Extracting Functional Properties from Verified Imperative Programs

Zixun Guo[1⋆], Yiyuan Cao[1,2⋆], Di Wang[1,2] ✉, and Zhenjiang Hu[1,2]

[1] Key Lab of HCST (PKU), MOE; SCS, Peking University, China
{yiyuan,2100012990}@stu.pku.edu.cn, {wangdi95,huzj}@pku.edu.cn
[2] Zhongguancun Laboratory, Beijing, China

Abstract. Verifying the correctness of imperative programs often requires proving properties of their functional models. Some auto-active verifiers let users supply such proofs as *lemma functions*, written in an effect-free language separate from executable code. We observe that, for proving functional-model properties, such restrictions are unnecessary. The key is a *purification* operation that extracts a pure proposition $\mathit{purify}(A)$ from a heap-dependent assertion A, together with an *extraction* rule showing that any valid total Hoare triple $[P]\,s\,[Q]$ yields the implication $\mathit{purify}(P) \implies \mathit{purify}(Q)$. Such extraction holds for arbitrary programs. We prove extraction sound both semantically, against the interpretation of total Hoare triples, and syntactically, by reflecting each program-logic inference rule into the meta-logic for a minimal heap-manipulating language. Consequently, theorems about functional models can be proved by verifying ordinary imperative programs, with no language-level distinction between proof code and executable code. We mechanize the theoretical results in Lean and illustrate its practical applicability on several worked examples checked in the VeriFast verifier.

Keywords: Program verification · Lemma functions · Hoare logic.

1 Introduction

Verifying the functional correctness of an imperative program often requires reasoning about its *functional model*. To verify a linked-list data structure, for instance, one reasons about the properties of logical functions over the inductive list it represents, such as the associativity of list concatenation:

$$\forall l_1, l_2, l_3.\, \mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3)). \tag{1}$$

The proof of this property proceeds by induction. This kind of reasoning is usually beyond the scope of what the program verifier can perform automatically.

To overcome this limitation, many auto-active verifiers provide ways to let users supply proofs manually [4, 6–8, 13]. For concreteness of illustration, we use

⋆ These authors contributed equally.

```
inductive list = nil | cons(int, list);

fixpoint list app(list l1, list l2) {
  switch (l1) {
    case nil: return l2;
    case cons(h, t): return cons(h, app(t, l2));
  }
}

lemma void app_assoc(list l1, list l2, list l3)
  requires true;
  ensures app(app(l1, l2), l3) == app(l1, app(l2, l3));
{
  switch (l1) {
    case nil:
    case cons(h, t): app_assoc(t, l2, l3);
  }
}
```

Fig. 1. A lemma function `app_assoc` in VeriFast establishing Eq. (1). Definitions of the inductive type `list` and the logical function `app` are provided for completeness.

a state-of-the-art C-program verifier, VeriFast [6], for the examples presented in this paper. VeriFast provides *lemma functions*, which are pieces of *ghost code* carrying no runtime meaning. Once its specification is verified, a lemma function may be invoked during verification of a target C program to make the proved functional property available as an assumption in the proof state. Fig. 1 shows a VeriFast lemma function establishing Eq. (1), with the induction and case analysis written as recursion and a pattern-matching `switch` construct. Here `fixpoint` declares a recursively defined total function, `lemma` marks ghost proof code, and `requires`/`ensures` give its specification.

In current practice, however, the language fragment for writing proof programs is segregated from executable code: (1) lemma functions must be written in an *effect-free* fragment: they may not allocate, mutate the heap, or call verified executable functions, and they must be shown to terminate. (2) lemma functions may use data types and special constructs unavailable to executable code, such as the pattern matching construct `switch` over values of inductive data types seen in Fig. 1. As a result, lemma functions require a different language fragment from executable programs, and verifying them requires the verifier to implement dedicated support.

This work. We observe that, for the purpose of proving properties of the functional model, these language-level extensions and restrictions on lemma functions are unnecessary. The proof-carrying nature of such code does not depend on it being effect-free ghost code invoked in the target program: one can extract functional-model theorems from *any* valid total Hoare triple.

The key is a *purify* operation that extracts a pure proposition from any heap-dependent assertion. It is defined semantically as

$$\mathit{purify}(A) = \exists h.\, [\![A]\!]\, h$$

asserting that *some* heap satisfies A, while forgetting the exact contents of that witness heap. We show that any valid total Hoare triple $[P]\, s\, [Q]$ yields an implication between the purifications of its precondition and postcondition, a rule we call *extraction*:

$$\frac{[P]\, s\, [Q]}{\mathit{purify}(P) \implies \mathit{purify}(Q)}\ \textsc{Extraction}$$

Extraction holds semantically for arbitrary valid total triples, with no restriction on s, because the program plays a purely existential role, much like a Skolem function: a total Hoare triple guarantees that every heap satisfying P is taken by s to some heap satisfying Q, so the mere existence of a P-heap witnesses the existence of a Q-heap. The program serves only as a *witness* that the transformation exists; its effects are irrelevant to the extracted proposition.

Consequently, a functional-model property such as Eq. (1) can be proved without special support from the verifier. In place of the lemma function of Fig. 1, one may verify an ordinary executable program operating on in-memory linked lists with a suitable specification. An example is shown in Fig. 2. It uses the following VeriFast notation: `?x` captures and creates an existential binding variable, `&*&` is separating conjunction, `p->f |-> v` owns field `f` with value `v`, and `open`/`close` unfold and fold predicates. It intentionally serves as a proof program: although it only traverses `p1` and need not perform useful computations, it is written in the executable language rather than the lemma-function fragment. The representation predicate `sll` relates the functional model of the linked list to its in-memory representation. This function is verified in the same way as for other executable functions. Extraction turns its successful verification result (assuming totality checking is enabled) into an implication between the purifications of its precondition and postcondition. A few laws for *purify* (cf. Section 3) expose the intended equation in the consequent; discharging the antecedent about the in-memory lists relies on a few reasonable *representability* conditions, which we discuss in Section 5.1.

Contributions. This paper studies the idea of extracting theorems from verified imperative programs. We make the following contributions.

- We define the *purify* operation and the *extraction* rule, which together extract a pure implication from any valid total Hoare triple. Soundness of the extraction rule is justified semantically against the interpretation of total Hoare triples, independent of any particular verification approach or verifier (Section 3).
- We give a proof-theoretic account of extraction for a minimal imperative language manipulating heaps: each program-logic inference rule is mirrored

```
struct node { struct node *next; int value; };

predicate sll(struct node *p, list l) =
  p == 0 ? l == nil :
  p->next |-> ?q &*& p->value |-> ?h &*& sll(q, ?t) &*& l == cons(h, t);

void exec_app_assoc(struct node* p1, struct node* p2, struct node* p3)
    //@ requires sll(p1, ?l1) &*& sll(p2, ?l2) &*& sll(p3, ?l3);
    /*@ ensures sll(p1, l1) &*& sll(p2, l2) &*& sll(p3, l3) &*&
                app(app(l1, l2), l3) == app(l1, app(l2, l3)); @*/
{
  if (p1 == 0) {
    //@ open sll(p1, l1);
    //@ close sll(p1, nil);
  } else {
    //@ open sll(p1, l1);
    struct node* q = p1->next;
    exec_app_assoc(q, p2, p3);
    //@ close sll(p1, l1);
  }
}
```

Fig. 2. A verified executable function `exec_app_assoc` whose extraction, together with representability of `sll` (Section 5.1), entails Eq. (1).

by a fixed pattern of meta-logic reasoning with a fixed set of purify laws, giving a structure-preserving translation of proofs (Section 4).
- We apply extraction to several worked examples checked with VeriFast, in which the proof structure is intrinsically connected to the program structure. The key technical step involved in turning the extracted implication into the intended property is a *representability* condition (Section 5).

The Lean mechanization of the theoretical results as well as the example programs checked in VeriFast are given in the online supplementary material [5].

2 Assertion Logic and Semantic Judgments

This section fixes the assertion logic and the semantic judgments used throughout the rest of the paper. We use separation logic because it describes the in-memory data structures of our examples concisely and locally. We define the meaning of total Hoare triples assuming a standard big-step semantics, abstracting away from the details of the language.

2.1 Assertion Syntax

We work in a standard separation logic built on top of a *meta-logic* (some higher-order logic), in which expression terms and pure propositions live. The

separation-logic assertions (of type **HProp**) used in this paper include the following constructs:

$$P, Q ::= \ulcorner \phi \urcorner \mid \mathsf{emp} \mid e_1 \mapsto e_2 \mid P * Q \mid P \vee Q \mid \forall x.\, P \mid \exists x.\, P \mid \cdots$$

where $\ulcorner \phi \urcorner$ lifts a pure proposition $\phi : \mathbf{Prop}$ of the meta-logic into an assertion, emp describes the empty heap, $e_1 \mapsto e_2$ denotes a singleton heap cell at address expression e_1 holding content e_2, and $P * Q$ is the separating conjunction.

2.2 Assertion Semantics

We interpret separation logic in the standard heap model: a heap $h : \mathbf{Heap}$ is a finite partial function from an infinitely countable address space to values (the specific choices of address space and value space are irrelevant to our topic); an assertion $P : \mathbf{HProp}$ denotes a predicate $[\![P]\!] : \mathbf{Heap} \to \mathbf{Prop}$. User-defined inductive predicates have the usual least-fixed-point interpretation. The other constructs are interpreted by induction on the structure of P:

$$\begin{aligned}
[\![\ulcorner \phi \urcorner]\!]\, h &\triangleq \phi \wedge h = \emptyset && (\phi : \mathbf{Prop}) \\
[\![\mathsf{emp}]\!]\, h &\triangleq h = \emptyset \\
[\![e_1 \mapsto e_2]\!]\, h &\triangleq h = [e_1 \mapsto e_2] \\
[\![P * Q]\!]\, h &\triangleq \exists h_1, h_2.\, h = h_1 \uplus h_2 \wedge [\![P]\!]\, h_1 \wedge [\![Q]\!]\, h_2 \\
[\![P \vee Q]\!]\, h &\triangleq [\![P]\!]\, h \vee [\![Q]\!]\, h \\
[\![\forall x.\, P]\!]\, h &\triangleq \forall v.\, [\![P[v/x]]\!]\, h \\
[\![\exists x.\, P]\!]\, h &\triangleq \exists v.\, [\![P[v/x]]\!]\, h
\end{aligned} \tag{2}$$

Here $\emptyset$ is the empty heap (a partial function defined nowhere), $[e_1 \mapsto e_2]$ is the singleton heap that maps address e_1 to value e_2 and leaves all other addresses undefined, and $h_1 \uplus h_2$ is the disjoint union of h_1 and h_2, defined only when $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Higher-order logic connectives are interpreted by pointwise lifting, as usual.

Entailment between assertions, written $P \vdash Q$, is not itself an assertion but a proposition of the meta-logic, interpreted as pointwise implication:

$$P \vdash Q \triangleq \forall h.\, [\![P]\!]\, h \implies [\![Q]\!]\, h\ .$$

2.3 Program Semantics and Hoare Triples

For most of the paper, we do not fix concrete syntactic constructs (except in Section 4), and assume only a notion of evaluation and total Hoare triples over heaps.

Remark 1. For simplicity of formal development, we do not model mutable local variables directly: variables in assertions are immutable meta-logic variables, so each denotes the same value in the pre- and postcondition. A mutable local x can instead be represented by a reserved heap cell at a fixed address a_x, with assignment to x encoded as a heap operation on a_x.

Big-step semantics. We use a standard big-step semantics, written

$$\langle s, h\rangle \Downarrow h'$$

read as: starting from the initial heap h, the statement s terminates with a final heap h'.

Total Hoare triples. We write $[P]\, s\, [Q]$ for a *total* Hoare triple with precondition P, statement s, and postcondition Q. A triple is valid if every initial heap satisfying P is taken by s to some final heap satisfying Q:

$$[P]\, s\, [Q] \triangleq \forall h : \mathbf{Heap}.\ [\![P]\!]\, h \implies \exists h'.\ \langle s, h\rangle \Downarrow h' \wedge [\![Q]\!]\, h'\ . \tag{3}$$

This definition assumes deterministic evaluation (modulo the address allocation choices), for simplicity. [3]

3 Purification and Extraction

This section introduces our two central notions, *purification* and the *extraction* rule, presents the basic laws of purification, and proves them sound semantically.

3.1 The Purification Operation

The purification operation maps a heap-dependent assertion to a pure proposition that holds exactly when the assertion is satisfied by *some* heap.

Definition 1. *The purification operation* $\mathit{purify} : \textbf{\textit{HProp}} \to \textbf{\textit{Prop}}$ *is defined by*

$$\mathit{purify}(P) \triangleq \exists h.\ [\![P]\!]\, h. \tag{4}$$

Proposition 1 (Purification Laws). *For all* $P, Q : \textbf{\textit{HProp}}$ *and all pure propositions* $\phi : \textbf{\textit{Prop}}$*, purification satisfies the following algebraic laws, which intuitively let us push it around the logical connectives:*

$$\begin{aligned}
\mathit{purify}(\ulcorner \phi \urcorner) &\iff \phi && \textsc{Purify-Pure}\\
\mathit{purify}(\mathsf{emp}) &\iff \mathit{true} && \textsc{Purify-Emp}\\
\mathit{purify}(e_1 \mapsto e_2) &\iff \mathit{true} && \textsc{Purify-Points-To}\\
\mathit{purify}(P * Q) &\implies \mathit{purify}(P) \wedge \mathit{purify}(Q) && \textsc{Purify-Star}\\
\mathit{purify}(P \vee Q) &\iff \mathit{purify}(P) \vee \mathit{purify}(Q) && \textsc{Purify-Or}\\
\mathit{purify}(\forall x.\, P(x)) &\implies \forall x.\, \mathit{purify}(P(x)) && \textsc{Purify-Forall}\\
\mathit{purify}(\exists x.\, P(x)) &\iff \exists x.\, \mathit{purify}(P(x)) && \textsc{Purify-Exists}
\end{aligned}$$

together with the entailment rule

$$\frac{P \vdash Q}{\mathit{purify}(P) \implies \mathit{purify}(Q)}\ \textsc{Purify-Entails}$$

[3] Our results can be generalized to nondeterministic settings, e.g., by using an *omni-bigstep* semantics [2].

Proof. Each law follows by unfolding Eq. (4) against the interpretation in Eq. (2); Purify-Unfold also unfolds the definition $\mathcal{I}(\bar{x}) \triangleq A_{\mathcal{I}}$. For example, for Purify-Star, we have:

$$\begin{aligned} \mathit{purify}(P * Q) &= \exists h.\, [\![P * Q]\!]\, h \\ &\iff \exists h, h_1, h_2.\, h = h_1 \uplus h_2 \wedge [\![P]\!]\, h_1 \wedge [\![Q]\!]\, h_2 \\ &\implies (\exists h_1.\, [\![P]\!]\, h_1) \wedge (\exists h_2.\, [\![Q]\!]\, h_2) = \mathit{purify}(P) \wedge \mathit{purify}(Q) \end{aligned}$$

where the last step simply drops the disjointness constraint $h = h_1 \uplus h_2$. □

The two laws Purify-Star and Purify-Forall hold only in the forward direction. The converse of Purify-Star fails because the disjointness constraint cannot be recovered. The converse of Purify-Forall fails because in general one cannot switch the order of an existential quantifier with a preceding universal quantifier.

3.2 The Extraction Rule

The extraction rule turns any valid total triple into a pure implication between the purifications of its precondition and postcondition, eliminating the program s regardless of its possible heap effects.

Definition 2. *The extraction rule is*

$$\frac{[P]\, s\, [Q]}{\mathit{purify}(P) \implies \mathit{purify}(Q)}\ \textsc{Extraction}$$

Theorem 1 (Soundness of Extraction). *The extraction rule is sound: for any total triple $[P]\, s\, [Q]$ interpreted as in Eq. (3), we have $\mathit{purify}(P) \implies \mathit{purify}(Q)$.*

Proof. Assume $[P]\, s\, [Q]$ and $\mathit{purify}(P)$. By Definition 1, pick h_0 with $[\![P]\!]\, h_0$. By Eq. (3) there is a final heap h_1 with $\langle s, h_0\rangle \Downarrow h_1$ and $[\![Q]\!]\, h_1$. This h_1 witnesses $\exists h.\, [\![Q]\!]\, h$, that is, $\mathit{purify}(Q)$. □

3.3 Discussion: Extraction and Skolemization

The conclusion of Definition 2 reads

$$(\exists h_1.\, [\![P]\!]\, h_1) \implies (\exists h_2.\, [\![Q]\!]\, h_2).$$

Pulling the existential quantifiers out of the implication's antecedent and consequent, we get the equivalent proposition:

$$\forall h_1.\, \exists h_2.\, [\![P]\!]\, h_1 \implies [\![Q]\!]\, h_2.$$

Assuming the usual choice principle, Skolemizing the existential yields the following view:

$$\exists f.\, \forall h_1.\, [\![P]\!]\, h_1 \implies [\![Q]\!]\, f(h_1).$$

Comparing this with the proof of Theorem 1 shows that *the semantics of the program* supplies the Skolem witness f.

4 Syntactic Extraction

Section 3 justified extraction *semantically*, appealing to the semantics of programs and the meaning of total Hoare triples. We now give a *syntactic* account, i.e., for a fixed language, we construct a proof of the extracted implication from the Hoare-logic derivation; unlike the semantic result, this account is relative to the program-logic rules displayed below and needs no operational semantics. Concretely, each program-logic rule is mirrored by a few reasoning steps in the meta-logic, so a program-logic proof reflects into a meta-logic proof of the corresponding purified implication.

This view is more demanding: it commits to a specific set of program-logic rules and purify laws, hence it is harder to generalize than the semantic argument. We therefore work with a minimal language. The payoff is a proof-theoretic reading of extraction and a route toward extracting structured proofs, not merely their existence.

4.1 A Minimal Heap-Manipulating Language

Language Syntax We consider a minimal imperative language with heap manipulation. Its syntax is given by:

$$
\begin{aligned}
e &::= x \mid n \\
\mathit{bop} &::= \mathsf{add} \mid \mathsf{sub} \mid \cdots \\
s &::= \mathsf{malloc}(e) \mid \mathsf{free}(e) \mid \mathsf{store}(e_t, e) \mid \mathsf{copy}(e_s, e_t) \mid \mathit{bop}(e_t, e_1, e_2) \\
&\quad \mid \mathsf{skip} \mid s; s \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{inv}\ P\ \mathsf{dec}\ e_d\ \mathsf{do}\ s
\end{aligned}
$$

Here, an expression e is simply a program variable x or an integer constant n. Statements s include the usual control-flow constructs, as well as the primitive commands: $\mathsf{malloc}(e)$ allocates a cell (initialized to an arbitrary value) and stores its address into the cell at e; $\mathsf{free}(e)$ frees the cell whose address is e; $\mathsf{store}(e_t, e)$ writes the value denoted by e into the cell at e_t; $\mathsf{copy}(e_s, e_t)$ copies the value stored at the cell at e_s into the cell at e_t; $\mathit{bop}(e_t, e_1, e_2)$ performs an arithmetic operation on the values stored at addresses e_1 and e_2, the result of which is stored in the cell at e_t. In the conditional construct if and the loop construct while, the expression e denotes an address; the then branch and loop body are taken when the cell at that address holds zero. The invariant P and the decrementing cell expression e_d (for termination checking) are incorporated as annotations into the loop construct.

Note that this language has no variable-assignment construct and mutability happens only in the heap. Hence program variables have constant values and can serve as parameters and auxiliary variables linking pre- and postconditions (cf. the SL-WHILE rule below). Mutable locals may be represented by reserved heap cells: for fixed addresses a_x and a_y, $x := y$ is encoded as $\mathsf{copy}(a_y, a_x)$ and $x := n$ as $\mathsf{store}(a_x, n)$.

Program-Logic Rules The structural rules of separation logic are given by:

$$\frac{[P]\,s\,[Q]}{[P * R]\,s\,[Q * R]}\ \text{SL-Frame} \qquad \frac{P \vdash P' \qquad [P']\,s\,[Q'] \qquad Q' \vdash Q}{[P]\,s\,[Q]}\ \text{SL-Conseq}$$

For the construct-specific rules, we first define two derived assertions for the result of a conditional test on the value stored at an address e:

$$B_{\mathsf{true}}(e) \triangleq e \mapsto 0 \qquad B_{\mathsf{false}}(e) \triangleq \exists v.\, e \mapsto v * \ulcorner v \neq 0 \urcorner\ .$$

They satisfy the equivalence $e \mapsto - = B_{\mathsf{true}}(e) \vee B_{\mathsf{false}}(e)$, where $e \mapsto -$ abbreviates $\exists v.\, e \mapsto v$. Then we have the construct-specific rules:

$$\frac{}{[P]\,\mathsf{skip}\,[P]}\ \text{SL-Skip} \qquad \frac{[P]\,s_1\,[R] \qquad [R]\,s_2\,[Q]}{[P]\,s_1; s_2\,[Q]}\ \text{SL-Seq}$$

$$\frac{[P * B_{\mathsf{true}}(e)]\,s_1\,[Q] \qquad [P * B_{\mathsf{false}}(e)]\,s_2\,[Q]}{[P * e \mapsto -]\,\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\,[Q]}\ \text{SL-If}$$

$$\frac{[P * B_{\mathsf{true}}(e) * D_s(e_d, N)]\,s\,[P * e \mapsto - * D_n(e_d, N)]}{[P * e \mapsto - * D_s(e_d, N_0)]\,W\,[P * B_{\mathsf{false}}(e) * D_e(e_d, N_0)]}\ \text{SL-While}$$

where $W \equiv \mathsf{while}\ e\ \mathsf{inv}\ P\ \mathsf{dec}\ e_d\ \mathsf{do}\ s$ is the loop, and the variant for ensuring termination is tracked at address e_d as follows:

$$\begin{aligned} D_s(e_d, N) &\triangleq e_d \mapsto N * \ulcorner N \geq 0 \urcorner, \\ D_n(e_d, N) &\triangleq \exists N'.\, e_d \mapsto N' * \ulcorner 0 \leq N' < N \urcorner, \\ D_e(e_d, N) &\triangleq \exists N'.\, e_d \mapsto N' * \ulcorner 0 \leq N' \leq N \urcorner. \end{aligned}$$

The primitive operations are specified by the following triples:

$$[e \mapsto v]\,\mathsf{malloc}(e)\,[\exists a.\, e \mapsto a * a \mapsto -] \qquad [e \mapsto a * a \mapsto -]\,\mathsf{free}(e)\,[e \mapsto a]$$

$$[e_t \mapsto -]\,\mathsf{store}(e_t, e)\,[e_t \mapsto e] \qquad [e_s \mapsto v * e_t \mapsto -]\,\mathsf{copy}(e_s, e_t)\,[e_s \mapsto v * e_t \mapsto v]$$

$$[e_1 \mapsto v_1 * e_2 \mapsto v_2 * e_t \mapsto -]\ bop(e_t, e_1, e_2)\,[e_1 \mapsto v_1 * e_2 \mapsto v_2 * e_t \mapsto v_1 \oplus v_2]$$

Here $bop \in \{\mathsf{add}, \mathsf{sub}\}$, with $\oplus$ the corresponding arithmetic operation.

Extra Rules for Purify Since we adopt a syntactic approach, we do not use the definition of *purify* in this section. Instead, we use only the rules listed in Proposition 1, along with the following extra ones:

$$\begin{aligned} \mathit{purify}(P * e \mapsto -) &\implies \mathit{purify}(P * e \mapsto v) && \text{Purify-Heap-Modify} \\ \mathit{purify}(P) &\implies \exists r.\, \mathit{purify}(P * r \mapsto v) && \text{Purify-Heap-Alloc} \end{aligned}$$

These extra rules are sound for the heap model of Section 2.2 (with an unbounded address space).

4.2 Extracting Proofs from Program-Logic Derivations

Our main result for syntactic extraction is the following theorem:

Theorem 2 (Soundness of Syntactic Extraction). *For every derivable total triple* $[P]\,s\,[Q]$ *in our language, a meta-logic proof of the corresponding purified implication* $\mathit{purify}(P) \implies \mathit{purify}(Q)$ *can be constructed in a structure-preserving way, using the aforementioned purification laws.*[4]

Proof. It is a direct corollary of Lemma 1 (stated and proved below) with the frame set as empty, i.e., $R = \mathsf{emp}$. □

We prove the following strengthened lemma that frames any assertion R from the start:

Lemma 1 (Framed Extraction). *For every derivable total triple* $[P]\,s\,[Q]$ *in our language, a meta-logic proof of* $\mathit{purify}(P * R) \implies \mathit{purify}(Q * R)$ *for arbitrary assertion* R *can be constructed in a structure-preserving way.*

Proof. A full proof of this theorem (mechanized in Lean) is given in the supplementary material. The proof goes by induction on the derivation of $[P]\,s\,[Q]$.

Case: SL-Skip. Here $Q = P$, and the claim $\mathit{purify}(P*R) \implies \mathit{purify}(P*R)$ is immediate by reflexivity of implication.

Case: SL-Seq. From $[P]\,s_1\,[M]$ and $[M]\,s_2\,[Q]$, the induction hypotheses give $\mathit{purify}(P * R) \implies \mathit{purify}(M * R)$ and $\mathit{purify}(M * R) \implies \mathit{purify}(Q * R)$ for every R; compose their proofs by transitivity of implication.

Case: SL-Frame. The conclusion is $[P * R_0]\,s\,[Q * R_0]$ from $[P]\,s\,[Q]$. Using associativity of $*$ and the induction hypothesis with frame $R_0 * R$,

$$\begin{aligned}\mathit{purify}((P * R_0) * R) &\iff \mathit{purify}(P * (R_0 * R))\\ &\implies \mathit{purify}(Q * (R_0 * R)) \iff \mathit{purify}((Q * R_0) * R).\end{aligned}$$

This is exactly the case that forces the lemma to quantify over *all* frames.

Case: SL-Conseq. From $P \vdash P'$, $[P']\,s\,[Q']$, and $Q' \vdash Q$. Monotonicity of $*$ gives $P * R \vdash P' * R$ and $Q' * R \vdash Q * R$, so by Purify-Entails and the induction hypothesis,

$$\mathit{purify}(P * R) \implies \mathit{purify}(P' * R) \implies \mathit{purify}(Q' * R) \implies \mathit{purify}(Q * R).$$

Case: primitive operations. For store, copy, and *bop*, the postcondition differs from the precondition only in the values written to certain cells (e.g., store replaces $e_t \mapsto v$ with $e_t \mapsto e$). After commuting the target cell to the end using associativity and commutativity of $*$, one application of Purify-Heap-Modify replaces the value stored in the target cell with the new one.

For malloc, use a similar pattern with Purify-Heap-Alloc and Purify-Star.

[4] This structure-preserving property can be stated precisely using categorical terms; we leave those technicalities implicit here.

Case: SL-If. Since $e \mapsto - = B_{\mathsf{true}}(e) \vee B_{\mathsf{false}}(e)$, distributing $*$ over $\vee$ and applying Purify-Or,

$$\mathit{purify}((P{*}e \mapsto -){*}R) \iff \mathit{purify}((P{*}B_{\mathsf{true}}(e)){*}R) \vee \mathit{purify}((P{*}B_{\mathsf{false}}(e)){*}R).$$

By the two induction hypotheses each disjunct implies $\mathit{purify}(Q * R)$, hence so does the disjunction.

Case: SL-While. We prove, by well-founded induction on $N_0 \geq 0$, that for every R,

$$\mathit{purify}(P * e \mapsto - * D_s(r, N_0) * R) \implies \mathit{purify}(P * B_{\mathsf{false}}(e) * D_e(r, N_0) * R).$$

Splitting $e \mapsto -$ as above and using Purify-Or, the antecedent yields one of two cases.

- $\mathit{purify}(P * B_{\mathsf{false}}(e) * D_s(r, N_0) * R)$: it corresponds to the guard failing and the loop exiting. We finish the case by the fact that $D_s(r, N_0) \vdash D_e(r, N_0)$.
- $\mathit{purify}(P * B_{\mathsf{true}}(e) * D_s(r, N_0) * R)$: the guard holds and the body runs. The SL-While premise is a subderivation, so its (outer) induction hypothesis applies; instantiating its schematic N to N_0 and framing by R,

  $$\mathit{purify}(P * B_{\mathsf{true}}(e) * D_s(r, N_0) * R) \implies \mathit{purify}(P * e \mapsto - * D_n(r, N_0) * R).$$

  Now $D_n(r, N_0) \vdash D_s(r, N')$ with $N' < N_0$, so the inner (well-founded) induction hypothesis at N' applies, giving $\mathit{purify}(P * B_{\mathsf{false}}(e) * D_e(r, N') * R)$; and $D_e(r, N') \vdash D_e(r, N_0)$ since $N' < N_0$.

In both cases the goal follows, completing the nested induction and the SL-While case. □

Each case of Lemma 1 translates one program-logic rule into a fixed pattern of meta-logic reasoning.

An Example To make the correspondence concrete, consider the following triple derivation. Let $\Psi \triangleq B_{\mathsf{false}}(e) \vee B_{\mathsf{true}}(e)$ (we omit the trivial $\Psi \vdash \Psi$ in the proof):

$$\dfrac{\dfrac{B_{\mathsf{true}}(e) \vdash \Psi \qquad \overline{[\Psi]\,\mathsf{skip}\,[\Psi]}}{[B_{\mathsf{true}}(e)]\,\mathsf{skip}\,[\Psi]} \qquad \dfrac{B_{\mathsf{false}}(e) \vdash \Psi \qquad \overline{[\Psi]\,\mathsf{skip}\,[\Psi]}}{[B_{\mathsf{false}}(e)]\,\mathsf{skip}\,[\Psi]}}{[e \mapsto -]\,\mathsf{if}\ e\ \mathsf{then\ skip\ else\ skip}\,[\Psi]}$$

The program does nothing; the proof merely repackages the precondition. The two entailment side conditions hold by the separation-logic rules $A \vdash A \vee B, B \vdash A \vee B$.

The construction process of Theorem 2 mirrors the derivation above into a meta-logic proof of $\mathit{purify}(e \mapsto -) \implies \mathit{purify}(\Psi)$:

$$\dfrac{\dfrac{B_{\mathsf{true}}(e) \vdash \Psi}{\mathit{purify}(B_{\mathsf{true}}(e)) \implies \mathit{purify}(\Psi)} \qquad \dfrac{B_{\mathsf{false}}(e) \vdash \Psi}{\mathit{purify}(B_{\mathsf{false}}(e)) \implies \mathit{purify}(\Psi)}}{\mathit{purify}(B_{\mathsf{true}}(e)) \vee \mathit{purify}(B_{\mathsf{false}}(e)) \implies \mathit{purify}(\Psi)}}{\mathit{purify}(e \mapsto -) \implies \mathit{purify}(\Psi)}$$

The top leaves are applications of Purify-Entails, mirroring the two SL-Conseq premises. The mid-level combines the two branch implications via the meta-logic rule $(A \implies C) \wedge (B \implies C) \to (A \vee B \implies C)$. The bottom step collapses $\mathit{purify}(B_{\mathsf{true}}(e)) \vee \mathit{purify}(B_{\mathsf{false}}(e))$ back to $\mathit{purify}(e \mapsto -)$ via the equivalence $e \mapsto - = B_{\mathsf{true}}(e) \vee B_{\mathsf{false}}(e)$ and Purify-Or.

5 Extraction in Practice

This section puts extraction to work. We first describe a technical condition for simplifying the extracted implication to the desired property, which we call *representability*. We then walk through worked examples checked in VeriFast.

5.1 Representability

Recall the motivating example from Section 1. Applying Extraction to the verified function of Fig. 2 yields

$$\begin{aligned}\mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3)) \implies \\ \mathit{purify}\big(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3) \\ * \ulcorner \mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3)) \urcorner\big),\end{aligned}$$

whereas the property we actually want is just

$$\mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3)).$$

By Purify-Star the consequent can be weakened to the conjunction of the purifications of the separating conjuncts, and the equation (being pure) is exposed by Purify-Pure; dropping the irrelevant conjuncts gives

$$\begin{aligned}\mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3)) \implies \\ \mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3)).\end{aligned}$$

Since p_1, p_2, p_3 do not occur in the consequent, this is equivalent to

$$\begin{aligned}\big(\exists p_1, p_2, p_3.\, \mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3))\big) \implies \\ \mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3)).\end{aligned} \tag{5}$$

It remains to discharge the antecedent of Eq. (5). Intuitively it asks *whether there is a heap holding concrete representations of the three lists.* With a finite address space, a sufficiently long list would have no representation; we therefore assume an infinite address space.

In general, for a data structure one picks a functional model T and a predicate $P(r, v)$ relating a concrete representation at address r to a model value $v : T$. The relevant property is that such a representation can always be added disjointly to any heap satisfying the surrounding assertion. Such a condition is formalized as follows.

Definition 3 (Representability). *A type T is* representable *by the predicate $P(r, v)$ if for all $v : T$ and all $Q :$ **HProp**,*

$$\mathit{purify}(Q) \implies \exists r.\, \mathit{purify}(Q * P(r, v))$$

holds, where r does not occur in Q.

For instance, the structure `node` in Fig. 2 represents a pair of a value and a pointer to the next node, i.e.

$$\mathit{node}(p, (v, n)) = p \mapsto v * (p + \mathrm{offset}_{\mathrm{next}}) \mapsto n,$$

and its representability states that any heap can be extended with a fresh `node` cell holding prescribed values, which follows from the infinite address space assumption.

The original goal reduces to the representability of a recursive predicate sll over lists, which we establish in two ways.

Logical approach. We argue by induction on the model value l, fixing a heap h satisfying Q.

- For $l = \mathsf{nil}$: taking $r = 0$ gives $\mathrm{sll}(0, \mathsf{nil}) = \ulcorner 0 = 0 \urcorner = \ulcorner \mathrm{true} \urcorner$, so the witness can be taken as h immediately.
- For $l = \mathsf{cons}(x, t)$: by the induction hypothesis, we have

  $$\mathit{purify}(Q) \implies \exists r_1.\, \mathit{purify}(Q * \mathrm{sll}(r_1, t)).$$

  Fixing such an r_1 and applying the representability of *node* to the values (x, r_1),

  $$\mathit{purify}(Q * \mathrm{sll}(r_1, t)) \implies \exists p.\, \mathit{purify}(Q * \mathrm{sll}(r_1, t) * \mathit{node}(p, (x, r_1))).$$

  Folding the *node* cell and the sublist back into a list yields

  $$\mathit{purify}(Q) \implies \exists p.\, \mathit{purify}(Q * \mathrm{sll}(p, \mathsf{cons}(x, t))),$$

  as required.

Programming approach. Representability also follows from the frame rule. Suppose we are given programs s_1, s_2 with

$$[\mathsf{emp}]\ s_1\ [\exists r.\, \mathrm{sll}(r, \mathsf{nil})]\ ,$$
$$[\mathrm{sll}(r, t)]\ s_2\ [\exists r_1.\, \mathrm{sll}(r_1, \mathsf{cons}(x, t))]\ .$$

Such programs can be constructed readily if the language has an always-successful allocation construct. Framing Q onto each,

$$[Q]\ s_1\ [Q * \exists r.\, \mathrm{sll}(r, \mathsf{nil})]\ ,$$
$$[Q * \mathrm{sll}(r, t)]\ s_2\ [Q * \exists r_1.\, \mathrm{sll}(r_1, \mathsf{cons}(x, t))]\ ,$$

and applying Extraction,

$$\begin{aligned} \mathit{purify}(Q) &\implies \mathit{purify}(Q * \exists r.\, \mathrm{sll}(r, \mathsf{nil})) \ , \\ \mathit{purify}(Q * \mathrm{sll}(r, t)) &\implies \mathit{purify}(Q * \exists r_1.\, \mathrm{sll}(r_1, \mathsf{cons}(x, t))) \ . \end{aligned}$$

Representability then follows by induction. One can even construct a recursive program from s_1 and s_2 to construct a representation of a list directly, without further inductive reasoning inside the meta-logic.

In either case sll is representable, and this is exactly what we need to discharge the antecedent of Eq. (5). Starting from $\mathit{purify}(\mathsf{emp})$ (Purify-Emp) and instantiating representability for sll three times, we obtain in turn, each for some fresh address,

$$\begin{aligned} &\mathit{purify}(\mathrm{sll}(p_1, l_1)) \ , \\ &\mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2)) \ , \\ &\mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3)) \ . \end{aligned}$$

Collecting the witnesses gives

$$\exists p_1, p_2, p_3.\, \mathit{purify}(\mathrm{sll}(p_1, l_1) * \mathrm{sll}(p_2, l_2) * \mathrm{sll}(p_3, l_3)) \ ,$$

precisely the antecedent of Eq. (5). Modus ponens then yields the associativity property $\mathsf{app}(\mathsf{app}(l_1, l_2), l_3) = \mathsf{app}(l_1, \mathsf{app}(l_2, l_3))$, completing the proof.

As these arguments show, proving representability is largely mechanical for representation predicates that relate the functional model to the concrete representation inductively.

5.2 Worked Examples

Apart from the example used in Section 1, we present two more examples checked in VeriFast. They use different computation patterns, recursion and iteration, and show how small specification and annotation changes can make ordinary programs carry functional-model proofs. In this sense, extraction supports an *intrinsic* proof style [16]: the proof structure can follow the program structure rather than being rewritten as a lemma function in a separate proof language.

Associativity of list concatenation, recursively. The first worked example revisits the motivating property of Eq. (1) from Section 1. Fig. 3 presents an implementation of linked-list append that carries its proof. It is extended with an extra `p3` parameter (to carry the third list) and the target functional equation as an additional postcondition. The extra property is verified along the same recursive structure as the execution, with no need to rewrite the argument as a pure lemma function like Fig. 1.

Sum of list concatenation, iteratively. The next example proves the following distribution property of list summation:

$$\mathsf{sum}(\mathsf{app}(l_1, l_2)) = \mathsf{sum}(l_1) + \mathsf{sum}(l_2) \ . \tag{6}$$

```
struct node *exec_app_assoc2(struct node *p1, struct node *p2, struct node *p3)
  //@ requires sll(p1, ?l1) &*& sll(p2, ?l2) &*& sll(p3, ?l3);
  /*@ ensures sll(result, app(l1, l2)) &*& sll(p3, l3) &*&
              app(l1, (app(l2, l3))) == app(app(l1, l2), l3); @*/
{
  if (p1 == 0) {
    //@ open sll(p1, _);
    return p2;
  } else {
    //@ open sll(p1, _);
    struct node* q = p1->next;
    p1->next = exec_app_assoc2(q, p2, p3);
    //@ close sll(p1, _);
    return p1;
  }
}
```

Fig. 3. Recursive linked-list append with an intrinsic associativity proof.

Again, by representability and extraction, it suffices to verify an executable function with a specification such as the following:

```
struct node *exec_app_sum(struct node *p1, struct node *p2)
  //@ requires sll(p1, ?l1) &*& sll(p2, ?l2);
  /*@ ensures sll(result, app(l1, l2)) &*&
              sum(app(l1, l2)) == sum(l1) + sum(l2); @*/
```

Figure 4 shows an excerpt of a proof-carrying implementation extending an iterative version of list append, satisfying the specification above. The predicate `lseg(st, ed, l)` represents a segment of the linked list from `st` to `ed` representing the list of values `l`. The extended implementation code is highlighted, including the computation of the sum of the first list and the extended facts in the invariant needed to relate that computation to the sum of the appended list. The key observation is that the loop is simultaneously traversing the first list for append and for summation, so the invariants for both views can be maintained together. The other lines, including the elided annotations, are unchanged.

5.3 Other Examples

Apart from the examples presented above, we implemented some other lemma functions from the list library of VeriFast as executable functions. Table 1 provides an overview of the functional properties they prove.

6 Related Work

Relationships between Proofs and Imperative Programs. Several lines of work [3, 11, 12] study relationships between imperative procedures and logical proofs. Our focus is the use of this relationship in practical program verification, where verified programs can provide pure functional-model facts. The setting and techniques used in Section 4 are particularly close to the system proposed in [12].

```
{
  if (p1 == 0) {
    //@ open sll(p1, _);
    return p2;
  } else {
    struct node *current = p1;
    int acc = 0;
    ...
    while (current->next != 0)
      /*@ invariant
        sll(p2, l2) &*& lseg(p1, current, ?l1_l) &*&
        current->value |-> ?value &*& current->next |-> ?next &*&
        sll(next, ?l1_r) &*& append(l1_l, cons(value, l1_r)) == l1 &*&
        &*& acc + value + sum(app(l1_r, l2)) == sum(app(l1, l2))
        &*& acc + value + sum(l1_r) == sum(l1) ; @*/
    {
      ...
      acc += current->value;
      current = current->next;
      ...
    }
    ...
    current->next = p2;
    ...
    return p1;
  }
}
```

Fig. 4. Implementation excerpt of `exec_app_sum`, extending an iterative list append with summation (the highlighted lines).

Compared with that work, we use separation logic assertions, omit return values and functions as values, and present inference rules with explicit program terms. Our work is complementary to theirs: we focus on the imperative-program-as-proof direction and on settings where the programming language is already available, such as auto-active verification of realistic programs.

Separation Logic. Separation logic [10, 14] extends Hoare logic to reason about memory, and more generally resources, in imperative programs. Many established verification tools [1, 6, 9, 13, 15] are based on or inspired by separation logic. It is possible to build our theory on top of elementary Hoare logic with a global heap interpretation, but doing so would make recursive data structures, such as the lists used in Section 5, harder to reason about.

Lemma Functions. VeriFast [6] and some other auto-active verifiers [4, 7, 8] support proof-oriented ghost code in the form of lemma functions. A lemma function has no runtime effect and can be erased before execution. Such ghost code can be used like ordinary code during verification while remaining absent from the compiled program. However, as mentioned in Section 1, lemma functions are usually syntactically distinct from executable programs: they forbid heap-mutating side effects and often restrict direct calls to executable functions. This

Table 1. Overview of implemented proofs.

Function	Description	Fact proved
`exec_len_singleton`	Singleton length	$\mathsf{len}(\mathsf{cons}(v, \mathsf{nil})) = 1$
`exec_app_nil`	Right unit of append	$\mathsf{app}(l_1, \mathsf{nil}) = l_1$
`exec_app_assoc`	Append associativity	$\mathsf{app}(l_1, \mathsf{app}(l_2, l_3)) = \mathsf{app}(\mathsf{app}(l_1, l_2), l_3)$
`exec_app_sum`	Sum over append	$\mathsf{sum}(l_1) + \mathsf{sum}(l_2) = \mathsf{sum}(\mathsf{app}(l_1, l_2))$
`exec_app_len`	Length over append	$\mathsf{len}(l_1) + \mathsf{len}(l_2) = \mathsf{len}(\mathsf{app}(l_1, l_2))$
`exec_reverse_append`	Reverse append	$\mathsf{rev}(\mathsf{app}(l_1, l_2)) = \mathsf{app}(\mathsf{rev}(l_2), \mathsf{rev}(l_1))$
`exec_reverse_reverse`	Reverse involution	$\mathsf{rev}(\mathsf{rev}(l)) = l$
`exec_mem_nth`	nth element membership	$0 \le i < \mathsf{len}(l) \implies \mathsf{mem}(\mathsf{nth}(i, l), l)$
`exec_mem_append`	Membership over append	$\mathsf{mem}(x, \mathsf{app}(l_1, l_2)) \iff \mathsf{mem}(x, l_1) \lor \mathsf{mem}(x, l_2)$
`exec_mem_index_of`	Index bound	$\mathsf{mem}(x, l) \implies 0 \le \mathsf{index}(x, l) < \mathsf{len}(l)$
`exec_nth_append_l`	Left-list element access	$0 \le i < \mathsf{len}(l_1) \implies \mathsf{nth}(i, \mathsf{app}(l_1, l_2)) = \mathsf{nth}(i, l_1)$
`exec_nth_append_r`	Right-list element access	$\mathsf{len}(l_1) \le i < \mathsf{len}(l_1) + \mathsf{len}(l_2) \implies \mathsf{nth}(i, \mathsf{app}(l_1, l_2)) = \mathsf{nth}(i - \mathsf{len}(l_1), l_2)$
`exec_drop_length`	Drop all	$\mathsf{drop}(\mathsf{len}(l), l) = \mathsf{nil}$
`exec_length_remove`	Remove length	$\mathsf{len}(\mathsf{rem}(x, l)) = \mathsf{len}(l) - (\mathsf{mem}(x, l)?1 : 0)$
`exec_length_drop`	Drop length	$0 \le n \le \mathsf{len}(l) \implies \mathsf{len}(\mathsf{drop}(n, l)) = \mathsf{len}(l) - n$
`exec_drop_0`	Drop zero	$\mathsf{drop}(0, l) = l$
`exec_take_0`	Take zero	$\mathsf{take}(0, l) = \mathsf{nil}$
`exec_take_length`	Take all	$\mathsf{take}(\mathsf{len}(l), l) = l$
`exec_drop_n_plus_one`	Drop one more	$0 \le n < \mathsf{len}(l) \implies \mathsf{drop}(n, l) = \mathsf{cons}(\mathsf{nth}(n, l), \mathsf{drop}(n + 1, l))$
`exec_append_take_drop_n`	Take/drop split	$0 \le n \le \mathsf{len}(l) \implies \mathsf{app}(\mathsf{take}(n, l), \mathsf{drop}(n, l)) = l$

restriction can prevent an already verified executable program from being reused directly as a proof witness.

Our method removes this *language-level distinction* for the specific purpose of extracting pure functional-model properties from valid total triples. The limitation is that *purify* forgets evolution of the heap. Thus, extraction is *not* a replacement for lemma functions when the goal is to prove a separation-logic entailment (e.g., a representation predicate of a non-empty list segment can be split into the prefix segment and a final cell) rather than a pure consequence of the functional model.

7 Conclusion

We have presented an extraction method that lets verified imperative programs serve as proofs of pure functional-model properties. On the theoretical side, we defined the *purify* operation, presented the extraction rule, and proved extraction sound both semantically and syntactically. To bridge the theory with practice, we introduced representability as the condition that removes concrete heap representations from extracted implications. Finally, we illustrated and tested the approach with VeriFast examples.

Future work includes evaluating more systematically when the intrinsic proof style supported by extraction reduces proof effort in realistic verification de-

velopments, and exploring the verifier support for discharging representability conditions automatically.

Acknowledgments. This research was supported by the National Cyber Security-National Science and Technology Major Project under Grant No. 2025ZD1503300

References

1. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1-4), 367–422 (2018). https://doi.org/10.1007/S10817-018-9457-5
2. Charguéraud, A., Chlipala, A., Erbsen, A., Gruetter, S.: Omnisemantics: Smooth handling of nondeterminism. ACM Trans. Program. Lang. Syst. **45**(1), 5:1–5:43 (2023). https://doi.org/10.1145/3579834
3. Churchill, M., Laird, J., McCusker, G.: Imperative programs as proofs via game semantics. In: Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada. pp. 65–74. IEEE Computer Society (2011). https://doi.org/10.1109/LICS.2011.19
4. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
5. Guo, Z.: Extracting functional properties from verified imperative programs (supplementary material) (Jun 2026). https://doi.org/10.5281/zenodo.20810021
6. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
7. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA1), 286–315 (2023). https://doi.org/10.1145/3586037
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
9. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2

10. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings. pp. 1–19. Lecture Notes in Computer Science, Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
11. Poernomo, I.H., Wirsing, M., Crossley, J.N.: Adapting Proofs-as-Programs - The Curry-Howard Protocol. Monographs in Computer Science, Springer (2005). https://doi.org/10.1007/0-387-28183-5
12. Powell, T.: Proofs as stateful programs: A first-order logic with abstract hoare triples, and an interpretation into an imperative language. Log. Methods Comput. Sci. **20**(1) (2024). https://doi.org/10.46298/LMCS-20(1:7)2024
13. Pulte, C., Makwana, D.C., Sewell, T., Memarian, K., Sewell, P., Krishnaswami, N.: CN: verifying systems C code with separation-logic refinement types. Proc. ACM Program. Lang. **7**(POPL), 1–32 (2023). https://doi.org/10.1145/3571194
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
15. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of C code with refined ownership types. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 158–174. ACM (2021). https://doi.org/10.1145/3453483.3454036
16. Stump, A.: Verified Functional Programming in Agda, ACM Books, vol. 9. ACM (2016). https://doi.org/10.1145/2841316