Mostly Automated Proof Repair for Verified Libraries

KIRAN GOPINATHAN, National University of Singapore, Singapore
MAYANK KEOLIYA, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

The cost of maintaining formally specified and verified software is widely considered prohibitively high due to the need to constantly keep code and the proofs of its correctness in sync—the problem known as *proof repair*. One of the main challenges in automated proof repair for evolving code is to infer invariants for a new version of a once verified program that are strong enough to establish its full functional correctness.

In this work, we present the first proof repair methodology for higher-order imperative functions, whose initial versions were verified in the Coq proof assistant and whose specifications remained unchanged. Our proof repair procedure is based on the combination of dynamic program alignment, enumerative invariant synthesis, and a novel technique for efficiently pruning the space of invariant candidates, dubbed *proof-driven testing*, enabled by the constructive nature of Coq's proof certificates.

We have implemented our approach in a mostly-automated proof repair tool called Sisyphus. Given an OCaml function verified in Coq and its unverified new version, Sisyphus produces a Coq proof for the new version, discharging most of the new proof goals automatically and suggesting high-confidence obligations for the programmer to prove for the cases when automation fails. We have evaluated Sisyphus on 10 OCaml programs taken from popular libraries, that manipulate arrays and mutable data structures, considering their verified original and unverified evolved versions. Sisyphus has managed to repair proofs for all those functions, suggesting correct invariants and generating a small number of easy-to-prove residual obligations.

CCS Concepts: • **Software and its engineering → Software verification**; *Software evolution.*

Additional Key Words and Phrases: mechanised proofs, separation logic, proof repair, invariant inference

ACM Reference Format:
Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI, Article 107 (June 2023), 25 pages. https://doi.org/10.1145/3591221

1 INTRODUCTION

The last two decades of research in software verification have demonstrated that large codebases can be formally verified with regard to non-trivial properties by means of constructing machine-checkable correctness proofs. The examples of such verified software systems range from OS kernels (Gu et al. 2016; Klein et al. 2009) and compilers (Kumar et al. 2014; Leroy 2006) to distributed protocols (Hawblitzel et al. 2015; Rahli et al. 2018) and cryptographic libraries (Erbsen et al. 2019).

By and large, the recent practical success of formal verification is enabled by (a) the advances in proof engineering and automation techniques (Ringer et al. 2019a), and (b) the vast body of research dedicated to the design and implementation of mechanised domain-specific Floyd-Hoare-style *program logics* (Floyd 1967; Hoare 1969) that allow for syntax-driven deductive symbolic reasoning about programs as well as for modularity of their specifications and proof reuse.

Authors' addresses: Kiran Gopinathan, National University of Singapore, Singapore, kirang@comp.nus.edu.sg; Mayank Keoliya, National University of Singapore, Singapore, mayank@u.nus.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2475-1421/2023/6-ART107
https://doi.org/10.1145/3591221

Problem Statement. While the combined power of deductive machine-assisted reasoning and proof automation can get one as far as to *prove* a realistic system correct and bug-free, it does not provide an immediate solution to the well known problem of *maintaining the proof* in response to the evolution of verified code and its specification (DeMillo et al. 1977). Such changes may involve:

(1) data type definitions and usages in the verified program and its proof,
(2) formal specifications and invariants ascribed to the system's components, and
(3) implementation logic of the system's components.

Any of the changes (1)–(3) will most likely require one to modify correctness proofs in order to keep them valid—the process known as *proof repair* (Ringer 2021). Recent work by Ringer et al. (2021, 2018) has made advances in automating proof repair for the first class of changes—in *data types*—by exploiting possible equivalences between old and new type definitions. The second class of changes—in *specifications*—has been acknowledged in several recent efforts (Klein et al. 2009; Wilcox et al. 2015), and some of them presented methodologies to make proofs more manageable, but only for specific case studies (Woos et al. 2016). To the best of our knowledge, the third, probably the most common, class—arbitrary changes in the *implementation logic*—has been largely unexplored from the perspective of proof repair; particularly so in the context of imperative programs.

In this work, we focus on the problem of repairing proofs about imperative programs in response to *local* changes in their code, and propose a practical solution to it.

Proof Repair for Libraries. A particular class of programs, whose specifications and data types change infrequently, are user-facing *libraries*. By their nature, to enable forward-compatibility, library functions are expected to preserve their API and the contracts describing their interaction with the user code. Because of this, verified software libraries (Appel and Naumann 2020; Charguéraud et al. 2017; Polikarpova et al. 2018) are a sweet spot for proof repair: it is not uncommon for library functions to undergo changes in their bodies for the sake of improved performance or readability, while modifications in their type definitions and specifications are relatively rare.

Given this setup (the spec is fixed) and the recent advances in automated deductive verification (Jacobs et al. 2011; Leino 2010; Müller et al. 2016; Sammler et al. 2021), the reader might wonder if it might be simpler to just prove the new version of a library function correct from scratch rather than try to adapt its old proof. The reason why this is not so straightforward is because of the main bane of fully-automated verification—*invariants* for loops and higher-order functions.

As an example, consider a fragment of an imperative program in a higher-order language, such as OCaml, that manipulates elements of an array `arr` via an effectful function `f` in a **`for`**-loop over the set of the array indices, and an updated version of the same program that replaces the **`for`**-loop by a call to a higher-order function **`Array`**`.iter` that iterates over array elements implicitly:

```
(* old version *)
for i = 0 to Array.length(arr) - 1 do f arr.(i) done
```

```
(* new version *)
Array.iter f arr
```

Proving that the new version satisfies the same specification as the old one would require us to provide a new invariant for the lemma describing the effect of a call to **`Array`**`.iter`, and finding such an invariant is a daunting task for general-purpose automated program verifiers. On the other hand, the programmer who performed this code refactoring with a knowledge about the relation between array elements, their indices, and the workings of **`Array`**`.iter`, should be able to derive a new invariant from an old one without too much trouble. Therefore, our hypothesis, supported by this example, is that an old proof might be helpful for deriving a new one in the situation when automated verification of a program from scratch would most definitely fail.

The goal of this work is to turn this intuition into a practical proof repair algorithm for libraries.

Approach and Key Challenges. Our goal is to implement the hinted above approach to proof repair by utilising the "knowledge about old invariants." We implemented our methodology as a

proof repair tool for OCaml programs verified in the CFML framework (Charguéraud 2011, 2020)—a version of a sequential higher-order Separation Logic (SL) (O'Hearn et al. 2001; Reynolds 2002) embedded into the higher-order logic of the Coq proof assistant.

The starting point for each proof repair task is an original OCaml program p, its SL specification $\phi \triangleq \{P\}\,\{Q\}$ (*i.e.*, its pre-/postconditions), the CFML proof $\mathcal{P}$ of p's correctness *w.r.t.* ϕ and an unverified new program version p' that is believed to satisfy ϕ, for which we would like to obtain the proof $\mathcal{P}'$. We identify the following two challenges that need to be addressed to construct $\mathcal{P}'$:

C1 The new program p' may be implemented using different computations; the relations between their results and the computations of the old program are not specified and hence need to be inferred in order to construct plausible invariant candidates.

C2 Given the arbitrary nature of assertions about the shape of data, even assuming that $\mathcal{P}'$ does not require new predicates, the search space of invariant candidates is enormous. Given the non-negligeable time required to validate them using commodity solvers, such as Z3 (de Moura and Bjørner 2008), and the inability of those solvers to handle many application-specific theories, it is not clear what the best strategy is to efficiently prune this search space.

We address both challenges using a somewhat unorthodox approach for a problem targeting deductive verification in a foundational proof assistant (*i.e.*, Coq)—via *dynamic analysis techniques*.

To address **C1**, we use a trace-based *program alignment* technique. We run p and p' on the same randomly generated input, collecting traces of their execution and recording the values of all program variables at each statement. As both programs still satisfy the same specification, p' should still compute the same or similar concrete values when run on the same input. Thus, by comparing the concrete values that flow through the variables of each program, we heuristically infer the mapping between the intermediate computations of both programs by relating statements in p and p' that have the most variables with similar concrete values.

The solution to **C2** is enabled by exploiting the nature of Coq proofs using a new technique that we call *proof-driven testing*. When reasoning about higher-order functions and program constructs (*e.g.*, loops) in program logics, proofs typically make use of *second-order* lemmas that encode the semantics of those higher-order constructs and take *invariants* to constrain the behaviour of their *first-order* arguments (*e.g.*, of a loop body). In turn, when mechanising such proofs within Coq, *proofs* of these second-order theorems become second-order *programs*, that, when instantiated with sufficient arguments and invariants, *compute* a *proof term* that witnesses the execution of the higher-order construct on these inputs. Furthermore, we observe that by applying these second-order theorems for loops to particular *concrete* inputs, we necessarily obtain a concrete trace of the program, annotated with additional *meta-data* capturing how the invariants are instantiated over the course of the execution with different concrete values and states. Proof-driven testing turns these observations into an efficient strategy for validating candidate invariants for a loop. It applies the loop's specification to concrete inputs and analyses the resulting proof term to extract an *executable* function that *tests* whether a given invariant holds over a particular execution trace.

In conjunction, our solutions to **C1** and **C2** make it possible to reconstruct a proof for the new program, inferring invariants for its loops and higher-order function applications. Our approach is *mostly-automated*: the verification conditions for the invariants are discharged using domain-specific automation. A relatively small amount of manual proof effort is required when automation fails to dispatch the residual proof obligations. Crucially, no manual work is required for stating invariants for the new program version.

Contributions. In summary, we make the following contributions.

- A methodology for mostly-automated proof repair for higher-order imperative programs written in OCaml and verified in Separation Logic embedded into Coq (Sec. 2).

- An algorithm for synthesising basic proof structure and invariant candidates for a refactored program using dynamic analysis and program alignment (Sec. 3).
- *Proof-driven testing*—a novel approach to test properties of data and programs for validity by extracting tests from proofs of higher-order facts that rely on those properties. We provide the intuition and formal description of the proof-driven testing methodology and show how to use it for efficiently pruning the search-space of candidate invariants required for proof repair (Sec. 4).
- Sisyphus—a proof repair tool for OCaml programs verified in Coq. We evaluated Sisyphus on a suite of 14 evolved programs, of which 10 were drawn from popular OCaml libraries, and found that all inferred invariants for new versions were valid and required relatively small amounts of manual effort to prove in comparison to the original programs (Sec. 5).

2 THE LABOURS OF SISYPHUS

We present an overview of Sisyphus by means of an illustrative example: repairing the proof of correctness between two versions of a real-world program from a widely-used OCaml library. The program in question is the function `Seq.to_array`, which converts a lazy sequence to an array, taken from the popular `containers` library between versions 3.6[1] and 3.7,[2] wherein it was updated to improve its performance. We manually verified the initial version of `to_array` in Coq via CFML; Sisyphus was then used to automatically generate a repaired proof for the updated program.

2.1 An Initial Verified `Seq.to_array`

Fig. 1 presents the original implementation of `to_array`, which converts a sequence into an array, where a sequence of type `'a t` is encoded as a thunked list:

```
type 'a t    = unit -> 'a node
and  'a node = Nil
             | Cons of 'a * (unit -> 'a node)
```

A sequence is represented by a function that, when evaluated, either returns `Nil` for the empty sequence, or returns a `Cons` cell with a head element and a tail that can be evaluated on-demand to retrieve the rest of the sequence. This encoding of sequences can even be used to represent lists of infinite length or encode arbitrary side-effects within the thunks of a sequence. However, for the purposes of this work, we will be restricting our focus to the case in which these sequences are finite and side-effect free, since `to_array` has undefined behaviour for other cases.

```
1  let to_array s =
2    match s () with
3    | Nil -> [| |]
4    | Cons (hd, _) ->
5      let sz = length s in
6      let a  = Array.make sz hd in
7      iteri
8        (fun i vl ->
9           a.(i) <- vl) s;
10     a
```

Fig. 1. Original `to_array`

The implementation of `to_array` is as follows. If the sequence contains at least one element, the function calculates its length and allocates a fresh result array with the capacity to store all of the elements of the sequence. The function then calls the *higher-order* iterator function `iteri`, whose argument function successively assigns elements of the sequence to the corresponding slots of the allocated array, which is eventually returned as the result. Having stepped through the function, it seems reasonable to believe that `to_array` is correct, and `containers` itself comes with an extensive test suite. But if we truly wish to *ensure* the correctness of this implementation of `to_array`, we should really *prove* its correctness.

2.1.1 Specification of `Seq.to_array`. In order to faithfully specify the effect of `to_array` on the heap, we turn to a formalism tailored for that purpose—Separation Logic (SL) (O'Hearn et al. 2001; Reynolds 2002)—a Hoare-style program logic for reasoning about heap-manipulating programs.

[1] `to_array` in `containers` 3.6: https://github.com/c-cube/ocaml-containers/blob/v3.6/src/core/CCSeq.ml#L397
[2] `to_array` in `containers` 3.7: https://github.com/c-cube/ocaml-containers/blob/v3.7/src/core/CCSeq.ml#L415

Assertions in Separation Logic capture constraints over disjoint partitions of the program's heap, called *heaplets*. These constraints can be combined using the *separating conjunction* $*$: the statement $H * H'$ asserts that the heap can be partitioned into two *disjoint* sub-heaps such that H holds on the first, and H' on the second. We use the standard notation $a \mapsto e$ to describe a heaplet represented by a single memory location with address a and contents e (pronounced "*a points to e*"); emp is an SL assertion satisfied by an empty heap. Using these assertions, we can thus capture the semantics of an imperative program using a Hoare triple $\{P\}\ c\ \{Q\}$, which asserts the following: starting in any heap state that satisfies P, after executing the program c, we must obtain a heap state satisfying Q. In this formalism, `to_array` can be ascribed the following SL specification:

$$\forall s\ \ell, \{s \mapsto \texttt{Seq}\ \ell\}\ (\texttt{Seq.to\_array}\ s)\ \exists a, \{a \mapsto \texttt{Array}\ \ell\} \tag{1}$$

The precondition, $\{s \mapsto \texttt{Seq}\ \ell\}$, states that the payload of the (universally quantified) input sequence s is modelled by a logical list ℓ, where the predicate `Seq` captures the fact that s is a finite sequence without side-effects. The postcondition, $\exists a, \{a \mapsto \texttt{Array}\ \ell\}$, uses CFML's convention of using existential quantifiers to encode return values and asserts that the function will return some pointer a that points to an array with contents described by ℓ, where the predicate `Array` encodes the fact that a points to an array on the heap.[3] In other words, the specification (1) asserts that `to_array` indeed converts a sequence to an array with the same payload, and now, if proven, provides a meaningful guarantee about the correctness of this function. Finally, note that while our post-condition does not constrain the input sequence s, this does not affect the usability of our specification as we have defined such sequences to be pure and effect-free, so they are immutable and thus allowed by CFML's affine logic to be duplicated before passing them to the function.

2.1.2 A mechanised proof for `Seq.to_array`. We are now ready to verify `to_array` in Coq. When proving properties about these kinds of heap-manipulating programs in Coq, the corresponding proofs follow by stepping through the code using the reasoning rules of the program logic to symbolically evaluate how its individual statements update the symbolic state.

Fig. 2 shows a correspondence between `to_array` and the Coq proof (done using the CFML embedding of SL) that establishes that the program indeed satisfies the spec (1). As is common with such SL implementations, each rule of the logic comes with an associated Coq *tactic* that applies the rule, automatically determining which heaplets are affected by the rule (*i.e.*, its *footprint*). As an example, consider the rule XAPP for verifying an application of a function f with a continuation c (ignore the highlighted premise for now):

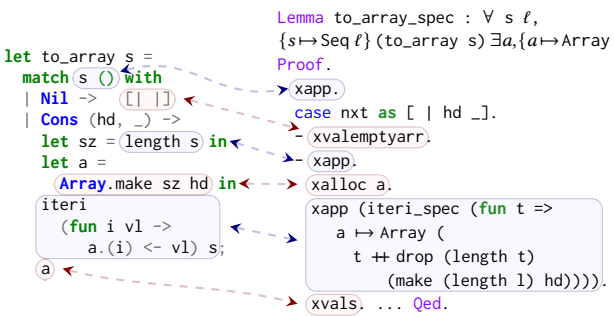

Fig. 2. Proof of `to_array` in CFML.

$$\textsc{xapp}\ \frac{\{P\}\ (f\ \overline{v})\ \exists x, \{Q'x\} \qquad \forall x, \{Q'x\}\ c\ x\ \{Q\}}{\{P\}\ (\texttt{let x} = f\ \overline{v}\ \texttt{in}\ c\ \texttt{x})\ \{Q\}} \tag{2}$$

In a Coq embedding of CFML, this rule is implemented as a tactic `xapp`, which applies a lemma that discharges the conclusion of the rule by emitting verification conditions as per its premise.

[3] We assume an OCaml-style (rather than C-style) memory model, in which locations can store arbitrary values, *e.g.*, arrays.

Consider the call to `Array.make` in `to_array` and the corresponding fragment in the proof in Fig. 2. In the program, this expression allocates an array of size `sz` initialised with the value `hd` in the heap and returns a pointer to the freshly allocated data. In the corresponding proof step, reasoning about this computation is handled using a tactic `xalloc`, which updates the symbolic program state (and is captured correspondingly by the Coq proof context) to reflect the semantics of the operation, as shown in the proof snippet above.

```
{s ↦ Seq ℓ}
let a = Array.make sz hd in
{s ↦ Seq ℓ ∗ a ↦ Array (repeat sz hd)}
```

Under the hood, the `xalloc` tactic itself operates by applying a reasoning rule for function application, `xapp` (2), to the specification of the library function `Array.make`:

$$\forall sz\ v, \{sz > 0; \mathsf{emp}\}\ (\texttt{Array.make}\ sz\ v)\ \exists a, \{a \mapsto \texttt{Array}\ (\texttt{repeat}\ sz\ v)\} \tag{3}$$

In particular, the spec (3) asserts that when calling `Array.make` with a size sz and value v where $sz > 0$, the return value of the function will be a pointer `a` to an array, whose contents are described by the logical expression `repeat` $sz\ v$; that is, the array has sz copies of v.

The most interesting part of the proof is reasoning about the use of the higher-order function `iteri`, which takes as arguments a possibly-effectful function `f` and a sequence s, and iterates through the sequence, calling `f` on each element. In order to characterise the state of the heap *after* its invocation, the specification (4) of `iteri` requires an *invariant* I that must be maintained by `f`:

$$\forall I\ \mathtt{f}\ s\ \ell, (\forall t\ v, \{I\ t\}\ \ (\mathtt{f}\ v)\ \ \{I\ (t +\!\!+ [v])\}) \to \{I\ [] * s \mapsto \mathsf{Seq}\ \ell\}\ (\mathtt{iteri}\ \mathtt{f}\ s)\ \{I\ \ell * s \mapsto\ \mathsf{Seq}\ \ell\} \tag{4}$$

Here, invariant I characterises the heap in terms of the *prefix* t of the sequence that has been visited by `iteri`. The premise of the specification asserts that the function call `f` v preserves the invariant I after visiting v, *i.e.*, I now holds over an extended prefix. The conclusion of the specification (in grey) states the effect of `iteri` on the state. Initially, none of the elements of s have been seen, so I must hold on the empty list `[]` in the precondition, constraining the corresponding part of the heap. As per the postcondition, after executing `iteri f` s, every element in the sequence has now been visited, and so the specification asserts that I now holds over *all* elements in the sequence, ℓ.

The proof in Fig. 2 makes use of the conclusion of the specification (4), referred to as `iter_spec` in the Coq code, by providing it as the highlighted premise of the rule (2). To do so, it *instantiates* `iter_spec` with a suitable invariant I. This invariant argument *does not* directly follow from the syntax of the program like other components of the proof, but, rather, must be *explicitly* provided by the user. The invariant provided in our proof states that at each iteration of `iteri`, the allocated array `a` will always start with the sequence of elements t that have been visited by the iteration:

$$\texttt{fun}\ t \Rightarrow \mathtt{a} \mapsto \texttt{Array}\ (t +\!\!+ \texttt{drop}\ (\texttt{length}\ t)\ (\texttt{repeat}\ (\texttt{length}\ \ell)\ \mathtt{hd})) \tag{5}$$

In particular, this invariant asserts that the value `a`, previously returned by the call to `Array.make`, will point to an array with the same length as ℓ, where the contents of the array start with t, the *prefix* of visited elements, and the remaining elements are all `hd`. At the precondition of `iteri`'s specification, we instantiate this invariant with the empty list, where `a` points to a list whose length is equal to that of ℓ and whose elements all have the value `hd`. Having executed `iteri`, the invariant in its postcondition is instantiated with the full sequence ℓ, and so we learn that the final contents of the array `a` must have the same length as ℓ, with its contents starting with the sequence ℓ and followed by an empty suffix. Therefore, we can then show that the contents of `a` are exactly the sequence ℓ, thereby satisfying the post-condition of `to_array` and concluding the proof.

2.2 A Recipe for Proof Repair

Our proof in Fig. 2 serves as a certificate of correctness for `to_array`, but, alas, only for one particular version of the function: if the implementation of `to_array` were to later change, then our certificate would no longer hold, and we would have to prove correctness of the program again. This definition

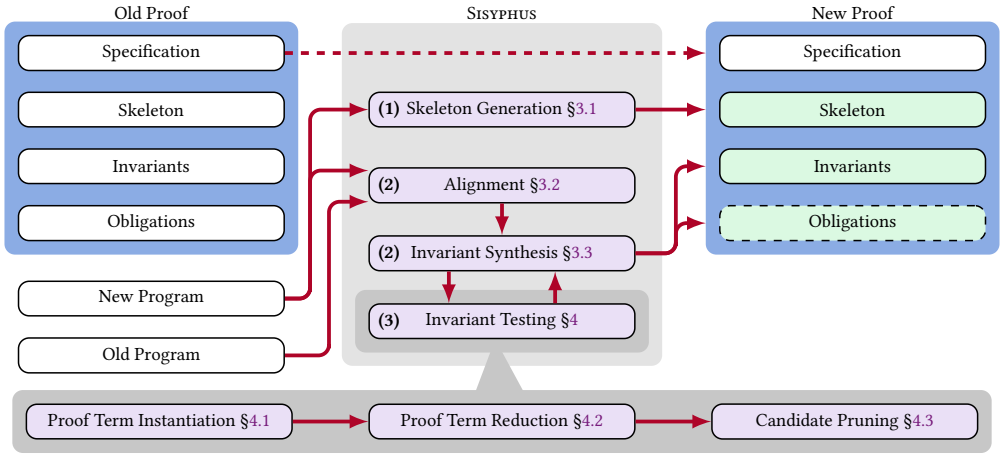

Fig. 4. Sisyphus overview. Boxes with white background represent user-provided components. Light-green boxes represent outputs generated by the tool with the dashed border around obligations representing a best-effort attempt at dispatching residual goals using `user_solve` or otherwise leaving them as admits. Solid arrows correspond to inputs and outputs, and the dashed arrow encodes that specifications are copied.

of `to_array` in the `containers` library was later updated to the implementation in Fig. 3, adopting a radically different, but more efficient implementation that avoids repeated evaluation of the input sequence. In the new implementation, the function first traverses the entire sequence in a single pass (line 3) using the `fold` iterator to fold over the elements of the sequence, using a *pure* function to accumulate a tuple of the length of the sequence and the elements of the sequence in reverse (line 4). Then, having allocated a result array of a suitable length, the program simply iterates over the reversed *list* of elements using `List.fold_left` (line 10), and assigns the elements to its result array in reverse (lines 11-12), gradually increasing the *suffix* of the array that is shared with the input. In this way, while both versions traverse the elements twice, the two implementations differ in the number of times the lazy sequence is "forced": in the original version the sequence is forced twice (by `length` and `iteri`), while the updated implementation only forces it once (using `foldi`). This can have significant performance benefits when the elements capture expensive computations.

While this new version significantly differs from the original implementation of `Seq.to_array` from Fig. 1, we can notice some striking similarities in their implementations. Most significantly, both programs follow the *same* high-level steps: first (1) to calculate the length of the sequence, then second (2) to allocate a result array with an appropriate size, and finally (3) to populate this array with the elements of the sequence. For example, in the old implementation, step (1) is completed using a dedicated `length` function, while in the new program, this is achieved in parallel with accumulating a reversed list of the elements of the sequence using `fold`. Can we use these similarities between the versions of the program to thus repair the correctness proof from Fig. 2?

```
1  let to_array s =
2    let sz, rls =
3     fold (fun (i,ls) x ->
4       i+1, x::ls) (0, []) s in
5    match rls with
6    | [] -> [| |]
7    | init :: rest ->
8      let a = Array.make sz init in
9      let idx = sz - 2 in
10     let _ = List.fold_left
11         (fun i x -> a.(i) <- x;
12           i - 1) idx rest in a
```

Fig. 3. New version of `to_array`

The remainder of this section describes our tool Sisyphus that does exactly that.

Overview of Sisyphus. The high level overview of Sisyphus is shown in Fig. 4, which highlights the three main stages in Sisyphus's proof repair process:

(1) First, following the syntactic structure of the new program, Sisyphus constructs its *proof skeleton*, with holes for the parts that cannot be immediately inferred (Sec. 2.2.1).
(2) Next, Sisyphus compares traces of the old and new programs on the same random inputs in order to recover high-level relations between the individual steps in their implementations. It uses these relations to discover relevant sections of the old proof, which are then used to synthesise candidate expressions to fill holes in the new proof skeleton (Sec. 2.2.2).
(3) Finally, Sisyphus uses a *fast* dynamic test to prune the space of synthesised expressions, instantiating the holes in the proof-skeleton and attempting to dispatch the resulting obligations using user-provided proof automation, thereby completing the proof script (Sec. 2.2.3).

In this way, Sisyphus implements a *mostly-automated* proof repair procedure — the tool will generate a proof-skeleton and invariants for the new proof, but residual obligations are handled in a best-effort fashion. In particular, pure domain-specific logical obligations often remain in SL proofs after symbolically reasoning about the program (*e.g.*, proving that a particular integer is a valid index into a list). Sisyphus allows the user to supply a tactic (which we will refer to as `user_solve`), that is invoked during repair to dispatch any such obligations. In order to construct such a tactic, one will typically use a mixture of Coq's hint databases and proof search such as `auto`/`eauto`. In the case the tactic fails to dispatch a goal, Sisyphus emits an `admit` for that particular subgoal, which the user must then fill in to complete the proof.

2.2.1 Building a proof skeleton. As we have seen in Sec. 2.1.2, when verifying programs using SL program logics in Coq, the structure of the corresponding proof scripts will often mirror the structure of the program being verified. Applying this insight in reverse, Sisyphus starts its repair process by constructing an initial skeleton proof script for the new program, traversing the program body and mapping program constructs to the relevant tactics to symbolically execute them. Any obligations that remain following this process are delegated to a user-provided solver tactic to dispatch (here, `user_solve`), or admitted and left for the user in cases when the solver fails. Using this strategy, Sisyphus can automatically generate a proof skeleton for the new `to_array` function (Fig. 5). Each program statement in Fig. 3 maps to a particular tactic application in the proof skeleton: function applications to `xapp` (lines 4 and 8), array allocation to `xalloc` (line 7), creating an empty array to `xvalemptyarr` (line 6), and branching in the program is mirrored by a case analysis in the proof (line 5).

```
1  Lemma to_array_spec : ∀ s ℓ,
2  {s ↦ Seq ℓ} (to_array s) ∃a, {a ↦ Array ℓ}
3  Proof.
4    (* .. *) xapp (* .. *).
5    (* .. *) case ls as [ | init rest].
6    - (* .. *) xvalemptyarr. { user_solve. }
7    - (* .. *) xalloc a. (* .. *)
8      xapp (list_fold_left_spec
9        (fun (acc: int) (t: list A) => □)).
10       { user_solve. }
11     (* .. *) xvals. { user_solve. }
12 Qed.
```

Fig. 5. Proof skeleton for new `to_array`

While this strategy automatically handles a large component of the burden of writing the new proof script, there may still be certain parts of the new proof which cannot be immediately filled in and must be left as holes. In our case, such a hole must be left when reasoning about the application of `List.fold_left` in the program, as its specification (6) takes an explicit invariant I to characterise how the program state is updated through its execution.

$$\begin{array}{c} \forall I\ \mathtt{f}\ s\ acc'\ \ell, (\forall\ acc\ t\ v, \{I\ acc\ t\}\ \ (\mathtt{f}\ acc\ v)\ \exists res, \{I\ res\ (t +\!\!+ [v])\}) \rightarrow \\ \{I\ acc'\ []\ *\ s \mapsto \mathtt{List}\ \ell\}\ (\mathtt{List.fold\_left\ f}\ s)\ \exists res, \{I\ res\ \ell\ *\ s \mapsto \mathtt{List}\ \ell\} \end{array} \tag{6}$$

The specification (6) of `fold_left` is fairly similar to the specification (4) of `iteri`; the key difference being that the invariant used to constrain the behaviour of the user-supplied function `f` now takes two parameters: an accumulator value *acc*, and a list t. As before, the t parameter represents a *logical* variable, the prefix of the sequence of elements that have been visited so far. The *acc* parameter represents a program-level value, which is the result accumulated by the fold.

The invariant provided to this specification must capture the shape of the heap that evolves over the execution of the fold at each iteration, and it does not easily follow from the program syntax, so Sisyphus leaves a hole (□) in the proof (Fig. 5) in place of the invariant body to be filled in later.

2.2.2 Transplanting invariants across proofs. To fill the remaining holes in the proof, Sisyphus builds on a simple observation that different versions of a program often share similar invariants in their proofs. More precisely, statements between different versions of a program that perform similar operations (*e.g.*, populating an array), will often share similarities in the assertions used to reason about the state they alter in their corresponding proofs, as in the snippets below:

```
(* old: populate prefix of array *)
iteri (fun i vl -> a.(i) <- vl) s
```

```
(* new: populate suffix of array *)
List.fold_left (fun i x -> a.(i) <- x; i-1) idx rest
```

This intuition leads to a two-step process for instantiating holes in the new proof-skeleton: first, (1) to determine which statements in the new program perform similar operations to those from the old program, and then (2) to use relevant invariants from the old proof to guide the generation of invariant candidates to fill the holes in the new proof.

Discovering similar computations. In order to instantiate holes in the proof skeleton, we require a mapping between the high-level steps of old and new programs. In other words, what we need is a program alignment between the two programs, which relates individual statements of both programs that correspond to the same high-level steps. These relations between the high-level steps in a program capture semantic properties that might be hard to determine statically, but, as it turns out, can be discovered dynamically using a simple strategy. Consider the old and new versions of `to_array`, which adopt the same general strategy to implement the conversion from a sequence to array (Fig. 6): (1) compute the length of the input sequence, (2) allocate an array for the result, and (3) populate the elements of the array. While both programs use the same steps, the implementations of these individual steps can significantly differ in execution. For example, when populating the result array in step (3), the original implementation fills the array from the beginning, while the new implementation does it by assigning elements from the end. However, after executing this step in both programs, the result array will contain exactly the elements of the input sequence—a fact that is easy to capture during executions of both programs:

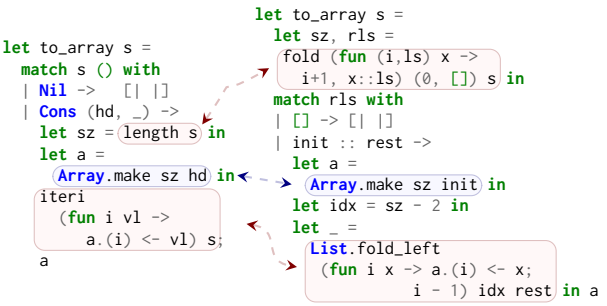

Fig. 6. Alignment between two versions of `to_array`

```
{sz = 3; a ↦ Array ([1; 1; 1])}
iteri (fun i vl -> a.(i) <- vl) s
{sz = 3; a ↦ Array ( [1; 2; 3] )}
```

```
{sz = 3 ∧ idx = 1; a ↦ Array ([3; 3; 3])}
List.fold_left (fun i x -> a.(i) <- x; i - 1) idx rest
{sz = 3 ∧ idx = 1; a ↦ Array ( [1; 2; 3] )}
```

To exploit this insight for generating new invariants, Sisyphus runs both versions of the program on the same randomly generated inputs and records observations of the runtime program states at each program point. Observations between the traces are ranked based on their *similarity* by using certain metrics to compare the values of program variables in each observation. In particular, Sisyphus uses two such metrics: comparing exact matches in runtime values, and barring an exact match, comparing the sizes of collections such as lists, albeit with a lower priority. That is, if many program variables have the same runtime values, then two observations are considered to be highly

Table 1. Mapping from Coq invariants to OCaml tests

Number	Logical embedding as Coq predicate	Executable embedding as OCaml function	Valid?
(1)	`(fun acc t => a ↦ Array (repeat (acc + 1) init ++ drop (acc + 1) ℓ))`	`(fun acc t -> Array.to_list a = (repeat (acc + 1) init ++ drop (acc + 1) ℓ))`	Yes
(2)	`(fun acc t => a ↦ Array ([] ++ drop 0 (repeat (length t) init)))`	`(fun acc t -> Array.to_list a = ([] ++ drop 0 (repeat (length t) init)))`	No

similar. A statement in one program is considered as similar to a statement in the other program when the observations at points surrounding the respective statements are similar.

In this way, Sisyphus builds up an *alignment* between the statements of both programs. For example, in Fig. 6 the call to `iteri` on the left is discovered to be similar to the call to `fold_left` on the right as both produce arrays with the same payload, even though they do so in different directions, hence both statements are marked as aligned.

Generating invariant candidates. When instantiating a hole in the new proof skeleton, Sisyphus first uses the calculated alignment to find the relevant statements of the old program that plausibly correspond to the same high-level step as the current statement of the new program that needs an explicit invariant in its proof. Looking at the proof steps from the old proof that correspond to these statements, Sisyphus then collects any expressions that occur in symbolic states preceding these steps and produces a family of *sketches* that are likely to capture the similarities in the invariants between the old and new proofs by replacing sub-expressions within those invariants with holes (_). For example, when looking to synthesise an invariant for the call to `fold_left` in the proof of the new program, Sisyphus discovers an aligned statement—a call to `iteri`—and the invariant (5) used to verify it in the corresponding proof (Fig. 2). It then extracts from the old invariant a number of sketches, one of them being `(_ ++ drop _ _)`, which captures the prefix/suffix argument from the proof of the old program and contributes the following (simplified) invariant template:

```
fun (acc: int) (t: list A) => a ↦ Array (_ ++ drop _ _)        (7)
```

Finally, Sisyphus uses this template (amongst others), along with any logical functions and constants in state assertions in the proof of the old and new program, to bootstrap an enumerative synthesis procedure for generating concrete invariant candidates for the current hole in the proof in Fig. 5.

2.2.3 Validating invariant candidates. The last step in Sisyphus' repair process is to validate the generated invariants and identify suitable candidates to instantiate the holes in the proof skeleton. As the generation strategy can produce large numbers of candidates, validating each candidate by trying to prove its correctness would quickly become intractable. As such, Sisyphus implements its candidate validation step in two phases, first running a *fast* dynamic test to quickly prune generated candidates, and then using the user-provided solver to actually prove the invariant. The key insight powering Sisyphus is that while such *dynamic* tests for invariants may be challenging to generate from scratch, we can actually *automatically* construct these tests from information hidden within the proofs of the higher-order functions, using a novel technique we dub *proof-driven testing*.

Consider the invariant candidates for the call to `fold_left` in the new program generated from the template (7) and listed in Tab. 1.[4] The first candidate accurately describes the invariant that holds for one iteration of `fold_left`'s argument function, while the second invariant is incorrect.

[4]We assume that the invariants are used in an environment where a and ℓ are bound, *e.g.*, the proof in Fig. 5.

Following the intuition presented in the earlier sections, the first invariant asserts that during the execution of `fold_left`, the contents of the array `a` will share an increasing *suffix* with the contents ℓ of the original sequence. The second invariant asserts that the contents of `a` will always be described by `repeat (length t) init`—that is, all elements in the array will always have the same value. Our goal is to test these invariants to quickly distinguish between the invalid invariant (2) and the correct invariant (1). To do so, we would like to convert the logical state properties asserted by the invariant and expressed as propositions in Coq's logic (second column of Tab. 1), into executable OCaml tests (third column) that check the program values of the program via a direct syntactic translation. Combining this with a suitable instantiation for the free variables in the expressions (*i.e.*, `a` and ℓ), we can construct an executable program to test the invariants.

```
let ℓ   = [ 1; 2; 3 ] in
let a   = [| 3; 3; 3; |] in
let f i x = a.(i) <- x; i-1 in
let len = 3 - 2 in
let inv = (* .. *) in
(* a = [ 3; 3; 3 ] *)
assert (inv len []);
let acc = f len 2 in
(* a = [ 3; 2; 3 ] *)
assert (inv acc [2]);
let acc = f acc 1 in
(* a = [ 1; 2; 3 ] *)
assert (inv acc [2;1])
```

Fig. 7. A concrete invariant test

Suppose we had access to the testing program presented in Fig. 7. This program encodes a particular *concrete* execution trace of `List.fold_left` within the new implementation of `to_array` called on a sequence with the elements $[1, 2, 3]$, annotated with appropriate assertions (`inv`). Specifically, after each step in the trace, the program includes an assertion that an invariant function `inv` holds with the current value of accumulator variable `acc` and the prefix of the elements that have been visited so far. If we use our first invariant candidate as the definition of the `inv` function, then this testing function will execute without raising an exception, as each time `inv` is called, the contents of the array `a` will indeed match the state expected by the invariant. Conversely, if we execute this program with the definition of `inv` assigned to the second invariant candidate, then the program will fail to execute to completion, as at the second call to the `inv` function, the contents of the array will be $[3; 2; 3]$ while the invariant will expect all elements to be the same.

```
let rec fold_left (* I *) f acc ls =
  (* assert (I acc []); *)
  let rec loop (* t *) acc ls =
   match ls with
   | [] -> acc
   | hd :: tl ->
     let acc' = f acc hd in
     (* assert (I acc' (t ++ [hd])); *)
     loop (* (t ++ [hd]) *) acc' tl in
  loop (* [] *) acc ls
```

Fig. 8. Implementation of `fold_left`

The challenge with constructing such a test program is that it manipulates logical variables and performs invariant checks that do not occur in the implementation of `fold_left` (as given in Fig. 8). Note however that these properties *could* actually be obtained if we considered the execution trace of a suitably annotated version of `fold_left` (*cf.* the comments in Fig. 8) that was parameterised by additional logical parameters (*i.e.*, `t`) and contained explicit invariant checks.

The key insight of Sisyphus is that we *do* have access to these annotations: they are in fact contained within *the proof of correctness* for `fold_left` with regard to the specification (6), which itself has to manipulate and maintain these variables and checks in order to establish that the invariant holds over the course of the entire program. Sisyphus *evaluates* proof terms of specifications for higher-order functions (as well as correctness proofs for rules of a program logic) on concrete inputs, extracting test specifications from the resulting reduced proof terms as presented in Fig. 7, a process we dub *proof-driven testing*. Using these generated tests, Sisyphus can quickly prune the generated candidates and suggest valid invariants necessary to complete the repaired proof.

2.2.4 Putting it all together. Recall that our starting point was an old program, its SL specification, and a Coq proof of its correctness (Sec. 2.1). Taking those artifacts and running them through the sequence of steps described in Sec. 2.2.1–Sec. 2.2.3, Sisyphus produced a nearly complete

repaired proof with explicitly-stated high-confidence invariants for calls to higher-order functions and loops. To fully complete the proof, the user has to mechanically establish the validity of the suggested invariants. These residual proof obligations typically boil down to proving new facts about (a) entailment of SL assertions and (b) mathematical properties of involved data types (*e.g.*, a new lemma relating list reversal and concatenation)—all outside of the scope of the original proof, and, therefore, beyond the reach of Sisyphus's proof repair capabilities.

In the following sections we provide detailed descriptions of the outlined algorithms for reconstructing proof skeletons, computing program alignment, and synthesising invariant candidates (Sec. 3), formally define proof-driven testing via Coq proofs (Sec. 4), and elaborate on the utility aspects of Sisyphus, including the proof burden for the residual verification conditions (Sec. 5).

3 PROOF RECONSTRUCTION AND SYNTHESIS OF INVARIANT CANDIDATES

3.1 Generating Proof Skeletons

Fig. 9 provides a subset of OCaml terms (ranged over by c) and CFML proof tactics (T), which we will be using for our presentation in the remainder of the paper. For the sake of uniform treatment, we require loops in OCaml programs to be encoded as applications of higher-order combinators that take a function, representing the loop's body, as their argument. This reduces the problem of inferring loop invariants to inferring invariants for higher-order function applications. In practice, this convention poses little problem: most idiomatic OCaml programs that loop are either already implemented using higher-order combinators or can be easily rewritten to do so. A tactic xapp that implements the CFML logic rules for function applications has two forms—the latter tackles application of higher-order functions and requires an explicit invariant, which is initially represented by a "blank" $\texttt{fun}\ \overline{v} \Rightarrow \square$ to be elaborated later.

$$
\begin{array}{lcll}
v & & & \text{variable names} \\
c & ::= & v \mid \texttt{fun}\ \overline{\texttt{v}} \Rightarrow \texttt{c} \mid \texttt{assert c} \\
 & & \mid c\ c \mid \texttt{let v =}\ \texttt{c in c} \\
 & & \mid \texttt{if c then c else c} \\
 & & \mid \texttt{match}\ v\ \texttt{with}\ \overline{C_i\ \overline{v_i} \rightarrow c_i} \\
T & ::= & \texttt{xval.} \mid \texttt{xif.}\ \overline{\{T\}} \\
 & & \mid \texttt{xletval v.} \mid \texttt{xmatch.}\ \overline{\{T\}} \\
 & & \mid \texttt{xapp.} \mid \texttt{xapp v (fun}\ \overline{\texttt{v}} \Rightarrow \square\texttt{).}
\end{array}
$$

Fig. 9. OCaml programs and Coq tactics

Fig. 10 shows the rules of proof skeleton reconstruction. It is implemented as a syntax-based translation $\rightsquigarrow_{\mathrm{x}}$ from OCaml programs to Coq sequences of Coq tactics. It also takes the actual verification goal (*i.e.*, pre-/postcondition of the residual program to be traversed) and emits a list of *residual obligations* $\overline{\varphi}$ that should be discharged separately. For example, the obligation emitted in the conclusion of xval, which is typically applied at the end of the proof (*cf.* Fig. 2), is the heap entailment $P \vdash Q\ v$, which is derived immediately from the residual goal $\{P\}\ \ \{Q\}$. Both rules xapp and xapp-hof deal with function applications, using the function dictionary $\mathcal{E}$ to retrive function specifications and adding their side conditions to the list of residual obligations. In addition to that,

$$
\textsc{xif}\ \frac{\{v = \texttt{true}; P\}\{Q\}\ c_t \rightsquigarrow_{\mathrm{x}} T_t; \varphi_{\mathrm{t}} \quad \{v = \texttt{false}; P\}\{Q\}\ c_f \rightsquigarrow_{\mathrm{x}} T_f; \varphi_{\mathrm{f}}}{\{P\}\ \{Q\}; \texttt{if}\ v\ \texttt{then}\ c_t\ \texttt{else}\ c_f \rightsquigarrow_{\mathrm{x}} \texttt{xif.}\ \{T_t\}\{T_f\};\ (\varphi_t, \varphi_f)}
$$

$$
\textsc{xmatch}\ \frac{\{v = C_i\ \overline{v_i}; P\}\{Q\}\ c_i \rightsquigarrow_{\mathrm{x}} T_i;\ \varphi_i}{\{P\}\ \{Q\}; \texttt{match}\ v\ \texttt{with}\ \overline{C_i\ \overline{v_i} \Rightarrow c_i} \rightsquigarrow_{\mathrm{x}} \texttt{xmatch.}\ \overline{\{T_i\}};\ \overline{\varphi_i}}
$$

$$
\textsc{xval}\ \frac{}{\{P\}\ \{Q\};\ v \rightsquigarrow_{\mathrm{x}} \texttt{xval.};\ P \vdash Q\ v}
$$

$$
\textsc{xapp}\ \frac{\mathcal{E}[f] = \forall\ \overline{x}, \overline{\psi} \rightarrow \{P\}\ f\ \overline{x}\ \exists x\ \{Q'\ x\} \quad \{(Q'\ x)[\overline{v}/\overline{x}]\}\{Q\};\ c\ x \rightsquigarrow_{\mathrm{x}} T;\ \varphi}{\{P\}\ \{Q\}; \texttt{let}\ x = f\ \overline{v}\ \texttt{in}\ c\ x \rightsquigarrow_{\mathrm{x}} \texttt{xapp.}\ T; (\overline{\psi}[\overline{v}/\overline{x}], \varphi)}
$$

$$
\textsc{xapp-hof}\ \frac{\mathcal{E}[f] = (s : \forall F, \overline{x}, I, \overline{\psi} \rightarrow \{P\}\ f\ F\ \overline{x}\ \exists x\ \{Q'\ x\}) \quad d \text{ is first-order} \quad \{(Q'\ x)[\overline{v}/\overline{x}, (\texttt{fun}\ \overline{w} \Rightarrow d)/\texttt{F}, (\texttt{fun}\ \overline{q} \Rightarrow \square)/\texttt{I}]\}\{\texttt{Q}\};\ \texttt{c}\ x \rightsquigarrow_{\mathrm{x}} T;\ \varphi}{\{P\}\ \{Q\}; \texttt{let}\ x = f\ (\texttt{fun}\ \overline{w} \Rightarrow d)\ \overline{v}\ \texttt{in}\ c\ x\ \rightsquigarrow_{\mathrm{x}} \texttt{xapp}\ s\ (\texttt{fun}\ \overline{q} \Rightarrow \square).\ T;\ (\overline{\psi}[\overline{v}/\overline{x}, (\texttt{fun}\ \overline{w} \Rightarrow d)/F, (\texttt{fun}\ \overline{q} \Rightarrow \square)/I], \varphi)}
$$

Fig. 10. Transformation from OCaml to CFML proof scripts (sequences of tactics)

ALGORITHM 3.1: Dynamic Program Alignment

```
Procedure DPA(p, p′)
    Input: Old program p, and a new program p′
    Output: Map α from program points in p′ to p
    R, R′ ← ProgramPoints(p), ProgramPoints(p′)
    for input in GenRandInputs(p) do
        tr, tr′ ← Trace(p, input), Trace(p′, input)
        for ⟨ρ, ρ′⟩ in R × R′ do
            ps ← ProgramStateAt(tr, ρ)
            ps′ ← ProgramStateAt(tr′, ρ′)
            scores[ρ′][ρ] ← Score(ps, ps′)
    for ρ′ in R′ do
        α[ρ′] ← HighestScore(scores[ρ′])
    return α

Procedure Score(ps, ps′)
    Input: Old and new program state ps, ps′
    Output: Integer score
    H, H′ ← Heap(ps), Heap(ps′)
    S, S′ ← Stack(ps), Stack(ps′)
    vals ← Normalize(H ∪ S)
    vals′ ← Normalize(H′ ∪ S′)
    score ← 0
    for val in vals do
        if val ∈ vals′ then
            score ← score + Size(val)
    return score
```

xapp-hof generates a "blank" invariant `fun` $\overline{q} \Rightarrow \square$, whose hole $\square$ is going to be filled up later with candidates. Our next step is to mine the building blocks for constructing such candidates from the old program's proofs using the *program alignment* technique.

3.2 Computing Program Alignment

Program alignment is the mapping from program points in the new program to those in the old one with *similar* program states—a crucial component in determining possible invariant candidates to fill holes in the reconstructed proof script. Note that alignment is a many-to-one correspondence: many new program points can map to one old program point (but not vice versa). Algorithm 3.1 shows the main step of computing the alignment. To measure similarity of two program states, it uses a simple `Score` function that only takes into the account the sizes of the aggregate values (*e.g.*, arrays and lists) present in both program states. Prior to computing the score, the `Normalize` function transforms aggregate data types with known canonical projections to a unified representation. For instance, both arrays and sequences are represented as lists; the user can supply their own projections. This enables Sisyphus to compare program states bearing the same logical values represented in different OCaml datatypes (*e.g.*, `seq` and `array` in `to_array`).

To compute the alignment α, Sisyphus generates a series of random inputs and executes both versions of the program on them to obtain execution traces tr and tr' correspondingly. It records the program state at each program point, computes the similarity to program points in the alternative version via the `Score` function, and finally associates each program point in the new program with the most similar program point in the old program.

3.3 Synthesising Invariant Candidates

Algorithm 3.2 provides an overview of the algorithm for template-based invariant synthesis. In our approach, invariants are encoded as Coq functions that take one or more parameters and construct a logical proposition that constrains the symbolic SL state (*e.g.*, the invariant (5)).

Recall that, as detailed in Sec. 3.1, we generate a "blank" invariant for every application of a higher-order function. Initially, this "blank" invariant consists of a single hole. To aid our synthesis, in its first step (Algorithm 3.2, left), the algorithm constructs an *invariant template* $I_\square$ for a program point ρ' in the new program. It does so by collecting all heaplets in the program's symbolic state, obtained by symbolic execution, immediately before ρ'. Each heaplet has the shape $v \mapsto \Pi\ \overline{e_i}$, where v is a logical or program-level variable, and Π is an OCaml data type constructor (*e.g.*, `Array` or `ref`),

ALGORITHM 3.2: Synthesis of invariant candidates

Procedure `MakeTemplate`(ρ', vs)
  Input: New program point ρ', invariant parameters vs
  Output: Invariant template $I_\square$
  $I_\square \leftarrow$ emp
  for $v \mapsto \Pi\ \overline{e_i}$ **in** `Heaplets`(ρ') **do**
    $I_\square \leftarrow I_\square * (v \mapsto \Pi\ \_)$
  for v **in** vs **do**
    $I_\square \leftarrow I_\square \wedge (v = \_)$
  return $I_\square$

Procedure `SynthesizeCandidates`($I_\square$, α, $\mathcal{P}$, p', ρ')
  Input: Invariant template $I_\square$, DPA α, old proof $\mathcal{P}$, new program p', new program point ρ'
  Output: List of invariant candidates matching $I_\square$
  $I_{old} \leftarrow$ `GetInvariant`($\alpha(\rho')$)
  sketches $\leftarrow$ `GetExpressionSketches`(I_{old})
  atoms $\leftarrow$ `CollectConsts`($\mathcal{P}$) $\cup$ `CollectFuns`($\mathcal{P}$)
  logvars $\leftarrow$ `LogicalVars`(specification of $\mathcal{P}$)
  return `EnumSynthesis`(*sketches*, *atoms* $\cup$ *logvars*, $I_\square$)

possibly applied to arguments $\overline{e_i}$ that are being replaced by a hole _. The template heaplets are then conjoined using the SL $*$ connective. Additionally, for every parameter to the invariant (obtained from the type of the corresponding lemma that requires it), we generate equality propositions of the form $v = \_$. For example, the template that was elaborated into the form (7) is

```
fun acc t => (a ↦ Array _) ∧ (acc = _) ∧ (t = _)
```  (8)

After inferring the template $I_\square$ of the desired invariant, we construct concrete candidates by filling it (Algorithm 3.2, right). To do so, the algorithm first retrieves the corresponding invariant from the aligned location in the old program and uses it to construct a set of sketches (*i.e.*, expressions with holes). It then collects constants and function symbols from the old proof, as well as logical variables from the ascribed specifications and uses those to fill the holes in the sketches, themselves used to instantiate holes in the template, thus completing the enumerative synthesis of candidates.

4 PROOF-DRIVEN INVARIANT TESTING

In this section, we describe the technique at the heart of Sisyphus, *proof-driven testing*, a means to test the validity of invariant candidates. We achieve that goal by automatically generating tests to check computable properties of program values and states, from proofs of higher-order lemmas.

To build an intuition for this process, let us start our reasoning by considering a simple *computable* property P on natural numbers, defined as $P\ n \triangleq 1 + n = n + 1$. A standard way to prove that P holds on *all* natural numbers is by induction, *i.e.*, by providing proofs of the facts $H_0 : P\ 0$ (*i.e.*, the induction base) and $H_i : \forall i, P\ i \to P\ (i + 1)$ (*i.e.*, the induction transition).

H_0 and H_i are *proof terms*: following their types, we can compose them into the expression

$$H_i\ 2\ (H_i\ 1\ (H_i\ 0\ H_0))) : P\ 3 \tag{9}$$

whose type will be $P\ 3$, therefore making it a proof term for the fact that the property P holds on the number 3. The proof of $P\ 3$ is, therefore, constructed by applying H_i, an inductive step, to two arguments: a concrete value 2, and a sub-term constructed to have the type $P\ 2$. This sub-term proving $P\ 2$ recursively is also constructed by applying H_i again, but this time to 1 and $(H_i\ 0\ H_0)$. The remarkable observation supported by this example is that this proof, *i.e.*, that P holds on the value 3, *contains* within it the information that P must necessarily also hold on the values 2, 1, and 0—from the fact that within its construction, it contains sub-terms proving $P\ i$ for $i \in \{0, 1, 2\}$.

What does this have to do with testing invariants? Imagine that we *don't know* if P holds for all numbers. A natural (pun intended) thing to do in this case would be to first *test* that P holds on *some* numbers, *e.g.*, 0, 1, 2, *etc.*, up to some large number (*e.g.*, 3). On the other hand, *assuming* that P can be proven by induction, we know that there *must* be witnesses (*i.e.*, proof terms) H_0 and H_i for statements $P\ 0$ and $\forall i, P\ i \to P\ (i + 1)$, correspondingly, such that for any concrete n, $P\ n$ can be derived by repeated application of H_i to smaller numbers and H_0, as demonstrated by (9).

$$\textsc{red-app}\quad (\texttt{fun}\ x : T, t)\ t' \Downarrow_{\mathrm{r}} t[t'/x] \qquad \textsc{red-constr}\quad \frac{t_i \Downarrow_{\mathrm{r}} t_i'}{C\ \overline{t_i} \Downarrow_{\mathrm{r}} C\ \overline{t_i'}} \qquad \textsc{red-match}\quad (\texttt{match}\ C_k\ \overline{t_j}\ \texttt{with}\ \overline{C_i\ \overline{x_j} \to t_i'}) \Downarrow_{\mathrm{r}} t_k'[\overline{t_j}/\overline{x_j}]$$

$$\textsc{red-fun}\quad \frac{t \Downarrow_{\mathrm{r}} t'}{(\texttt{fun}\ x : T \Rightarrow t) \Downarrow_{\mathrm{r}} (\texttt{fun}\ x : T \Rightarrow t')} \qquad \textsc{red-fix}\quad \frac{t_0' = C\ \overline{t'} \qquad F = (\texttt{fix}\ f\ \overline{x_i : T_i}\ \Rightarrow t)}{F\ \overline{t_i'} \Downarrow_{\mathrm{r}} t[F/f, \overline{t_i'}/\overline{x_i}]}$$

Fig. 12. Selected reduction rules for CIC_ω

By combining these two observations, we can have our main revelation: For any *concrete* n, the proof term t_n for the proposition $P\ n$ *contains* sub-terms of types $P\ i$ for $i < n$ within itself; by traversing t_n, we can identify each of those types $P\ i$ and convert it into a *dynamically checkable assertion* over a boolean expression $P(i)$, which *must not fail* if P holds universally.

As it turns out, the same observation, *i.e.*, that a proof-term for a higher-order lemma about a property can be used for extracting tests for this property, is not specific to statements about natural numbers! Exactly the same principle applies to testing properties of *program executions*, *e.g.*, when P is a guessed *invariant over program states* parameterised by program and logical variables. In this case, concrete values of theose variables can be supplied by subterms of the corresponding lemma proofs that are partially evaluated on concrete inputs.

4.1 Instantiating Proof Terms with Concrete Inputs

For the rest of this section we restrict our language to a subset of CIC_ω, the calculus of Coq, its syntax adopted from Timany and Jacobs (2015) and given in Fig. 11.

$$\begin{array}{lcl} v & & \text{variable names} \\ t & ::= & v \mid t\ t' \mid \forall x : t, t' \mid \texttt{fun}\ x : t \Rightarrow t' \\ & & \mid\ C\ \overline{t} \mid \texttt{match}\ t\ \texttt{with}\ \overline{C_i\ \overline{v_i} \Rightarrow t_i} \\ & & \mid\ \texttt{fix}\ f\ \overline{v_i : t_i} : t \Rightarrow t' \end{array}$$

Fig. 11. Syntax of selected CIC_ω terms

The first step in our process is to instantiate higher-order lemmas with concrete arguments and reduce their proof terms to a head-normal form using rules from Fig. 12. Once an expression is in head-normal form, the types of its non-reducible subterms will contain the property of interest applied to concrete values, allowing for further test extraction.

As an example, consider instantiating and reducing the induction principle `nat_ind` for natural numbers. The induction principle is implemented as a higher-order lemma whose type is

$$\forall (P : \texttt{nat} \to \texttt{Prop}), P\ 0 \to (\forall i, P\ i \to P\ (i+1)) \to \forall (x : \texttt{nat}), P\ x \tag{10}$$

and whose proof term is a recursive function that pattern-matches over two constructors of `nat`:

$$\texttt{fun}\ P\ H_0\ H_i \Rightarrow \texttt{fix}\ F\ (n : \texttt{nat}) : P\ n \Rightarrow\ \texttt{match}\ n\ \texttt{with}\ \mid 0 \Rightarrow H_0\ \mid S\ n' \Rightarrow H_i\ n'\ (F\ n') \tag{11}$$

According to its (dependent) type (10), `nat_ind` takes four parameters, of which the first three are either properties or their proofs (which is indicated by their types), while the fourth one has type `nat`. In the scenario of our interest, we apply lemmas to concrete (*i.e.*, non-proposition) arguments, as those correspond to values that can be used for testing the validity of propositions (in this case, passed as arguments to P). Therefore, to formally replicate the example (9), we should instantiate `nat_ind` with three *symbolic* values P, H_0, and H_i indicating proof terms or properties whose values are irrelevant for test generation (but whose types are important), and the fourth *concrete* value 3. By repeatedly applying the rules RED-APP, RED-FIX, and RED-MATCH to this expression, we reduce it to the following partially-evaluated form:

$$\texttt{nat\_ind}\ P\ H_0\ H_i\ 3\ \Downarrow_{\mathrm{r}}^{*}\ H_i\ 2\ (H_i\ 1\ (H_i\ 0\ H_0)) \tag{12}$$

The reduced form on the right reveals the familiar sub-terms whose types contain applications of P to concrete values. We proceed by using this result to extract a testable specification for P.

$$\textsc{vanilla-app}\ \frac{\Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{v}} c'}{\Gamma; t\ t' \rightsquigarrow_{\mathrm{v}} c\ c'} \qquad \textsc{vanilla-ind}\ \frac{C\ \overline{t_i} : \tau \qquad \Gamma; t_i \rightsquigarrow_{\mathrm{v}} c_i}{\Gamma; C\ \overline{t_i} \rightsquigarrow_{\mathrm{v}} C\ \overline{c_i}} \qquad \textsc{vanilla-fun}\ \frac{x, \Gamma; t \rightsquigarrow_{\mathrm{v}} c}{(\mathsf{fun}\ x : T \Rightarrow t) \rightsquigarrow_{\mathrm{v}} (\mathsf{fun}\ x \Rightarrow c)}$$

$$\textsc{vanilla-match}\ \frac{t \rightsquigarrow_{\mathrm{v}} c \qquad \overline{x_i}, \Gamma; t_i' \rightsquigarrow_{\mathrm{v}} c_i}{\Gamma; \mathsf{match}\ t\ \mathsf{with}\ \overline{C_i\ \overline{x_i : T_i} \to t_i'} \rightsquigarrow_{\mathrm{v}} \mathsf{match}\ c\ \mathsf{with}\ \overline{C_i\ \overline{x_i} \to c_i}} \qquad \textsc{vanilla-erase}\ \frac{t : \tau \qquad \tau : \mathsf{Prop}}{t \rightsquigarrow_{\mathrm{v}} ()}$$

$$\textsc{vanilla-fix}\ \frac{f, \overline{x_i}, \Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t_i' \rightsquigarrow_{\mathrm{v}} c_i'}{\Gamma; (\mathsf{fix}\ f\ \overline{x_i : T_i} \Rightarrow t)\ \overline{t_i'} \rightsquigarrow_{\mathrm{v}} \mathsf{let\ rec}\ f\ \overline{x_i} = c\ \mathsf{in}\ f\ \overline{c_i'}}$$

$$\textsc{extract-prop}\ \frac{\Gamma \vdash t : \tau \qquad \tau = t_1 \ldots t_n \to \tau' \qquad \tau' : \mathsf{Prop}}{\Gamma; t \rightsquigarrow_{\mathrm{e}} (\mathsf{fun}\ x_1 \ldots x_n \Rightarrow ())} \qquad \textsc{extract-test}\ \frac{\Gamma \vdash t : \tau \qquad \tau : \mathsf{Prop} \qquad \Gamma \vdash \tau \rightsquigarrow_{\mathrm{t}} p \qquad \Gamma; t \rightsquigarrow_{\mathrm{e}} c}{\Gamma; t \rightsquigarrow_{\mathrm{e}} \mathsf{assert}\ p; c}$$

$$\textsc{extract-gen}\ \frac{\Gamma; v\ \overline{t_i} \rightsquigarrow_{\mathrm{f}} c}{\Gamma; v\ \overline{t_i} \rightsquigarrow_{\mathrm{e}} c}$$

Fig. 13. Vanilla extraction rules from CIC$_\omega$ to OCaml (top) and new rules for test extraction (bottom). Highlighted premises are domain-specific and are instantiated for a particular set of properties.

4.2 Extracting Tests from Reduced Proof Terms

The top part of Fig. 13 presents a simplified form of the extraction relation ($\rightsquigarrow_{\mathrm{v}}$) from Coq to OCaml (Letouzey 2008). The key rule that is relevant for our purposes is VANILLA-ERASE, which handles the removal of logical parameters for the sake of producing efficient executable OCaml code: when a Coq expression t has type in **Prop**, which means that it is a proof term, then Coq's vanilla extraction simply returns the unit value `()` without even inspecting t's structure. As we wish to visit the sub-terms of logical properties to generate tests, this rule must be suitably adapted.

Our extraction mechanism (Fig. 13, bottom), updates $\rightsquigarrow_{\mathrm{v}}$ with the additional three rules : EXTRACT-TEST, EXTRACT-PROP, and EXTRACT-GEN (which will be explained in Sec. 4.3). The rule EXTRACT-PROP extends the extraction making it traverse terms with type in **Prop**. The key addition is the rule EXTRACT-TEST, which implements our earlier intuition that sub-terms witnessing a property can encode dynamically checkable assertions for which the property must hold. In particular, whenever we visit a sub-term inhabiting a type τ that can be converted into an executable test p (via a *reflection* step $\rightsquigarrow_{\mathrm{t}}$), the extraction emits an assertion that p must hold within the extracted computation. The heavy lifting in this translation is done by the $\rightsquigarrow_{\mathrm{t}}$ reflection rule, which is domain-specific and is instantiated by the user for a particular set of decidable properties.

To wrap up, let us get back to our running example. We shall instantiate $\rightsquigarrow_{\mathrm{t}}$ with the bespoke reflection function (on the right) tailored for our property $P\ n \triangleq 1 + n = n + 1$, and use the extraction rules to convert the normalised proof term (12) into the test-specification (13).

$$\textsc{nat-refl-eq}\ \frac{\Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{v}} c'}{\Gamma; (t = t') \rightsquigarrow_{\mathrm{t}} c = c'}$$

$$\begin{aligned} &H_i\ 2\ (H_i\ 1\ (H_i\ 0\ H_0)) \ \rightsquigarrow_{\mathrm{t}}^* \\ &\mathbf{assert}\ (1 + 3 = 3 + 1); \mathbf{assert}\ (1 + 2 = 2 + 1); \mathbf{assert}\ (1 + 1 = 1 + 1); \mathbf{assert}\ (1 + 0 = 0 + 1) \end{aligned} \tag{13}$$

Notice that the four asserts in the OCaml program above correspond to the subterms $(H_i\ 2\ \ldots)$, $(H_i\ 1\ \ldots)$, $(H_i\ 0\ H_0)$, and H_0, as each one inhabits the type of a concrete instantiation of P.

4.3 Testing Properties in Separation Logic

Leaving the world of properties over `nat`, let us now tackle our original goal, using *proof-driven testing* to test separation logic properties, such as invariants. In particular, we consider an embedding of CFML in Coq, and present how our extraction rules (*cf.* Fig. 14) can be extended for this domain.

$$\textsc{refl-emp}\ \frac{}{\Gamma; \mathsf{emp} \rightsquigarrow_{\mathrm{t}} \mathsf{true}} \qquad \textsc{refl-sep}\ \frac{\Gamma; t \rightsquigarrow_{\mathrm{t}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{t}} c'}{\Gamma; t * t' \rightsquigarrow_{\mathrm{t}} c \,\&\&\, c'} \qquad \textsc{refl-pts}\ \frac{\Gamma; t \rightsquigarrow_{\mathrm{t}} c \qquad \mathcal{R}(f) = F}{\Gamma; v \mapsto f\ t \rightsquigarrow_{\mathrm{t}} F\ v = c}$$

$$\textsc{extract-xapp}\ \frac{\Gamma; H_c \rightsquigarrow_{\mathrm{e}} c \qquad \Gamma; P \rightsquigarrow_{\mathrm{t}} p}{\Gamma; \mathsf{xapp}\ P\ Q'\ Q\ f\ v\ b\ H_f\ H_c \rightsquigarrow_{\mathrm{f}} \mathsf{assert}\ p;\ \mathsf{let}\ x\ =\ f\ v\ \mathsf{in}\ c\ x}$$

Fig. 14. A reflection for SL properties (top); An extraction rule for XAPP (bottom)

Consider a prototypical CFML rule XAPP for function applications, first seen in Sec. 2.1.2. When embedded as a lemma (xapp) within Coq, XAPP takes the following type:

$$\forall (P : \mathsf{hprop})\ Q'\ Q\ f\ v\ b, \quad (\{P\}\ (f\ v)\ \exists x, \{Q'\ x\}) \rightarrow (\forall x, \{Q'\ x\}\ b\ x\ \{Q\}) \rightarrow \{P\}\ (\mathbf{let}\ x\ =\ f\ v\ \mathbf{in}\ b\ x)\ \{Q\} \tag{14}$$

That is, the xapp lemma takes eight arguments, where the first three are properties over the symbolic heap (with type hprop), the next three take values, and the last two take SL proofs. Notice that, from the type of xapp, for the heap properties we pass as input (*i.e.*, P, Q', and Q) no witness is passed to the lemma. Rather, the lemma expects proofs of Hoare triples such as $\{P\} \ldots \exists x, \{Q'\ x\}$. We do not show the proof term of xapp here, but it does not construct proof terms inhabiting P, Q, *etc.*, either; instead, it composes the witnesses for the lemma's argument Hoare triples. As such, if we were to apply our extraction rules from *cf.* Fig. 14 to a concrete instantiation of xapp, the resulting program would *not* include any explicit assertions about properties over the heap.

Our goal is thus to extend our test extraction to test SL properties that hold over the heap. We first instantiate the reflection relation $\rightsquigarrow_{\mathrm{t}}$ with the rules REFL-EMP, REFL-SEP, and REFL-PTS (*cf.* Fig. 14, top) to handle terms in hprop, CFML's encoding of heap properties. The key rule in this encoding is REFL-PTS that uses a mapping $\mathcal{R}$ to extract heap predicates (*e.g.*, Array) to corresponding OCaml functions that manipulate their logical contents (*e.g.*, of_list). Applying these rules, we can now reflect logical assertions over the heap into executable OCaml tests similar to the program in Fig. 7.

To assert these SL properties within our test, we must tune our extraction procedure such that the test programs appropriately maintain the heap, testing the predicates derived from the passed heap assertions at the right program points. For this purpose, we now turn to the EXTRACT-GEN rule (*cf.* Fig. 13), and instantiate it for CFML, extending $\mathcal{F}$ to map CFML's reasoning rules to transformations that capture their semantics. For example, EXTRACT-XAPP (*cf.* Fig. 14, bottom) presents the corresponding instantiation for XAPP. The essential part of the rule is the bespoke invocation of the $\rightsquigarrow_{\mathrm{t}}$ procedure that converts the precondition P to the assertion **assert** p installed right before the call to f. The rule EXTRACT-XAPP does not convert the other two argument properties of xapp, Q' and Q, which both take value arguments, into assertions. This extension is straightforward but would require to generalise $\rightsquigarrow_{\mathrm{t}}$ to handle parameterised properties; in the interest of time we did not carry out this exercise. The encodings for the remaining rules of CFML follow the same strategy and are not presented here, but can be found in our implementation.

ALGORITHM 4.1: Invariant testing

Procedure TestInvariant(p', f, t_f, I)
  Input: Program p', HOF f, proof term for f $t_f : args \rightarrow I \rightarrow$ Spec, invariant I
  Output: Passes if no assertion is violated
  $st, args \leftarrow$ RunUpto(p', f, GenInput(p))
  $t \leftarrow$ Instantiate($t_f, args, I$)
  Reduce($t \Downarrow_{\mathrm{r}} t'$)
  Extract($t' \rightsquigarrow_{\mathrm{e}} \delta$)
  RunFromState($st, \delta\ ()$)

Putting it all together, Algorithm 4.1 describes how SISYPHUS uses proof-driven testing to prune candidate invariants when instantiating holes in the generated proof skeleton for a given program. When testing an invariant candidate I for a higher-order function f, whose application occurs in the code of the new program p', TestInvariant first executes the enclosing program p' on a randomly

generated input and observes both the program state st at the call site and the concrete arguments to f. The concrete arguments and the invariant are passed to t_f to construct an instantiated reduced proof term t, which is extracted into a OCaml test program δ, which is executed in the context st. If δ raises an exception during its execution, then I is invalidated and can be pruned away.

5 IMPLEMENTATION AND EVALUATION

We have implemented Sisyphus in 19k lines of OCaml. The table on the right summarises the distribution of implementation effort, in terms of the approximate lines of code of each component of the development.[5] In order to implement proof-driven testing, we include a lightly modified version of Coq's reduction algorithm to allow reduction within proof terms. Note that our CFML instantiation of proof-driven testing only requires around 1600 specific LOC, which is likely a good estimate of the additional effort that one might expect in order to extend Sisyphus to handle proof repair in other Coq embeddings of Separation Logic. In order to dispatch any obligations generated during the repair process, we implemented a domain-specific solver as a small collection of Ltac-based tactics (~700 LOC). An initial implementation of the tool used Z3 (de Moura and Bjørner 2008) for this purpose; however, we found that it was not effective at reasoning about the generated obligations, taking several minutes on even simple goals.

Component	LOC
Proof-skeleton generator (§3.1)	2700
Program alignment (§3.2)	1400
Enumerative synthesis (§3.3)	1600
Modified Coq reduction (§4.1)	7600
Proof-driven test extraction (§4.2)	2000
Reflection & extraction for CFML (§4.3)	1600
Miscellanea (*e.g.*, logging, stats, etc)	1900
Total	18800

We conducted an empirical evaluation of Sisyphus to answer the following research questions:

- **RQ1**: Is Sisyphus effective at repairing proofs for real world programs: can it find correct invariants and how much manual effort is required to complete the proof?
- **RQ2**: How efficient is Sisyphus: does it generate invariants in a reasonable amount of time?
- **RQ3**: What are the classes of changes in programs that Sisyphus handles poorly or not at all?

Benchmarks. In order to answer these questions, we constructed a benchmark suite of 14 evolved OCaml programs as summarised in Tab. 2. Of these programs, a majority, 10, were drawn from real-world code-bases, found by mining the version control of popular OCaml libraries (*e.g.*, Jane Street's `base`, `containers`, *etc.*—*cf.* the accompanying artefact for their exact provenance), with the remaining ones constructed by us.

Programs were selected to exhibit the use of a diverse range of refactorings. In particular, we classify the changes of programs in our benchmarks into four classes: (1) IterOrd, for changes in iteration order (*cf.* the change in `to_array`), (2) DataStr, for changes in intermediate data structures, (3) Mutable/Pure for the transformation of programs using loops with mutation to pure variants, and (4) Pure/Mutable for the converse. The benchmarks we consider manipulate a range of common OCaml data-structures: arrays, lazy sequences, mutable singly-linked-lists (SLL), stacks, queues and trees. The main roadblock in tackling larger classes of data structures for the case studies was in the lack of available verified implementations of these data structures. For example, though the CFML development provides a simplified version of the OCaml Map data structure, implemented using a balanced binary tree, it does not verify most of its associated functions.

Methodology. Our methodology to evaluate Sisyphus on these benchmark programs was as follows: For each library function in our benchmark suite, we first extracted the function and its dependencies into a standalone file, replacing the use of `for`- and `while`-loops with suitable higher-order loop combinators (*cf.* Sec. 3.1). Then, in order to construct the initial proofs for each

[5]The accompanying artefacts, Sisyphus implementation and benchmarks, are available online (Gopinathan et al. 2023).

Table 2. Categorisation of benchmark programs by the type of program changes and language features and data structures they use (left); Comparison of additional effort required to dispatch the obligations in proof scripts produced by Sisyphus (right). † indicates when the new version of the program was constructed by the authors, and ‡ indicates when both versions of the programs were constructed by the authors.

Example	Data Structure	Refactoring	Time (old)			Time (new)	# Admits / # Obligations
			Spec	Proof	Total		
seq_to_array	Array, Seq	IterOrd, DataStr	1hrs	1hr	2hrs	17m	3/5
make_rev_list†	Ref	Mutable/Pure	5m	5m	10m	-	0/2
tree_to_array†	Array, Tree	IterOrd, DataStr	4hrs	1hr	5hrs	18m	2/4
array_exists	Array	Mutable/Pure	10m	20m	30m	12m	2/4
array_find_mapi	Array, Ref	Pure/Mutable	30m	1hr	1.5hrs	12m	2/5
array_is_sorted	Array	Pure/Mutable	1hr	3hrs	4hrs	2m	2/5
array_findi	Array	Pure/Mutable	30m	1hr	1.5hrs	9m	3/7
array_of_rev_list	Array	DataStr	5m	1hr	1hr	3m	2/3
array_foldi	Array	Pure/Mutable	10m	5m	15m	-	0/1
array_partition	Array	DataStr	30m	2hrs	2.5hrs	5m	3/3
stack_filter‡	Stack	DataStr	1hr	30m	1.5hrs	11m	3/3
stack_reverse‡	Stack	DataStr	1.5hrs	30m	2hrs	30s	1/1
sll_partition‡	SLL	Mutable/Pure, IterOrd	1hr	1hr	2hrs	-	0/2
sll_of_array‡	Array, SLL	IterOrd	1.5hrs	30m	2hrs	-	0/1

program, one of the authors manually verified the implementations of each of the benchmark programs and recorded the time taken. Sisyphus was then invoked to construct proofs for the new versions of each benchmark program. Finally, any residual obligations that were left as admits by the tool were proven manually by another author and timed. Both authors were equally familiar with Coq/CFML before writing any proofs, and the purpose of this experiment was to evaluate the manual effort in proving the remaining obligations in relation to the initial proof effort.

RQ1: Effectiveness and Utility. Our experiments demonstrate that Sisyphus is effective at repairing the proofs of real world programs. Sisyphus was able to automatically construct new proofs for all programs in our benchmarks: *all* synthesised invariants were valid, and we were able to manually prove any remaining proof obligations. The fourth to the sixth columns of Tab. 2 describe the comparison of the manual effort, in terms of the time taken, to prove and specify the original programs in comparison to using Sisyphus and manually dispatching remaining obligations. The last column of the table (# Admits / # Obligations) describes the number of verification conditions that Sisyphus was unable to dispatch automatically and were left as admits for the user to prove. We were able to construct Coq proofs for all such remaining sub-goals manually.

As can be seen from the table, all obligations took fewer than 20 minutes to dispatch, while specifying and proving the original programs took considerably longer, demonstrating the utility of Sisyphus for maintaining verified code-bases. We found that most of the challenge in completing proofs was in reasoning about properties of the involved functions that were not considered in the original development but relied on by the generated invariants. For example, in our case study for `array_partition`, the generated invariant made use of an expression of the form `filter` p (`filter` $p\ \ell$). Since filtering is idempotent, such repetition is redundant, but no such property had been proven before, so dispatching the obligation required us to prove it manually.

RQ2: Efficiency. Our experiments were conducted on a commodity laptop (3.5 GHz Apple M2 MacBook Air with 8GB RAM). We have found Sisyphus to be efficient at repairing proofs, taking fewer than 2 minutes to execute on most examples in our benchmarks.

Table 3. Statistics of Sisyphus when repairing proofs of verified OCaml programs. The first 3 columns list the total number of invariant candidates that were generated and their breakdown by heap and pure parts, to be tested independently. The last 5 columns describe the time taken by Sisyphus, with the breakdown per individual component: generation of candidates, extraction of tests, running the tests, and the remaining tasks, which include computing program alignment and interaction with the Coq runtime.

Example	Candidates			Time (s)				Total (s)
	Heap	Pure	Total	Generation	Extraction	Testing	Remaining	
seq_to_array	3.4×10^7	1.8×10^4	6.2×10^{11}	28.57	1.95	20.36	5.28	58
make_rev_list	-	30	30	$\leq 10ms$	3.36	$\leq 10ms$	11.95	15
tree_to_array	5.0×10^6	8.2×10^3	4.0×10^{10}	6.75	1.95	2.98	13.32	25
array_exists	-	25	25	$\leq 10ms$	3.30	$\leq 10ms$	13.23	17
array_find_mapi	13	34	442	$\leq 10ms$	2.13	$\leq 10ms$	13.95	17
array_is_sorted	64	70	4.5×10^3	$\leq 10ms$	2.04	$\leq 10ms$	15.38	18
array_findi	4.9×10^3	34	1.7×10^5	$\leq 10ms$	2.13	$\leq 10ms$	19.07	22
array_of_rev_list	1.5×10^6	-	1.5×10^6	1.72	2.82	0.96	15.62	21
array_foldi	24	-	24	$\leq 10ms$	488.89	$\leq 10ms$	15.00	504
array_partition	1.6×10^6	-	1.6×10^6	3.51	69.73	2.62	17.53	95
stack_filter	71	-	71	$\leq 10ms$	81.88	$\leq 10ms$	21.53	104
stack_reverse	22	-	22	$\leq 10ms$	88.42	$\leq 10ms$	16.94	105
sll_partition	630	-	630	$\leq 10ms$	426.62	$\leq 10ms$	16.43	443
sll_of_array	2.4×10^4	-	2.4×10^4	0.02	55.98	0.01	13.33	69

In the two cases where Sisyphus takes longer than 2 minutes, `array_foldi` and `sll_partition`, most of the time is spent on performing the extraction of tests itself, rather than generating and pruning candidates. Our implementation of proof-driven testing has not been particularly optimised, simply reusing Coq's infrastructure to implement extraction, so further improvements could be obtained by adopting a more specialised implementation. The second to fourth columns of Tab. 3 list the number of generated invariant candidates; Sisyphus uses a lazy generation strategy, so we only record the number of candidates until Sisyphus finds a suitable invariant or gives up. Note that for many of the benchmarks, Sisyphus is able to use program alignment to considerably reduce the space of candidates, often leaving fewer than 100 candidates. Furthermore, whenever possible Sisyphus conducts testing of pure and heap-related parts of invariant candidates independently, rather than by taking their product, thus contributing to the efficiency of the search.

RQ3: Failure Modes. An important assumption of Sisyphus's repair process is that components of the old proof, such as the lemmas and functions that it uses, will be sufficient to prove the new program correct. As we have seen, this is not always the case. In the case of `array_partition`, Sisyphus was unable to fully automatically repair the proof, as the new proof required a lemma about repeated filters that had not been present in the old proof. In cases where the new program requires use of a function not present in the old proof, Sisyphus's search space will not even contain an invariant that can pass proof-driven testing, and the repair process will fail entirely.

For example, consider an optimised version of `to_array` on the right that uses a batching strategy: instead of collecting the elements of the sequence into a list, it accumulates a list of batches of elements and then separately inserts each batch into the result array as on the right. Sisyphus will fail to repair this program as the invariants for its loops will require an operation to reason about flattening lists of lists; however, the old proof for `to_array` does not make use of any functions that can capture this operation.

```
let batches = (* .. *) in
let res =
   Array.make (* .. *) in
List.iter (fun batch ->
  let dst = (* .. *) in
  Array.copy batch res dst)
  batches
```

6 RELATED WORK

Proof refactoring and repair. Most of the existing tools for automated proof refactoring allow for direct manipulations of standalone proofs while preserving their validity. Levity is a tool for the Isabelle/HOL proof assistant that simplifies the motion of lemmas between different packages while ensuring that the automation procedures relying on those lemmas do not break (Bourke et al. 2012). Tactitian (Adams 2015) is a tool for refactoring proof scripts in HOL Light. Refactor-Agda (Wibergh 2019) and Chick (Robert 2018) are refactoring tools for programs in dependently-typed languages that can serve as proof terms to some theorems. None of these tools address the problem of adapting a proof *about a program* in response to changes in that program.

The work by Ringer (2021) focuses on proof repair in two particular scenarios: (*i*) synthesis-by-example of patches to proofs of theorems whose statements were changed to use new data types (Ringer et al. 2018) and (*ii*) equivalence-preserving changes in data types used by a verified functional program and its specification (Ringer et al. 2021, 2019b). Neither of these approaches address arbitrary local changes in the code of a verified program; nor do they consider foundational verification of imperative programs, via embedded program logics or otherwise.

Techniques from non-dependently typed provers. Our work draws some parallels with earlier research in non-dependently typed proof assistants by Matichuk (2012) on automatic extraction of function annotations for programs in Isabelle/HOL. Matichuk describes a technique that operates on proofs about stateful programs in monadic Hoare logic to extract intermediate state assertions into a standalone set of function annotations, that can then be re-used to verify other inter-dependent properties. While this approach has similarities to our proof-driven-testing algorithm, there are some constraints arising from the non-constructive setting that limit the generality of this technique.

In particular, Matichuk's approach requires proofs to use a particular monadic Hoare logic that has been adjusted to collect intermediate assertions, while proof-driven testing allows extracting tests from a larger class of theorems in pre-existing logics and is able to obtain this information for free by introspecting the proof terms "as is". Furthermore, the machinery in the 2012 paper only considers first-order imperative programs, while our proof-driven testing mechanism enables inferring invariants for higher-order programs with combinators. That said, *assuming* that Matichuk's approach could be extended to proofs about higher-order combinators such that it stored annotations describing how those proofs instantiate the invariants (as in our Fig. 8), we speculate that such annotations could be used to generate tests similar to the one in Fig. 7.

KeYTestGen (Ahrendt et al. 2016) and StaDy (Petiot et al. 2014) take advantage of annotations required by correctness proofs to generate tests with a high degree of coverage. On a conceptual level, in contrast with KeYTestGen and StaDy, our approach is inherently *higher-order*, as it extract tests for invariants (*i.e.*, *properties* of programs) from proofs of lemmas that use those invariants, while the mentioned tools target testing of programs themselves.

Invariant inference. Inference of loop invariants for imperative programs using static (Flanagan and Leino 2001; Henzinger et al. 2004; Qin et al. 2013) and dynamic (Ernst et al. 1999, 2000) analysis, as well as machine learning techniques (Si et al. 2018) is a well-studied research topic.

Magill et al. (2006) were amongst the first to describe a heuristic procedure for automatically inferring SL invariants via static analysis relying on predicate abstraction (Das et al. 1999; Graf and Saïdi 1997) for programs manipulating linked lists. In contrast with Magill et al.'s work, our approach is based on dynamic analysis techniques and does not require a predefined set of predicates, neither heap- nor pure ones, as those are drawn from the old proofs.

The data-driven tools Locust (Brockschmidt et al. 2017) and SLING (Le et al. 2019) use dynamic analysis to derive SL formulas describing the shape of a state manipulated by a C program as well as

the program's loop invariants. Both of these tools target proofs of memory safety and do not consider full functional correctness *w.r.t.* arbitrary specifications, limiting the language of pure assertions to the first-order logic with arithmetic comparisons. For the validation of inferred invariants, Locust relies on the automated non-foundational verification tool GRASShopper (Piskac et al. 2014), while the authors of SLING report poor experience with SMT solvers and had to resort to manual (*i.e.*, non machine-assisted) invariant checking (*cf.* Sec. 5.3 of Le et al. (2019)).

Of the state-of-the-art deductive verification tools, only Why3 (Filliâtre and Paskevich 2013) allows for a limited form of invariant inference using numeric abstract domains and does not support arbitrary effectful functions or statements about shape properties of data (*e.g.*, constraining contents of an array). We are not aware of any approach comparable to Sisyphus in its ability to automatically infer complex invariants which constrain both heap and data for higher-order imperative programs, and then use these invariants to reconstruct proofs for modified programs.

Automated foundational proofs in Separation Logic. Foundational approaches to program verification encode the meta-theory and the rules of a domain-specific logic (*e.g.*, a version of SL) in terms of the logic of the host verifier (*e.g.*, Coq). In this work, we considered libraries written in a higher-order imperative language and verified in a foundational encoding of SL into Coq, which enabled proof reconstruction and efficient pruning of invariant candidates as described in Sec. 3 and 4. Our contributions are complementary to the efforts in automated foundational SL-based verification of sequential (Gonthier et al. 2011; Sammler et al. 2021) and concurrent (Mulder et al. 2022) heap-manipulating programs, as all those tools require explicit invariants.

Even though our current implementation of Sisyphus only supports a particular Coq-embedded SL, namely, CFML (Charguéraud 2020), we believe that our approach would be applicable to many other foundational SL implementations: HTT (Nanevski et al. 2010), Bedrock (Chlipala 2011), VST (Appel 2011), CHL (Chen et al. 2015), FCSL (Sergey et al. 2015), and the large family of logics based on the Iris framework (Iris Project 2022). As explained in Sec. 4.3, to support a custom SL embedding, one would have to elaborate test extraction by implementing reflection procedures for embedding-specific encodings of SL assertions and rules.

7 CONCLUSION

In this work we have presented Sisyphus, a tool for the mostly-automated repair of verified libraries. Our repair procedure is implemented through two dynamic analyses: a dynamic program-alignment for recovering relations between old and new programs, and *proof-driven testing* for quickly pruning candidate invariants, with the latter being a novel construction introduced in this work. Our experimental results show that the combination of these techniques is effective at constraining the search space of repairs, and allows Sisyphus to produce repaired proofs in only a few minutes for most cases; while the generated proofs are sometimes incomplete, the remaining proof obligations are typically of a reasonable difficulty and pose significantly less challenge than writing the initial proof. Our current implementation only supports the CFML embedding of SL; however, our techniques have been designed where possible to be parametric over the logic, so we expect can be generalised to other logics and programming languages. We leave these opportunities as exciting future work in making maintainable verified software less of a Sisyphean task.

ACKNOWLEDGMENTS

We thank Andreea Costea, Leonidas Lampropoulous, Yunjeong Lee, Peter Müller, and George Pîrlea for their feedback on drafts of this paper. We also thank the anonymous PLDI'23 PC and AEC reviewers as well as our shepherd Adam Chlipala for their constructive and insightful comments.

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and Tier 1 grant T1 251RES2108.

DATA AVAILABILITY

The software artefact accompanying this paper is available online (Gopinathan et al. 2023). The artefact contains the source code and build scripts for Sisyphus, a corpus of OCaml programs that can be used to reproduce the experimental results described in Sec. 5, a self-contained Docker file to automate setting up the development environment, and a `README` file in markdown that provides detailed step-by-step instructions for running Sisyphus and the experiments.

REFERENCES

Mark Adams. 2015. Refactoring Proofs with Tactician. In *SEFM (LNCS, Vol. 9509)*. Springer, 53–67. https://doi.org/10.1007/978-3-662-49224-6_6

Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. 2016. Proof-based Test Case Generation. In *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS, Vol. 10001. Springer, 415–451. https://doi.org/10.1007/978-3-319-49812-6_12

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel and David A. Naumann. 2020. Verified sequential Malloc/Free. In *ISMM*. ACM, 48–59. https://doi.org/10.1145/3381898.3397211

Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and Experiences in Managing Large-Scale Proofs. In *11th International Conference Intelligent Computer Mathematics (CICM) (LNCS, Vol. 7362)*. Springer, 32–48. https://doi.org/10.1007/978-3-642-31374-5_3

Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *SAS (LNCS, Vol. 10422)*. Springer, 66–87. https://doi.org/10.1007/978-3-319-66706-5_4

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *ICFP*. ACM, 418–430. https://doi.org/10.1145/2034773.2034828

Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

Arthur Charguéraud, Jean-Christophe Filliâtre, François Pottier, and Mário Pereira. 2017. VOCAL – A Verified OCaml Library. In *ML Family Workshop*.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*. ACM, 18–37. https://doi.org/10.1145/2815400.2815402

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245. https://doi.org/10.1145/1993498.1993526

Satyaki Das, David L. Dill, and Seungjoon Park. 1999. Experience with Predicate Abstraction. In *CAV (LNCS, Vol. 1633)*, Nicolas Halbwachs and Doron A. Peled (Eds.). Springer, 160–171. https://doi.org/10.1007/3-540-48683-6_16

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. 1977. Social Processes and Proofs of Theorems and Programs. In *POPL*. ACM, 206–214. https://doi.org/10.1145/512950.512970

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/SP.2019.00005

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE*. ACM, 213–224. https://doi.org/10.1145/302405.302467

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *ICSE*. ACM, 449–458. https://doi.org/10.1145/337180.337240

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (LNCS, Vol. 7792)*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME (LNCS, Vol. 2021)*. Springer, 500–517. https://doi.org/10.1007/3-540-45251-6_29

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to make ad hoc proof automation less ad hoc. In *ICFP*. ACM, 163–175. https://doi.org/10.1145/2034773.2034798

Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. *Reproduction Artefact for Article "Mostly Automated Proof Repair for Verified Libraries"*. https://doi.org/10.5281/zenodo.7703886

Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (LNCS, Vol. 1254)*. Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, 1–17. https://doi.org/10.1145/2815400.2815428

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

The Iris Project. 2022. Iris: a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant. https://iris-project.org/ Online; last accessed 6 November 2022.

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 (LNCS, Vol. 6617)*. Springer, 41–55.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic. In *PLDI*. ACM, 788–801. https://doi.org/10.1145/3314221.3314634

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *4th Conference on Computability in Europe (CiE) (LNCS, Vol. 5028)*. Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring Invariants in Separation Logic for Imperative List-Processing Programs. *The third workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)* 1, 1, 5–7.

Daniel Matichuk. 2012. Automatic Function Annotations for Hoare Logic. In *Proceedings Seventh Conference on Systems Software Verification (SSV) (EPTCS, Vol. 102)*. 46–56. https://doi.org/10.4204/EPTCS.102.6

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI*. ACM, 809–824. https://doi.org/10.1145/3519939.3523432

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. 261–274. https://doi.org/10.1145/1706299.1706331

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 105–114. https://doi.org/10.1109/SCAM.2014.19

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (LNCS, Vol. 8413)*. Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. 2018. A fully verified container library. *Formal Aspects Comput.* 30, 5 (2018), 495–523. https://doi.org/10.1007/s00165-017-0435-1

Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Xin Chen. 2013. Loop invariant synthesis in a combined abstract domain. *J. Symb. Comput.* 50 (2013), 386–408. https://doi.org/10.1016/j.jsc.2012.08.007

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *ESOP (LNCS, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington, USA.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019a. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *PLDI*. ACM, 112–127. https://doi.org/10.1145/3453483.3454033

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *CPP*. ACM, 115–129. https://doi.org/10.1145/3167094

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019b. Ornaments for Proof Reuse in Coq. In *ITP (LIPIcs, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:19. https://doi.org/10.4230/LIPIcs.ITP.2019.26

Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs.* Ph. D. Dissertation. University of California, San Diego, USA.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *NeurIPS*. 7762–7773.

Amin Timany and Bart Jacobs. 2015. First Steps Towards Cumulative Inductive Types in CIC. In *ICTAC (LNCS)*. Springer, 608–617. https://doi.org/10.1007/978-3-319-25150-9_36

Karin Wibergh. 2019. *Automatic refactoring for Agda.* Master's thesis. Chalmers University of Technology and University of Gothenburg.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. ACM, 154–165. https://doi.org/10.1145/2854065.2854081