined to be `(cons 1 (cons 2 nil))` solely based on its proof, without ever having to spell out this term as a whole.

3 OVERVIEW: WRITING AND COMPILING A SAMPLE PROGRAM

3.1 Guided Tour Through the `memset` Example

This subsection gives an overview of our approach by means of a detailed discussion of the sample program in Figure 1. The sample program contains many notations, and in this section, we are not yet attempting to explain what exactly each notation unfolds to. Instead, we are just trying to give an intuitive understanding of their meanings. For reference, a listing of all notations can be found in Appendix A.

3.1.1 Polyglot Source File Can be Read as C or Coq at the Same Time [Lines 1-27]. The code in Figure 1 is a Coq file accepted by unmodified Coq 8.17.1. By (ab)using Coq's notation system, we can insert program snippets that look like C code. If the file is preceded by our framework-specific C header and an opening C comment, it becomes a C file that can be compiled with GCC.

3.1.2 Function Signature Using Only One Type [Line 6]. Since we only support the subset of C that is also supported by Bedrock2, all variables have the same type, namely `uintptr_t` (defined in `stdint.h`). According to the standard, that is an unsigned integer type large enough to hold a pointer value, but we rely on the observation that in practice, compiler implementations define it as 32-bit and 64-bit unsigned int on 32-bit and 64-bit machines, respectively.

3.1.3 Specifications Using Separation Logic and $\mathbb{Z}$ [Lines 7-13]. The C signature is followed by a function specification enclosed in a `/**# #**/` comment that lists ghost arguments, a precondition over the initial event trace `t` and the initial memory `m`, and a postcondition over the final event trace `t'` and final memory `m'`. The parts between `<{ }>` are separation-logic assertions. We use `*` symbols as bullet points for lists of separation-logic clauses to be joined by separating conjunction, so `*` can also be read as the traditional star operator from separation logic, just with the additional liberty of allowing a series of separating conjunctions to start with a superfluous initial `*`. The `array` predicate takes as arguments the predicate for its elements `(uint 8)`, followed by its number of elements, its list of elements, and its start address.

To make our specifications as trustworthy as possible, we need to avoid accidential integer overflows in the specifications, so we generally use unbounded integers ($\mathbb{Z}$) in our specifications rather than bounded integers (`word`), except in situations with many bitwise operations and where integer overflow is the desired outcome. Therefore, we often need to interpret bounded integers (values that were computed by our programs) as unbounded integers in order to mention them in specifications. To interpret a `word` value `x` as an unsigned $\mathbb{Z}$, we use the notation `\[x]` (which expands to the `word.unsigned` function), and there is also a `word.signed` function (for which we have not yet invented a notation because we use it less frequently). The reverse direction, coercing a $\mathbb{Z}$ into a `word`, does not need to distinguish between signed and unsigned integers, because in both cases, it simply takes the 32 least significant bits of the unbounded integer's binary representation (where a negative number is considered to start with an infinite series of 1s on the left). We call this coercion `word.of_Z` and abbreviate it with `/[x]`, but since it drops the more significant bits, we try to use it as little as possible.

3.1.4 The Initial Proof Goal [Line 14]. We use Coq's **`Derive`**[1] command (§ 2.3) to start the correctness proof of a function that has not yet been defined but will be defined at the same time as we write the proof. The **`Derive`** command opens a proof goal which could be summarized, using the notation from § 2.1, as $\vdash\ P(t,s,m)\ \rightarrow\ \mathrm{wp}\ (t,s,m)\ ?body\ Q$, where P stands for the precondition from lines 8-10, Q stands for the postcondition from lines 11-13, and $?body$ is an evar (§ 2.2) acting as a placeholder

[1] A note for Haskell users: Unlike in Haskell, the **`Derive`** keyword in Coq is in no way related to the **`Instance`** keyword. We mark specifications as type-class instances to enable our tactics to look up the spec of a callee by its string name.

for the function body that is going to be defined. The state triple (t, s, m) contains an event trace t, a partial mapping s from variable names to values, and a memory m. The initial s contains just the function arguments, so in this example, it equals `map.of_list [|("a", a); ("b", b); ("n", n)|]`.[2] The wp judgment takes an initial state, a command, and a desired postcondition, and it tells us what we have to prove in order to ensure that after running the command on the initial state, the postcondition holds.[3]

3.1.5 *C Snippets Acting As Proof-Script Steps [Lines 15-26].*

Each C snippet is enclosed between a closing comment `.**/` and an opening comment `/**.` and is actually just a notation for a tactic. The first proof step, `.**/ { /**.`, introduces the precondition as hypotheses and performs some setup to start the proof.

3.1.6 *Applying Weakest-Precondition Rules [Lines 16-24].*

The assignment on line 16 is a notation for a tactic that applies the wp rule for assignment shown on the right.[4] It has a built-in sequence command, so applying it to a wp goal whose command is an evar instantiates that evar and leaves behind a new evar `?rest` for the subsequent commands.

The snippet on line 22 applies the WP-WHILE rule shown on the right.[5] It requires an invariant *Inv*, a proof that *Inv* holds for the initial state; a proof that *Inv* implies that evaluating the condition e is safe; a proof that if the condition is nonzero (true), running the loop body c always leads to a state that satisfies *Inv* again; and a proof that if the condition is zero (false), the code after the loop is correct.

Our framework contains rules for all language constructs, and they are all proven sound with respect to the semantics of Bedrock2 (expressed in omnisemantics [Charguéraud et al. 2023]).

WP-SET

$$\frac{\begin{array}{c}\text{eval\_expr}\ s\ m\ e\ v\\ \text{wp}\ (t, s[x := v], m)\ rest\ P\end{array}}{\text{wp}\ (t, s, m)\ (x := e; rest)\ P}$$

WP-WHILE

$$\frac{\begin{array}{l}\qquad\qquad Inv\ \sigma\\ \forall \sigma'.\ Inv\ \sigma' \Rightarrow \exists b.\\ \quad \text{eval\_expr}\ \sigma'\ e\ b\ \land\\ \quad (b \neq 0 \Rightarrow \text{wp}\ \sigma'\ c\ Inv)\ \land\\ \quad (b = 0 \Rightarrow \text{wp}\ \sigma'\ rest\ P)\end{array}}{\text{wp}\ \sigma\ (\texttt{while}\ e\ \text{do}\ c; rest)\ P}$$

Fig. 2. Some weakest-precondition rules

3.1.7 *Expressing the Loop Invariant as a Diff from the Current Symbolic State [Lines 17-21 in Figure 3c].*

The WP-WHILE lemma requires a loop invariant. Automatically inferring loop invariants is a hard problem, and we do not attempt to solve it. But spelling out loop invariants manually is also quite cumbersome. Therefore, we use an approach in-between these two extremes, based on the observation that the loop invariant often looks quite similar to the symbolic state just before the loop. Instead of requiring that the user spells out the *whole invariant*, we only require that the user expresses the *insight* needed to obtain the right invariant, expressed as a tactic script (Figure 3c) that transforms the symbolic state before the loop (Figure 3a) into a generalized and/or strengthened symbolic state (Figure 3b) which our framework then mechanically packages into a loop invariant (Figure 3d).

[2] Note that for list literals, we use the notation `[|x; y; z|]` instead of Coq's standard notation `[x; y; z]`, because we want to use bracket notation to index into lists, so the term `f [b]` would become ambiguous: It could be the application of function `f` to the singleton list containing `b`, or it could be the `b`-th element of list `f`. We experimented with type-based operator overloading (§ 5.7.2), but it did not seem worth the trouble.

[3] Note that even though weakest-precondition generators are often presented as threading a postcondition through a program *backwards*, we can actually also use them to step through a program in *forward* direction – we just need to evaluate the weakest-precondition generator under normal-order evaluation (i.e. left-to-right) instead of the standard call-by-value order where arguments get evaluated first.

[4] The rule that actually gets applied is specially tailored to work well with our proof automation, see § 5.2.

[5] For simplicity, we show a termination-insensitive variant, but the real lemma also requires a termination argument and is specially tailored for our proof automation, see § 5.2.

```
state : currently displaying
... 6 lines of section vars omitted ...
fs : list (string * func)
fs_ok : functions_correct fs ?Goal
Scope0 : ____ FunctionParams ____
a, b, n : word
bs : list Z
R : mem ⟶ Prop
Scope1 : ____ FunctionBody ____
t : trace
i : word
m, m0, m1 : mem
H0 : m0 |= array (uint 8) \[n] bs a
H1 : m1 |= R
D : m0 \*/ m1 = m
Hp1 : \[b] < 2 ^ 8
Def0 : i = /[0]
============================
ready
```

(a) Symbolic state (proof goal) after processing the first line of the function body in Figure 1

```
17  swap bs with
18    (List.repeatz \[b] \[i] ++ bs[\[i]:])
19    in #(array (uint 8)).
20  loop invariant above i.
21  delete #(i = ??).
```

(c) Snippet of Ltac code that was folded into ⋯ in Figure 1. The # notation is used to reference a hypothesis matching a pattern, instead of using its autogenerated (and thus subject-to-change) name.

```
state : currently displaying
... 6 lines of section vars omitted ...
fs : list (string * func)
fs_ok : functions_correct fs ?Goal
Scope0 : ____ FunctionParams ____
a, b, n : word
bs : list Z
R : mem ⟶ Prop
Scope1 : ____ FunctionBody ____
t : trace
Scope2 : ____ LoopInvOrPreOrPost ____
i : word
m, m0, m1 : mem
H0 : m0 |= array (uint 8) \[n]
     (List.repeatz \[b] \[i] ++ bs[\[i]:]) a
H1 : m1 |= R
D : m0 \*/ m1 = m
Hp1 : \[b] < 2 ^ 8
============================
ready
```

(b) Symbolic state (proof goal) after processing the Ltac code in (c)

```
fun (measure : word) (ti : trace)
    (mi : mem) (li : locals) ⇒
  exists i : word, ands [|
    measure = n ^- i; ti = t;
    li = map.of_list [|("a", a); ("b", b);
                       ("i", i); ("n", n)|];
    seps [|array (uint 8) \[n]
              (List.repeatz \[b] \[i]
               ++ bs[\[i]:]) a; R|] mi;
    \[b] < 2 ^ 8|]
```

(d) Loop invariant automatically built by packaging everything below __LoopInvOrPreOrPost__ in (b)

Fig. 3. Loop-invariant definition using a diff script (c) instead of explicitly spelling it out

3.1.8 Heapletwise Separation Logic [Background for Line 23]. It is useful to name each separation-logic clause and to make it available to Ltac's **match** command, which finds hypotheses matching a given pattern. Therefore, instead of using one big separation-logic clause `(P * Q * R) m`, we split it into one hypothesis per clause. This strategy requires explicitly splitting the memory `m` into a heaplet corresponding to each clause, which takes up some space in the display of the proof goal, but it can be handled completely automatically and therefore does not affect the user experience too negatively. This splitting then leads to three new heaplets `m0`, `m1`, `m2`; an equation saying that their disjoint union equals `m`, written as `m0 \*/ m1 \*/ m2 = m`; and three hypotheses `P m0`, `Q m1`

Table 1. Different Ways of Compiling

	Feed Coq File to GCC	Bedrock2's Ugly-Printer & GCC	Bedrock2 Compiler
Readability of exported code	OK (see e.g. Figure 1)	Decipherable (many casts & parentheses)	It is assembly
Instruction-set architecture support	Everything supported by GCC	Everything supported by GCC	Only RISC-V
Performance of compiled code	Good	Good	Bad
Additions to trusted code base	Notations to parse C into Bedrock2, GCC, load/store C header	Bedrock2's ugly-printer, GCC, load/store C header	Only the riscv-coq specification

and `R m2`. To make them more easily recognizable as memory hypotheses, we use the `mi |= Pi` notation, which just expands to `Pi mi`. See for example hypotheses `H0`, `H1`, and `D` in Figure 3a, and compare them to the precondition of the `memset` function on line 8 in Figure 1.

3.1.9 Accessing Memory That Is Part of a Bigger Separation-Logic Clause [Line 23]. `store8(a + i, b)` stores the lowest 8 bits of `b` to the `i`-th element of the array at `a`. According to the loop invariant, we have the following separation-logic clause:

```
H0 : m0 |= array (uint 8) \[n] (List.repeatz \[b] \[i] ++ bs[\[i]:]) a
```

However, the wp lemma for the store commands (omitted for space reasons) expects a separation-logic clause with just one `(uint 8)` element, so we need to split the array appropriately. Our tactics take care of this automatically, leading to the following three clauses:

```
H2 : m0 |= array (uint 8) \[i] (List.repeatz \[b] \[i]) a
H3 : m2 |= uint 8 bs[\[i]] (a ^+ i)
H7 : m4 |= array (uint 8) (\[n] - \[i] - 1) bs[\[i] + 1 :] (a ^+ i ^+ /[1])
```

The store then replaces `bs[\[i]]` with `b` in `H3`, and since the splitting tactic posed a hypothesis that acts as a reminder to merge the three clauses back together later, we end up with the following clause after the store:

```
H1 : m |= array (uint 8) \[n] (List.repeatz \[b] \[i] ++ [|\[b]|] ++ bs[\[i] + 1 :]) a
```

3.1.10 Proving That the Current Symbolic State Satisfies Expectations [Lines 25 and 26]. The closing brace at the end of the loop body creates a proof that the symbolic state obtained by executing the loop body satisfies the loop invariant again, and the closing brace at the end of the function body creates a proof that the final symbolic state satisfies the postcondition given on lines 11-13. In this example, the proofs are found completely automatically, but in more complex examples, the automation might leave some goals open for the user to prove manually. This completes our tour of the `memset` example.

3.2 Tradeoffs Between Three Different Ways of Compiling

Finally, we might also want to compile and run our code. Table 1 compares three different ways of compiling code that was verified in our framework. The C-parsing notations of our framework expand to Bedrock2 ASTs, defined as a Coq inductive datatype, so the correctness proofs are statements about these ASTs. The verified Bedrock2 compiler consumes the same ASTs and is proven correct against the same semantics as used by our framework, so when it comes to minimizing the TCB, this is the preferred approach. For better performance and support of ISAs other than just RISC-V, one can choose to compromise on TCB minimality: If one trusts our notations to parse C as well as Coq's implementation of its notation system, one can feed our Coq files (which are also C files if preceded by our header defining loads and stores and an opening comment `/*`) to GCC (and likely also to other C compilers), or if one prefers to trust Bedrock2's pretty-printer (called ugly-printer by its author), one gets less readable C code but otherwise similar characteristics.

One might also wonder whether it would make sense to compile our programs with CompCert. In practice, this would probably work, but we do not have a compatibility proof between Bedrock2 semantics and CompCert C semantics, and such a statement would not be provable because of differences such as e.g. that comparisons between integers that were obtained by casting pointers are undefined behavior in CompCert C.

4 USER INTERFACE

4.1 New Separation-Logic Concepts

To better drive separation-logic proof automation and make some expressions more concise, we introduce a few properties of separation-logic predicates:

4.1.1 Predicate Size. Often a separation-logic predicate `P` occupies some range of memory addresses, and we need to know the length in bytes of that range. Therefore, we define `PredicateSize P` to be an alias of $\mathbb{Z}$, mark it as a type class, and register a hint for each predicate, so that we can use type-class search to find the size of a predicate. The predicate `(array elemPred n xs a)` can then use an implicit, automatically inferred argument `elemSize` of type `(PredicateSize elemPred)`, to state that at address `a`, we have an array of the `n` elements in list `xs`, where the `i`-th element of `xs` is asserted using `(elemPred xs[i] (a+i*elemSize))`.

4.1.2 Support for Adjacent Sep Clauses: `sepapp` *and* `sepapps`*.* Often, we want to lay out several predicates adjacent to each other.[6] To avoid having to write out offsets explicitly, we introduce a new definition that we call *separating append*, written `sepapp P1 P2 addr`. It takes two separation-logic predicates `P1` and `P2` of type `word → mem → Prop`, where the `word` stands for the address at which the predicate begins, and also takes an implicitly inserted argument `P1size` of type `PredicateSize P1` (which can be found by type-class search as explained above) and an address `addr`, and it is defined as the separating conjunction `P1 addr * P2 (addr ^+ /[P1size])`. We also define a `sepapps` predicate that takes a list of predicates, infers their sizes, and lays them out adjacently.

4.1.3 n-Fillable Predicates. We call a predicate P n-fillable if for any n-byte buffer at address a, there exists a value v such that the predicate $P\ v\ a$ holds. This concept is useful to know whether we can cast the byte buffer returned by `malloc` into a predicate P.

4.2 Defining Record Predicates Using C Syntax

Our framework supports defining separation-logic predicates using C syntax. For example, given a Coq record type for nodes of singly linked lists, `Record node := { data: word; next: word }`, we can create a separation-logic predicate called `node_t` that asserts that at a given address, a representation of a given `node` record is found. Using `sepapps` and some custom notations, we can define a predicate that looks like a C struct definition (first definition in Figure 4). The two other definitions in that Figure express the same predicate but using a notation for `sepapps` or `sepapps` directly, respectively.

[6] So far, we have only considered packed records, so we do not automatically insert spacing to respect alignment constraints.

```
Definition node_t(r: node):
  word ⟶ mem ⟶ Prop := .**/
typedef struct
__attribute__ ((__packed__)) {
  uintptr_t data;
  uintptr_t next;
} node_t;
/**.
```

```
Definition node_t(r: node): word ⟶ mem ⟶ Prop :=
  <{ + uintptr (data r)
     + uintptr (next r) }>.
Definition node_t(r: node): word ⟶ mem ⟶ Prop :=
  sepapps
    (cons (mk_sized_predicate (uintptr (data r)) 4)
    (cons (mk_sized_predicate (uintptr (next r)) 4)
    nil)).
```

Fig. 4. Three equivalent definitions, using different notations

4.3 IDE Extensions

Our framework can be used in any IDE for Coq. However, there are three very common operations for which we implemented keyboard shortcuts in 40 lines of Emacs Lisp: Showing/hiding of the Ltac block under the cursor (i.e. folding tactics into ···), inserting spaces until the end of line followed by a C-closing/Ltac-opening marker `/**.` and then processing the line, and inserting and processing one `step` command (§ 4.8.3).

4.4 Expressing a Loop Invariant as a Diff from the Current Symbolic State

Before each loop, the user of our framework must turn the symbolic state into a shape that our framework can use as a loop invariant. The example in Figure 3 should be helpful to illustrate the general process that we are going to explain in detail now. All modifications are expressed in Ltac and are of two kinds:

The first is that the user needs to separate variables and hypotheses that remain unchanged during the loop from those that may change during the loop, by using the command `loop invariant above x`, where `x` is the name of a variable or hypothesis. This command adds a `LoopInvOrPreOrPost` marker above `x` to separate unchanged (above) from changing (below) variables and hypotheses. After adding this marker, one can use Coq's builtin Ltac commands `move x before y` and `move x after y` to move hypotheses and variables up and down, until each is on the correct side of the separating marker. The variables below the marker will turn into existentials in the loop invariant, and the hypotheses will turn into a big conjunction (expressed as `ands [|...|]`). The variables and hypotheses above the marker do not appear in the loop invariant, except that the local variables above the marker are asserted to keep their values throughout the loop, and the hypotheses naturally remain available during the proof of the loop body without requiring further intervention.

The second kind of modification is related to generalizing the state. For instance, a variable `i` that equals one particular value before the loop might need to be generalized to be within a range by `prove (0 <= i < n)`; and by `delete #(i = ??)`, which finds the first hypothesis of shape `i = ??` and deletes it. Other common modifications of this kind include viewing a list of unprocessed items as the concatenation of an empty processed list and a remainder of unprocessed items, then forgetting that the unprocessed and processed list are the empty and whole list, respectively. A similar example is also in Figure 3c, where we replace the list `bs` of initial garbage data by the concatenation of repeating `\[b]` zero times (zero being the initial value of `i`) and the suffix of `bs` starting at `i`. And finally, it is sometimes also needed to introduce additional variables, so that a value that happens to be the same in two hypotheses can differ during the loop, which can be achieved using the `pose (a := b)` command, and `change b` **`with`** `a` **`in`** `H`, and finally, `clearbody a` to forget that `a` equals `b`.

4.5 Treating While Loops as Tail-Recursive Calls

Certain loops can be more easily verified by viewing them as tail-recursive functions with pre- and postconditions parameterized over ghost variables [Tuerk 2010]. Before each loop iteration, the precondition must hold, and at the end of the loop body, one has to show that the current state implies the precondition with smaller ghost variables, and one also has to show that the postcondition with small ghost variables implies the postcondition with bigger ghost variables.

For instance, when iterating over a data structure, the ghost variables can include a representation of the data structure and a frame, and the former shrinks with each iteration, while the latter grows with each iteration, so that we can forget the parts of the memory that are not relevant anymore.

We implement support for while and do-while loops in this style, using the symbolic state before the loop, appropriately generalized and strengthened through a diff script by the user, as a precondition, and the function's postcondition as the postcondition of the tail-recursive view of the loop. Since we do not want users to spell out loop postconditions manually, we do not support yet this tail-recursive view for cases where the code after the loop still needs to access the memory that was "forgotten" (pushed into the frame) during the loop. In such cases, one would have to factor the code into two functions or resort to a traditional while loop with just one invariant.

4.6 Variable-Naming Scheme

Our tactics make sure that a program variable named `"x"` always has its corresponding value bound to a Coq variable named `x`. When a variable gets reassigned, the old value is renamed into `x'`, and `x` is used for the new value.

4.7 Context Packaging and Merging for if-then-else

For if-then-else, we use the lemma WP-IF.[7] Note that the `if` below the line belongs to the object language (Bedrock2), whereas the `if` above the line belongs to the metalanguage (Coq).

$$\textsc{wp-if}\quad \frac{\begin{array}{c}\mathsf{eval\_expr\_as\_bool}\ \sigma\ e\ b\\ (b = \mathtt{true}\ \rightarrow\ \mathsf{wp}\ \sigma\ \mathit{thn}\ Q_1)\\ (b = \mathtt{false}\ \rightarrow\ \mathsf{wp}\ \sigma\ \mathit{els}\ Q_2)\\ \forall \sigma'.\,(\mathtt{if}\ b\ \mathsf{then}\ Q_1\ \sigma'\ \mathsf{else}\ Q_2\ \sigma')\ \rightarrow\ \mathsf{wp}\ \sigma'\ \mathit{rest}\ P\end{array}}{\mathsf{wp}\ \sigma\ (\mathtt{if}(e)\{\mathit{thn}\}\mathtt{else}\{\mathit{els}\};\mathit{rest})\ P}$$

When WP-IF gets applied, evars are created for the result b of evaluating the condition e, for the code snippets thn, els, and $rest$, as well as for the postconditions of the two branches, Q_1 and Q_2. The tactics first evaluate the condition e into a Boolean b. Then, the user can provide more snippets that make up the code of the then-branch. When providing the snippet `.**/ } else { /**.`, the then-branch is closed, and the evar $?Q_1$ is instantiated by our tactics to a conjunction of all the hypotheses in the current context. When the user closes the else-branch, $?Q_2$ is instantiated in the same way, and before the first command after the if-then-else is processed, the two symbolic states (expressed by Q_1 and Q_2) are merged by pushing down the metalanguage `if` as far as possible by detecting parts in Q_1 and Q_2 that have the same structure. The tactics bind the value of b to a fresh variable, so that we can mention it many times without becoming overly verbose. This merging results in symbolic states containing hypotheses with many if-then-else expressions like e.g. in the following:

```
H1 : m0 |= uint 32 (if c' then in1 else if c then in2 else in0) a0
Def7 : w1 = (if c' then /[in0] else /[in1])
```

A `/* split */` option is available that can be inserted after the `if` condition if one prefers to continue the proof separately after the if-then-else rather than using a merged state. However, this option is only available if the if-then-else is at the end of a block with a concrete postcondition (i.e.

[7] The lemma that we actually use is specially tailored to work well with our tactics, as described in § 5.2.

a loop invariant or the function's postcondition), because splitting the proof of the code after the if-then-else into two separate proofs would require writing down all the code snippets (which drive the proof) twice, which would not result in the desired C code when treating the Coq file as a C file.

4.8 Optimize the User Experience for Failing Proofs Instead of Working Proofs

In the past, most frameworks for automated program proofs have focused on presenting automated proofs that work. However, we must recognize that the default case that users of program verification tools face is the case where the prover fails or seems to run forever, either because the program or the specification contains a bug, or because a user-provided invariant is not strong enough, or because the prover lacks some domain-specific insight or hint that needs to be provided by the user.

We believe that debugging these situations, and being able to determine quickly which of the above is the case, is the primary usability criterion for a program-verification tool, much more important than the number of lines of proof script that users need to provide manually.

Therefore, we adhere to three principles described in the following subsections.

4.8.1 Do Not Run "Forever" on Failing Proofs. We carefully designed our proof automation in such a way that it never runs for longer than a few seconds, and if it does, we consider it a bug.

4.8.2 Actionable Error Messages. If the tool fails to prove a goal, it should provide the user with an error message containing information on what it tried and what (currently unprovable) conditions might enable it to make more progress.

As an example, let us look more closely at separation-logic cancellation, which is required e.g. before function calls, to match the caller's symbolic heap to the callee's symbolic heap. The strategy is repeatedly to delete separation clauses that appear both in the caller's heap and in the callee's heap. Since the clauses in the callee's heap typically contain evars for the callee's ghost arguments (because ghost arguments do not get mentioned in the source code), our procedure carefully only instantiates an evar if there is a unique choice. Sometimes, e.g. when a record field or a slice of an array is passed to the callee, cancellation needs to split a caller's clause before it can proceed. So, if the callee expects a chunk of memory covering the range starting at address a of size n, we need to find a clause in the caller's heap covering a superrange of that range, starting at an address a' and of a size n' such that the subset relation on half-open intervals $[a, a+n[\subseteq [a', a'+n'[$ holds, written in Coq as `subrange a n a' n'`.

Say we want to implement and verify a function with the following signature:

```
uintptr_t safeCopySlice(uintptr_t src, uintptr_t srcOfs, uintptr_t srcLen,
     uintptr_t unsafeN, uintptr_t dst, uintptr_t dstOfs, uintptr_t dstLen)
```

Its full specification spans 25 lines of code and is given in Appendix B but is more easily expressed in English: Given a byte array of length `srcLen` at address `src` and a byte array of length `dstLen` at address `dst`, we want to copy `unsafeN` bytes starting at offset `srcOfs` in the source array to offset `dstOfs` in the destination array. We already know that `srcOfs` and `dstOfs` are within bounds, but `unsafeN` comes from an untrusted origin and needs to be checked. We might start with the following:

```
Derive safeCopySlice SuchThat (fun_correct! safeCopySlice) As safeCopySlice_ok.    .**/
{                                                                             /**. .**/
  if (srcOfs + unsafeN <= srcLen && dstOfs + unsafeN <= dstLen) /*split*/ {   /**. .**/
    Memcpy(dst + dstOfs, src + srcOfs, unsafeN);                              /**.
```

After processing the proof just until before the `Memcpy` call, our symbolic state contains 16 uninteresting lines listing variables that we elide, followed by some more interesting lines shown in Figure 5.

```
H0 : m0 |= array (uint 8) \[srcLen]
             srcData src
H2 : m2 |= array (uint 8) \[dstLen]
             (dstData ++ dstUninit) dst
H3 : m3 |= R
D : m0 \*/ (m2 \*/ m3) = m
Hp1 : \[srcOfs] <= \[srcLen]
Hp2 : len dstData = \[dstOfs]
Scope2 : ____ IfCondition ____
H : \[srcOfs ^+ unsafeN] <= \[srcLen]
H1 : \[dstOfs ^+ unsafeN] <= \[dstLen]
Scope3 : ____ ThenBranch ____
==========================
ready
```

Fig. 5. Proof goal before Memcpy

At this point, the user could start wondering whether the word addition in hypothesis H, i.e. \[srcOfs ^+ unsafeN], could also be expressed as an addition on $\mathbb{Z}$, i.e. as \[srcOfs] + \[unsafeN], and why the tool did not do that, even though it usually does, or the user can also just carry on and move the proof cursor past the Memcpy call. Doing so changes the conclusion of the goal to (find_hyp_for_range (dst ^+ dstOfs) (\[unsafeN] * 1) X), where X is a bigger goal that we elided for presentation purposes. find_hyp_for_range is a Gallina definition of an identity function that takes two phantom arguments that it ignores, plus a third one, X, that it returns. It serves as a marker to inform the tactics as well as the user that the tool is performing cancellation and looking for a separation-logic clause in the caller's symbolic heap whose range starts at (dst ^+ dstOfs) and spans (\[unsafeN] * 1) bytes. The tool also emits the following error message: "Exactly one of the following claims should hold:" [|subrange (dst ^+ dstOfs) (\[unsafeN] * 1) src (\[srcLen] * 1); inrange src (dst ^+ dstOfs) (\[unsafeN] * 1); subrange (dst ^+ dstOfs) (\[unsafeN] * 1) dst (\[dstLen] * 1); inrange dst (dst ^+ dstOfs) (\[unsafeN] * 1)|] ! This message might look quite unintelligible at first, but we will show how it is *actionable* in the sense that it points the user to something to try to prove that is unprovable and will make the user understand the bug. The message contains four semicolon-separated claims and says that exactly one of them should hold. We can ignore the two inrange claims, because they are only needed to split separation-logic clauses on the callee side, which we do not expect to happen here, so we are left with just two subrange claims, and the second one looks like it *should* be provable (whereas the first one tries to relate completely unrelated ranges, one in dst, the other in src). Since we are in Coq's proof mode, right after the call to Memcpy, we can insert the following Ltac snippet to try to prove the subrange claim that we think should hold:

```
assert (subrange (dst ^+ dstOfs) (\[unsafeN] * 1) dst (\[dstLen] * 1)). {
  unfold subrange. bottom_up_simpl_in_goal.
```

It leads to a new goal with conclusion \[dstOfs] + \[unsafeN] <= \[dstLen], but the most closely matching hypothesis is H1 (see Figure 5), which performs the addition on word instead of on $\mathbb{Z}$. So now, the user might complain about how limited the proof automation of this Live Verification tool is and attempt to prove the goal manually, by invoking Coq's standard Search command with the pattern \[_ ^+ _] as its argument, whose top two results are a lemma which shows that for all words x and y, \[x ^+ y] = word.wrap (\[x] + \[y]); and one which shows that if \[x] + \[y] < 2 ^ width, then \[x ^+ y] = \[x] + \[y]. By this point, it should have become clear that the program contains an overflow bug: If unsafeN is very big, the addition overflows and results in a small number that might satisfy the condition tested by the **if** on the first line of safeCopySlice, but the Memcpy will still copy unsafeN bytes and overwrite out-of-bounds data, which could be exploited by attackers for arbitrary code execution. In fact, this (seemingly simple) type of bug (overflow on unsigned integer addition that computes the required amount of memory) led to the stagefright bugs, which in 2015 exposed the majority of Android users to no-click remote code execution on mere reception of a malicious MMS message.[8] Replacing the test by the following, overflow-safe variant resolves the problem:

```
if (unsafeN <= srcLen - srcOfs && unsafeN <= dstLen - dstOfs) /*split*/ {   /**. .**/
```

For reference, the corrected full program (which does not do anything more in addition to what was already shown except returning 1 or 0 depending on whether `unsafeN` was accepted) is given in Appendix B. We note that all its proof steps are completely automated, including the splitting of the source and destination arrays before the `Memcpy` call, pasting them back together after the call, and matching that result to the desired postcondition.

4.8.3 Safe Steps – Avoiding Backtracking for Better Proof Debuggability. To make our proof automation more debuggable, we avoid backtracking as much as possible and instead use mechanisms that allow us to know whether a proof step is *safe*, i.e. will not turn a provable goal into an unprovable one. We expose a tactic called `step` to the user, and when a proof does not work, the user can disable the automatic invocation of side-condition solving by replacing the `/**.` after a snippet by `/*?.` and then manually invoke `step` many times and watch step-by-step what the prover does and how it affects the proof goal.

To give an example of safe and unsafe steps, if we have a goal `?xs ++ ?ys = vs1 ++ vs2`, i.e. two evars on the left and normal variables on the right, it would be tempting to just instantiate `?xs` to `vs1` and `?ys` to `vs2`. However, this choice might make another goal in which the two evars appear as well unsolvable, because the correct choice for `?xs` might be `vs1 ++ vs2[:1]`, and the correct choice for `?ys` would then be `vs2[1:]`. On the other hand, on a very similar-looking goal, `vs1 ++ ?ys = vs1 ++ vs2`, it is safe to instantiate `?ys` to `vs2`, because that is the only possible choice.

We use a user-extensible hint database of judments of the form `safe_implication P Q`, which is defined as `P` implies `Q`. The opposite direction usually also holds, but in some cases, `Q` does not quite imply `P`, yet the only reasonable way to prove `Q` is to reduce it to proving `P`, so we do not require the opposite implication direction. For examples like the above, our hint database of safe steps contains the rules `safe_implication (ys1 = ys2) (xs ++ ys1 = xs ++ ys2)` and `safe_implication (xs1 = xs2) (xs1 ++ ys = xs2 ++ ys)`.

As an example of how safe steps can help debug failing proofs, consider the last proof step of the insert function of a binary search tree, in the case where a new leaf had to be allocated for the value to be added. Assume that the programmer correctly initialized all fields of the new leaf but forgot to link the leaf to the parent node. The return value of the function is specified to be 0 if the memory allocator failed and 1 if it succeeded. Figure 6 shows the postcondition that needs to be proven in this case.

To debug why this postcondition cannot be proven automatically, the user can insert and process[9] many invocations of `step` and see how they try to solve the goal step-by-step. Each step also prints a short description of what it did. Here, we just summarize the most interesting steps. One of the first steps gets rid of the trivial equality `t = t`, and a subsequent step notices that since `\[/[1]] = 0` can never hold, it is *safe* to attempt proving only the right-hand side of the disjunction. Further steps then start cancelling the separation-logic formula with clauses from the hypotheses (not shown in this paper) and manage to prove everything except a remaining goal `(is_empty_set (fun x : Z ⇒ x = \[vAdd] ∨ s x))`, which asks us to prove that a set, expressed as a lambda returning a proposition, is empty, even though it clearly contains at least one element, namely `\[vAdd]`. Here we see a case where reducing unprovability to the smallest

[8] https://en.wikipedia.org/wiki/Stagefright_(bug), https://nvd.nist.gov/vuln/detail/CVE-2015-3864, https://www.exploit-db.com/docs/39527

[9] We provide an Emacs macro to do so efficiently (see § 4.3), but simple copy-paste works as well.

```
(t = t ∧
 (\[/[1]] = 0 ∧
  <{ * allocator_failed_below 12
     * (EX rootp : word, <{ * uintptr rootp p
                            * (EX sk : tree_skeleton, bst' sk s rootp) }>)
     * R }> m4 ∨
  \[/[1]] = 1 ∧
  <{ * allocator
     * (EX rootp : word,
          <{ * uintptr rootp p
             * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
     * R }> m4))
```

Fig. 6. Postcondition that needs to be proven in one case at the end of a (buggy) binary-search-tree insert, where `bst' sk s a` asserts that at address `a` is found a binary search tree whose tree structure is `sk` and whose contents correspond to the set `s`, represented as a propostion over values.

possible core is actually too much, so that it is not easily understandable anymore why the tool asks us to prove this contradictory goal. But, fortunately, one of the intermediate goals that the user encounters while invoking `step` repeatedly is more enlightening: It asks the user to prove `bst' ?x (fun x : Z ⇒ x = \[vAdd] ∨ s x) /[0]`, i.e. that at address 0, there is a binary search tree containing `vAdd`, the value being inserted. However, what we expect to prove is that this binary search tree is at some nonzero address `p`, which points us directly to our bug, namely that the pointer that should point to our newly allocated leaf still is 0 instead of `p`.

So, to summarize, this example shows that sometimes (in fact, often, in our experience), neither the full initial unprovable goal nor its smallest unprovable core is very enlightening, but the most enlightening goal is somewhere in-between during the automated proof process, and giving the users a means of running this automated proof process step-by-step enables them to understand more quickly why a goal cannot be proven.

5 IMPLEMENTATION NOTES

5.1 Parsing C in Coq

Using Coq's notation system , we can declaratively write a parser for a big enough subset of C. Our ASTs use strings to represent identifiers, but we do not want double quotes around these strings to appear in our C code. Unfortunately, there is no officially supported way of getting rid of these quotes in Coq, so we resort to a somewhat sinister trick by Pit-Claudel and Bourgeat [2021, §3].

5.2 Tailored Weakest-Precondition Lemmas

Based on wp rules like the ones in Figure 2, we prove another layer of wp rules (two of which are shown in Figure 7) that is tailored to work well with the proof-automation tactics.

While WP-SET uses two separate hypotheses for the evaluation of the expression e and the remainder of the program *rest*, `wp_set` uses a judgment called `dexpr1` whose last argument is a proposition that needs to hold after evaluating `e`, so that changes to the symbolic state (i.e. changes to the hypotheses of the proof goal) that are made while evaluating `e` are also visible to the proof code for the rest of the program. For instance, if the evaluation of `e` contained some memory access that treats some byte buffer as a record, the proof for `e` will change the corresponding hypothesis

```
Lemma wp_set: forall fs x e v t m l rest post,
  dexpr1 m l e v (update_locals [|x|] [|v|] l (fun l' ⇒ wp_cmd fs rest t m l' post)) →
  wp_cmd fs (cmd.seq (cmd.set x e) rest) t m l post.

Lemma wp_while {measure: Type} (v0: measure) (e: expr) (c: cmd) t (m: mem) l fs rest
  (Inv: measure → trace → mem → locals → Prop) {lt} {post: trace → mem → locals → Prop}:
  Inv v0 t m l →
  well_founded lt →
  (∀ v t m l, Inv v t m l →
    ∃ b, dexpr_bool3 m l e b
      (loop_body_marker (wp_cmd fs c t m l (fun t m l ⇒ ∃ v', Inv v' t m l ∧ lt v' v)))
      (pop_scope_marker (after_loop fs rest t m l post))
      True) →
  wp_cmd fs (cmd.seq (cmd.while e c) rest) t m l post.
```

Fig. 7. Tailored Weakest-Precondition Lemmas

from a byte-array predicate to a record predicate, and it is usually desirable to preserve this change for the rest of the program.

Lemma wp_while (Figure 7) is based on WP-WHILE (Figure 2) but adds a termination measure that needs to decrease at the end of each iteration according to a user-provided less-than predicate lt which needs to be well_founded. The lemma contains some markers such as loop_body_marker, pop_scope_marker, and after_loop (an alias of wp_cmd) that inform the tactics what to do. It uses a judgment called dexpr_bool3, whose last three arguments are propositions that need to hold in case the Boolean b obtained from evaluating the expression e turns out to be true, false, or either, respectively. For example, a loop searching through a tree where null pointers are used for leaves might start with while (p && load(p) != key), and during the evaluation of the condition, in the case where p is non-null, this fact allows us to turn the memory assertion which says that at p, we either have a leaf or a node into one that says we certainly have a node at p, and using dexpr_bool3 instead of a simple conjunction that gets split into separate subgoals allows us to keep this modification visible to the evaluation of the loop body.

5.3 Extracting Pure Facts From Sep Clauses

A separation-logic formula often contains some pure (i.e. heap-indepenent) facts, either by explicitly asserting them or because its definition implies them. For example, a (ring_buffer cap vs a) judgment declaring a circular buffer of capacity cap at address a containing the elements in list vs might imply the pure fact len vs <= cap.

In order to make such pure facts available to our solver for arithmetic side conditions, we define the judgment purify R P := ∀ (m: mem), R m → P, and whenever we define a new separation-logic predicate R, we also prove a corresponding purify lemma and register it in a custom hint database. Before running side-condition solvers, our framework searches the hint database for a purification rule of the form (purify R _) for each separation-logic clause R and applies all the rules it finds.

5.4 Pattern-Based Selective Warning Suppression

If the framework encounters a separation-logic clause for which it cannot find a purify hint or a PredicateSize, it emits a warning, because often, this is the reason a proof does not go through. But some clauses do not contain pure facts or do not have constant sizes. For these, we want to suppress the warning selectively. To do so, we use a Coq hint database to which we add the patterns of all warnings that should be suppressed. Compared to most other warning-suppression mechanisms, which only allow warnings to be suppressed by their kinds, ours also allows suppressing them based on their arguments, without any additional implementation effort: We just piggy-back on Coq's very general building blocks.

5.5 Mixed Word/Integer Arithmetic Side Conditions

When reasoning about array accesses and simplifying symbolic expressions indexing into lists, many arithmetic side conditions need to be solved. Since our specifications are written in terms of ℤ, but the programs operate on a bounded `word` type, we obtain side conditions that mix the two. We solve such a goal as follows: First, if it is an equality or inequality on words, we use an injectivity lemma to reduce it to an equality or inequality on ℤ. Next, we push down all conversions from word to ℤ (written as `\[x]`), transforming them into modulos. For instance, `\[a ^+ b]` gets rewritten to `(\[a] + \[b]) mod 2 ^ 32`. Then, we eliminate modulos using the Euclidean equations, leading to terms like `\[a] + \[b] - 2 ^ 32 * k`, where `k` is `(\[a] + \[b]) / 2 ^ 32`. For efficiency, our implementation merges these two steps into one. This push-down of `\[_ OP _]` into `\[_] OP \[_]` with modulos is applied recursively until only variables or unintepreted functions are wrapped in `\[_]`. Bounds are then asserted, since interpreting a word as an unsigned ℤ always leads to a number between `0` and `2 ^ 32`. Finally, we invoke Coq's linear-arithmetic solver `lia`.

5.6 Undoable, Reusable ℤification

We call the preprocessing described in the previous subsection ℤification. Before solving arithmetic side conditions, it has to be applied to the conclusion, as well as to all arithmetic hypotheses. Our bottom-up term-simplification procedure needs to invoke arithmetic-side-condition solving hundreds of times in order to find which simplifications to apply. For instance, when encountering a list slice starting at `i` of a list append like `(xs ++ ys ++ zs)[i:]`, we need to test whether `i` is ≤ `0`, points somewhere into `xs`, `ys`, or `zs`, or whether it exceeds the whole length, which already amounts to 5 separate queries. ℤifying all hypotheses from scratch for each arithmetic side condition would be unacceptably slow. Instead, we implement ℤification in such a way that it does not modify any hypotheses but just makes a ℤified copy of each arithmetic hypothesis. Each time the user adds a new C snippet, we run hypothesis ℤification once and reuse the ℤified hypotheses for many side conditions, and just as the last step before marking the goal as ready for the next C snippet, we clear all the ℤified hypotheses, so that a clean and concise context is presented to the user.

5.7 Discussion

In the following, we discuss a few design alternatives that we decided not to pursue further.

5.7.1 Why Not a Stand-Alone Tool? Building our framework inside Coq required us to go through some contortions, especially to make tactic invocations look like C snippets – clearly, Coq was not designed to do this.

In order to build a software-verification tool that provides a live display of the current symbolic state, we could also have built a stand-alone tool from scratch, which might have saved us some trouble and, if implemented well, might also have been more performant because it could be more specialized to our task, thus not having to pay the cost of being run inside a tool as generic as Coq.

However, Coq still has several advantages that made us choose it: Coq provides many term-manipulation facilities, including concise term matching, and its foundational proofs, i.e. proofs that are checked by its small proof-checking kernel, guarantee soundness, so that bugs in our framework cannot lead to wrong proofs, which allows us to modify the tool more freely and confidently, without worrying about soundness at each modification. Finally, working with Coq paves the way for connecting to the many other interesting verified artifacts in the Coq ecosystem.

5.7.2 Limiting the Number of Conversions and Avoiding Operator Overloading. To avoid accidential overflows in our specifications, we write them using unbounded integers $\mathbb{Z}$, but the values treated by our programs are bounded 32-bit integers, and loading and storing 8-bit and 16-bit values is supported as well. Moreover, certain values in the specifications cannot be negative, so they would belong to $\mathbb{N}$. We tried using separate types for $\mathbb{N}$, $\mathbb{Z}$, 8-bit, 16-bit, and 32-bit words, but it led to two problems:

First, since Coq does not support subtyping natively, coercion functions are needed between different number types. Writing and displaying them explicitly becomes very verbose quickly, and relying on Coq's implicit-coercion feature did not work well. Coercions are inserted during type-checking, so patterns, which are untyped, do not have them inserted, which can lead to confusion. Coercions also make it harder to copy-paste a term from the goal into the proof script, because one might miss a coercion that only gets inserted because of the surrounding context.

The second problem was operator overloading: We would like to use some short infix notation for common operators like addition, subtraction, etc. Coq provides a mechanism called notation scopes that works well as long as no polymorphic functions are used, because when parsing the arguments of a function, Coq relies on the signature of the function to determine in which notation scope (e.g. the notation scope for $\mathbb{N}$ or for $\mathbb{Z}$) to parse the arguments. Another popular mechanism for operator overloading is to use type classes. For instance, the infix notation `(a + b)` might be defined as `(TypeclassBasedAdd a b)`, where `TypeclassBasedAdd` takes an implicit argument that is a type-class instance implementing addition on the type of `a` and `b`. However, if we simplify terms or obtain terms from third-party libraries not using such a type class-based notation system, they contain the plain `(Nat.add a b)` instead of our type class-based one, so they will not be printed the same and will not match our terms syntactically. Similar problems occur with a related approach based on canonical structures. We also tried an operator-overloading approach using notations with tactics in terms that type-checks the operands and picks the right operator based on the type of the operands, resulting in plain terms like `(Nat.add a b)`, combined with ambiguous printing-only notations that use the same + symbol for addition on all types. It was a bit heavy-weight and did not work in patterns (because they are not type-checked), so we stopped using it.

Finally, we decided to restrict ourselves to just two number types: 32-bit words and $\mathbb{Z}$. This approach only requires three coercions: truncating a $\mathbb{Z}$ to a bounded integer (which we write as `/[x]`), interpreting a word as a signed integer (which we use less frequently and write as `word.signed x`), and interpreting a word as an unsigned integer (which we write as `\[x]`). It also only requires two sets of infix arithmetic operators, so we use the regular operators for $\mathbb{Z}$ and operators prefixed by `^` such as `^+`, `^-`, etc. for words.

5.7.3 Implementation Language. Our framework is implemented using a mix of tactic scripts and lemmas and definitions in Gallina (Coq's specification language) that are specifically tailored to work well with our tactic scripts. For compatibility with other code from the Bedrock2 ecosystem, we refrain from modifying Coq itself (even though such modifications might have simplified certain parts); and for easy compatibility with new Coq versions, we refrain from writing any OCaml plugins, because Coq's OCaml API tends to change with each Coq version. The tactics are implemented in a mix of Ltac1 and Ltac2.

5.7.4 Ltac1 vs Ltac2: When to Prefer an Untyped Language With Undocumented Semantics. Ltac1 is an untyped language without clearly specified semantics. For instance, whether a variable refers to a binder declared in Ltac, to a binder declared in a Gallina term quoted inside Ltac code, or to the name of a hypothesis in the current proof context is decided at runtime, in a barely documented manner. It can also happen that a thunk being passed to a function and meant to be evaluated lazily can accidentally be evaluated eagerly. Another common source of surprises is that whether a tactic is a pure function returning a term or an imperative program modifying the current proof goal is also decided at runtime.

Ltac2 addresses these shortcomings by being a typed language with straightforward call-by-value semantics and unambiguous quotation mechanisms to make it clear what variables refers to. In addition, it offers some low-level APIs that Ltac1 does not have.

Given this situation, one might expect that the unambiguously preferred choice for the whole framework would be Ltac2. However, this is not the case in our experience:

First, even though Ltac2 has been developed over several years now, it still lacks support for many tasks that can be done in Ltac1 much more easily, so when writing Ltac2, a considerable amount of time is spent working around non-fundamental limitations related to not-yet-implemented features. And second, Ltac1 code is exceptionally concise, in a manner that really matters: In our experience, there seems to be a certain verbosity threshold below which a tactic programmer can read and understand tactic code very quickly and easily, and Ltac1 is the only programming language we know to be below this verbosity threshold. The reason for Ltac2 often being above it seems to be on one hand that it is typed and more principled, i.e. it requires being explicit about many things that are implicit in Ltac1; and on the other hand, that less effort has been spent yet on defining concise notations for Ltac2. We are curious to see how future evolution of Ltac2 affects these considerations.

For now, we use a mix of Ltac1 and Ltac2, preferring Ltac2 for bigger, more complex functions, where the benefit of catching errors before tactic runtime is considerable, and for situations where low-level term APIs are needed.

6 EVALUATION

6.1 Scope of Sample Programs

We used our tool to verify a few sample functions listed in Table 2, trying to cover an interesting set of low-level memory-handling patterns. It includes splitting a byte buffer into a linked list of free blocks in the init function of a simple malloc library with a fixed block allocation size, lookup and insertion functions for a binary search tree and a crit-bit tree [Bernstein 2006] where we exploit the pre-/postcondition loop verification style by Tuerk [2010] to avoid the need for "tree-with-a-hole" predicates, passing record fields and subarrays to functions with automatic splitting of the callers' record or array predicates, and functions with up to three if-then-else constructs.

The crit-bit tree example shows that we can also support data structures with more involved validity constraints, at the expense of more manual proof lines, though we believe that a more automated proof style could reduce the number of lines of proof. This example also provides a datapoint on usability of our framework, because the example was developed by an undergraduate student who did not participate in the development of the framework and had started to learn Coq less than three months before completing this proof of crit-bit lookup and insert functions.

6.2 Qualitative Discussion of Loop-Invariants-as-Diff Approach

As illustrated by the example in § 3.1.7, in our framework, users express loop invariants as diffs from symbolic state before loops. Table 3 shows why we prefer this middle ground over the two extremes in the design space, manually spelled-out invariants or automatically inferred invariants.

By robustness, we mean how likely it is that after a small modification of the program, the proof still works. Manually spelled-out loop invariants are very likely to require some update after a program modification, whereas an invariant expressed as a diff that just encodes the insight

Table 2. Statistics on our case studies. The two numbers in the Loops column indicate the number of invariant-based loop proofs and the number of of pre-/post-based loop proofs, respectively.

File	Funcs	Snippets	Lines	Time[s]	Loops
`onesize_malloc`	3	24	345	20.25	1 + 0
`tree_set`	4	66	389	73.63	0 + 2
`swap`	2	10	44	3.77	–
`swap_record_fields`	2	6	83	4.00	–
`fibonacci`	1	17	83	8.74	1 + 0
`memset`	1	7	41	9.29	1 + 0
`sort3`	1	22	51	36.31	–
`critbit`	8	122	1881	255.11	2 + 2
`swap_subarrays`	1	3	48	15.77	–
`sort3_separate_args`	1	22	58	22.87	–
`linked_list`	2	16	252	10.06	1 + 1
`nt_uint8_string`	1	11	299	60.76	0 + 1
`min`	3	32	71	6.97	–

and avoids mentioning irrelevant details is more likely to remain applicable. Automated invariant inference tends to be not so stable under modification of the proof context, because the presence of a new but unrelated term might send the invariant search down a wrong path, so that an invariant that was found within reasonable time before the program change might time out after the change.

By proof performance, we mean the running time it takes to produce and check the correctness proof. Executing the diff script corresponds to proving that the symbolic state before the loop implies the loop invariant, i.e. proof work that any framework needs to do, so we do not count it.

The expressivity of the manual and the diff approach is maximal, because any invariant expressible in the logic can be used, whereas in the automatic approach, only those that the heuristics find within a reasonable time limit can be used.

Another advantage of our approach (shared with the approach of manually providing loop invariants) is that we can display a symbolic state at any point inside the loop body even if the loop body has not been completely written yet or some parts of the proof fail because of a bug in the code or because of a missing hint or tweak. In contrast, the fully automatic approach only knows that it picked a reasonable loop invariant if the correctness proof of the whole loop worked out.

Currently, the star ratings in Table 3 are not based on measurements but on anecdotal evidence, so it is cautious to view them as conjectures. In the future, we hope to back them up with measurements, but currently, our framework is still in an early prototype phase where most new examples that we verify point us to some bugs and limitations in the framework that we fix on the fly, but for a meaningful evaluation, one should not make fixes to the framework while evaluating it.

6.3 Some Statistics

Some file-by-file statistics are shown in Table 2. The first column lists the number of functions in each file, and the second lists the number of snippets, which typically corresponds to the number of lines of C code. The total number of lines of each file (third column) is much bigger, because the files also contain specifications, definitions needed to state the specifications, helper lemmas, file-specific proof automation and hints, as well as proof code interspersed between the C snippets.

Table 2 also shows the total time Coq takes to verify each file. Typically, processing each snippet takes just a couple of seconds, and in our experience, it is just right below the threshold of what is

Table 3. Tradeoffs in the design space around loop-invariant automation. Conjectured ratings on a one-star (worst) to three-star (best) rating scale.

	manual	by diff	automatic
verbosity	★	★★	★★★
robustness	★	★★	★★
proof performance	★★★	★★★	★
fully expressive	✓	✓	✗
can display state	✓	✓	✗
total	5★ + 2✓	7★ + 2✓	6★ + 0✓

bearable for interactive development (and whenever it exceeded that perceived threshold, we spent more effort on speeding up the proof automation).

The final column shows the number of loops in each file, expressed as $x + y$, where x is the number of loops proven with an invariant expressed as a diff script from the symbolic state before the loop, and y is the number of loops proven with a family of pre/postcondition pairs (in the style popularized by Tuerk [2010]) by expressing the precondition as a diff script and automatically generalizing the function's postcondition to use it as the loop's postcondition.

So far, our experience seems to confirm our conjectures from Table 3. Once our framework has matured to a point where we do not anymore feel compelled to make framework improvements with every new sample program, we plan to evaluate our conjectures from Table 3 more rigorously.

7 RELATED WORK

Dafny [Leino 2013, 2017] is a high-level programming language with a specification language and SMT-based, highly automated proving of verification conditions. The development experience is very interactive, as the IDE continuously checks the verification conditions. Our framework is still far from reaching the level of automation of Dafny but does have a few advantages over Dafny:

- It allows to reason about (a subset of) C, which is more low-level and more efficient, and can reason about low-level operations like casting byte arrays to records.
- Users can extend the proof automation with domain-specific verification procedures.
- By repeatedly invoking our `step` tactic, users can watch how our system solves side conditions and can easily debug cases where our solver fails.
- The correctness of the tool itself need not be trusted, only Coq's kernel, which is much smaller than Coq's tactic system and our tool's tactics, and also much smaller than the Dafny tool and the SMT solver it uses.
- Finally, and perhaps most importantly, our tool can provide a concise representation of everything the prover knows, in the form of the proof context (list of hypotheses) of Coq's current proof goal. We believe that such a concise summary of all known facts is similar to what attentive programmers need to keep track of in their minds while programming, so displaying it on-screen can assist the programmer. In Dafny, there is no such representation, and the only way to find out whether the prover knows a given fact is to write it down as an assertion at the program point in question and see if Dafny can prove it.

VeriFast [Jacobs et al. 2011] is a separation-logic-based C verification tool. Its symbolic debugger can display the current symbolic state to the user at any program point, and users can affect the symbolic state by invoking lemma functions in ghost code (comments) in the source program. VeriFast is implemented in OCaml. As far as we know, there is no easy way to add domain-specific verification automation on a per-function or per-module basis, while our own approach provides various Ltac hooks and hint databases that users can extend and provides smooth integration between framework code and user code, because both are written in the same language (Ltac).

Boogie, the intermediate verification language powering Dafny, used to have a verification debugger [Le Goues et al. 2011] providing counterexamples for failed verification conditions. However, it appears that it was not popular enough to be maintained, and was eventually removed from the codebase [Qadeer 2020]. In the design space between automatically inferred and manually spelled-out loop invariants, Boogie chooses an interesting middle ground: It infers some simple loop invariants and combines them with those written explicitly by the user [Barnett et al. 2006].

Rupicola [Pit-Claudel et al. 2022] and its predecessor Fiat [Delaware et al. 2015] are extensible, user-configurable compilers from functional programs written in Coq to Bedrock2 and Bedrock1 code, respectively. The user specifies the compilation strategy and lets the framework derive the code accordingly. The Isabelle Refinement Framework [Lammich 2015, 2017] applies similar techniques in Isabelle/HOL. In contrast, our framework is designed for users who already have a clear idea of what low-level code they want and feel that configuring the compiler until it emits the desired low-level code would be more work than just writing down the code.

The Verified Software Toolchain (VST) [Cao et al. 2018] is a tool based on Hoare logic and separation logic, implemented in Coq, for proving correctness of C programs. It uses a similar style of stepping through a program line-by-line, using Coq's context of hypotheses to keep track of the symbolic state. Instead of using Hoare triples $\{P\}c\{Q\}$ like VST, we use wp judgments of the form $\forall s.\ P\ s \Rightarrow \mathsf{wp}\ c\ s\ Q$, so the precondition is already separated and can more easily be moved into Coq's context of hypotheses. In VST, one has to recompile the source program and reload the whole proof each time one wants to change the source program, and sometimes, it is hard to relate the positions in the proof script to positions in the source code.

Like Bedrock2 (which our Live Verification framework targets), CakeML can also be used to create end-to-end-verified software-hardware stacks [Lööw et al. 2019]. All their program verification happens at the ML level, whereas we believe that certain performance-critical pieces of software need to be written in more imperative and low-level languages, which is the subject of our paper.

Why3 [Bobot et al. 2015; Filliâtre and Paskevich 2013] is a tool for interactive development and verification of programs. It provides a programming and specification language called WhyML and can also be used as an intermediate language to verify C, Java, and Ada programs. It discharges its verification conditions to automated as well as interactive external theorem provers.

CAPS [Chaudhari and Damani 2014, 2015], which stands for Calculational Style of Programming, uses a tactic-based approach to derive programs from specifications and uses Why3 as its backend.

8 CONCLUSION AND FUTURE WORK

We have presented a tool for verifying low-level programs using the Coq proof assistant, in a way that continually provides a concise representation of the current symbolic state as the user writes the program. Additionally, our tool stands out by its support for diff-based loop invariants, its option to allow users to extend the proof automation with domain-specific procedures, its small trusted code base that does not include the tool itself, and its compatibility with the Bedrock2 ecosystem that enables end-to-end proofs, which also check that the assumptions that the different tools make about each other are compatible.

It seems to us that the size of the biggest case study in Bedrock2 [Erbsen et al. 2021] was mostly bottlenecked by the lack of automation and usability of the program logic. Similar limitations apply to other Coq-based C verification tools like e.g. VST [Cao et al. 2018] as well. With our live-verification framework, we hope to make a step towards more convenient verification of low-level code in Coq, eventually enabling bigger end-to-end verified stacks.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants CNS-2130671, CCF-2217064, and CCF-2313023 and gifts from Amazon and Google.

DATA-AVAILABILITY STATEMENT

All our code is available at https://github.com/mit-plv/bedrock2/tree/LiveVerifPLDI24.
An artifact [Gruetter et al. 2024] associated with this paper was evaluated and is available at https://doi.org/10.5281/zenodo.10806323.

A LISTING OF NOTATIONS

In this glossary, we give semi-formal definitions of all notations used in the code examples. For each notation (or group of notations), we point to a section in the paper and also indicate the Coq file where the precise definition can be found. Snippets in `typewriter` font are literal parts of the syntax of the notation, while names in *italics* are placeholders.

Specification of a function § 3.1.3, `LiveVerif/src/LiveVerif/LiveProgramLogic.v`

```
.**/ uintptr_t fname(uintptr_t arg1, ..., uintptr_t argN) /**#
       ghost_args := ghost1 ... ghostM;
       requires t m := Pre;
       ensures t' m' r := Post #**/  /**.
```

The first line of this notation is chosen to be compatible with C function signatures. Note that since `uintptr_t` is currently the only supported type, it is a literal part of the notation and cannot be replaced by anything else. $arg_1, \dots arg_N$ are the function argument names, and $ghost_1, \dots ghost_M$ are ghost arguments (whose type is usually inferred by Coq, but can also be annotated explicitly like in $(ghost_i \colon\ MyType_i)$, where $MyType_i$ can be any Coq type). *Pre* is the precondition, and it can refer to the binders t and m, which stand for the event trace and memory before the function call. *Post* is the postcondition, and in addition to t and m, it can also refer to t' and m', which stand for the event trace and memory after the function call, and if the function has a return value, to a binder r representing that return value. It roughly expands to the following formula, which is parameterized over the function environment e:

$$\lambda e.\, \forall arg_1 \dots arg_N\ ghost_1 \dots ghost_M\ t\ m\,,\ Pre \Rightarrow \mathsf{wp}\ e\ fname(arg_1, \dots, arg_N)\ t\ m\ (\lambda\ t'\ m'\ r\,.\, Post)$$

There's also a variant of this notation with a `void` return type, where the `ensures` clause only takes $t'\ m'$ instead of $t'\ m'\ r$.

Providing the next C snippet § 3.1.5, `LiveVerif/src/LiveVerif/LiveProgramLogic.v`

`.**/` *snippet* `/**.`

a tactic notation to provide the next snippet of C code. Typically, we write the `.**/` at the end of the preceding line, and fill in whitespace between *snippet* and the `/**.` The *snippet* is parsed using a custom grammar for C snippets, and passed as an argument to the tactic `next_snippet`, which applies the appropriate weakest-precondition rule, and then repeatedly applies the `step` tactic to solve the side conditions.

Providing the next C snippet without automation § 4.8.3, `LiveProgramLogic.v`,

`.**/` *snippet* `/*?.`

the same as `.**/` *snippet* `/**.`, but without running the `step` tactic

Bullet-point list of separation-logic clauses § 3.1.3, `bedrock2/src/bedrock2/SepBulletPoints.v`

`<{ *` $clause_1$
`...`
`*` $clause_{N-1}$
`*` $clause_N$ `}>`
	stands for `(sep` $clause_1$ `... (sep` $clause_{N-1}$ $clause_N$`) ...)`, where `sep` is the standard separating conjunction (usually written as $*$ in the literature).

Bullet-point list of sepapp clauses § 4.1.2, 4.2, `bedrock2/src/bedrock2/SepappBulletPoints.v`

`<{ +` $clause_1$
`...`
`+` $clause_{N-1}$
`+` $clause_N$ `}>`
	stands roughly for `(sepapp` $clause_1$ `... (sepapp` $clause_{N-1}$ $clause_N$`) ...)`, where `sepapp` means *separating append* (§ 4.1.2)

Marker hypotheses § 4.4, `LiveVerif/src/LiveVerif/PackageContext.v`

`____` *scopekind* `____`	in a proof goal, marks the beginning of a scope of kind *scopekind*

Referring to hypotheses by pattern Figure 3c, `bedrock2/src/bedrock2/find_hyp.v`

`#(`*pattern*`)`	returns the name of the first hypothesis that matches *pattern*, which may contain ?? placeholders

Heapletwise separation logic § 3.1.8, `bedrock2/src/bedrock2/HeapletwiseHyps.v`

m_1 `\*/` m_2	disjoint union between two heaplets (pieces of memory)
m `|=` P	heaplet m satisfies predicate P, defined as $(P\ m)$

Notations for bounded integers § 3.1.3, 5.7.2, `bedrock2/src/bedrock2/WordNotations.v`

`\[`w`]`	interprets a `word` w as an unsigned integer, returning a $\mathbb{Z}$
`/[`z`]`	coerces a $\mathbb{Z}$ into a `word` by dropping the more significant bits
`^+` `^-` `^*`	addition, subtraction, multiplication on `word`

List notations § 3.1.4, `deps/coqutil/src/coqutil/Datatypes/ZList.v`

ℓ`[`i`]`	i-th element of list ℓ
ℓ`[`i `:=` x`]`	ℓ with its i-th element replaced by x
ℓ`[:`i`]`	the first i elements of ℓ
ℓ`[`i`:]`	ℓ with its first i elements skipped
ℓ`[`i`:`j`]`	the slice of ℓ from index i (inclusive) to index j (exclusive)
`[|`x_1`;` x_2`; ...;` x_N`|]`	list literal
h `::` t	list cons
a `++` b	list append

Potentially less-well-known standard Coq notations

`~`P	negation of proposition P, defined as $P \to \bot$
`?`x	an evar (see § 2.2)
a `^` b	a raised to the b-th power

```
#[export] Instance spec_of_safeCopySlice: fnspec :=                                .**/

uintptr_t safeCopySlice(
  uintptr_t src, uintptr_t srcOfs, uintptr_t srcLen,
  uintptr_t unsafeN,
  uintptr_t dst, uintptr_t dstOfs, uintptr_t dstLen
) /**#
  ghost_args := srcData dstData dstUninit (R: mem ⟶ Prop);
  requires t m :=
    <{ * array (uint 8) \[srcLen] srcData src
       * array (uint 8) \[dstLen] (dstData ++ dstUninit) dst
       * R }> m ∧
    \[srcOfs] <= \[srcLen] ∧
    len dstData = \[dstOfs];
  ensures t' m' r := t' = t ∧
   ((r = /[0] ∧
     <{ * array (uint 8) \[srcLen] srcData src
        * array (uint 8) \[dstLen] (dstData ++ dstUninit) dst
        * R }> m') ∨
    (r = /[1] ∧
     <{ * array (uint 8) \[srcLen] srcData src
        * array (uint 8) \[dstLen] (dstData ++
                                    srcData[\[srcOfs]:][:\[unsafeN]] ++
                                    dstUninit[\[unsafeN]:]) dst
        * R }> m')) #**/                                                      /**.
Derive safeCopySlice SuchThat (fun_correct! safeCopySlice) As safeCopySlice_ok.    .**/
{                                                                             /**. .**/
  if (unsafeN <= srcLen - srcOfs && unsafeN <= dstLen - dstOfs) /*split*/ {   /**. .**/
    Memcpy(dst + dstOfs, src + srcOfs, unsafeN);                              /**. .**/
    return 1;                                                                 /**. .**/
  }                                                                           /**. .**/
  else {                                                                      /**. .**/
    return 0;                                                                 /**. .**/
  }                                                                           /**. .**/
}                                                                             /**.
Qed.
```

Fig. 8. Specification and implementation of safeCopySlice

B SAMPLE FUNCTION USED TO ILLUSTRATE ACTIONABLE ERROR MESSAGES

We give the specification as well as the final, correct implementation of safeCopySlice in Figure 8. Note that all the proof steps are completely automated, including the splitting of the source and destination arrays before the Memcpy call, pasting them back together after the call, and matching that result to the desired postcondition.

REFERENCES

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. Vol. 4111. Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Daniel J. Bernstein. 2006. Crit-Bit Trees. https://cr.yp.to/critbit.html

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer* 17, 6 (Nov. 2015), 709–727. https://doi.org/10.1007/s10009-014-0314-5

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (March 2023), 5:1–5:43. https://doi.org/10.1145/3579834

Dipak L. Chaudhari and Om Damani. 2014. Automated Theorem Prover Assisted Program Calculations. In *Integrated Formal Methods*, Elvira Albert and Emil Sekerinski (Eds.). Vol. 8739. Springer International Publishing, Cham, 205–220. https://link.springer.com/10.1007/978-3-319-10181-1_13

Deepak L. Chaudhari and Om P. Damani. 2015. CAPS, A Calculational Assistant for Programming from Specifications. https://www.cse.iitb.ac.in/~dipakc/CAPS/

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL 2015*. ACM, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. *PLDI'21* (2021). https://doi.org/10.1145/3453483.3454065

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Code Artifact for Live Verification in an Interactive Proof Assistant. https://doi.org/10.5281/zenodo.10806323

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*. Vol. 6617. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. http://link.springer.com/10.1007/978-3-642-20398-5_4

Peter Lammich. 2015. The Isabelle Refinement Framework. *Kolloquium Programmiersprachen* (2015). https://www.complang.tuwien.ac.at/kps2015/proceedings/KPS_2015_submission_13.pdf

Peter Lammich. 2017. Refinement to Imperative HOL. *Journal of Automated Reasoning* 62, 4 (2017), 481–503. https://doi.org/10.1007/s10817-017-9437-1

Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. 2011. The Boogie Verification Debugger (Tool Paper). In *Software Engineering and Formal Methods (Lecture Notes in Computer Science)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer, Berlin, Heidelberg, 407–414. https://doi.org/10.1007/978-3-642-24690-6_28

K. Rustan M. Leino. 2013. Developing Verified Programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE)*. 1488–1490. https://doi.org/10.1109/ICSE.2013.6606754

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (Nov. 2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *PLDI 2019*. Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

Clément Pit-Claudel and Thomas Bourgeat. 2021. An Experience Report on Writing Usable DSLs in Coq. In *CoqPL'21: The Seventh International Workshop on Coq for PL*, Assia Mahboubi and Amin Timany (Eds.). https://pit-claudel.fr/clement/papers/koika-dsls-CoqPL21.pdf

Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In *PLDI 2022*. Association for Computing Machinery, New York, NY, USA, 918–933. https://doi.org/10.1145/3519939.3523706

Shaz Qadeer. 2020. ModelViewer and BVD Projects. https://github.com/boogie-org/boogie/issues/293

Thomas Tuerk. 2010. Local Reasoning about While-Loops. *VSTTE 2010* (2010).

Received 2023-11-16; accepted 2024-03-31