Path-based Inductive Synthesis for Program Inversion

Saurabh Srivastava
University of California, Berkeley
saurabhs@cs.berkeley.edu

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Swarat Chaudhuri
Pennsylvania State University
swarat@cse.psu.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Abstract

In this paper, we investigate the problem of semi-automated inversion of imperative programs, which has the potential to make it much easier and less error prone to write programs that naturally pair as inverses, such as insert/delete operations, compressors/decompressors, and so on. Viewing inversion as a subproblem of program synthesis, we propose a novel synthesis technique called Path-based Inductive Synthesis (PINS) and apply it to inversion. PINS starts from a program P and a template T for its inverse. PINS then iteratively refines the space of template instantiations by exploring paths in the composition of P and T with symbolic execution. PINS uses an SMT solver to intelligently guide the refinement process, based on the paths explored so far. The key idea motivating this approach is the *small path-bound hypothesis*: that the behavior of a program can be summarized with a small, carefully chosen set of its program paths.

We evaluated PINS by using it to invert 14 programs such as compressors (e.g., Lempel-Ziv-Welch), encoders (e.g., UUEncode), and arithmetic operations (e.g., vector rotation). Most of these examples are difficult or impossible to invert using prior techniques, but PINS was able to invert all of them. We also found that a semi-automated technique we developed to *mine* a template from the program to be inverted worked well. In our experiments, PINS takes between one second to thirty minutes to synthesize inverses. We believe this proof-of-concept implementation demonstrates the viability of the PINS approach to program synthesis.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program Synthesis; D.2.5 [*Testing and Debugging*]: Symbolic Execution

General Terms Languages, Algorithms, Theory

Keywords PINS, Program Inversion, Inductive Synthesis, Symbolic execution, Testing-inspired Synthesis

1. Introduction

Recently, there has been significant interest in *program synthesis*, in which an automated tool helps the programmer derive program source code from its specification [28, 27, 38, 33, 26, 13]. One particularly interesting synthesis subproblem is *program inversion*, which is the task of constructing a P^{-1} given a program P such that executing P followed by P^{-1} is the identity function. Pairs of inverses are quite common in software, e.g., insert/delete operations on data structures, encryption/decryption, compression/decompression, rollback in transactions, etc.. Automatic program inversion could potentially allow programmers to write only one of each inverse pair, halving the time required to write and maintain such code and eliminating bugs due to inconsistencies between the inverses.

However, existing program synthesis techniques are not well-suited to the inversion task. Proof-theoretic synthesis [38] requires inferring loop invariants, which are unrealistic for programs such as compressors or encoders. Counterexample-guided inductive synthesis [33] requires finitizing the domain (e.g., specifying a bound on loop unrollings and lengths of arrays), and we have found this is a significant burden. Other approaches that are specifically for inversion, rather than general program synthesis, are either hard to implement [8, 12] or are quite restricted in the programs that can be inverted [11, 24, 41, 11].

In this paper, we present a new program synthesis algorithm, *Path-based inductive synthesis (PINS)*, that takes a major step in addressing these challenges. At a high level, the PINS algorithm is straightforward. It iteratively uses symbolic execution to simulate paths through a known program P concatenated with a *template* of an unknown program P^{-1}. The template contains an ordinary program, except it can contain unknown expressions and predicates, i.e., "holes" that need to be filled in. The template also includes sets Π_e of candidate expressions and Π_p of candidate predicates that can instantiate unknowns. PINS uses the *path conditions* generated from symbolic execution to create a query for an SMT solver, and the solution of the query assigns elements of Π_e and Π_p to the unknowns such that P followed by P^{-1} is an inverse along the corresponding paths.

The key idea that underlies PINS is what we call the *small path-bound hypothesis*: for many programs, all behavior can be summarized by examining a small, carefully chosen set of its program paths. This same hypothesis underlies program testing, and just as in testing, it is critical that PINS picks the right set of paths to explore—and this is potentially quite challenging in synthesis, in which we are trying to find useful paths through a program with unknowns. Thus, PINS includes a heuristic technique that tries to guide symbolic execution down program paths likely to reinforce correct solutions and eliminate incorrect solutions. This heuristic helps ensure that when PINS finishes, its output is highly likely (though not guaranteed) to be correct. To further validate correctness, the programmer can use a range of techniques. In our experiments, we used manual code inspection, testing with concrete instantiations of the paths explored by PINS, and bounded model checking [1] to validate PINS's output.

PINS has a number of attractive features compared to other synthesis approaches. First, by using symbolic execution, PINS is able to reason about program paths precisely, but without needing strong loop invariants that prove functional correctness. Second, PINS need not finitize the problem domain, since it relies on SMT solving. Finally, SMT solving has another benefit: it allows PINS to model external library functions with axioms, rather than needing source code. For example, we can add axioms about string library functions such as $\forall s, c : \texttt{strlen}(\texttt{append}(s, c)) = \texttt{strlen}(s) + 1$, where s and c are string and character types, respectively. This makes the synthesis task smaller and more modular.

One challenge in using PINS, or any other template-based synthesis approach (e.g., proof-theoretic synthesis [38], invariant-based synthesis for hybrid systems [39] or Sketching [33]), is determining what template to use. For program inversion, we have developed a semi-automated technique for *mining* the template from the program to be inverted. In our approach, an automated tool generates initial guesses for Π_p and Π_e based on the original program text, and the programmer similarly constructs a template program based on the control flow of the original program. The user then runs PINS, using its output to further guide manual refinement of the template. We have found that this approach is much easier than other synthesis techniques that require constructing a template from scratch [38, 39, 33].

We implemented PINS and used it to invert 14 small but complex programs ranging in size from 5 to 25 lines of code. Our benchmarks include several compressors (e.g., Lempel-Ziv-Welch), encoders (e.g., UUEncode), and arithmetic operations (e.g., rotation of a vector). For these benchmarks, PINS worked well in reducing the huge space of possible template instantiations to only a few final candidates. In 11 cases, PINS produced 1 candidate inverse, which was correct. In the remaining 3 cases, PINS produced at most 4 candidates, at least one of which was correct.

For these benchmarks, we found that our semi-automated template mining strategy worked well; we needed only a few changes from the initial template for synthesis to succeed. We also found that the ability to add axioms was critical for our benchmarks, 8 of which called external libraries.

Our results also support the small path-bound hypothesis. Synthesizing inverses for our benchmarks required exploring between 1 and 14 paths, with a median of 5 paths. The running time of PINS varied significantly, ranging from 1 second to 30 minutes. The high variability is due to the large semantic differences between programs and the unpredictable nature of SMT/SAT solving. More engineering may reduce these times much further, but we think that for a prototype, these times are still encouraging, especially as programmer time is far more expensive than processing time.

We believe that PINS takes an important step forward in program synthesis, and presents a promising new approach to the program inversion program.

2. Path-based Inductive Synthesis

We begin our presentation by describing the PINS algorithm. As a running example, we will consider synthesizing an inverse to the function shown in Figure 1, which performs in-place run-length encoding of an input array A of length n. The function imperatively updates the arrays A (destructively) and N to hold the compressed output and the count of how often each compressed element appeared in the input, respectively. The output variable m gives the length of the compressed output. Our aim is to invert this function, i.e., to produce code that generates a new A' to contain the original contents that were in A, using the counts in N.

2.1 Synthesis templates

To constrain the search space for synthesis, PINS uses a *synthesis template* supplied by the programmer. In PINS, a template is a triple (P, Π_e, Π_p), where P is a program that may contain *unknown expressions* ε and *unknown predicates* ρ, and Π_e and Π_p are sets

```
runlength(inout datatype *A, in int n,
          out int *N, out int m)
{
   int i,r;
   assume(n ≥ 0);
   i:=0;
   m:=0;
   while (i < n)
      r := 1;
      while (i+1 < n ∧ A[i] = A[i+1])
         r := r+1; i := i+1;
      A[m] := A[i];
      N[m] := r;
      m := m+1;
      i := i+1;
}
```

Figure 1. In-place run length encoding of an array A of n data elements, where the encoded output is of length m. Another output array N holds the counts.

of expressions and predicates (without unknowns) that ε and ρ, respectively, may range over. Formally, template programs P are defined by the following language:

$$\begin{array}{lcl} P, s & ::= & x, \ldots, x := e, \ldots, e \mid s; s \mid \texttt{if}(*)\ s\ \texttt{else}\ s \\ & & \mid \texttt{while}(*)\ s \mid \texttt{assume}(p) \mid \texttt{exit} \mid \texttt{in}(x, ..) \mid \texttt{out}(x, ..) \\ e & ::= & \varepsilon \mid x \mid e\ op_a\ e \mid \texttt{sel}(e, e) \mid \texttt{upd}(e, e, e) \mid \mathbf{f}(\vec{e}) \\ p & ::= & \rho \mid * \mid e\ op_r\ e \end{array}$$

Program statements consist of parallel assignments to variables, sequencing, conditionals, while loops, and assume statements. We also add a form exit, to mark the exit of the program, and in and out, which indicate which program variables are inputs and outputs, respectively. These last two forms are used to help construct the synthesis template (Section 3).

Note that conditionals and loops are non-deterministic in our language. As is standard, we can encode $\texttt{if}(p)\ s_1\ \texttt{else}\ s_2$ in our language as $\texttt{if}(*)\ (\texttt{assume}(p); s_1)\ \texttt{else}\ (\texttt{assume}(\neg p); s_2)$, and $\texttt{while}(p)\ s$ as $(\texttt{while}(*)\ (\texttt{assume}(p); s)); \texttt{assume}(\neg p)$. For notational convenience, we may use the skip statement as well, which can be modeled in the language as assume(true).

Expressions may either be unknown expressions ε, variables x, arithmetic operations $e\ op_a\ e$, array reads $\texttt{sel}(e_1, e_2)$ (read element e_2 from array e_1), non-destructive array writes $\texttt{upd}(e_1, e_2, e_3)$ (return a new array that is the same as e_1, except element e_2 has value e_3), and uninterpreted function symbols $\mathbf{f}$ (used to model external library calls). Similarly, predicates used in assume expressions may be unknown ρ or known predicates p, which are known expressions compared with relational operators op_r. Then Π_e is a set of expressions that do not contain occurrences of ε, and analogously for Π_p.

As mentioned in the introduction, for program inversion we construct a template for the inverse and then concatenate that to the original program to produce the synthesis template for PINS. We defer discussion of exactly how the inverse template is constructed until Section 3. For our running example, the input to PINS is shown in Figure 2. For clarity, we have written while loops with guards, though as mentioned above we actually encode these with non-deterministic choice and assume. Lines 1–11 are the same as in Figure 1, except they have been translated into our formal language. Lines 12–18 contain the inverse template. Notice that in this particular example, the inverse template has essentially the same shape as the original program, and that the primed variables used by the inverse arise from (unprimed) variables used in the original program. The sets Π_e and Π_p range over expressions extracted from the original program, but translated to work over both primed and selected unprimed variable names.

```
in($A, n$);
assume($n \geq 0$);
$i, m$:=$0, 0$;
while ($i < n$)
    $r$ := 1;
    while ($i + 1 < n \wedge$ sel($A, i$) = sel($A, i + 1$))
        $r, i$ := $r + 1, i + 1$;
    $A$ := upd($A, m,$ sel($A, i$));
    $N$ := upd($N, m, r$);
    $m, i$ := $m + 1$, $i + 1$;
out($A, N, m$);
$i', m' := \varepsilon_1, \varepsilon_2$;
while ($\rho_1$)
    $r' := \varepsilon_3$;
    while ($\rho_2$)
        $r', i', A' := \varepsilon_4, \varepsilon_5, \varepsilon_6$;
    $m' := \varepsilon_7$;
out($A', i'$); exit;
```

$$\Pi_e = \left\{ \begin{array}{c} 0, 1, m'+1, m'-1, r'+1, r'-1, i'+1, i'-1, \\ upd(A', m', sel(A, i')), upd(A', i', sel(A, m')), sel(N, m') \end{array} \right\}$$

$$\Pi_p = \{\ m' < m, r' > 0, sel(A', i') = sel(A', i'+1)\ \}$$

Figure 2. Program from Figure 1 composed with the template.

$$\textsc{Assn}\quad \frac{v_i = V(x_i) + 1 \qquad V' = V[x_i \mapsto v_i] \qquad \forall i \in 1..n \qquad \phi' = \phi \wedge \bigwedge_{j \in 1..n} (x_j^{v_j} = e_j^V)}{S; \Phi \vdash \langle (x_1, \ldots x_n := e_1, \ldots, e_n); \phi; V \rangle \to \langle \phi'; V' \rangle}$$

$$\textsc{Assume}\quad \frac{\phi \wedge (S(p))^V \not\Rightarrow \texttt{false} \qquad \phi' = \phi \wedge p^V}{S; \Phi \vdash \langle \texttt{assume}(p); \phi; V \rangle \to \langle \phi'; V \rangle}$$

$$\textsc{Seq}\quad \frac{S; \Phi \vdash \langle s_1; \phi; V \rangle \to \langle \phi_1; V_1 \rangle \qquad S; \Phi \vdash \langle s_2; \phi_1; V_1 \rangle \to \langle \phi_2; V_2 \rangle}{S; \Phi \vdash \langle (s_1; s_2); \phi; V \rangle \to \langle \phi_2; V_2 \rangle}$$

$$\textsc{Exit}\quad \frac{\phi \notin \Phi}{S; \Phi \vdash \langle \texttt{exit}; \phi; V \rangle \to \langle \phi; V \rangle}$$

$$\textsc{Cond}\quad \frac{S; \Phi \vdash \langle s_i; \phi; V \rangle \to \langle \phi'; V' \rangle \qquad i = 1 \text{ or } 2}{S; \Phi \vdash \langle (\texttt{if}(*)\ s_1\ \texttt{else}\ s_2); \phi; V \rangle \to \langle \phi'; V' \rangle}$$

$$\textsc{Loop}\quad \frac{S; \Phi \vdash \langle \texttt{if}(*)\ (s; \texttt{while}(*)\ s)\ \texttt{else skip}; \phi; V \rangle \to \langle \phi'; V' \rangle}{S; \Phi \vdash \langle \texttt{while}(*)\ s; \phi; V \rangle \to \langle \phi'; V' \rangle}$$

$$\textsc{InOut}\quad \frac{io \in \texttt{in}, \texttt{out}}{S; \Phi \vdash \langle io(x, ..); \phi; V \rangle \to \langle \phi; V \rangle}$$

Figure 3. Symbolic execution with unknowns.

2.2 Symbolic execution of templates

As discussed in the introduction, PINS iteratively uses *symbolic execution* to simulate chosen paths through a template. Symbolic execution has been studied extensively in the literature [25], but our use of symbolic execution has an interesting twist: we need to simulate programs containing unknowns. What makes this possible is that the only unknowns we permit are expressions and predicates, both of which are pure, and thus their evaluation does not affect the program state.

At each step of symbolic execution, we maintain a *path condition* ϕ that describes the path taken and the current and past state of all program variables. To distinguish different definitions of the same variable, we maintain a *version map* V that assigns a version number to each variable. As variables are reassigned, their version numbers monotonically increase, similarly to variable renaming in static single assignment form. As a slight abuse of notation, we will use the version map 0 to denote assigning all variables the version number 0. When an unknown expression ε or predicate ρ is evaluated, we simply pair it with the version map V at that program point, which we write as ε^V or ρ^V. Then at the end of symbolic execution, since the path condition contains the history of all variables, it gives us sufficient information to interpret uses of ε and ρ at any point during the prior execution. For notational convenience, we use the same evaluation strategy for known expressions and predicates, and we write e^V or p^V for the (known or unknown) expression or predicate evaluated under version map V.

As symbolic execution is a subroutine in PINS, it takes two auxiliary inputs. First, our symbolic executor is given a *solution* S, which is an assignment from unknown expressions and predicates to known expressions and predicates, respectively. The solution arises from a previous iteration of PINS, and is used to guide symbolic execution down paths that tend to more quickly refine the solution space; we discuss this further in Section 2.3. Second, our symbolic executor is given a *path condition set* Φ, which is the set of path conditions that have been previously explored and hence should not be simulated again.

Figure 3 formalizes our symbolic executor as a judgment of the form $S; \Phi \vdash \langle s; \phi; V \rangle \to \langle \phi'; V' \rangle$, which says that, using solution S and avoiding path condition set Φ, evaluating statement s beginning with path condition ϕ and version map V yields a new path condition ϕ' and version map V'.

We discuss the rules briefly. In Rule Assn, we increment the version numbers of all assigned variables to yield a new version map V', and we create a new path condition ϕ' containing equalities for the newly assigned variables. Notice here we pair the expressions e_j with the old version map V, since the e_j are evaluated at the start of the assignment.

In Rule Assume, we write $S(p)$ to mean the predicate p with unknowns replaced according to S. Note that S may be a partial map, and if S does not provide a mapping for p then $S(p)$ equals p. Thus, this rule requires that the conjunction of the current path condition and the assumed predicate is satisfiable according to the solution. In our implementation, we use an SMT query to determine the satisfiability of this constraint.

Rule Seq is standard. Rule Exit ensures that the path has not previously been explored. Rule Cond non-deterministically executes one of its branches. Rule Loop unrolls a loop one time. Finally, Rule InOut ignores in and out expressions, which are only used for constructing the synthesis template and the specification formula for inversion.

2.3 Synthesis algorithm

Given a synthesis template P, the goal of PINS is to find a *solution* S that assigns known expressions and predicates to unknown expressions and predicates, such that $S(P)$ satisfies a given specification. For program inversion, the specification is particularly simple—at a high level, the program should be the identity function. In our running example, at the end of execution, the array A should contain the same elements as at the beginning. While in this paper our focus is on program inversion, PINS is a general algorithm, and we believe it can also be used for other synthesis problems.

The PINS algorithm is shown as Algorithm 1. The input to the algorithm is a synthesis template (P, Π_e, Π_p), a specification *spec*, and a bound m on the number of solutions to request from

```
    Input: Synthesis template $(P, \Pi_e, \Pi_p)$, specification spec,
           bound $m$ on number of solutions from solver.
    Output: Solution $S$ or "No Solution."
 1  begin
 2  |   $\Phi := \emptyset$;
 3  |   $C :=$ terminate$(P)$;
 4  |   while (true) do
 5  |   |   $sols :=$ solve$(C, \Pi_p, \Pi_e, m)$;
 6  |   |   if $sols = \emptyset$ then
 7  |   |   |_  return "No Solution"; /* Refine abstraction */
 8  |   |   if stabilized$(sols, m)$ then
 9  |   |   |_  return $sols$;
10  |   |   $S :=$ pickOne$(sols)$;
11  |   |   $S; \Phi \vdash \langle P; \mathtt{true}; 0\rangle \to \langle \phi; V'\rangle$;
12  |   |   $\Phi := \Phi \cup \{\phi\}$;
13  |   |_  $C := C \wedge$ safepath$(\phi, V', \mathtt{spec})$;
14  end
```

Algorithm 1: The `PINS` algorithm.

the solver (discussed below). We next give an overview of the algorithm, deferring details of the subroutines `terminate`, `solve`, `stabilized`, `pickOne`, and `safepath` until after the overview.

Throughout its execution, `PINS` maintains a set Φ of program paths that have been symbolically executed so far, and a constraint C that includes constraints gathered from prior symbolic executions. As the algorithm progresses, Φ increases and C accumulates additional constraints. Φ is initially empty, and C is initially set to $\mathtt{terminate}(P)$, which constrains loops in P to terminate.

The main loop of `PINS` iteratively refines the space of solutions as follows. On line 5, we compute (at most) m solutions to the constraints C, using Π_p and Π_e for the possible values of unknown predicates and expressions in C. If the resulting set of solutions $sols$ is empty, then there is no valid template instantiation that satisfies *spec*. Typically this means the template needs to be refined; we describe this process in Section 3.

If $sols$ has `stabilized`, meaning it did not change from the last iteration, then we exit and return $sols$. Otherwise, we refine the solution space. On line 10, we set S to be one of the computed solutions, chosen by the heuristic `pickOne`. Then we symbolically execute P, using the solution S to guide the execution, and beginning with the path condition `true` and the version map that assigns version 0 to all variables, which we write simply as 0. Recall that in Rule ASSUME of Figure 3, we require that the guard p is satisfiable according to S—thus, the path taken in the symbolic execution on line 11 is a feasible path in the program $S(P)$. As we will discuss below, this means that the path taken will tend to generate constraints that either reinforce S if it is a valid solution, or contradict S if it is an invalid solution.

Finally, we add the path condition ϕ from the symbolic execution to Φ, and we add to C the constraint $\mathtt{safepath}(\phi, V', spec)$, which specifies that the path taken meets the specification *spec*. We then repeat this process until no solutions are possible, or the set of solutions has stabilized.

Next, we discuss the subroutines used by `PINS` in more detail and how `PINS` handles uninterpreted functions.

Safety constraints The constraint generated by `safepath` on Line 13 specifies that the symbolic execution, which generated path condition ϕ and final version map V', satisfies *spec*:

$$\mathtt{safepath}(\phi, V', spec) \doteq \forall X : \phi \Rightarrow spec^{V'}$$

where X is the set of all program variables at all versions. Notice that in $spec^{V'}$, program variables in *spec* will be interpreted in terms of V', i.e., at their final values at the end of execution.

EXAMPLE 1. *Suppose in Figure 2 we take a path that immediately exits the loop on line 4, enters the loop on line 13, exits the loop on line 15, exits the loop on line 13, and then exits the program. The safety constraint (left) generated for this path (right) is:*

$$\begin{array}{ll}
n^0 \geq 0 \wedge i^1 = 0 \wedge m^1 = 0 \wedge & \ldots n \geq 0; i, m := 0, 0; \\
i^1 \geq n^0 \wedge i'^1 = \varepsilon_1^{V_1} \wedge m'^1 = \varepsilon_2^{V_1} \wedge & \ldots i \geq n; i', m' := \varepsilon_1, \varepsilon_2; \\
\rho_1^{V_2} \wedge r'^1 = \varepsilon_3^{V_2} \wedge & \ldots \rho_1; r' := \varepsilon_3; \\
\neg\rho_2^{V_3} \wedge m'^2 = \varepsilon_7^{V_3} \wedge & \ldots \neg\rho_2; m' := \varepsilon_7; \\
\neg\rho_1^{V_4} & \ldots \neg\rho_1 \\
\Rightarrow spec^{V_4} &
\end{array}$$

where we abbreviate the version maps as $V_1 = \{n \mapsto 0, m \mapsto 1, i \mapsto 1\} \cup 0$, $V_2 = V_1 \cup \{m' \mapsto 1, i' \mapsto 1\}$, $V_3 = V_2 \cup \{r' \mapsto 1\}$, *and* V_4 *is* V_3 *but with* $m' \mapsto 2$. *The identity specification spec is derived syntactically from* $\mathtt{in}(A, n)$ *and* $\mathtt{out}(A', i')$) *as* $n^0 = i'^{V_4} \wedge \forall k : 0 \leq k < n^0 \Rightarrow A^0[k] = A'^{V_4}[k]$. *This path turns out to be infeasible for the actual synthesized inverse (shown in Section 3), but this does not mean that the path is redundant during synthesis. To the contrary, the path imposes constraints on the unknown expressions and predicates that eliminate any candidate for which this path is feasible, thereby pruning the search space.*

Termination constraints The termination constraints generated on Line 3 serve two important purposes. First, they prevent synthesis of programs that diverge, since such programs trivially satisfy any partial specification but are uninteresting in this domain. Second, they ensure that the symbolic execution runs on Line 11 themselves terminate, since those runs are guided by a solution to the constraints.

We generate termination constraints by reasoning about each loop separately. Consider a loop $l = \mathtt{while}(*)\{\mathtt{assume}(\rho_l); B_l\}$, where the guard ρ_l is an unknown and the body is B_l. We assume there is a corresponding ranking function η_l that is an unknown, ranging over a set of expressions Π_r (discussed below). We impose two kinds of constraints on η_l to ensure the loop terminates.

First, we assume the ranking function is related to the loop guard ρ_l, and generate a constraint

$$\mathtt{bounded}(l) \doteq \forall X.\ \rho_l^0 \Rightarrow ({\eta_l}^0 \geq 0)$$

that the guard implies a lower bound on the ranking function. Notice that this constraint does not involve any path condition, and hence the relationship it implies must hold across all possible values of the program variables.

Next, we need to generate a constraint that the ranking function decreases on every iteration through the loop. We introduce an *unknown* loop invariant ${\rho'}_l$, and generate the following constraints:

$$\begin{array}{l}
\mathtt{decrease}(l) \doteq \\
\qquad \bigwedge_{\langle\phi,V\rangle \in init} \forall X.\ \phi \Rightarrow {\rho'}_l^V \quad \wedge \\
\qquad \bigwedge_{\langle\phi,V\rangle \in body} \forall X.\ \phi \wedge {\rho'}_l^0 \Rightarrow {\rho'}_l^V \quad \wedge \\
\qquad \bigwedge_{\langle\phi,V\rangle \in body} \forall X.\ \phi \wedge \rho_l^0 \wedge {\rho'}_l^0 \Rightarrow {\eta_l}^V < {\eta_l}^0
\end{array}$$

where *body* is a set of tuples $\langle\phi; V\rangle$ from symbolic executions of the loop body, and *init* is a set of tuples from symbolic executions that start at the program entry and end at the loop entry. From top to bottom, these constraints specify that the invariant holds at the beginning of the loop; that it is maintained during an execution of the loop body; and that the invariant and the loop guard imply that the ranking function decreases on each iteration. Note that although these constraints look complex, in fact the loop invariant required for termination is often far simpler than a loop invariant required for functional correctness. For example, typically we only required an invariant that is a linear relation (real programs shown terminating by others [5, 6] illustrate that linear relations typically suffice), and we never required a quantified invariant.

To compute *body* and *init*, we use two heuristics. First, to compute *body* we symbolically execute all possible paths through the loop body (starting with the empty path conditions and empty version map 0), always taking the exit branch of any inner loop to keep the set of paths finite. Thus, we are assuming that inner loops do not affect the termination of outer loops. Second, we initialize *init* to be empty, and each time `PINS` symbolically executes some path on Line 11, we take the prefix of that path up to the start of the loop, and add the corresponding *init*-related constraint to C. Thus, we are only constraining the invariant to hold on a finite number of paths, rather than on every path, similar to dynamic approaches that infer likely invariants [10].

Other than the (transition) invariant involving `decrease` and `bounded`, in our experiments, additional inductive safety invariants are typically not required to prove termination. (An example of where we might need an inductive invariant, e.g., $C > 0$, might be when $x := x + C$ is the iteration counter increment.) Other work also shows that termination for typical programs can be proven without inductive reasoning [19].In this simpler case, the SMT/SAT solver discovers it can set the invariant to simply be `true`, in which case $\texttt{decrease}(l)$ simplifies to $\bigwedge_{\langle\phi,V\rangle\in body} \forall X : \phi \wedge \rho_l^0 \Rightarrow {\eta_l}^V < {\eta_l}^0$.

Putting `bounded` and `decrease` together, we have

$$\texttt{terminate}(P) \doteq \bigwedge_{l \in \texttt{loops}(P)} \texttt{decrease}(l) \wedge \texttt{bounded}(l)$$

where $\texttt{loops}(P)$ is the set of all loops in P.

EXAMPLE 2. *Consider the loop, let us call it l_1, on Line 13 in Figure 2. The loop guard is ρ_1, and we assume an unknown ranking function η_{l_1}. For clarity we show the simplified version of* `decrease` *(assuming a loop invariant is not required) as above. Then the termination constraint for the loop is:*

$$\begin{array}{lr} \rho_1^0 \wedge r'^1 = \varepsilon_3^0 \wedge \neg\rho_2^{V_1} \wedge m'^1 = \varepsilon_7^{V_1} \Rightarrow ({\eta_{l_1}}^{V_2} < {\eta_{l_1}}^0) & ..\texttt{decrease}(l_1) \\ \rho_1^0 \Rightarrow ({\eta_{l_1}}^0 \geq 0) & ..\texttt{bounded}(l_1) \end{array}$$

where $V_1 = \{r' \mapsto 1\} \cup 0$ and $V_2 = \{m' \mapsto 1\} \cup V_1$. We will show later that the valid inverse to this program instantiates the unknowns as $\{\rho_1 \mapsto (m' < m), \rho_2 \mapsto (r' > 0), \varepsilon_3 \mapsto (\texttt{sel}(N, m')), \varepsilon_7 \mapsto (m' + 1), \eta_l \mapsto (m - m' - 1)\}$ in which case each conjunct reduces as follows. `bounded`*(l) reduces to the following trivial constraint:*

$$(m' < m)^0 \Rightarrow ((m - m' - 1)^0 \geq 0)$$

Also, $\texttt{decrease}(l)$ *reduces to*

$$\begin{array}{l} (m' < m)^0 \wedge r'^1 = (sel(N, m'))^0 \wedge \neg(r' > 0)^{V_1} \wedge m'^1 = (m' + 1)^{V_1} \\ \Rightarrow ((m - m' - 1)^{V_2} < (m - m' - 1)^0) \end{array}$$

which if we keep the relevant conjuncts simplifies to $m'^0 < m^0 \wedge m'^1 = m'^0 + 1 \Rightarrow (m^0 - m'^1 < m^0 - m'^0)$, which holds.

One issue we have not yet discussed is how to determine Π_r, the expressions η_l may range over. We could ask the user to supply Π_r as part of the synthesis template, but we have found that we can derive Π_r as follows: For each inequality in Π_p, we convert it into an equivalent relation of the form $e \geq 0$ using simple symbolic manipulation, and add e to Π_r. For example, if $(n > s) \in \Pi_p$, we convert it to $n - s - 1 \geq 0$ and thus add $n - s - 1$ to Π_r. We found this simple approach to be effective in practice.

Solving for and enumerating m solutions As we can see from the definitions of `safepath` and `terminate`, the constraint C maintained by `PINS` has the form $\forall X.C'$, where X is the set of program variables (note that we have lifted the quantification over X to the top level). We wish to solve for the unknown predicates and expressions, and thus the `solve` procedure tries to find m solutions for a constraint $\exists \rho_i \varepsilon_j \eta_l \rho_l . \forall X.C'$ where the unknown predicates, expressions, ranking functions, and dynamic invariants can range over Π_p, Π_e, Π_r, and Π_p, respectively.

Our implementation of `solve` builds upon our earlier work, which uses *constraint-based invariant inference* [36, 38, 17, 18]. The invariant inference problem is to solve a constraint of the form $\exists I_k \forall X.vc$, where the I_k are unknowns in a template assumed for the invariant, X is the set of program variables, and vc is a verification condition. Notice that this constraint is similar in shape to the constraints generated by `PINS`, and thus we can adapt the solving strategy from this prior work to solve our constraints.

Very briefly, constraint-based invariant inference works by calling an SMT solver a polynomial number of times to extract information from each constraint in vc. It then uses that information to construct a SAT formula that contains boolean indicator variables, whose solution maps back to an assignment of each I_k to one of the possible predicates it may range over [36].

During invariant inference using a program's vc (which is a conjunction of implications, each constructed from some fragment of the program), the unknowns I_k in the invariant templates appear at most twice in each vc implication: at the beginning of the loop, and at the end. However, the constraints generated in `PINS` can include unknown predicates and expressions that are paired with many different version maps. We therefore needed to extend the earlier work to handle version maps. It turns out that the core theory and algorithm we previously developed [36] holds under multiple versions, and so we only need to make the following implementation changes to build `solve`: (a) we added version maps to track states of variables, (b) we incorporated a stack depth parameter so we can distinguish variable versions across recursive calls, and (c) we replaced the verification condition generator with the symbolic executor from Figure 3.

One nice property of using an SMT/SAT-based reduction for `solve` is that we can easily ask for m different solutions from the solver: Given one solution S for C, we can then pass the constraint $C \wedge \bigvee(\alpha_i \neq S(\alpha_i))$ to the solver to get a different solution, where $\alpha_i \in dom(S)$. As we discuss next, we use this enumeration of possible solutions to determine when to halt iteration of `PINS`.

Stabilization of solutions `PINS` stops iteration and returns the current set of solutions when that set is the same as in the last iteration and its size is less than m (Line 8). We actually perform this check by comparing the sizes of the current and last value of *sols*—since the set of constraints C is only added to, if the size of *sols* is the same from one iteration to the next, the solutions themselves must also be the same.

When `PINS` exits, it is not guaranteed that the returned solutions are correct; this should be clear, because `PINS` typically only explores a subset of the possible paths through the template program. In our experiments, stabilization happened when only one to four solutions remained, and thus `PINS` winnowed down a very large space of solutions to only a small set of possibilities. Given this set, the programmer can validate the solutions using other approaches, such as as manual inspection, test case generation, or model checking. We discuss these approaches more in Section 2.5.

Picking one solution On Line 10 of the `PINS` algorithm, we pick one solution out of *sols* for subsequent symbolic execution. Ideally, we will choose an incorrect solution $S_{bad} \in sols$. Since incorrect solutions are usually incorrect on many paths, we expect that if we explore a path that is feasible in S_{bad}, the constraints generated will show that S_{bad} violates *spec*, and thus S_{bad} will be pruned from the search space (as will many more incorrect solutions, most likely). We found that a good heuristic for picking such a solution is to find one that contradicts many constraints in Φ. More precisely, we define

$$\texttt{infeasible}(S) = |\{\phi \in \Phi \mid S(\phi) \Rightarrow \texttt{false}\}|$$

and then pick a solution $S \in sols$ with the highest $\texttt{infeasible}(S)$. We break ties randomly.

To understand why this heuristic works, consider an incorrect solution $S_{bad} \in sols$. Since S_{bad} has survived previous iterations of PINS, it must agree with C, which was generated from many paths. But as the number of paths represented by C increases, the chance that S_{bad} survived because it satisfies *spec* along paths in Φ diminishes. Instead, it is much more likely that S_{bad} survived because the paths in Φ are infeasible in S_{bad}, i.e., $S_{bad}(\phi)$ is false for $\phi \in \Phi$. (Notice this makes the left-hand side of the implication in `safepath` false, and hence `safepath` is trivially satisfied.) Thus, if we pick a solution S with a high $\texttt{infeasible}(S)$, there is a good chance it is an incorrect solution.

Note that even if `pickOne` selects a solution S that is valid, it will still tend to help PINS converge: Since S is valid, it will survive the next round of iteration, but the additional constraints it imposes on the symbolically executed path should help prune out invalid solutions.

We experimentally compared using `infeasible` to implement `pickOne` versus random selection, and we found that random selection yields runtimes that are 20% longer than with `infeasible`.

Axiomatization for modular synthesis Our language for template programs includes calls of the form $\mathbf{f}(\vec{e})$, where $\mathbf{f}$ is an uninterpreted function. In PINS, these are used to model calls to external libraries. Not surprisingly, library calls are common in practice—in our experiments, 8 of 14 benchmarks use library calls.

Because PINS uses SMT/SAT solving over symbolic constraints, we can readily model the behavior of these library calls as additional axioms over the uninterpreted functions, which are passed directly through to the solver. For example, in our experiments we model strings as an abstract data type with three functions `append`, `strlen`, and `empty` that satisfy axioms such as:

$$\begin{aligned}
&\texttt{strlen}(\texttt{empty}()) = 0\\
\forall x, y.\ &\texttt{strlen}(\texttt{append}(x, y)) = \texttt{strlen}(x) + \texttt{strlen}(y)\\
\forall x, c.\ &\texttt{strlen}(\texttt{append}(x, `c')) = \texttt{strlen}(x) + 1
\end{aligned}$$

In another instance, we treat an angle as an abstract data type, with trigonometric interface functions `cos` and `sin`. We also use such abstract reasoning for operations that are difficult for SMT solvers, e.g, we can add an axiom $\forall x \neq 0.\texttt{mul}(x, \texttt{div}(1, x)) = 1$ to allow enhanced reasoning about multiplication and division.

Overall, we found the ability to axiomitize library calls to be extremely useful, and as a general technique it helps split the synthesis problem into more modular pieces.

2.4 Discussion

Now that we have presented the PINS algorithm, we can revisit the *small path-bound hypothesis* described in the introduction: That for many programs, all behavior can be summarized by examining a small, carefully chosen set of paths.

PINS is designed around this hypothesis. In each iteration, it explores one path, and then uses the constraints $\texttt{safepath}(\phi, V', spec)$ from that exploration to further restrict the solution space. Specifically, PINS finds a solution S to constraints C and uses S to guide symbolic execution on the next iteration. As discussed earlier, if S is incorrect, it will likely be eliminated, and if S is correct, it will be reinforced and will likely eliminate other, incorrect solutions.

Contrast this with, for example, random path exploration. Suppose a template program contains two nested loops in sequence, as does the run-length example in Figure 2. For this example, a single element array results in the identity compression. For a two element array, the elements can be identical or different. Interesting compression happens only in an array of length three or more. So, we would want to explore the program's behavior on such nontrivial cases. But even if we only consider paths that traverse each loop at most three times, there are still 7,225 unique paths. We tried random path exploration, but we found it did not work even for the simplest examples.

The path-bound hypothesis underlies the idea of software testing as well, and we think it holds for synthesis for the same reasons: a small set of carefully chosen paths can cover both the main behavior and the corner cases of a program because that is typically how programs and algorithms are designed. While the set is small, it needs to be carefully constructed. In the case of software testing, these paths are either provided by a human, or possibly through symbolic execution. For synthesis, we propose PINS's mechanism of directing path exploration using candidate solutions.

2.5 Validating solutions

When PINS terminates, it outputs a set of solutions that satisfy all of the program paths explored during iteration. As discussed earlier, this does not guarantee that the synthesized output is correct over all possible paths, and so to gain further confidence in the output solutions, developers can take several additional steps.

First, developers can manually inspect the solutions for correctness. In our experience, this was fairly easy. For 11 out of 14 of our benchmarks, there was a single solution to inspect, and in all 11 cases it was correct. For the other three benchmarks, there were at most 4 remaining solutions. In one benchmark, we found that the remaining two solutions were indeed both valid, while in the other two benchmarks, we found that only one solution was valid. In each of the three cases, the solutions differed in at most one assignment, so it was no harder to understand the set of solutions than a single solution. We believe that PINS helps the programmer because it can be easier to check a program for correctness than create it.

Second, developers can examine the set of paths explored by PINS. For each path condition ϕ, our implementation uses the SMT solver to output a concrete input that will take that path (by solving $(\exists X.\phi)$ restricted to input variables at version 0). These inputs are concrete test cases that necessarily meet the specification, and we found them very helpful in understanding the generated solution. More particularly, the concrete tests helped us intuitively understand the solution's behavior, without needing to trace through long symbolic paths.

Finally, the programmer can use formal verification techniques to validate the solution. For example, in our experiments we tried using the bounded model checker CBMC [1] to check the synthesized inverses, and we succeeded in doing so for 6 of the 14 benchmarks (more details in Section 4). One limitation of CBMC is that it cannot incorporate new axioms (which we use to model library functions); other formal reasoning techniques that can support axioms may be able to verify the remaining cases.

3. Semi-automated template mining

In this section, we discuss how we construct a synthesis template (P, Π_e, Π_p) for a program inverse. Our approach is inspired by an insight from Dijkstra, who observed that in some cases, a program can be inverted by reversing its control flow edges and assignment statements [8]. Based on this observation, we help the programmer derive the template components from the text of the program to be inverted. We should emphasize that our approach is meant to assist the programmer in constructing a template, but it is still up to a human to identify the final template to use.

We begin by automatically *mining* candidate sets for Π_p and Π_e, in three steps. First, we traverse the original program text and return all expressions e that appear in assignments $x := e$, and all predicates p that appear in assumptions $\texttt{assume}(p)$. (Recall this is the only place predicates can appear, since we have transformed conditionals and while loops to use non-deterministic choice and `assume`.) We also record which variables are used with `in` and `out`. Next, to these sets we apply *projections* that, given an input expression or predicate, return a set of candidate expressions and predicates. For inversion some of the projections we use are the identity $\lambda x.\{x\}$; subtraction inversion $\lambda(e_1 - e_2).\{e_1 + e_2\}$; addition inversion $\lambda(e_1 + e_2).\{e_1 - e_2\}$; copy inversion $\lambda\texttt{upd}(A, i, \texttt{sel}(B, j)).\{\texttt{upd}(B, j, sel(A, i))\}$; and array read $\lambda(\texttt{sel}(A, i)\ op_r\ X).\{\texttt{sel}(A, i)\}$. We also have a projection that uses the `out` call over `int`s to construct a predicate as $\lambda\texttt{out}(m).\{m' < m\}$, for integer m. In essence, these projections capture specific domain knowledge—in this case, that program inversion often requires inverting operations. In total, we use eight projections, including the ones above, for inversion, and we apply all projections to all possible inputs. For example, since identity is one of our projections, all of the expressions and predicates in the original program are included in the output of the projection phase. Finally, we rename the variables after projection to fresh names.

We leave it up to the programmer to choose the structure of the template program P. For inversion, we found that a good starting place is to make a template program with the same control flow structure the original program text, but replacing guards with unknowns. For each assignment statement, we either simply replace its right-hand side with an unknown, or we opt to invert it, replacing an assignment $x := e$ with a parallel assignment of unknowns to all program variables in e. We also decide whether to keep sequences as-is or reverse them. In general, we found it was not hard to use this heuristic to come up with the inversion template. For instance, the inverse template in our running examples (Lines 12–18 in Figure 2) corresponds to the original program, with the control flow as-is in the outer loop and reversed in the inner loop, intuitively because the inverse needs to traverse the array in order, but then undo the run-length encoding (flipping the direction of the inner loop).

The output of the mining procedure is candidates for Π_p and Π_e. In practice, the automatically mined sets are typically too large for synthesis to succeed if we use them directly, as the space of candidate programs is exponentially related to the number of unknowns and the number of predicate and expression options. However, they give the programmer an excellent starting place for developing the final candidate sets for the template. Making some guesses, we pick a subset of the mined sets and a template program, and then attempt synthesis. If `PINS` succeeds, we are done. If `PINS` times out, we choose a smaller or different subset. If `PINS` eliminates all solutions, then we examine the paths explored by `PINS` (see Section 2.5), which typically provide enough information to determine how to change Π_p and Π_e—either by modifying some element in the chosen subset, or by adding some new predicates and expressions that were not mined. In rare cases, these paths also indicate missing assignments, which leads us to add those to the program template. In our experiments, we found that once we identified the correct subset of Π_p and Π_e, we only needed to manually modify at most a few predicates and expressions, which we easily inferred from the paths explored by `PINS`.

Debugging templates using `PINS` The `PINS` approach is also a significant step forward from previous template-based approaches, e.g., Sketch and proof-theoretic synthesis [38], in providing user guidance when synthesis fails. Both these previous systems simply fail with UNSAT when the template is not expressive enough, with little further assistance. In contrast, if PINS terminates without finding an inverse, the paths explored by PINS provide a witness to the non-invertibility using the template—there is no instantiation of the template that makes all of those paths valid inverses. In our experience, by inspecting those paths we can understand why PINS failed, and distinguish cases in which the template needs to be extended from cases in which inversion is impossible. In fact, our initial implementation of LZ77 (see Section 4 and Appendix A) had an off-by-one bug in it, and inspection of the paths generated by PINS led us to find the bug.

Inverting a run-length encoder To give more insight into the process of inverting a program with `PINS`, we describe how to invert the running example from Figure 2. Running the first step of template mining yields the following predicates and expressions:

$$\left\{\begin{array}{c} 0, 1, m+1, r+1, i+1, upd(A, m, sel(A, i)), upd(N, m, r) \\ sel(A, i) = sel(A, i+1), n \geq 0, i+1 < n, i < n \end{array}\right\}$$

Then applying the projections and renaming variables yields:

$$\left\{\begin{array}{c} 0, 1, m'+1, m'-1, r'+1, r'-1, i'+1, i'-1, \\ upd(A', m', sel(A', i')), upd(A', i', sel(A', m')), \\ upd(N, m', r'), sel(N, m') \\ sel(A', i') = sel(A', i'+1), sel(A', i'), \\ m' < m, r' > 0 \end{array}\right\}$$

Notice that since n does not have a corresponding variable in the decoder, all expressions and predicates referring to it are automatically deleted. The last two predicates come from an inversion projector that scans loop iterators and `out` statements. Now we want to remove elements of this set that are unneeded, since they will slow down the synthesis process. We can see right away that we can remove $upd(N, m', r')$, because the decoder will have no need to modify the array of counts N, and $sel(A', i')$, since A' holds the compressed data, which the decoder should only write to and not read from.

Next we select a template program P. Since we expect the decoder to scan its input starting at the beginning, we choose an outer loop that has assignments to the same variables in the same order. Since we expect the decoder to be reversing the compression process for each element, we choose an inner loop where the body is reversed, i.e., for the variables read from A, r, i, and n (A and n are read in the loop guard) their renamed versions A', r' and i' are written to. We then try running `PINS`, and discover that constraint solving has bogged down. With a little more experimentation, we remove the assignment to i', A' and N that corresponds to assignments between lines 8–10 in Figure 2, yielding the template program at the bottom of that figure.

We run `PINS` again, and this time it terminates but claims no solution exists. It provides the three paths that were used to eliminate all solutions. We examine the paths, and notice that the third one enters the inner loop of the template program and assigns to A'. We examine of current Π_e, and we notice that it contains no expressions that read from A, the array containing the compressed data—thus clearly there is a problem, since A' should contain data expanded from A.

As a fix, we opt to change occurrences of $sel(A', x)$ to $sel(A, x)$ within the upd expressions in Π_e, yielding a $\Pi_p \cup \Pi_e$ of

$$\left\{\begin{array}{c} 0, 1, m'+1, m'-1, r'+1, r'-1, i'+1, i'-1, \\ upd(A', m', sel(A, i')), upd(A', i', sel(A, m')), \\ sel(N, m'), sel(A', i') = sel(A', i'+1), m' < m, r' > 0 \end{array}\right\}$$

Using the current template, `PINS` then takes 7 iterations and 36 seconds total to prune the search space down to one candidate:

```
$i', m' := 0, 0;$
while ($m' < m$)
    $r' :=$ sel$(N, m')$;
    while ($r' > 0$)
        $r', i', A' := r' - 1, i' + 1,$ upd$(A', i',$ sel$(A, m'))$;
    $m' := m' + 1$;
```

This is a dramatic reduction in the search space; for this particular example, there were $11^7 \times (2^3)^2 \approx 2^{30}$ possible inverses given the synthesis template. (Note that each unknown predicate can be instantiated with a subset, denoting conjunction, from Π_p.)

We then manually inspected the inverse to ensure it was correct, and we also verified it using CBMC with bounds of 10 loop unrollings and at an array A of length at most 4 ($n \leq 4$). We also attempted to synthesize the inverse using the Sketch tool [33]. We rewrote our template as a sketch (it takes 53 lines of code in that format), and Sketch was able to synthesize the solution in 156 seconds, using the same bounds of 10 unrollings and $n \leq 4$. (See Section 4.3 for more discussion.)

4. Experiments

We implemented a symbolic executor based directly on rules in Figure 3, and used it to implement PINS.[1] As discussed earlier, our implementation of `solve` is an extension of our prior work [36, 37]. We use Z3 [7] as an SMT solver.

Recall that PINS is parameterized by the number of solutions m to request from the solver. In our experiments, we chose $m = 10$, which we found worked well—it provided enough solutions so that our `pickOne` heuristic could work effectively, while not requiring too much solver time.

Benchmarks We used PINS to synthesize inverses for the 14 programs listed in the leftmost column of Table 1. All of these programs are small, but they are also complex. PINS succeeded in inverting all of these programs, and we should emphasize that, to our knowledge, no other automated technique is able to do so given the same information.

The first group of programs is compressors. We have already discussed the run-length encoder, and we include two variants, one that compresses and decompresses in place, as in Figure 1, and one that uses a separate array for the compressed data. We also invert two well-known compression algorithms, Lempel-Ziv 77 (*LZ77*) and Lempel-Ziv-Welch (*LZW*) [42, 40]. We coded the compressors in C from a description of the algorithm [2].

The second group of programs convert among different formats. The *Base64* program converts binary input to printable ASCII characters. *UUEncode* outputs four printable characters for every three bytes of binary input, plus adds a header and a footer to the output. Again, we coded the compressors in C from their standard descriptions. *Pkt wrapper* wraps a data object (with a set of fields inside) into a variable length packet format by traversing the fields and adding a preamble (the length of the field) to the data bytes for the field. *Serialize* is a toy data structure serialization program that recursively walks over data objects (recursing into any non-primitive types) and writes out a flattened representation. Our serializer is small because it relies on external functions to check whether a field is primitive, get the next field, etc.. We encode the behavior of these external functions using axioms.

The last group of programs perform arithmetic computations. $\sum i$ is a simple iterative computation that adds i to a running sum in the ith iteration. (Our inverse works by iteratively subtracting i from the sum, rather than trying to solve the quadratic $n(n+1)/2$.) *Vector shift*, *Vector scale*, and *Vector rotate* perform the named operation on a set of points on the Euclidean plane, represented as a pair of arrays X and Y. In practice, a programmer would realize that these computations can be reversed by negating the inputs to the program, but that is domain-specific knowledge the synthesizer does not have. Instead, the synthesizer discovers a specialized un-shifter, -scaler, and -rotater that iterate through the vectors, *semantically* negating the operation performed. This is non-trivial for scaling, where the synthesizer needs to be able to reason about $1/x$, and for rotation, where the inverse of $(x', y' := x\texttt{cos}(t) - y\texttt{sin}(t), x\texttt{sin}(t) + y\texttt{cos}(t))$ is $(x'', y'' := x'cos(t) + y'sin(t), y'cos(t) - x'\texttt{sin}(t))$.

[1] Our implementation of PINS is available on the web [3]

Benchmark	LoC	$\Pi_p \cup \Pi_e$			Inv. LoC	Num. Axms
		Mined	Subset	Mod		
In-place RL	12	16	14	1	10	0
Run length	12	16	10	0	10	0
LZ77	22	16	10	3	13	0
LZW	25	20	15	4	20	15
Base64	22	13	10	2	16	3
UUEncode	12	8	7	4	11	3
Pkt wrapper	10	12	8	1	16	2
Serialize	8	7	7	1	8	6
$\sum i$	5	8	6	2	5	0
Vector shift	8	11	7	0	7	0
Vector scale	8	9	7	2	7	1
Vector rotate	8	13	7	0	7	1
Permute count	11	12	7	2	10	0
LU decomp	11	14	9	0	12	2

Table 1. Template mining characteristics.

The *Permute count* program is *Dijkstra's permutation* program from his original note on program inversion [8]. He considered a program that, given a permutation π, computes for the ith element of π the number of elements between $0 \mathinner{.\,.} i$ that are less that $\pi(i)$. The inverse program computes the permutation from an array of these counts. Dijkstra manually derived it from the original program, while PINS synthesizes the inverse from the template.

Finally, *LU-decomposition* performs that operation in-place on a matrix using the Doolittle algorithm [30]. The inverse, which has been manually derived before [4] and which we synthesize using the synthesis template, is a program that multiplies the lower triangular and upper triangular matrices in-place.

4.1 Template construction

Table 1 summarizes the results of the template mining process on our benchmarks. For each benchmark, we first list its size in terms of lines of code, followed by three columns that measure the size of $\Pi_p \cup \Pi_e$. First, we list the size of the candidate sets mined from the original program; then, we give the size of the subset we guessed initially; and lastly, we list the number of modifications we needed to make to elements within the chosen subset for successful synthesis. The second-to-last column lists the size of the portion of the template program containing the inverse (e.g., Lines 12 and below in Figure 1). In this measure, we count loop guards as being on their own line, and we count a parallel assignment to k variables as k lines (since that is what it will ultimately be expanded to).

We found that picking the subset of $\Pi_p \cup \Pi_e$ was fairly straightforward, as we could easily eliminate obviously redundant or useless elements. Then an initial run of PINS gave us enough information to infer the few changes to $\Pi_p \cup \Pi_e$ we needed to make. Overall, the template mining process proved tremendously helpful in coming up with a synthesis template. Although it was still non-trivial to find the correct final template, we found it was much easier to tweak a candidate template than come up with one from scratch.

The rightmost column in Table 1 reports the number of axioms we used during synthesis, as discussed in Section 2.3. These axioms are quite generic over the uninterpreted functions we were abstracting, e.g., the axioms over strings from Section 2.3, and can be reused across many different program synthesis problems that use the same ADTs. In the current implementation, we select axioms for each benchmark because of current limitation of SMT solvers—specifically, quantifier instantiation is expensive, and so we limit how often it occurs by limiting the number of axioms. However, SMT solvers are evolving rapidly, and we expect in the future to write standard libraries of axioms once and use the same libraries across many different synthesis problems.

Benchmark	Srch. Sp. Redn	Num. Iter.	Total Time (s)	|SAT|
In-place RL	$2^{30} \rightarrow 1$	7	36.16	837
Run length	$2^{25} \rightarrow 1$	7	26.19	668
LZ77	$2^{25} \rightarrow 2$	6	1810.31	330
LZW	$2^{31} \rightarrow 2$	4	150.42	373
Base64	$2^{37} \rightarrow 4$	12	1376.82	598
UUEncode	$2^{20} \rightarrow 1$	7	34.00	177
Pkt wrapper	$2^{20} \rightarrow 1$	6	132.32	2161
Serialize	$2^{11} \rightarrow 1$	14	55.33	69
$\sum i$	$2^{15} \rightarrow 1$	4	1.07	51
Vector shift	$2^{16} \rightarrow 1$	3	4.20	187
Vector scale	$2^{16} \rightarrow 1$	3	4.41	191
Vector rotate	$2^{16} \rightarrow 1$	3	39.51	327
Permute count	$2^{3} \rightarrow 1$	1	8.44	4
LU decomp	$2^{5} \rightarrow 1$	1	160.24	10

Table 2. Performance of `PINS`

Benchmark	Validation			Sketch
	Manual	Tests	CBMC	
In-place RL	ok	2	34.59s	157s
Run length	ok	2	0.62s	30s
LZ77	1 of 2 ok	5	1.93s	29s
LZW	2 of 2 ok	3	—	—
Base64	1 of 4 ok	4	—	—
UUEncode	ok	6	—	—
Pkt wrapper	ok	1	—	—
Serialize	ok	5	—	—
$\sum i$	ok	2	1.15s	fail
Vector shift	ok	1	113.74s	2s
Vector scale	ok	1	—	—
Vector rotate	ok	1	—	—
Permute count	ok	1	1.06s	172s
LU decomp	ok	1	—	—

Table 3. Validating the solutions generated by `PINS`.

4.2 Performance

Table 2 shows the performance for `PINS` on our benchmarks. The second column reports the approximate size of the search space and the number of solutions returned when `PINS` terminates. For example, for the in-place run-length encoder, the synthesis template has roughly 2^{30} possible instantiations, and `PINS` eliminates all but one of them. `PINS` returned one solution for 11 of the 14 benchmarks, and only a few solutions in other cases. Thus, we can see that `PINS` is highly effective in refining the set of possible inverses. For all programs `PINS` found at least one correct solution; we defer a more detailed discussion of correctness to Section 4.3.

The next column in Table 2 reports the number of full loop iterations until `PINS` converges. Since `PINS` calls `solve` at the top of the loop before deciding whether to exit, i full loop iterations corresponds to $i+1$ solver calls (see Algorithm 1). On each full iteration, we explore one program path, so these numbers support our path-bound hypothesis: for these programs, indeed only a small number of paths were required to characterize the programs' behavior.

The third column in the table lists the running times for `PINS`. We can see that the time varies widely, from one second for $\sum i$ up to 30 minutes for *LZ77*. Even so, these times may be shorter than the time required for a programmer to manually write the inverses. The variability of the times arises from the large semantic differences between these programs and the vagaries of SMT/SAT solvers. We also separately measured how much of the running time was due to each of the various steps within `PINS`, and we found that symbolic execution (which makes SMT queries) and SMT reduction to SAT take more than 90% of the running time. The actual SAT formulas produced are quite small, as shown in the last column of the table, and solving those and computing our `pickOne` heuristic take little of the running time. (The detailed distribution for individual benchmarks is available in Appendix B.)

4.3 Validation

After `PINS` terminates, we need to validate the inverses, since `PINS` does not guarantee their correctness. Table 3 shows the results of validation following the methodology outlined in Section 2.5. The second column counts how many solutions were correct according to manual inspection. We can see that most of the solutions returned by `PINS` were in fact correct. For the three benchmarks that yielded multiple solutions, *LZ77*, *LZW*, and *Base64*, the solutions were very similar, differing by a single assignment in most cases. Thus, after inspecting one solution, it was easy to understand the others.

When `PINS` exits, it also outputs concrete test cases, which are concrete assignments to the inputs that will cause the final solutions to take the paths `PINS` explored. The third column of the table lists the number of such test cases; note that it might be smaller than the number of iterations, since some of the previously explored paths may be infeasible in the final solutions. We found these tests helpful when doing our manual inspection.

The fourth column reports the time for running the bounded model checker CMBC [1] to verify the final solutions. Note that to run CBMC, we needed to bound the number of loop unrollings and, in many cases, the sizes of arrays. Using CMBC gives us further confidence that the synthesized programs are correct. However, most of the benchmarks needed axioms for library functions, and we found no easy way to apply CBMC to those programs—we would instead need to write implementations of the library functions. For some axioms, such as $\forall x \neq 0.\texttt{mul}(x, \texttt{div}(1, x)) = 1$, even an implementation would be insufficient, as this particular axiom essentially adds a capability to the solver.

Sketch For comparison, we tried running Sketch [33] on the same synthesis templates we used in our experiments. As with CBMC, Sketch does not have a way to include axioms to model library functions, and so we could only run Sketch on 6 of our benchmarks; we felt that writing the library functions would significantly change the input to the synthesizer, and thus would not be a meaningful comparison. For instance, it dramatically changes the synthesis problem to have an implementation for $\texttt{inv}(x) = 1/x$ as opposed to the axiom $\forall x \neq 0.\texttt{mul}(x, \texttt{div}(1, x)) = 1$, or to have implementations for $\texttt{cos}(t)$ and $\texttt{sin}(t)$ as opposed to the axiom $\forall t.\texttt{cos}^2(t) + \texttt{sin}^2(t) = 1$. Sketch also requires that the user specify bounds on array sizes.

Sketch was able to synthesize an inverse for 5 out of the 6 benchmarks that did not make external calls; the running times are reported in the rightmost column of Table 3. Sketch only failed on one benchmark—we let it run for an hour, and even after we eliminated all unknown predicates from the template, Sketch still did not terminate. In this case, the program forces Sketch to unroll a loop maxint times, which by default is 32 in Sketch (which sets integers to be 5 bits), and this large number of unrollings seems to cause the timeout. It may be possible to synthesize this example by reducing the number of bits in integers below 5.

For LZ77, Sketch resolves the template an order of magnitude faster than `PINS`, but note that this is only after we spent considerable effort getting it to terminate, which happened only when we reduced the bound on the array size to 4.

In general, we found that using Sketch was more challenging than we expected, as it took us a significant amount of experimentation to come up with the right bounds. This bounds requirement also highlights a key difference between `PINS` and Sketch: `PINS` solves for *all possible input values* along a small set of paths, but Sketch ensures correct synthesis on *all paths within the finitized* *space*. The details of the parameters we found through experimentation are available in Appendix B.

4.4 Limitations and future work

Our results show that PINS is promising, but it still has several limitations. Scalability is clearly a challenge, both for PINS and for other program synthesis tools. It would be interesting future work to investigate when synthesizing small components of larger applications would be useful, and to improve the scalability of PINS itself. Another challenge is the need to discover templates. We think that templates are a useful tool—they give the developer a way to narrow the search space for synthesis, but, at least in our experience, they are much easier to develop than the programs to be synthesized. For inversion in particular, our template mining approach proved very helpful. It would be worthwhile to study whether similar mining approaches could be used for other domains, and user studies of synthesis could shed light on how much input a human is willing to provide to a synthesis tool. Finally, PINS relies on SMT solving, and more scalable and powerful SMT solvers would enable larger, less-constrained search spaces and faster synthesis.

An interesting direction for future work is to automatically refine templates, rather than requiring the user to do so. One logical starting place is CEGAR, as the paths explored by PINS are analogous to the counterexamples used by CEGAR (although we have observed that CEGAR's refinement cannot be used directly). Inverting many-to-one programs is another interesting question. There are two formulations of the problem we could consider. First, we could try to find an inverse that returns *some*, instead of all, elements of the original program's input. We believe it is possible to handle this formulation with PINS directly, with just a different template. Second, we could try to find an inverse that generates all of the original inputs. This is more difficult; we believe it requires that we specify a different specification than identity, but that the core of PINS should still be applicable.

5. Related Work

Deriving program inverses A range of techniques for program inversion have been previously proposed. Dijkstra [8] and Gries [12] suggest using a set of proof rules to derive an inverse. Our template mining approach is inspired by this idea, but it is hard to envision this approach succeeding in an automated way on many examples. A related proposal uses local inversion plus proof rules that compose the locally inverted fragments into a complete program [4, 9, 31]. These techniques work if all values in the reversed computation can be obtained analytically, which can be a challenge. To our knowledge, these systems have not been automated.

One very interesting inversion technique consists of approximating the behavior of the program with a grammar. If the output of the original program can be parsed using a deterministic grammar, then the inverse of the grammar (written using local inversion, i.e., reading the grammar backwards) corresponds to the program inverse [11, 24, 41]. Grammar-based inversion only works when the input to the program to be inverted has the right (context-free) form—grammar-based approaches can invert run-length encoding, but not the dictionary-constructing compressors LZW and LZ77. Also, grammar-based inversion has been applied to functional programs, where there are no destructive state updates and thus inversion is arguably simpler. Additionally, it is also not clear to us if grammar-based inversion works on any of the arithmetic examples.

Program inverses for restricted cases have been considered in the context of some larger problems. As part of an approach to generate divide-and-conquer parallel functional programs, Morita et al show how to compute a weak right inverse to a program fragment [29]. It is unclear whether this approach can handle loops. Kanade et al propose a simple inversion subroutine to aid representation dependence testing [23]. This latter approach is based on proof rules, and seems hard to use in the general case.

Inductive and deductive program synthesis Inductive synthesis generalizes from finite instances to yield an infinite state program [21]. Deductive synthesis, in contrast, refines a specification to derive the program [28, 32]. Our approach is mostly inductive synthesis since we generalize from a finite set, but it has elements of deductive synthesis because we use paths and symbolic reasoning. Since we use symbolic paths and not concrete traces, each additional path captures the behavior of multiple concrete runs, and more of the space is explored in each iteration. Additionally, while inductive synthesis cannot provide the full formal guarantees of deductive synthesis, symbolic paths provide a close approximation. Lastly, while previous inductive synthesizers refine using either only positive reinforcing examples or only negative counterexamples, we refine using both positive and negative example paths.

Proof-theoretic synthesis [38] and similar approaches for hybrid systems [39] are deductive synthesis techniques, which encode the synthesis problem as a search for invariants, and therefore needs to infer complicated invariants (and requires a formal verifier with support for such reasoning). In contrast, PINS relies on directed path exploration and symbolic execution-based reasoning, and does not reason about invariants. Thus, PINS allows us to synthesize inverses that are infeasible using proof-theoretic synthesis.

As discussed earlier, Sketching [33] is an inductive synthesis technique that uses a model checker to refine the space of candidates. PINS differs from Sketch in several ways. First, Sketch uses domain specific reductions to finitize loops for stencil [33], concurrent [34], and bit-streaming [35] programs, and is engineered to solve the resulting loop-finitized problem. In contrast, PINS finitizes the solution space using templates, but does not finitize loops or the input. We found that this is an important consideration, because in our experiments it was tricky to choose the right finitization for Sketch; smaller sizes lead to faster solving, but may restrict the space such that the synthesized candidates are correct only on the bounded values but not for arbitrary inputs. It took us several hours to get the finitization right for our experiments. Second, PINS prunes using symbolic paths, while Sketch prunes using concrete executions; since multiple concrete executions may follow a single path, a few iterations of PINS suffice for the synthesis. Lastly, PINS uses SMT reasoning to generate concise SAT instances that are easily solved; Sketch uses bit-blasting, which generates large formulas that may be hard to solve [15].

Some previous synthesis techniques can successful synthesize acyclic programs. Gulwani et al have developed SMT-based techniques for synthesizing acyclic programs over a user-specified set of components, which they apply to the synthesis of bit-vector programs [15, 22]. Gulwani et al have also shown that exhaustive search pruned by heuristics, inspired by techniques from the artifical intelligence community, can synthesize ruler and compass-based macros for geometrical drawings [16]. Kuncak et al have developed decision procedures for synthesis of functions specified within a decidable logic [26]. While these techniques do not rely on templates, they are restricted to loop-free programs, in contrast to PINS, which synthesizes programs with loops.

Gulwani et al have developed synthesis techniques based on divide-and-conquer paradigm that can synthesize loopy programs for string [14] and table manipulation [20]. However, these techniques are based primarily on inductive synthesis, and are well-suited for *end-users*, who find it easy to provide examples, but would find the task of providing a formal specification quite daunting. In contrast, the technique presented in this paper can leverage formal logical specifications.

6. Conclusion

We have presented a program synthesis approach called PINS that synthesizes programs by exploring relevant paths in a template program and ensuring that the program meets the specification over those paths. PINS leverages symbolic execution and SMT solving to synthesize programs that take unbounded inputs and may have unbounded loops. An important consideration is finding the right set of paths to efficiently prune the space of possible candidates, for which PINS includes a directed path exploration strategy that is parameterized by a remaining candidate solution. We apply PINS to the task of semi-automated program inversion. We show that it is possible to mine the synthesis template using domain-specific projection operators, which significantly reduces the burden on the user. Our results on inverting 14 small benchmarks suggest that PINS is a promising new approach to inversion in particular, and, we believe, to program synthesis in general.

Acknowledgments

This work was supported in part by NSF grant 1019343 to the Computing Research Association for the CIFellows Project, and by NSF grants CCF-0346982 and CCF-0953507.

References

[1] CBMC. `http://www.cprover.org/cbmc/`.

[2] LZW and LZ77. `http://en.wikipedia.org/wiki/Lempel-Ziv-Welch` and `http://en.wikipedia.org/wiki/LZ77_and_LZ78`.

[3] PINS. `http://www.cs.umd.edu/~saurabhs/vs3/PINS/`.

[4] Wei Chen. A formal approach to program inversion. In *CSC: Proc. of the ACM conference on Cooperation*, pages 398–403, 1990.

[5] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *CAV'08*.

[6] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3, 2008. `http://research.microsoft.com/projects/Z3/`.

[8] Edsger W. Dijkstra. Program inversion. In *Program Construction,* `http://www.cs.utexas.edu/~EWD/ewd06xx/EWD671.PDF`, pages 54–57, London, UK, 1979. Springer-Verlag.

[9] David Eppstein. A heuristic approach to program inversion. In *IJCAI*, pages 219–221, 1985.

[10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[11] Robert Glück and Masahiko Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inf.*, 66(4):367–395, 2005.

[12] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., 1987.

[13] Sumit Gulwani. Dimensions in program synthesis (invited talk paper). In *ACM Symposium on PPDP*, 2010.

[14] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[15] Sumit Gulwani, Susmit Kumar Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[16] Sumit Gulwani, Vijay Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[17] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[18] Sumit Gulwani and Ashish Tiwari. Constraint-based approach for analysis of hybrid systems. In *CAV*, pages 190–203, 2008.

[19] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI'10*, pages 292–304, 2010.

[20] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[21] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[22] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[23] Aditya Kanade, Rajeev Alur, Sriram Rajamani, and G Ramalingam. Representation dependence testing using program inversion. In *FSE*, 2010.

[24] Masahiko Kawabe and Robert Glück. The program inverter lrinv and its structure. In *PADL*, pages 219–234, 2005.

[25] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

[27] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1), 1980.

[28] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[29] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *PLDI'07*.

[30] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *LU Decomposition and Its Applications*, chapter 2.3, pages 34–42. 1993.

[31] Brian J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Asp. Comput.*, 9(3):331–348, 1997.

[32] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16:1024–1043, 1990.

[33] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[34] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[35] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Prog. by sketching for bit-stream. prgs. In *PLDI*, pages 281–294, 2005.

[36] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[37] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS3: SMT solvers for program verification. In *CAV*, 2009.

[38] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[39] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, pages 305–319, 2009.

[40] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[41] Daniel M. Yellin. *Attribute grammar inversion and source-to-source translation.* Springer-Verlag New York, Inc., 1988.

[42] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(5):337–343, 1977.

A. Code examples: LZ77 and LZW

To illustrate concretely the difficulty in inverting these benchmarks, Figure 4 shows the code for LZ77 and LZW compressors that we invert. This code and the code for the other benchmarks is available

```
void main(int $*A$, int $n$) {
    int $*P, *N, *C$;
    int $i, j, k, c, p, r$;
    in($A, n$);
    assume($n \geq 0$);
    $i$ := 0; $k$ := 0;
    while ($i < n$) {
        $c$ := 0; $p$ := 0; $j$ := 0;
        while (j < i) {
            $r$ := 0;
            while ($i + r < n - 1 \wedge A[j + r] = A[i + r]$)
                $r$ := $r + 1$;
            if (c < r)
                $c$ := $r$; $p$ := $i - j$;
            $j$ := $j + 1$;
        }
        $P[k]$ := $p$; $N[k]$ := $c$; $C[k]$ := $A[i + c]$;
        $i$ := $i + 1 + c$; $k$ := $k + 1$;
    }
    out($P, N, C, k$);
}
```

(a)

```
void main(int $n$, BitString $A$) {
    BitString $*D$;
    int $*B, i, p, k, j, r, size, x, go$;
    in($A$,$n$);
    assume($n \geq 1$);
    $D[0]$ = "0"; $D[1]$ = "1";
    $i$ := 0; $p$ := 2; $k$ := 0;
    while ($i < n$) {
      $j$ := $i$; $r$ := 0; $size$ := $-1$;
      while ($j < n \wedge r \neq -1$) {
        $x$ := 0; $r$ := $-1$;
        while ($x < p$) {
          if ($D[x]$ =substr($A, i, j$))
            $r$ := $x$;
          $x$ := $x + 1$;
        }
        if ($r \neq -1$)
          { $go$ := $r$; $size$ := $j - i + 1$; }
        $j$ := $j + 1$;
      }
      $B[k]$ := $go$; $k$ := $k + 1$;
      $D[p]$ := substr($A, i, j - 1$); $p$ := $p + 1$;
      $i$ := $i + size$;
    }
    out($B, D, k$);
}
```

(b)

Figure 4. The compressors for (a) LZ77, and (b) LZW, inverted by PINS.

	Percentage of total time				
Benchmark	Sym. Exe.	SMT Red.	SAT Sol.	pickOne	Total Time (s)
In-place RL	41%	51%	6%	2%	36.16
Run length	45%	45%	7%	3%	26.19
LZ77	98%	1%	<0.1%	<0.1%	1810.31
LZW	68%	29%	<1%	3%	150.42
Base64	42%	57%	<1%	<1%	1376.82
UUEncode	84%	12%	1%	3%	34.00
Pkt wrapper	1%	96%	3%	<1%	132.32
Serialize	92%	7%	<1%	<1%	55.33
$\sum i$	50%	38%	4%	8%	1.07
Vector shift	21%	73%	2%	4%	4.20
Vector scale	21%	73%	2%	4%	4.41
Vector rotate	6%	93%	<1%	<1%	39.51
Permute count	96%	2%	<1%	2%	8.44
LU decomp	88%	11%	<0.1%	1%	160.24

Table 4. Breakdown of PINS running time.

on the web [3]. Note that for both programs, none of the dictionary construction has been abstracted away. The only abstraction used is in LZW to model strings, e.g., the call to substr extracts a substring between two indices from the given string. (Also note that we assume all pointers point to valid memory.)

B. Experimental parameters and details

This appendix contains some additional details on the reported experimental results.

Table 4 breaks down the time taken by PINS into the four steps of the algorithm that involve constraint solving. The second column gives the time spent in the symbolic executor (which needs to perform SMT queries to determine which branches are feasible; see ASSUME in Figure 3). The third column gives the time spent in SMT reduction in solve, which transforms the constraints into a SAT formula, using SMT queries in the process. The fourth column gives the time spent in SAT solving, and the fifth column gives the time spent in pickOne. Clearly the vast majority of the time is spent in the first two steps.

	CBMC		Sketch		
Benchmark	Unroll	Size	Unroll	Size	|SAT|
In-place RL	10	4	10	5	2,183k
Run length	10	∞	10	8	718k
LZ77	5	4	8	4	12,456k
$\sum i$	10	∞	fail	fail	fail
Vector shift	5	∞	10	5	35k
Permute count	10	5	8	8	1.76k

Table 5. Parameters required and running times for CBMC and Sketch.

Table 5 lists the bounds we chose in running CBMC and Sketch, discussed in Section 2.5. We only list the programs that did not need axioms for library functions. For both tools we give the bound on loop unrollings and the bound on sizes of input arrays. For Sketch, we also list the average size of the SAT formula it produces (these are averages of the "find" values, as opposed to the "check" values, across the algorithm).

We arbitrarily started with 10 as the number of unrollings, and then reduced that sufficiently to ensure termination. For CBMC, in half of the benchmarks we had to specify a bound on the size of the input arrays, to get it to terminate. We experimentally found a low enough size for verification, which was 5 in one case and 4 in two others. Sketch *always* requires the input arrays be bounded. We arbitrarily tried a value of 8, which worked for two programs, but had to come down to 5 and 4 for others. Note these bounds are on the sizes of the input arrays. Sketch internally also bounds the bit-widths of all primitive types, e.g., integers are 5 bit wide. Because these systems bit-blast the formula, they are also susceptible to the types of the variables used. For instance, loops can be unrolled much more if instead of ints we use chars. PINS explores the right set of paths that allow it to effectively search as vast a space as explored through bit-blasting over a small finitized space.