Fixing Code That Explodes Under Symbolic Evaluation

Sorawee Porncharoenwase, James Bornholt, and Emina Torlak

University of Washington, Seattle WA 98195, USA
{sorawee,bornholt,emina}@cs.washington.edu

Abstract. Effective symbolic evaluation is key to building scalable verification and synthesis tools based on SMT solving. These tools use symbolic evaluators to reduce the semantics of all paths through a finite program to logical constraints, discharged with an SMT solver. Using an evaluator effectively requires tool developers to be able to identify and repair performance bottlenecks in code under all-path evaluation, a difficult task, even for experts. This paper presents a new method for repairing such bottlenecks automatically. The key idea is to formulate the *symbolic performance repair* problem as combinatorial search through a space of semantics-preserving transformations, or *repairs*, to find an equivalent program with minimal cost under symbolic evaluation. The key to realizing this idea is (1) defining a small set of generic repairs that can be combined to fix common bottlenecks, and (2) searching for combinations of these repairs to find good solutions quickly and best ones eventually. Our technique, SymFix, contributes repairs based on deforestation and symbolic reflection, and an efficient algorithm that uses symbolic profiling to guide the search for fixes. To evaluate SymFix, we implement it for the Rosette solver-aided language and symbolic evaluator. Applying SymFix to 18 published verification and synthesis tools built in Rosette, we find that it automatically improves the performance of 12 tools by a factor of 1.1×–91.7×, and 4 of these fixes match or outperform expert-written repairs. SymFix also finds 5 fixes that were missed by experts.

Keywords: symbolic evaluation · performance optimization

1 Introduction

Tools based on SMT solving have automated vital programming tasks in many domains, from verifying safety-critical properties of medical software [33] to synthesizing fast computational kernels for cryptographic applications [35]. These tools employ *symbolic evaluation* [26, 4] to reduce the semantics of all paths through a loop-free (i.e., *finite*) program to logical constraints. The resulting constraints are then used to express queries about program behavior as logical satisfiability queries, discharged with an SMT solver. Since the solvability of such queries hinges on the compactness and simplicity of the underlying constraints, effective symbolic evaluation is key to building effective solver-aided tools.

Building a tool used to require crafting a custom symbolic evaluator, a difficult task that can take years of expert work. Today, this burden is much lower thanks to reusable symbolic evaluators provided by *solver-aided host languages* [45, 47] and frameworks [9, 39]. To build a tool, developers simply write an interpreter for the tool's source language in the solver-aided host language. When this interpreter executes a source program, the host's symbolic evaluator reduces both the interpreter and the program to constraints. The interpreter can control its symbolic evaluation, and thus the encoding, through constructs [44, 38] exposed by the host language and through the structure of its implementation [7]. By exploiting these control mechanisms, developers can create, in weeks, state-of-the-art tools [29] that outperform a custom symbolic execution engine [30, 41].

But if an interpreter performs poorly on a host symbolic evaluator, finding and fixing the bottleneck can be daunting. Recent work on *symbolic profiling* [7] explains why: classic performance engineering techniques assume a single path of execution, and the all-path execution model of symbolic evaluation violates this assumption. As a result, standard profiling tools (e.g., time-based) fail to identify the code that needs to be optimized, and standard optimizations (e.g., breaking early out of a loop) can make performance asymptotically worse under symbolic evaluation. Symbolic profiling addresses the first problem, providing a new performance model for symbolic evaluation and an automatic technique for identifying performance bottlenecks in solver-aided code. The second problem, however, remains open, with developers relying on experience and ad-hoc experimentation to optimize their code.

To address this problem, we present a new method for automatic repair of common performance bottlenecks in solver-aided code. The key idea is to formulate the *symbolic performance repair* problem as combinatorial search in a space of semantics-preserving transformations, or *repairs*. Our technique, SymFix, takes as input a solver-aided program and a workload, and it searches the repair space for a semantically equivalent program that minimizes the cost of symbolic evaluation [7] on the input workload. The choice of repairs and the search strategy are critical to the usefulness and completeness of SymFix. This paper contributes a small set of generic repairs that combine to fix common bottlenecks, and an effective algorithm for combining repairs into (optimal) fixes.

What makes a generic repair useful for code under symbolic evaluation? Intuitively, a repair is useful if its application reduces the cost of symbolic evaluation for a large class of programs. This cost depends on the program's control structure and the evaluator's strategy for splitting and merging states [7, 27, 46]. So useful repairs change the program's control structure or evaluation strategy.

Based on this insight, we develop a set of three repairs that employ *deforestation* [48] to simplify program structure and *symbolic reflection* [46] to simplify the evaluation strategy. Deforestation is a classic optimization for functional programming languages that eliminates intermediate lists. Under concrete evaluation, deforestation improves performance by a constant factor. Under symbolic evaluation, however, it can improve performance asymptotically when the intermediate lists are symbolic. We use deforestation based on build/foldr fusion [22]

as one of our repairs. We also develop two repairs for host languages that support symbolic reflection—a set of language constructs that a program can use to inspect its symbolic state and control its symbolic evaluation (e.g., by forcing a split on a merged state). These two repairs work by creating more opportunities for concrete evaluation. As such, they can both improve performance asymptotically and, in some cases, fix divergence due to loss of concreteness.

The search space defined by our repairs is finite for every program, so it supports complete and optimal search. But it is also intractably large for real programs. We therefore formulate SymFix as an anytime algorithm, equipped with a pruning mechanism that exploits *precedence* of repairs and a prioritization heuristic that exploits symbolic profiling information. The pruning mechanism is inspired by partial order reduction [15]: if two repairs can always be reordered so that one is applied before the other without changing the result, SymFix explores only one of the orders. The prioritization heuristic uses ranking information computed by symbolic profiling to decide what parts of the program to repair first. In particular, symbolic profiling takes as input a program and a workload, and ranks the locations in the program from most to least likely bottlenecks. SymFix uses this ranking to quickly drive the search toward most promising solutions.

We implement SymFix for Rosette [43, 46, 45], a solver-aided host language that extends Racket [37, 18] with support for symbolic evaluation, reflection, and profiling. To evaluate SymFix's effectiveness, we apply it to 15 solver-aided tools [2, 5, 6, 8, 10, 12, 14, 25, 36, 33, 46, 50, 51] studied in the paper on symbolic profiling [7], as well as 3 more recent tools [29, 31, 35]. SymFix improves the performance of 12 tools by a factor of 1.1×–91.7×, and 4 of these fixes match or outperform those written by experts. SymFix also finds 5 fixes that were missed by experts. We further show that the improvements made by SymFix generalize to unseen workloads, and that its search strategy is essential for finding useful fixes.

In summary, this paper makes the following contributions:

1. A formulation of the symbolic performance repair problem as combinatorial search in a space of semantics-preserving transformations, or repairs.
2. SymFix, a new technique for solving this problem. SymFix contributes a set of repairs based on deforestation and symbolic reflection, and an effective anytime algorithm for combining these repairs into useful fixes.
3. An implementation of SymFix for the Rosette solver-aided language [43, 46].
4. An evaluation of SymFix's effectiveness on 18 published solver-aided tools built in Rosette, showing that it can find repairs that outperform expert fixes and that generalize to unseen workloads.

The rest of the paper illustrates symbolic performance repair on a small example (§2); formulates the problem of repairing performance bottlenecks in solver-aided code (§3); presents the SymFix algorithm, repairs, and implementation for Rosette (§4); shows the effectiveness of SymFix at repairing bottlenecks in real solver-aided tools hosted by Rosette (§5); discusses related work (§6); and concludes with a summary of key points (§7).

2 Overview

This section illustrates symbolic performance repair on a small solver-aided program (Figure 1). The program is adapted from Serval [29], a framework for verifying systems code at the instruction level. Serval is built in Rosette [43], and it supports creating scalable automated verifiers by writing interpreters. Serval's authors show how to profile this program with a symbolic profiler, and manually fix the bottleneck using a custom construct implemented as a Rosette macro. We first revisit this analysis to highlight the challenges of repairing bottlenecks in solver-aided code, and then show how SymFix repairs the problem automatically and generically, using a repair based on symbolic reflection [46].

```
1  ; cpu state: program counter and registers
2  (struct cpu (pc regs) #:mutable)
3
4  ; interpret a program from a cpu state
5  (define (interpret c program)        ; A
6    (define i (fetch c program))       ; B
7    (match i
8      [(list opcode rd rs imm)
9       (execute c opcode rd rs imm)
10      (when (not (equal? opcode 'ret))
11        (interpret c program))]))
12
13 ; fetch an instruction at the current pc
14 (define (fetch c program)
15   (define pc (cpu-pc c))
16   (vector-ref program pc))           ; C
17
18 ; read register rs
19 (define (cpu-reg c rs)
20   (vector-ref (cpu-regs c) rs))
21
22 ; write value v to register rd
23 (define (set-cpu-reg! c rd v)
24   (vector-set! (cpu-regs c) rd v))
25
26 ; execute instruction (opcode rd rs imm)
27 (define (execute c opcode rd rs imm)
28   (define pc (cpu-pc c))
29   (case opcode
30     [(ret)
31      (set-cpu-pc! c 0)]              ; D
32     [(bnez)
33      (if (! (= (cpu-reg c rs) 0))
34          (set-cpu-pc! c imm)         ; E
35          (set-cpu-pc! c (+ 1 pc)))] ; F
36     [(sgtz)
37      (set-cpu-pc! c (+ 1 pc))
38      (if (> (cpu-reg c rs) 0)
39          (set-cpu-reg! c rd 1)       ; G
40          (set-cpu-reg! c rd 0))]     ; H
41     [(sltz)
42      (set-cpu-pc! c (+ 1 pc))
43      (if (< (cpu-reg c rs) 0)
44          (set-cpu-reg! c rd 1)       ; I
45          (set-cpu-reg! c rd 0))]     ; J
46     [(li)
47      (set-cpu-pc! c (+ 1 pc))        ; K
48      (set-cpu-reg! c rd imm)]))      ; L
49
50 (define sgnt #( ; sign in ToyRISC
51  (sltz 1 0 #f)  ; 0. r1 = r0<0 ? 1 : 0
52  (bnez #f 1 4)  ; 1. branch to 4 if r1!=0
53  (sgtz 0 0 #f)  ; 2. r0 = r0>0 ? 1 : 0
54  (ret #f #f #f) ; 3. return
55  (li 0 #f -1)   ; 4. r0 = -1
56  (ret #f #f #f) ; 5. return
57  ))
58
59 (define-symbolic X Y integer?)
60 (define c (cpu 0 (vector X Y)))
61 (interpret c sgnt)
62 (verify
63   (assert (= (cpu-reg c 0) (sgn X))))
```

Fig. 1: A ToyRISC interpreter and program in Rosette, adapted from Serval [29].

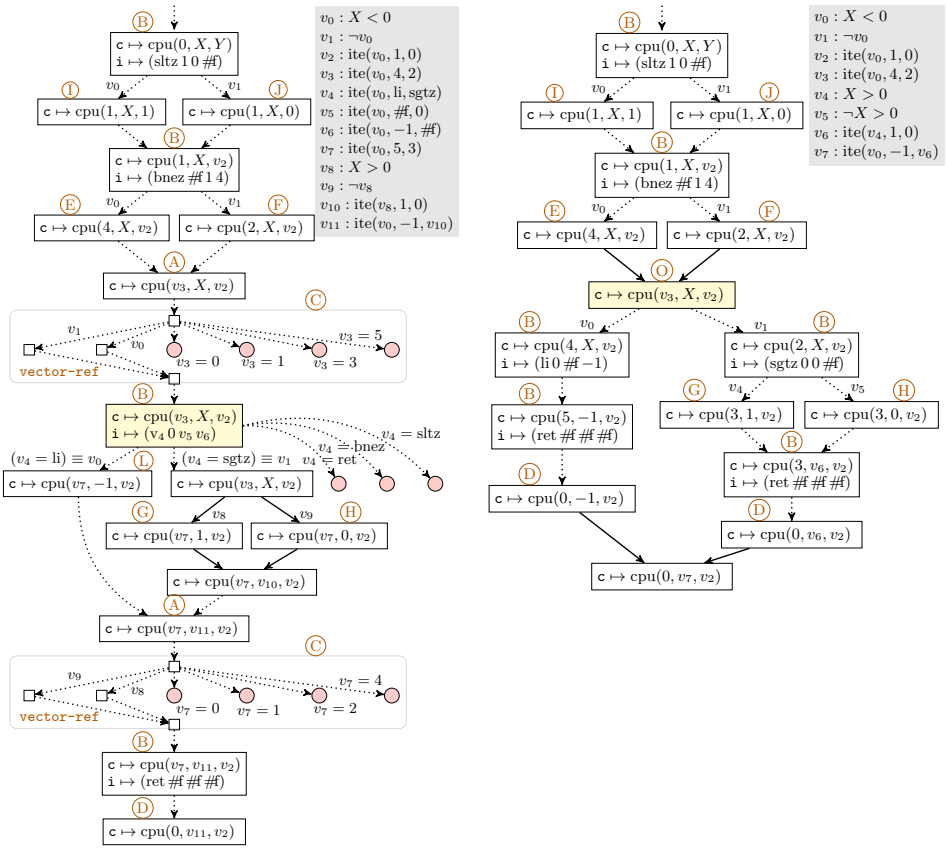
Solver-aided programming. Figure 1 shows a small program [29] written in Rosette, a solver-aided host language that extends Racket [37] with support for symbolic evaluation. Rosette programs behave like Racket programs when executed on concrete values. But Rosette also *lifts* programs, via symbolic evaluation, to operate on *symbolic values*. These values are used to formulate *solver-aided queries*, such as verifying that a program satisfies its specification, expressed as assertions, on all inputs. The example verifies a program in ToyRISC, a small subset of RISC-V [1], by lifting its interpreter to work on symbolic values.

The ToyRISC interpreter (lines 1–48) implements a simple recursive procedure for executing a ToyRISC program from a given CPU state. The state consists of a program counter and vector of two registers, r_0 and r_1, both holding integers. A program is a sequence of instructions that manipulate the state. An instruction is a list of four values, `(opcode rd rs imm)`, specifying the instruction's opcode, destination and source registers, and the immediate constant. Unused arguments are denoted by `#f`; for example, the return instruction takes no arguments, denoted by `(ret #f #f #f)`. In addition to the return instruction, which halts the execution (line 10), the language also includes instructions for conditional branching (`bnez`), loading values into registers (`li`), and comparing register values to zero (`sgtz` and `sltz`). The example ToyRISC program, `sgnt`, uses these instructions to compute the sign of the value in register r_0, storing the result back into r_0 and using r_1 as a scratch register.

The `sgnt` program is correct if it produces the same result as the host sign procedure, `sgn`, for all valid CPU states. To verify `sgnt`, we first use Rosette's `define-symbolic` form to create two fresh symbolic integers, X and Y, and bind them to the variables `X` and `Y` (line 59). Next, we use these variables to create a CPU state `c` with the program counter set to 0 and registers set to X and Y (line 60). The symbolic state `c` represents all valid concrete CPU states. Finally, we interpret `sgnt` on `c` and use Rosette's `verify` query to search for a counterexample to the assertion that register r_0 holds the sign of X. A counterexample to this query would bind the symbolic values X and Y to concrete integers that trigger the assertion failure. But since `sgnt` is correct, the query returns `(unsat)` to indicate the absence of counterexamples.

Symbolic evaluation and profiling. When interpreting `sgnt` on the symbolic state `c`, Rosette evaluates all paths through the interpreter code and reduces their meaning to symbolic expressions over X and Y. For example, after the call to the interpreter (line 61), register r_0 of `c` holds the symbolic value $\text{ite}(X < 0, -1, \text{ite}(0 < X, 1, 0))$, which encodes the meaning of `sgnt`. This value is part of the *symbolic heap* that Rosette generates while exploring the interpreter's *symbolic evaluation graph* [7] (Figure 2a). The heap consists of all symbolic values created during evaluation. The graph is a DAG over program states and guarded transitions between states, and its shape reflects the evaluator's strategy for path splitting and state merging. The symbolic heap and evaluation graph characterize the behavior of solver-aided code under every (forward) symbolic evaluation strategy, and controlling their complexity is key to good performance [7].

To help with this task, Rosette provides a *symbolic profiler*, SymPro, that monitors the heap and the graph to identify performance bottlenecks. SymPro computes summary statistics about the effect of each procedure call on these structures, such as the number of symbolic values added to the heap, and the number of path splits and state merges added to the graph. It then uses these statistics to rank the calls to suggest likely bottlenecks. When applied to ToyRISC, SymPro identifies the calls to `execute` at line 9 and `vector-ref` at line 16 as the likely bottlenecks. But how does one diagnose and fix these bottlenecks?



(a) Original (Figure 1)

(b) Repaired (Figure 3)

Fig. 2: Simplified symbolic evaluation heap and graph for the original (a) and repaired (b) ToyRISC code. Heaps are shown in gray boxes. Nodes in a symbolic evaluation graph are program states, and edges are guarded transitions between states, labeled with the condition that guards the transition. Edges ending at pink circles denote infeasible transitions. Dotted edges indicate elided parts of the graph. Circled letters are program locations, included for readability.

Manually repairing bottlenecks. The authors of ToyRISC reasoned [29] that the first location returned by SymPro "is not surprising since `execute` implements the core functionality, but `vector-ref` is a red flag." Examining the merging statistics for `vector-ref`, they concluded that `vector-ref` is being invoked with a symbolic program counter to produce a "merged symbolic instruction" (highlighted in Figure 2a), which represents a set of concrete instructions, only some of which are feasible. Since `execute` consumes this symbolic instruction (line 9), its evaluation involves exploring infeasible paths, leading to degraded performance on our example and non-termination on more complex ToyRISC programs.

```
(define (interpret c program)       ; A
  (serval:split-pc [cpu pc] c       ; O
    (define i (fetch c program))    ; B
    (match i
      [(list opcode rd rs imm)
      (execute c opcode rd rs imm)
      (when (not (equal? opcode 'ret))
        (interpret c program))])))
```

(a) Manual repair [29]

```
(define (interpret c program)       ; A
  (split-all (c)                    ; O
    (define i (fetch c program))    ; B
    (match i
      [(list opcode rd rs imm)
      (execute c opcode rd rs imm)
      (when (not (equal? opcode 'ret))
        (interpret c program))])))
```

(b) Generated repair

Fig. 3: Manual and SymFix repair for ToyRISC code.

Having diagnosed the problem, the authors of ToyRISC then reasoned that the fix should force Rosette to split the evaluation into separate paths that keep the program counter concrete. Such a fix can be implemented through *symbolic reflection* [46], a set of constructs that allow programmers to control Rosette's splitting and merging behavior. In this case, ToyRISC authors used symbolic reflection and metaprogramming with macros (which Rosette inherits from Racket) to create a custom construct, `split-pc`, that forces a path split on CPU states with symbolic program counters. Applying `split-pc` to the body of `interpret` (Figure 3a) fixes this bottleneck (Figure 2b)—and ensures that symbolic evaluation terminates on all ToyRISC programs. But while simple to implement, this fix is hard won, requiring manual analysis of symbolic profiles, diagnosis of the bottleneck, and, finally, repair with a custom construct based on symbolic reflection.

Repairing bottlenecks with SymFix. SymFix lowers this burden by automatically repairing common performance bottlenecks in solver-aided code. The core idea is to view the repair problem (§3) as search for a sequence of semantics-preserving repairs that transform an input program into an equivalent program with minimal symbolic cost—a value computed from the program's symbolic profiling metrics. To realize this approach, SymFix solves two core technical challenges (§4): (1) developing a small set of generic repairs that can be combined into useful and general repair sequences for common bottlenecks, and (2) developing a search strategy that discovers good fixes quickly and best fixes eventually.

SymFix can repair complex bottlenecks in real code as well as or better than experts (§5). It can also repair ToyRISC, finding the fix in Figure 3b. This fix has the same effect as the expert `split-pc` fix but uses a generic `split-all` construct. The construct forces a split on the value stored in a variable depending on its type: if the value is a `struct`, the split is performed on all of its fields that hold symbolic values. The `split-all` construct can be soundly applied to any bound occurrence of a local variable in a program, leading to intractable search spaces even for small programs. For example, there are 55 bound local variables in ToyRISC, so the `split-all` repair alone can be used to transform ToyRISC into 2^{55} syntactically distinct programs. SymFix is able to navigate this large search space effectively, matching the expert fix in a few seconds.

3 Symbolic Performance Repair

As §2 illustrates, performance bottlenecks in solver-aided code are difficult to repair manually. This section presents a new formulation of this problem that enables automatic solving. Our formulation is based on two core concepts: repairs and fixes. A *repair* is a semantics-preserving transformation on programs. A *fix* combines a sequence of repair steps, with the goal of reducing the *cost* of a program under symbolic evaluation. The *symbolic performance repair problem* is to find a fix, drawn from a finite set of repairs, that minimizes this cost. We describe repairs and fixes first, present the symbolic performance repair problem next, and end with a discussion of key properties of repairs that are sufficient for solving the repair problem in principle and helpful for solving it in practice.

3.1 Repairs, fixes, and symbolic cost

Repairs. A *repair* transforms a program to a set of programs that are syntactically different but semantically equivalent (Definition 1 and 2). A repair operates on programs represented as abstract syntax trees (ASTs). It takes as input an AST and a node in this AST, and produces an ordered set of ASTs that transform the input program at the given node or one of its ancestors. This interface generalizes classic program transformations by allowing repairs to produce multiple ASTs. The classic interface is often implemented by heuristically choosing one of many possible outputs that an underlying rewrite rule can generate. Our interface externalizes this choice, while still letting repairs provide heuristic knowledge in the order of the generated ASTs, as illustrated in Example 1. This enables an external algorithm to drive the search for fixes, with advice from the repairs.

Definition 1 (Program). *A* program *is an abstract syntax tree (AST) in a language $\mathcal{P}$, consisting of labeled nodes and edges. A program $P \in \mathcal{P}$ denotes a function $[\![P]\!] : \Sigma \to \Sigma$ on* program states*, which map program variables to values. Programs P and P' are syntactically equivalent if their trees consist of identically labeled nodes, connected by identically labeled edges. They are semantically equivalent iff $[\![P]\!]^{\mathcal{P}}\sigma \equiv [\![P']\!]^{\mathcal{P}}\sigma$ for all program states $\sigma \in \Sigma$, where $[\![\cdot]\!]^{\mathcal{P}} : \mathcal{P} \to \Sigma \to \Sigma$ denotes the concrete semantics of $\mathcal{P}$.*

Definition 2 (Repair). *A* repair *$R : \mathcal{P} \to \mathcal{L} \to 2^{\mathcal{P}}$ is a function that maps a program and a location to an ordered finite set of programs. A location $l \in \mathcal{L}$ identifies a node in an AST. The set $R(P, l)$ is empty if the repair R is not applicable to P at the location l. Otherwise, each program $P_i \in R(P, l)$ satisfies two properties. First, P and P_i differ in a single subtree rooted at l or an ancestor of l in P. Second, P and P_i are semantically equivalent.*

Example 1. Consider a repair R_1 that performs the rewrite $e * 2 \to e \ll 1$ on integer expressions. There are three ways to apply this rewrite to the program $P =$ `1 + (a * 2) * 2` at the node `a` or its ancestors, and R_1 orders them as follows:

```
1 + (a << 1) << 1 ; 0: Apply the rewrite exhaustively.
1 + (a << 1) * 2  ; 1: Apply the rewrite just to a's parent.
1 + (a * 2) << 1  ; 2: Apply the rewrite just to a's grandparent.
```

The order of the generated ASTs suggests that applying the rewrite exhaustively is most useful, followed by applying it from the inside out.

Fixes. A *fix* composes a sequence of *repair steps* into a function from programs to programs (Definition 3 and 4). A repair step $\langle R, l, i\rangle$ specifies the repair R to apply to a program, the location l at which to apply it, and the index i of the program to select from the resulting ordered set of programs. In essence, a repair step turns a repair into a classic program transformation by choosing one of the repair's outputs, and fixes can compose these steps to create new transformations, as illustrated in Example 2.

Definition 3 (Repair step). *A* repair step $\langle R, l, i\rangle$ *consists of a repair R, program location l, and non-negative integer i. A step denotes the function* $[\![\langle R, l, i\rangle]\!] : \mathcal{P} \cup \{\bot\} \to \mathcal{P} \cup \{\bot\}$ *as follows:* $[\![\langle R, l, i\rangle]\!]P = R(P, l)[i]$ *if* $P \neq \bot$ *and* $|R(P, l)| > i$*; otherwise the result is* $\bot$*. We write* $R(P, l)[i]$ *to mean the* i^{th} *program in the ordered set* $R(P, l)$*.*

Definition 4 (Fix). *A* fix $F = [\langle R_1, l_1, i_1\rangle, \ldots, \langle R_n, l_n, i_n\rangle]$ *is a finite sequence of one or more repair steps. A fix F denotes the function that composes the repair steps of F, i.e.,* $[\![F]\!] = [\![\langle R_n, l_n, i_n\rangle]\!] \circ \ldots \circ [\![\langle R_1, l_1, i_1\rangle]\!]$*. We say that fix is* successful *for a program P if* $[\![F]\!]P \neq \bot$*.*

Example 2. Consider the fix $F = [\langle R_1, \mathtt{a}, 0\rangle, \langle R_2, \mathtt{a}, 0\rangle]$, where R_1 is the repair from Example 1 and R_2 performs the rewrite $(e \mathtt{<<} 1) \mathtt{<<} 1 \to e \mathtt{<<} 2$. Applying F to the program P from Example 1 produces the program `1 + (a << 2)`: $[\![\langle R_2, \mathtt{a}, 0\rangle]\!]([\![\langle R_1, \mathtt{a}, 0\rangle]\!]P) = [\![\langle R_2, \mathtt{a}, 0\rangle]\!]$`(1 + (a << 1) << 1)` = `1 + (a << 2)`. In other words, the fix F composes its repair steps to rewrite the second subexpression of P using the rule $(e * 2) * 2 \to e \mathtt{<<} 2$.

Cost. There are many ways to combine repairs into fixes for a given program, even when the program is small and repairs are few (Example 3). To choose a fix that is *useful* for improving the performance of a program under *symbolic evaluation*, we need a way to measure the *cost* of symbolic evaluation (Definition 5). We address this challenge by building on the observation that the behavior of symbolic evaluators is characterized by two structures: the symbolic heap and the symbolic evaluation graph. Our framework defines symbolic evaluation as a function from programs and program states to these structures (Definition 6), and the cost of symbolic evaluation as a function from these structures to (natural) numbers (Definition 7). The details of the cost function are not important for the framework, although they are important in practice: the symbolic cost should correlate with concrete metrics that are meaningful to developers (e.g., end-to-end running time), and SymFix uses a cost function (§4) that is simple but effective (§5). What matters, however, is that the symbolic evaluator is a total function, which means that we consider only finite computations. In particular, we make the standard assumption that programs $P \in \mathcal{P}$ are free of input-dependent loops, and are therefore guaranteed to terminate under symbolic evaluation, ensuring that we can compute the cost for every fix.

Definition 5 (Useful fix). *A fix F is* useful *for a program $P \in \mathcal{P}$, program state $\sigma \in \Sigma$, symbolic evaluator $S : \mathcal{P} \to \Sigma \to \mathcal{G}$, and cost $c : \mathcal{G} \times \mathcal{H} \to \mathbb{N}$, if $[\![F]\!]P \neq \bot$ and $c(S([\![F]\!]P, \sigma)) < c(S(P, \sigma))$.*

Definition 6 (Symbolic evaluator). *A* symbolic evaluator $S : \mathcal{P} \to \Sigma \to \mathcal{G} \times \mathcal{H}$ *is a function that takes as input a program $P \in \mathcal{P}$ and program state $\sigma \in \Sigma$, and outputs a pair $\langle G, H \rangle$, where $G \in \mathcal{G}$ is a symbolic evaluation graph and $H \in \mathcal{H}$ is a symbolic heap [7]. A* symbolic heap $H = (V_H, E_H)$ *is a directed acyclic graph (DAG) with labeled nodes and edges. Heap nodes are* symbolic values*, and heap edges connect compound symbolic values to the symbolic or concrete values from which they are built. A* symbolic evaluation graph $G = (V_G, E_G)$ *is a DAG where nodes $V_G \subseteq \Sigma$ are program states and edges are transitions between states, each labeled with a (symbolic or concrete) boolean value that guards the transition and a program location in P that caused the transition. The graph G has $\sigma \in V_G$ as its sole source node. The heap H contains all symbolic values that appear in G as part of a program state or as an edge label. If $H = (\emptyset, \emptyset)$ is empty, then G consists of a single path from σ to $[\![P]\!]^{\mathcal{P}}\sigma$, where all edges are labeled with $\top$.*

Definition 7 (Symbolic cost). *A* symbolic cost *function $c : \mathcal{G} \times \mathcal{H} \to \mathbb{N}$ assigns a cost, expressed as a natural number, to the results of symbolic evaluation.*

Example 3. Consider again the fix F, repairs R_1 and R_2, and program P from Example 1 and 2. In addition to F, there are seven different ways to compose repair steps over R_1 and R_2 into fixes for P; two are equivalent to F and five to the outputs of R_1 on P. Intuitively, F produces the best program for all workloads, and in this case, the intuition is captured by a simple cost function that measures the size of the symbolic heap, i.e., $c(\langle G, H \rangle) = |V_H|$. For example, letting $\sigma = \{\texttt{a} \mapsto A\}$, where A is a symbolic integer, we can compute the cost of P, the output of the fix $[\langle R_1, \texttt{a}, 0 \rangle]$, and the output of the fix F as follows:

$$\begin{aligned}
c(S(\texttt{1 + (a * 2) * 2}, \sigma)) &= |\{v_0 : A * 2, v_1 : v_0 * 2, v_2 : 1 + v_1\}| &= 3 \\
c(S(\texttt{1 + (a << 1) << 1}, \sigma)) &= |\{v_0 : A \texttt{<<} 1, v_1 : v_0 \texttt{<<} 1, v_2 : 1 + v_1\}| &= 3 \\
c(S(\texttt{1 + (a << 2)}, \sigma)) &= |\{v_0 : A \texttt{<<} 2, v_1 : 1 + v_0\}| &= 2
\end{aligned}$$

As expected, the program produced by F has the lowest cost.

3.2 The symbolic performance repair problem

The *symbolic performance repair problem* is to find a fix, drawn from a finite set of repairs, that minimizes the symbolic cost of a program on a given workload (Definition 8). To make this problem solvable in principle, it is sufficient to ensure that the set of repairs is *terminating* [17], preventing the repairs from being indefinitely applicable to any program (Definition 9). To help solve the repair problem in practice, we can use a general property of repairs, *precedence*, to prune fixes during search without missing any programs (Definition 10). A partial order $\preceq_{\mathsf{R}}$ is a precedence relation on a set of repairs R if every successful fix over R can be turned into an equivalent fix by permuting its repair steps to respect $\preceq_{\mathsf{R}}$. To search for a fix over R with $\preceq_{\mathsf{R}}$, it is sufficient to explore successful fixes that order all repair steps according to $\preceq_{\mathsf{R}}$. Example 4 illustrates

these definitions, and we use them in the next section to develop the SymFix algorithm for solving the repair problem.

Definition 8 (Symbolic performance repair). *Let $P \in \mathcal{P}$ be a program, $\sigma \in \Sigma$ a program state, R a finite set of repairs for $\mathcal{P}$, S a symbolic evaluator for $\mathcal{P}$, and c a symbolic cost function for S. The* symbolic performance repair problem *is to find a useful fix F over R that minimizes the cost of evaluating P on σ; i.e., for all useful fixes $F' \neq F$ over R, $c(S([\![F]\!]P, \sigma)) \leq c(S([\![F']\!]P, \sigma))$.*

Definition 9 (Terminating repair set). *Let R be a finite set of repairs for the language $\mathcal{P}$. We say this set is* terminating *if for every program $P \in \mathcal{P}$, there is an upper bound on the length of every successful fix for P drawn from R.*

Definition 10 (Repair precedence). *Let R be a finite set of repairs and $\preceq_{\mathsf{R}}$ a partial order on R. Let* spine *be a function that projects out the repairs from a fix, i.e.,* $\text{spine}(F) = [R_1, \ldots, R_n]$ *for* $F = [\langle R_1, l_1, i_1\rangle, \ldots, \langle R_n, l_n, i_n\rangle]$. *We say that $\preceq_{\mathsf{R}}$ is a* precedence *on R if for every program P and every successful fix F for P drawn from R, there is a fix F' such that $[\![F]\!]P = [\![F']\!]P$ and* $\text{spine}(F')$ *permutes* $\text{spine}(F)$ *to respect $\preceq_{\mathsf{R}}$, i.e.,* $\forall i, j.\, \text{spine}(F')[i] \preceq_{\mathsf{R}} \text{spine}(F')[j] \implies i \leq j$.

Example 4. Recall the program P, repairs R_1 and R_2, fix F, and cost c from Examples 1–3. The repair set $\mathsf{R} = \{R_1, R_2\}$ is terminating; $R_1 \preceq_{\mathsf{R}} R_2$; and F is a solution to the symbolic performance repair problem for P, R, and c.

4 The SymFix Algorithm and Repairs

This section presents the SymFix system for solving the symbolic performance repair problem. SymFix consists of two components: an anytime algorithm for searching the space of fixes drawn from a terminating set of repairs, and a set of three generic repairs for functional solver-aided languages with symbolic reflection. We present the algorithm first and prove its correctness and optimality (§4.1). We then describe the repairs and a total precedence relation on them, and argue that they form a terminating set (§4.2). We end by highlighting the key details of our implementation of SymFix for the Rosette language (§4.3).

4.1 Profile-guided search for fixes

The SymFix algorithm (Figure 4) solves the symbolic performance repair problem for a cost function based on symbolic profiling. As shown in prior work [7], the metrics computed by a symbolic profiler closely reflect the overall running time of solver-aided code (i.e., symbolic evaluation together with solving time), and reducing these metrics is key to improving performance. In addition to computing these metrics, which measure the size and shape of the symbolic heap and evaluation graph, a symbolic profiler M also ranks all locations in a program from most to least expensive to evaluate. The SymFix algorithm uses both of these outputs: it searches for a fix that minimizes the sum of the profiling metrics for a given program and workload, and the search is guided by the profiling ranks.

```
1  function SymFix(P_in, σ, S, M, R, ⪯_R)
2      function Info(P, F)                          ▷ Symbolic profile sorts P's locations from most to least
3          ⟨L_P, m_P⟩ ← M(S(P, σ))                  ▷ likely bottlencks & collects k profiling metrics.
4          c_P ← Σ_{0≤i<k} m_i                      ▷ P's cost is the sum of its profiling metrics.
5          return {P ↦ {cost ↦ c_P, locs ↦ L_P, fix ↦ F}}
6      function Next(P, info)                       ▷ Picks a successor of P, if any, with an extra repair.
7          F ← info[P][fix]                         ▷ Get the fix that generated P.
8          for R in R do                            ▷ Iterate over the repairs in R that do not precede
9              if ⋀_{R_i ∈ spine(F)} R_i = R ∨ R ⋠_R R_i then   ▷ any repairs in P's fix,
10                 for l in info[P][locs] do        ▷ then over the ranked locations in P,
11                     for P_j ∈ R(P, l) do         ▷ and then over the ordered results
12                         if info[P_j] = ⊥ then    ▷ to find a new program P_j.
13                             return ⟨P_j, append(F, ⟨R, l, j⟩)⟩   ▷ Return P_j and its fix.
14         return ⟨⊥, ⊥⟩                            ▷ No new programs can be obtained from P.
15     function Search(P_in)
16         W, info ← {P_in}, Info(P_in, [])         ▷ Initialize the work set and info map.
17         minCost ← info[P_in][cost]               ▷ Set P's cost as current best cost.
18         while W ≠ ∅ do
19             P ← min(W, λP.info[P][cost])         ▷ Choose the cheapest P ∈ W to work on.
20             ⟨P′, F′⟩ ← Next(P, info)             ▷ Get a successor P′ of P and its fix.
21             if ⟨P′, F′⟩ ≠ ⟨⊥, ⊥⟩ then            ▷ If P′ exists,
22                 W, info ← W ∪ {P′}, info ∪ Info(P′, F′)   ▷ add P′ to W and info;
23                 if info[P′][cost] < minCost then          ▷ and if P′ is best so far,
24                     minCost ← info[P′][cost]              ▷ update minCost and
25                     print P′                              ▷ output P′.
26             else
27                 W ← W \ {P}                      ▷ No new programs can be obtained from P.
28     Search(P_in)
```

Fig. 4: The SymFix search algorithm takes as input a program P_{in} in a language $\mathcal{P}$, a workload σ, a symbolic evaluator S for $\mathcal{P}$, a symbolic profiler M for S, a terminating set of repairs R for $\mathcal{P}$, and a precedence relation $\preceq_{\mathsf{R}}$ on R. It searches the space of fixes drawn from R to find a program that is equivalent to P_{in} and minimizes the cost of symbolic evaluation on σ according to the profiler M.

The algorithm relies on the SEARCH procedure to explore the space of fixes for a program P_{in} and a terminating set of repairs R. SEARCH performs exhaustive (rather than greedy) best-first search over this space. It starts by initializing the work set W with the input program P_{in}; the *info* map with a binding from P_{in} to its profiling metrics, cost, and the empty fix; and the minimum cost *minCost* with the cost of P_{in}. The main search loop then picks a program P from the work set, applies one repair step to P to get a new program P' (corresponding to a fix F' that extends P's fix by one step), and adds P' to both W and *info*. If the new program P' has lower cost than *minCost*, SEARCH prints it and updates *minCost* accordingly. But if no new programs can be obtained from P by applying a repair from R, then P has no more children in the underlying search graph, and SEARCH removes it from the work set W. The search continues as long as there are programs in W, so the entire search graph is eventually explored.

To make the algorithm practically useful, SEARCH employs the procedure NEXT to explore the most promising fixes first and to prune the search space without losing completeness. SEARCH selects the cheapest fix F to extend (line 19), and NEXT constructs the repair step $\langle R, l, j\rangle$ to add to F. To construct

$\langle R, l, j\rangle$, NEXT first choses a repair R that does not strictly precede any of the repairs in F, according to the precedence relation $\preceq_{\mathsf{R}}$. Then, it uses profiling rankings and the repair's ordering heuristics to select the location l and the result index j. This ensures that SEARCH explores only fixes that respect $\preceq_{\mathsf{R}}$, and that it tries to repair most likely bottlenecks first.

The SymFix algorithm is sound, complete, and optimal (Theorem 1). It produces correct fixes that are semantically equivalent to the input program (soundness). It always finds a useful fix if one exists in the space defined by the given set of repairs (completeness). And it eventually finds the best such fix that minimizes the symbolic profiling cost on the given workload (optimality).

Theorem 1. *Let P_{in} be a program, σ a workload, R a terminating set of repairs, and $\preceq_{\mathsf{R}}$ a precedence relation on R. Then* SYMFIX$(P_{in}, \sigma, S, M, \mathsf{R}, \preceq_{\mathsf{R}})$ *terminates and satisfies the following conditions. (1) If* SEARCH *produces a program at line 25, then every such program P' is semantically equivalent to P_{in} (soundness). (2) For every cost $C < info[P_{in}][cost]$, if there is a fix over R with cost C, then line 25 will produce a program P' with $info[P'][cost] \leq C$ (completeness and optimality).*

Proof (sketch). First, note that SymFix explores a search graph where nodes are programs; two nodes are related by a repair step drawn from R; and a path in the graph corresponds to a fix over R that respects $\preceq_{\mathsf{R}}$. All paths through this graph are finite because R is terminating (Definition 9). There are also finitely many such paths because each node has finitely many outgoing edges (repair steps), which follows from the finite number of repairs, locations in a program, and repair outputs. So, (1) the underlying search graph is finite, and (2) by definition of $\preceq_{\mathsf{R}}$ (Definition 10), it contains the same programs (nodes) as the search graph that includes all fixes (paths) over R. Next, note that (3) SymFix adds each program in this graph to the work set W exactly once, and (4) each added program is removed after all of its children have been visited, i.e., added to the *info* map. These facts (1–4) imply that the algorithm terminates after visiting each program in the space defined by R. Completeness and optimality then follow from lines 17, 23–25, and soundness follows from the definition of repairs (Definition 2).

4.2 Effective repairs for functional hosts with symbolic reflection

The effectiveness of SymFix hinges on the choice of the repair set R. An ideal repair set includes a few key repairs that can be combined into useful fixes for most common performance bottlenecks. This section presents three such repairs for solver-aided languages with functional programming primitives and symbolic reflection. We use Rosette to illustrate these repairs, but they are applicable to any solver-aided language or framework with similar features (e.g., [47, 39, 13]).

Deforestation. Higher-order combinators (e.g., `map`, `fold`, and `filter`) are commonly used to operate on lists. Using these combinators generates intermediate lists that are immediately consumed and discarded, slowing down concrete evaluation by a constant factor. Under symbolic evaluation, however, the resulting slow down is asymptotically worse, as the following example demonstrates.

```
(define (sum-slow xs)                ; Sums the positive numbers in xs using an
  (foldr + 0 (filter positive? xs)))  ; intermediate list (the result of filter).

(define (sum-fast xs)                ; Sums the positive numbers in xs without
  (foldr (lambda (e acc)             ; creating any intermediate lists.
           (if (positive? e)
               (+ e acc)
               acc))
         0 xs))

> (define-symbolic xs integer? [100])  ; xs is a list of 100 symbolic integers.

> (time (sum-slow xs))                 ; Adds 520,000 values to the symbolic heap.
cpu time: 5119 real time: 4954 gc time: 2194

> (time (sum-fast xs))                 ; Adds 100 values to the symbolic heap.
cpu time: 3 real time: 3 gc time: 0    ; Times are given in milliseconds.
```

Deforestation [48] is a classic program transformation that eliminates intermediate lists produced by list combinators. As such, it makes a powerful repair for performance bottlenecks in solver-aided code. In the above example, it automatically transforms `sum-slow` into `sum-fast`, avoiding the expensive call to `filter` that creates a symbolic intermediate list when the input `xs` is symbolic. Many variants of deforestation exist for different functional languages; for Rosette, SymFix uses a repair based on build/foldr fusion [22]. This repair applies deforestation exhaustively at a given location and outputs at most one program.

Path splitting. Deforestation changes the behavior of a program under symbolic evaluation by restructuring its implementation. But if the host language supports symbolic reflection [46], we can control the evaluation more directly, by using dedicated constructs to force path splitting [44] (or state merging [38]) at specific program locations. We have seen an example of this in §2, where SymFix used a path splitting construct to fix the ToyRISC interpreter. In Rosette, this construct takes the form `(split-all (`x`) `E`)`, where x is an identifier and E an expression over x. If x is bound to a symbolic value that ranges over a small finite set of concrete values, $\{v_1, \ldots, v_n\}$, then `split-all` splits the evaluation of E into n paths, one for each value that x can take, i.e., $x = v_i \vdash$ `(let ([`x v_i`]) `E`)` for $1 \leq i \leq n$. Otherwise, `split-all` acts as the identity transformation on E. Because path splitting increases the number of paths that are evaluated, it must be applied carefully to avoid path explosion—a task we delegate to automated search.

The SymFix path splitting repair works as follows. Given a program location l in a procedure body P, it outputs all valid ways to insert `(split-all (`x`) `E`)` into P, so that E contains the location l, x is bound in E's context, and there is no other split on x in E or its context. So, nested splits on the same identifier, `(split-all (`x`) (...(split-all (`x`) ...)`, are disallowed. The resulting set of transformed programs is finite but large, and the repair heuristically prefers splits with broadest scope (i.e., where E is the highest ancestor of l in P).

Value splitting. Path splitting allows programs to exert local control on the symbolic evaluation strategy, by concretizing a specific symbolic value at a specific program location. In principle, it is possible to combine many path splitting repairs to implement a global change in the evaluation strategy, such as concretizing every operation on a given user-defined type. In practice, however, this

would require prohibitively long and complex fixes. We therefore develop a global value splitting repair that assumes the host language provides a mechanism for controlling how all values of a given type are merged and split. In Rosette, this is done with a *transparency* annotation, illustrated in the following example.

```
(require rosette/lib/match)

(struct Cell (v) #:transparent)

; Return a new Cell that doubles
; the value v of c.
(define (twice c)
  (match c
    [(Cell v) (Cell (+ v v))]))

; Create a symbolic Cell.
(define-symbolic b boolean?)
(define c (if b (Cell 1) (Cell 0)))

; Fields of transparent structs are merged,
; so 'twice' works on symbolic values.
> c
(Cell (ite b 1 0))
> (twice c)
(Cell (+ (ite b 1 0) (ite b 1 0)))

; The symbolic heap now contains 4 values:
; b, ¬b, ite(b, 1, 0), ite(b, 1, 0) + ite(b, 1, 0).
```

```
(require rosette/lib/match)

(struct Cell (v)) ; Opaque struct.

; Return a new Cell that doubles
; the value v of c.
(define (twice c)
  (match c
    [(Cell v) (Cell (+ v v))]))

; Create a symbolic Cell.
(define-symbolic b boolean?)
(define c (if b (Cell 1) (Cell 0)))

; Fields of opaque structs are not merged,
; so 'twice' works on concrete values.
> c
{[b ⊢ (Cell 1)] [(! b) ⊢ (Cell 0)]}
> (twice c)
{[b ⊢ (Cell 2)] [(! b) ⊢ (Cell 0)]}

; The symbolic heap now contains 2 values,
; b, ¬b, but the graph has more paths.
```

The SymFix value splitting repair toggles the transparency annotation on user-defined structures in a way that preserves soundness. Under Rosette semantics, it is sound to make structs less transparent (i.e., the transparency annotation can be removed) but not more. So given a location within a struct declaration, the value splitting repair produces at most one program. Like path splitting, this repair creates more opportunities for concrete evaluation, at the cost of adding more paths to the symbolic evaluation graph.

Termination and precedence. The SymFix repairs form a terminating set with a total precedence relation $R_V \preceq_{\mathsf{R}} R_D \preceq_{\mathsf{R}} R_P$ that orders value splitting first, deforestation second, and path splitting last. To see this, first note that value splitting applies to structs, while neither of the other repairs does, so R_V can be freely reordered with R_D and R_P. Next, observe that if deforestation R_D follows path splitting R_P, then either they were applied to disjoint locations, or R_P was applied to an expression that is moved but not transformed by deforestation (e.g., `xs` in the `sum-slow` example). In either case, the same effect can be achieved by applying R_P after R_D (though not vice versa). Finally, note that R_V and R_D can be applied to the same location at most once, and R_P can be applied at most N times, where N is the number of bound identifiers in the enclosing context. Hence, the set $\{R_V, R_D, R_P\}$ is terminating.

4.3 Implementation

We implemented the SymFix algorithm and repairs for Rosette. All three repairs require side effect analysis to preserve soundness, and we implement a simple conservative analysis that allows repairs only on expressions built out of procedures and constructs known to be safe. Because the repairs are totally ordered, we apply them in stages so that all of our fixes are of the form $R_V^* R_D^* R_P^*$. While

our repair framework assumes that programs have no unbounded loops, Rosette places no bounds on loops by design [46], so it is possible to write a Rosette program that does not terminate under symbolic evaluation. Our implementation deals with diverging and slow executions with timeouts.

5 Evaluation

To evaluate the effectiveness of SymFix, we address three research questions:

RQ1 : Can SymFix repair the performance of state-of-the-art solver-aided tools, and how do its fixes compare to those written by experts?
RQ2 : Do the fixes found by SymFix generalize to different workloads?
RQ3 : How important is SymFix's search strategy for finding useful fixes?

All results in this section were collected using an Intel Core i7-7700K at 4.20 GHz with 16 GB of RAM, running Racket v7.4. Each timing result is the average of 10 executions of the corresponding experiment.

5.1 Can SymFix repair the performance of state-of-the-art solver-aided tools, and how do its fixes compare to experts'?

To demonstrate that SymFix is effective on state-of-the-art solver-aided tools, we collected a suite of 15 tools [2, 5, 6, 8, 10, 12, 14, 25, 36, 33, 46, 50, 51] built in Rosette from a prior literature survey [7], together with 3 more recent tools [29, 31, 35]. For each of these Rosette programs, we applied SymFix to identify and repair performance bottlenecks.

Figure 5 shows the results. For each program, we report the original running time in seconds, and the cost of the original program as estimated by SymFix. We report three sources of repairs: fixes found by SymFix, fixes found by a baseline greedy algorithm discussed in §5.3, and manual fixes from prior work [7, 29]. We used a one-hour timeout for all experiments. For each fix, we report the relative speedup and cost decrease compared to the original run time and cost. A dash "–" indicates the absence of data due to timeouts or the lack of known manual fixes. One original program (Cosette) does not terminate within an hour, so we report only its repaired running times and costs.

SymFix finds fixes that improve the performance of 12 programs, with the improvements ranging from 1.1× to 91.7×. SymFix also finds 2 fixes that lower the symbolic cost and runtime only slightly, marked as 1.0× in Figure 5. The "#" column reports the number of iterations of SymFix's search procedure needed to find the fix, and "$|F|$" reports the number of repair steps in the fix. Most fixes are found in fewer than 10 iterations, and most have up to 2 repair steps.

Of the 15 benchmarks from prior work, 7 were manually fixed by the authors of that paper. For two of these benchmarks (Neutrons and RTR), the expert finds a significantly better fix than SymFix or finds some fix while SymFix finds none. Overall, SymFix matches or outperforms experts on 4 benchmarks, and

		Original		SymFix				Greedy		Manual	
Program	LoC	Time	Cost	Time	Cost	$\|F\|$	#	Time	Cost	Time	Cost
Bagpipe	3317	17 s	6.0e4	1.0×	1.0×	1	6	–	–	–	–
Bonsai[†]	641	27 s	1.5e6	1.3×	1.3×	2	21	1.1×	1.1×	1.0×	0.9×
Cosette[§]	2709	–	–	21 s	6.8e5	3	33	–	–	15 s	7.4e5
Ferrite	350	13 s	9.8e5	2.8×	3.8×	4	11	–	–	1.6×	1.1×
Fluidics	145	10 s	6.5e5	1.9×	1.7×	1	1	1.9×	1.7×	2.1×	1.8×
FRPSynth	304	3 s	2.3e4	3.1×	1.6×	4	93	1.4×	1.3×	–	–
GreenThumb	934	1179 s	2.0e5	1.3×	1.1×	1	1	1.3×	1.1×	–	–
IFCL	574	53 s	6.2e5	–	–	–	–	–	–	–	–
Memsynth	3362	15 s	2.0e6	1.1×	1.1×	1	2	1.0×	1.1×	–	–
Neutrons	37317	29 s	5.6e6	2.0×	2.3×	3	5	2.0×	2.3×	193.7×	869.9×
Nonograms	6693	8 s	1.5e5	1.1×	1.4×	7	46	–	–	–	–
Quivela	5946	47 s	2.9e6	91.7×	218.4×	6	7	90.1×	187.3×	86.1×	218.5×
RTR	2007	282 s	1.6e7	–	–	–	–	–	–	7.2×	4.1×
Serval[‡]	8641	116 s	7.3e6	6.2×	80.7×	1	1	–	–	6.2×	80.7×
Swizzle	1240	7 s	3.1e5	1.8×	1.3×	2	18	–	–	–	–
SynthCL	3732	16 s	7.5e5	–	–	–	–	–	–	–	–
Wallingford	3866	2 s	8.5e3	1.0×	1.0×	1	2	–	–	–	–
WebSynth	2057	7 s	1.0e6	–	–	–	–	–	–	–	–

[†] The manual repair was made unnecessary by a subsequent Rosette improvement.
[§] The repair by SymFix involves independent changes from users.
[‡] The repair by SymFix uses user-supplied repairs.

Fig. 5: Summary of fixes found by SymFix, a baseline greedy search, and experts. "LoC" is the number of lines of code in a given benchmark; "Cost" is SymFix's cost function for search; "$|F|$" is the number of repair steps in the fix found by SymFix; and "#" is the number of fixes explored in one hour before the reported best one is found.

it finds fixes for 5 benchmarks with no expert fix. We inspected all the fixes manually, and discuss interesting cases below.

For **Bonsai** (a synthesis tool for checking type-system soundness [12]), **Neutrons** (a verifier for safety-critical systems [33]), and **RTR** (a refinement type checker for Ruby), the manual fixes were sound but not semantics-preserving, so SymFix cannot discover them. For Bonsai, the manual fix was made unnecessary by a subsequent Rosette improvement, but SymFix still discovers a new repair that improves the performance further. For Neutrons, SymFix cannot recover the manual fix but does find a concretization opportunity offering a 2.0× speedup. For RTR, SymFix does not find a useful fix, suggesting future opportunities to exploit *conditional* repairs that are only sound under certain preconditions [40].

For **Cosette**, an automated prover for deciding the equivalence of two SQL queries [14], the original implementation did not terminate within one hour. The expert fix allowed Cosette to terminate in 15 seconds. Because SymFix needs to execute the original program during the search for repairs, we imposed a timeout of 60 seconds per execution. SymFix finds a fix that reduces Cosette's run time to 21 seconds, comparable to the manual fix. This new fix combines path splitting and deforestation of the map–reduce pattern Cosette uses to filter SQL tables. Finding the deforestation repair required converting Cosette's recursive implementation of this pattern into a higher-order version, but the Cosette developers

		Original		SymFix	
Program	Input	Time (s)	Cost	Time	Cost
Bonsai	nanodot	17 s	7.8e5	1.2×	1.1×
Cosette	q2	1 s	4.6e4	2.2×	9.8×
Cosette	q3	–	–	33 s	1.3e6
Ferrite	chrome	99 s	2.1e7	16.2×	15.6×
FRPSynth	program0	2 s	1.9e4	0.8×	0.8×
Quivela	test-etm-10	19 s	6.7e5	1.8×	1.8×
Serval	enosys	105 s	8.0e5	1.8×	11.3×
Swizzle	stencil	6 s	2.1e5	1.1×	1.1×
Swizzle	aos-sum	5 s	5.4e4	1.1×	1.0×

Fig. 6: Effectiveness of SymFix's repairs from Figure 5 on alternative workloads.

made this change independently to implement the manual fix; SymFix exploited this new structure to find another fix that allows Cosette to terminate in seconds.

For **Fluidics**, a tool for synthesizing programs that control a digital microfluidics array [51], the expert-written fix involves a change to the core data structure the tool uses to represent the array. This change is outside the scope of SymFix's search space. However, SymFix instead discovers a different fix that uses path splitting and requires no changes to the data structure. This fix offers a 1.9× speedup instead of 2.1× for the manual one, but it is made automatically and allows the tool's developers to retain their preferred data structure.

For **Ferrite**, a tool for checking file-system crash consistency [5], SymFix improves upon the expert-written fix by finding additional opportunities for concretization through path and value splitting. These changes make Ferrite close to 2× faster than the expert-repaired version.

For **GreenThumb**, a tool for developing superoptimizers [36], SymFix finds a concretization opportunity that the expert did not. The concretization both improves symbolic evaluation and alters the shape of the SMT formula so that SMT solving is 1.1× faster. SymFix also finds previously unknown concretization opportunities for **FRPSynth**, a tool for synthesizing functional reactive programs [31], and **Swizzle**, a tool for synthesizing GPU kernels [35], leading to a 3.1× and 1.8× speed-up, respectively.

For **Serval**, a toolset for automatic verification of systems software [29], SymFix does not discover a significant fix using its built-in repairs. But Serval comes with its own set of *symbolic optimizations*, which were originally designed for manual application [29]. Using these optimizations as repairs, SymFix discovers the manual fix, showing that its algorithm works well with a variety of repairs.

5.2 Do the fixes found by SymFix generalize to different workloads?

SymFix generates each of the fixes in Figure 5 using a single input to the respective program. To determine whether discovered repairs generalize to *different* program inputs, we identified the programs in Figure 5 that have alternative inputs available and executed the repaired versions on them.

Figure 6 shows the performance of each program on alternative inputs, both before and after the fix that SymFix discovered in Figure 5. In all but one case,

the fix generalizes to the new input and improves the program's performance. The relative performance improvement varies from Figure 5 due to different problem sizes; for example, the new Ferrite input is much larger than the original and so spends comparatively less time in the fixed procedure. The one exception is the "program0" input to FRPSynth, which is 20% slower than the original version. Manual inspection of this fix shows that the last of its 4 repair steps overfits to the initial input, and the first 3 steps improve the performance on both inputs.

5.3 How important is SymFix's search strategy for finding fixes?

SymFix employs a complete form of best-first search, guided by symbolic profiling ranks. To evaluate the importance of these design choices, we consider two alternative algorithms without them:

Random implements a complete best-first search that is not guided by profiling ranks, and instead chooses a location randomly at line 10 in Figure 4; and

Greedy implements the standard greedy best-first search, which applies only the first repair produced by NEXT at line 20 and never backtracks (by removing P from W unconditionally at line 27).

The **Random** algorithm discovers no useful fix for any benchmark within a one hour timeout. This is not surprising since the space of fixes is exponential in the number of potential repair locations, and there are thousands of such locations in each benchmark. The results for the **Greedy** algorithm are reported in the last two columns of Figure 5. **Greedy** finds a useful fix for only half (7) of the benchmarks repaired by SymFix, and none of its fixes are better than those found by SymFix. These results show that the key features of the SymFix algorithm are vital for fixing performance bottlenecks in real solver-aided tools.

6 Related Work

Profile-guided optimization. Compilers often support *profile-guided* optimization, in which the compiler uses profile data to guide its optimization phases (see Gupta et al. [23] for a survey). For example, the efficacy of inlining depends on factors including cache size and access patterns that are best determined by executing the program in the intended environment. Pettis and Hansen [34] introduce a profile-guided code layout algorithm that tries to position commonly used code together in memory to improve spatial locality. As another example, many JIT compilers for both static and dynamic languages will *specialize* methods based on type information observed at run time [20, 32] (e.g., specializing virtual calls for a particular concrete receiver). SymFix takes inspiration from these approaches, using profile data to guide the application of semantics-preserving repairs. But unlike them, SymFix focuses on optimizing a program's symbolic evaluation strategy rather than its utilization of machine resources.

Not all profile-guided optimization techniques are automated. Optimization coaching [42] is an interactive tool that gives programmers feedback about the

optimizations a compiler applied to their program, and optimizations that it tried unsuccessfully. SymFix does not provide interactivity, but because its repairs are high level, it can follow the optimization coach approach of reporting them to the programmer as syntactic changes to their input program.

Symbolic profiling. Because SymFix uses profile data to guide the search for fixes, its effectiveness depends on high quality profiles. SymFix builds on *symbolic profiling* [7], a technique for profiling the behavior of symbolic evaluation engines. Symbolic profiling generalizes across a spectrum of symbolic evaluation techniques, and so SymFix's approach could generalize to other engines that support symbolic profiling (e.g., Crucible [21]). Other profiling techniques measure different aspects of automated tools. The Z3 Axiom Profiler [3] measures axiom instantiations in the Z3 [16] SMT solver's quantifier theory module. It can be used to detect optimization opportunities at the SMT level. Using such profilers to extend SymFix to the SMT level is an interesting direction for future work.

Optimizing symbolic evaluation. A number of approaches exist for interactively improving the performance of tools based on symbolic evaluation. Wagner et al. [49] introduce a configuration for optimizing compilers to prioritize generating code that is amenable to symbolic execution. Cadar [11] develops a suite of compiler optimizations that make code easier to evaluate symbolically. Nelson et al. [29] develop a set of custom symbolic optimizations that can be manually applied to build scalable verifiers for low-level languages (e.g., RISC-V [1], LLVM [28], x86 [24], and eBPF [19]) on top of a generic verification framework. SymFix is complimentary to these approaches: it can automatically apply custom optimizations to verifiers for low-level code, and these verifiers can further benefit from the custom compiler optimizations applied to their input programs.

7 Conclusion

This paper presented a new approach to repairing performance bottlenecks in code under symbolic evaluation. Our approach rests on three technical contributions. We formulate the symbolic performance repair problem as combinatorial search for a fix that applies a sequence of semantics-preserving repairs to a program and a workload; the resulting fixed program is guaranteed to be equivalent to the input program, and to have minimal symbolic evaluation cost on the input workload. To solve this repair problem, we develop SymFix, a system with two key components: (1) a small set of generic repairs based on deforestation and symbolic reflection, and (2) an anytime search algorithm that uses symbolic profiling to guide the exploration of this space. Our evaluation shows that SymFix can discover useful fixes for state-of-the-art verification and synthesis tools, matching or outperforming experts, and that the fixed programs continue to work well across different workloads. As more programmers employ symbolic evaluation to automate verification and synthesis tasks for new domains, SymFix can help them build better tools more easily.

Bibliography

[1] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. RISC-V Foundation (Jun 2019)

[2] Amazon Web Services: Quivela (2018), https://github.com/awslabs/quivela

[3] Becker, N., Müller, P., Summers, A.J.: The axiom profiler: Understanding and debugging SMT quantifier instantiations. In: Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 99–116. Prague, Czech Republic (Apr 2019)

[4] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 193–207. Amsterdam, The Netherlands (Mar 1999)

[5] Bornholt, J., Kaufmann, A., Li, J., Krishnamurthy, A., Torlak, E., Wang, X.: Specifying and checking file system crash-consistency models. In: Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 83–98. Atlanta, GA, USA (Apr 2016)

[6] Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 467–481. Barcelona, Spain (Jun 2017)

[7] Bornholt, J., Torlak, E.: Finding code that explodes under symbolic evaluation. Proc. ACM Program. Lang. (OOPSLA), 149:1–149:26 (Oct 2018)

[8] Borning, A.: Wallingford: Toward a constraint reactive programming language. In: Proceedings of the Constrained and Reactive Objects Workshop (CROW). Málaga, Spain (Mar 2016)

[9] Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 239–254. Salt Lake City, UT, USA (Mar 2014)

[10] Butler, E., Torlak, E., Popović, Z.: Synthesizing interpretable strategies for solving puzzle games. In: Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG). No. 10, Hyannis, MA, USA (Aug 2017)

[11] Cadar, C.: Targeted program transformations for symbolic execution. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 906–909. Bergamo, Italy (Aug 2015)

[12] Chandra, K., Bodik, R.: Bonsai: Synthesis-based reasoning for type systems. Proc. ACM Program. Lang. 2(POPL), 62:1–62:34 (Jan 2018)

[13] Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 265–278 (2011)

[14] Chu, S., Wang, C., Weitz, K., Cheung, A.: Cosette: An automated prover for SQL. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems (CIDR). Chaminade, CA, USA (Jan 2017)
[15] Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
[16] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Budapest, Hungary (Mar 2008)
[17] Dershowitz, N., Jouannaud, J.: Rewrite systems. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 243–320 (1990)
[18] Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010)
[19] Fleming, M.: A thorough introduction to eBPF (Dec 2017), https://lwn.net/Articles/740157/
[20] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 465–478. Dublin, Ireland (Jun 2009)
[21] Galois, Inc.: Crucible (2018), https://github.com/GaloisInc/crucible
[22] Gill, A.J., Jones, S.L.P.: Cheap deforestation in practice: An optimizer for haskell. In: IFIP Congress (1994)
[23] Gupta, R., Mehofer, E., Zhang, Y.: Profile guided compiler optimizations. In: The Compiler Design Handbook: Optimizations and Machine Code Generation, chap. 4. CRC Press (Sep 2002)
[24] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (2015), revision 53
[25] Kazerounian, M., Vazou, N., Bourgerie, A., Foster, J.S., Torlak, E.: Refinement types for ruby. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 269–290. Los Angeles, CA, USA (Jan 2018)
[26] King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
[27] Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 89–98. Beijing, China (Jun 2012)
[28] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO). Palo Alto, California (Mar 2004)
[29] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with Serval.

In: Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP). pp. 225–242 (2019)
[30] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-button verification of an OS kernel. In: Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP). pp. 252–269 (2017)
[31] Newcomb, J.L., Bodik, R.: Using human-in-the-loop synthesis to author functional reactive programs (2019)
[32] Paleczny, M., Vick, C., Click, C.: The Java HotSpot Server Compiler. In: Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology (JVM). Monterey, CA, USA (Apr 2001)
[33] Pernsteiner, S., Loncaric, C., Torlak, E., Tatlock, Z., Wang, X., Ernst, M.D., Jacky, J.: Investigating safety of a radiotherapy machine using system models with pluggable checkers. In: Proceedings of the 28th International Conference on Computer Aided Verification (CAV). vol. 2, pp. 23–41. Toronto, ON, Canada (Jul 2016)
[34] Pettis, K., Hansen, R.C.: Profile guided code positioning. In: Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 16–27. White Plains, NY, USA (Jun 1990)
[35] Phothilimthana, P.M., Elliott, A.S., Wang, A., Jangda, A., Hagedorn, B., Barthels, H., Kaufman, S.J., Grover, V., Torlak, E., Bodik, R.: Swizzle inventor: Data movement synthesis for GPU kernels. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 65–78 (2019)
[36] Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 297–310. Atlanta, GA, USA (Apr 2016)
[37] Racket: The Racket programming language (2017), https://racket-lang.org
[38] S2E: S2E: Exponential analysis speedup with state merging (2019), http://s2e.systems/docs/StateMerging.html
[39] Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 488–498. Saint Petersburg, Russian Federation (Aug 2013)
[40] Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Conditionally correct superoptimization. In: Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 147–162. Pittsburgh, PA, USA (Oct 2015)
[41] Sigurbjarnarson, H., Nelson, L., Castro-Karney, B., Bornholt, J., Torlak, E., Wang, X.: Nickel: A framework for design and verification of information flow control systems. In: Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI). pp. 287–305 (2018)

[42] St-Amour, V., Tobin-Hochstadt, S., Felleisen, M.: Optimization coaching: Optimizers learn to communicate with programmers. In: Proceedings of the 27th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 163–178. Tuscon, AZ, USA (Oct 2012)
[43] Torlak, E.: Rosette (2018), http://github.com/emina/rosette
[44] Torlak, E.: The Rosette guide: Symbolic reflection (2019), https://docs.racket-lang.org/rosette-guide/ch_symbolic-reflection.html
[45] Torlak, E., Bodik, R.: Growing solver-aided languages with Rosette. In: Proceedings of the 2013 ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!). pp. 135–152. Indianapolis, IN, USA (Oct 2013)
[46] Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 530–541. Edinburgh, United Kingdom (Jun 2014)
[47] Uhler, R., Dave, N.: Smten with satisfiability-based search. In: Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 157–176. Portland, OR, USA (Oct 2014)
[48] Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Proceedings of the Second European Symposium on Programming. pp. 231–248 (1988)
[49] Wagner, J., Kuznetsov, V., Candea, G.: -Overify: Optimizing Programs for Fast Verification. In: Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS). Santa Ana Pueblo, NM, USA (May 2013)
[50] Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A., Tatlock, Z.: Scalable verification of border gateway protocol configurations with an smt solver. In: Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 765–780. Amsterdam, The Netherlands (Oct 2016)
[51] Willsey, M., Ceze, L., Strauss, K.: Puddle: An Operating System for Reliable, High-Level Programming of Digital Microfluidic Devices. In: Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Wild and Crazy Ideas Session. Williamsburg, VA, USA (Mar 2018)