Completely Automated Equivalence Proofs

QI ZHOU, Georgia Institute of Technology
DAVID HEATH, Georgia Institute of Technology
WILLIAM HARRIS, Georgia Institute of Technology

Verifying partial (i.e., termination-insensitive) equivalence of programs has significant practical applications in software development and education. Conventional equivalence verifiers typically rely on a combination of given relational summaries and suggested synchronization points; such information can be extremely difficult for programmers without a background in formal methods to provide for pairs of programs with dissimilar logic.

In this work, we propose a completely automated verifier for determining partial equivalence, named PEQUOD. PEQUOD automatically synthesizes expressive proofs of equivalence conventionally only achievable via careful, manual constructions of product programs. To do so, PEQUOD synthesizes relational proofs for selected pairs of program paths and combines the per-path relational proofs to synthesize relational program invariants. To evaluate PEQUOD, we implemented it as a tool that targets Java Virtual Machine bytecode and applied it to verify the equivalence of hundreds of pairs of solutions submitted by students for problems hosted on popular online coding platforms, most of which could not be verified by existing techniques.

ACM Reference format:
Qi Zhou, David Heath, and William Harris. 2016. Completely Automated Equivalence Proofs. 1, 1, Article 1 (January 2016), 24 pages. DOI: 10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

In many practical contexts, determining if two programs are functionally equivalent is a critical problem. Prominent instances of this problem include determining (1) if a given program written in a high-level source language is equivalent to a given (typically optimized) program that executes on a target machine architecture [28, 37], (2) if consecutive versions of a program module preserve critical program behavior, (3) if one given program is an obfuscation of the other, or (4) if a program provided by a student or hiring candidate in response to a challenge problem is equivalent to a trusted reference solution. Checking student solutions, in particular, is perhaps more critical than ever before, given increasing enrollments in computer science courses and the rapid development of online programming courses [42].

While verifying even only termination-insensitive (i.e., partial) equivalence has been the subject of a significant body of work, many previous techniques are either intended to be applied to verify equivalence of programs generated from particular transformations [28, 33, 37], or can only be applied to programs that use restricted control structures (e.g., are loop-free [24]) or data operations (e.g., only linear arithmetic on scalar data, without operations on dynamically-allocated memory [34, 47]). Other approaches only generate proofs for a bounded number of control paths [36, 38] or inputs [42].

One strategy that can potentially be followed to prove the equivalence of many programs is to reduce the problem of verifying equivalence of programs \( P_0 \) and \( P_1 \) to synthesizing a product program that soundly models all steps of \( P_0 \) and \( P_1 \), accompanied by inductive invariants of the product program that imply the equivalence of \( P_0 \) and \( P_1 \). Unfortunately, current approaches that follow such a strategy either only attempt to synthesize product programs in a class that is too...
restricted to prove equivalence of many practical programs, such as the class of sequential compositions [9, 15, 45], or require additional information about a target product program to be provided manually [6, 10, 17, 19, 44].

In this paper, we present a novel verifier for partial equivalence, named PEQUOD, which is not subject to the limitations given above. I.e., PEQUOD can be applied to pairs of programs with arbitrary control structure and that use arbitrary data operations, and can potentially synthesize proofs ranging over a class of product programs that is much more expressive than those that have been synthesized by previous automatic verifiers.

The key challenge addressed by PEQUOD is, given programs \( P_0 \) and \( P_1 \), to synthesize both a product program of \( P_0 \) and \( P_1 \) and suitable inductive invariants automatically. Previous approaches either require the structure of a product program to be provided manually, or that first attempt to guess the structure of a product program using heuristics, and then synthesize invariants for the product program by adapting techniques used by automatic verifiers of safety properties. Unfortunately, it is difficult to communicate the requirements of a product program to users without experience in program analysis (such as novice programmers). Proposed heuristics can only be applied in practice to programs have syntactic similarities that typically only hold for multiple versions of the same program. However, it is difficult to develop heuristics that can be applied to programs that have been developed by independent developers, such as a solutions submitted by independent groups of students.

PEQUOD addresses this key challenge by synthesizing both the product program and its inductive invariants simultaneously. In particular, PEQUOD selectively enumerates control paths of \( P_0 \) paired with those of \( P_1 \). For each enumerated pair of control paths \( p_0 \) and \( p_1 \), PEQUOD first determines if some runs of the paths from equivalent inputs result in non-equivalent outputs, in which case it determines that \( P_0 \) and \( P_1 \) are not equivalent. Otherwise, PEQUOD efficiently synthesizes a proof that each run of \( p_0 \) and each run of \( p_1 \) from equivalent inputs result in equivalent outputs. PEQUOD combines proofs synthesized for multiple pairs of paths, and then attempts to extract from them a product program and its inductive invariants using a novel symbolic search algorithm. An extensive body of previous work has developed automatic verifiers that synthesize inductive invariants of a single program from invariants of program paths in order to prove that a program satisfies a given safety property [5, 21, 22, 32]. The contribution of the proposed work is to adapt such a strategy to simultaneously synthesize a product program and its invariants in order to prove that given programs are equivalent.

We have implemented a prototype of PEQUOD that verifies the partial equivalence of programs given in Java Virtual Machine (JVM) bytecode and have applied PEQUOD to verify the partial equivalence of 369 pairs of solutions to challenge problems hosted on online coding platforms [13, 27]. Implementations of previous automated equivalence verifiers could verify only one of pairs of programs that we found.

The rest of this paper is organized as follows. §2 provides an informal overview of our approach, PEQUOD, by example. §3 reviews the technical foundations for our work, and §4 presents PEQUOD in detail. §5 presents an empirical evaluation of PEQUOD. §6 compares PEQUOD to related work on equivalence verification, and §7 concludes.

2 OVERVIEW

In this section, we illustrate PEQUOD by example. In §2.1, we present as a running example a pair of programs that were submitted independently as solutions to an online coding problem. In §2.2, we give a proof that the two solutions are partially equivalent, expressed as relational invariants over pairs of control locations. In §2.3, we illustrate how PEQUOD synthesizes the proof automatically.
2.1 Climbing Stairs: a coding challenge problem

Figure 1 contains the pseudocode for two solutions to the Climbing Stairs Problem hosted on the coding platform LeetCode [27]. The Climbing Stairs Problem is to take an integer \( n \) and return the number of distinct ways to climb \( n \) steps, where steps can be climbed one or two at a time. If \( n \leq 1 \), then the solution is one.

\texttt{climbStairs0} and \texttt{climbStairs1} are two correct solutions to the problem, submitted by independent programmers. \texttt{climbStairs0} first checks if its argument \( n \) is less than or equal to 1, and if so, immediately returns 1 (line 5). Otherwise, \texttt{climbStairs0} executes a loop with counter \( i \) incremented from 2 to \( n \) (lines 9—14). The loop maintains the invariant that at each step, \( \text{sum} \) stores the number of sequences in which to climb \( i \) stairs, \( \text{cur} \) stores the number of sequences in which to climb \( i - 1 \) stairs, and \( \text{prev} \) stores the number of sequences in which to climb \( i - 2 \) stairs. In each step through the loop, \texttt{climbStairs0} copies the value in \( \text{sum} \) to \( \text{cur} \) (line 11), increments the value in \( \text{sum} \) by \( \text{prev} \) (line 12), and copies the value in \( \text{cur} \) to \( \text{prev} \) (line 13). \texttt{climbStairs0} iterates until \( i \geq n \) and then returns the value stored in \( \text{cur} \) (lines 15).

\texttt{climbStairs1} is similar to \texttt{climbStairs0}, but maintains the invariant that the variable \( \text{count2} \) stores the number of sequences in which to climb \( i - 1 \) stairs and \( \text{count1} \) stores the number of sequences in which to climb \( i - 2 \) stairs. While \( \text{count1} \) and \( \text{count2} \) are used in \texttt{climbStairs1} similarly to how \( \text{cur} \) and \( \text{sum} \) are used in \texttt{climbStairs0}, they are initialized to distinct values to establish \texttt{climbStairs1}'s loop invariant (lines 7—8). Given the same input, \texttt{climbStairs1} performs one more iteration of its loop than \texttt{climbStairs0}.

2.2 Equivalence of \texttt{climbStairs0} and \texttt{climbStairs1}

\texttt{climbStairs0} and \texttt{climbStairs1}, when given equal inputs on which they terminate, exit in states with return equal values; i.e., the programs are partially equivalent.

PEQUOD, given programs \( P_0 \) and \( P_1 \) attempts to determine if they are partially equivalent by synthesizing a product program of \( P_0 \) and \( P_1 \), denoted \( P' \), accompanied by suitable inductive invariants [6–8]. A product program of \( P_0 \) and \( P_1 \) is a program in which each location is a pair of a location of \( P_0 \) and a location of \( P_1 \) and each state is a pair of a state of \( P_0 \) and a state of \( P_1 \). In each step of execution, the product program chooses a stepping component program—either \( P_0 \) or \( P_1 \) —based on its state, and then non-deterministically chooses an instruction of the chosen component program on which to step. Thus, there are potentially infinitely many product programs of fixed programs \( P_0 \) and \( P_1 \). Each product program has the same state space, but in each step, chooses the stepping program based on a different predicate on its current state.

The equivalence of \( P_0 \) and \( P_1 \) is certified by inductive invariants of \( P' \) (1) the invariant at the pair of initial locations of \( P_0 \) and \( P_1 \) is supported by the assumption that the components of state corresponding to \( P_0 \) and \( P_1 \) have equivalent
We give symbolic relations over key pairs of locations as formulas over a logical vocabulary consisting of variables that climbStairs0 will now describe a proof of equivalence of climbStairs0 for climbStairs0 are proof that climbStairs0 support the assertion that if supported by the assumption that climbStairs0, at each pair of identical line numbers, chooses to step on climbStairs0, and at all other pairs of locations, chooses to step on climbStairs1. We will now describe a proof of equivalence of climbStairs0 and climbStairs1 as inductive invariants of the fixed product program climbStairs’. However, a key feature of PEQUOD is that it does not require a fixed product program to be given manually or as the result of heuristics. Instead, PEQUOD synthesizes both a product program and its invariants simultaneously. Such a technique is essential for automatically verifying the equivalence of programs that, unlike the relatively simple examples of climbStairs0 and climbStairs1, have dissimilar control structure or data variables.

Inductive invariants of climbStairs’ can be represented as a map from pairs of control locations to symbolic relations. We give symbolic relations over key pairs of locations as formulas over a logical vocabulary consisting of variables that occur in climbStairs0 and climbStairs1, denoted with subscripts 0 and 1. In this paper, we only consider symbolic relations defined over constraints in linear arithmetic, because this is sufficient to axiomatize the semantics of the simple programs that we describe. Our implementation of PEQUOD for JVM bytecode synthesizes invariants in a more expressive logic that can describe states with dynamically-allocated objects and arrays, namely the combination of the theories of linear arithmetic and arrays.

The relational invariant over lines 2 and 2, denoted l(2, 2), establishes that the components of the state of climbStairs’ for climbStairs0 and climbStairs1 have equal arguments. I.e., l(2, 2) is

\[ n_0 = n_1 \]

The relational invariant for line 5 in climbStairs0 and 5 in climbStairs1, denoted l(5, 5), establishes that any pair of states in climbStairs0 and climbStairs1 at such locations will result in states with equivalent return values. I.e., l(5, 5) is

\[ \text{result}_0 = \text{result}_1 \]

The relational invariant for line 9 of climbStairs0 and line 9 of climbStairs1, denoted l(9, 9), establishes that for each run of climbStairs’, (1) the value of i in climbStairs0 is one greater than the value of i in climbStairs1, (2) the value of sum in climbStairs0 is equal to the value of count2 in climbStairs1, and (3) the values of n in climbStairs0 and climbStairs1 are equal. I.e., l(9, 9) is

\[ i_0 = i_1 + 1 \land \text{sum}_0 = \text{count2}_1 \land n_0 = n_1 \]

The relational invariant for line 15 of climbStairs0 and line 15 of climbStairs, denoted l(15, 15), establishes that the components of the state of climbStairs’ for climbStairs0 and climbStairs1 have equal values in their return variables. I.e., l(15, 15) is

\[ \text{result}_0 = \text{result}_1 \]

The symbolic relations for the pairs of locations given above define inductive invariants for climbStairs’ that are supported by the assumption that climbStairs0 and climbStairs1 execute from states with equal arguments, and that support the assertion that if climbStairs0 and climbStairs1 terminate, they have equal return values. Thus, the invariants are proof that climbStairs0 and climbStairs1 are partially equivalent.
Fig. 2. Invariants for pairs of prefixes of a control path $p_0$ of $\text{climbStairs0}$ paired with a path $p_1$ of $\text{climbStairs1}$. Each node $n$ represents a pair of subpaths of $p_0$ and $p_1$. The sequence of control locations from the bottom up to row of $n$ contain its path in $p_0$; the sequence of control locations from the left to the column of $n$ contains its path in $p_1$. $n$ is annotated with a relational invariant over all runs of its pair of paths.

2.3 Synthesizing a product program and its invariants

PEQUOD, given programs $P_0$ and $P_1$, attempts to synthesize a product program of $P_0$ and $P_1$ accompanied by inductive invariants by iteratively maintaining invariants of sets of pairs of $P_0$ and $P_1$’s paths. If the invariants $I$ are defined for a path $p_0$ of $P_0$ and path $p_1$ of $P_1$, then $I$ maps $p_0$ and $p_1$ to a symbolic relation between all pairs of states reached after $P_0$ executes $p_0$ and $P_1$ executes $p_1$ from states with equal arguments.

For example, Figure 2 depicts path-pair invariants for all pairs of prefixes of a complete path $p_0$ of $\text{climbStairs0}$ and a complete path $p_1$ of $\text{climbStairs1}$. $p_0$ is the control path of $\text{climbStairs0}$ that executes the loop in lines 9—14 once, and $p_1$ is the control path of $\text{climbStairs1}$ that executes the loop in lines 9—14 twice. I.e., $p_0$ and $p_1$ are the paths executed by their programs on input $n = 3$. The relational invariants for path pairs $([2], [2]), ([2, 9], [2, 9], ([2, 9], [2, 9, 9]), ([2, 9, 15], [2, 9, 9, 15])$ are the entries in the location-pair invariants $I$ for $\text{climbStairs0}$ and $\text{climbStairs1}$ given in §2.2. The invariants for all other pairs of prefixes of $p_0$ and $p_1$ are given explicitly in Figure 2.

In each of PEQUOD’s iterations, it determines if the maintained path-pair invariants $I_M$ define inductive invariants of some product program of $P_0$ and $P_1$. In particular, for path-pair invariants $I_p$, if the map $\text{LocRel}_I[p]$ from each pair of locations $L_0$ and $L_1$ to the disjunction of invariants in $I_p$ for all pairs of paths ending with $L_0$ and $L_1$ are inductive
invariants of some product program of $P_0$ and $P_1$, then $I_p$ are inductive path-pair invariants. If PEQUOD finds a subset of bindings of $I_M$ (i.e., some restriction of $I_M$) that is inductive, then PEQUOD determines that $P_0$ and $P_1$ are equivalent. If not, PEQUOD selects a path $p_0$ of $P_0$ and $p_1$ of $P_1$ on which its maintained path-pair invariants are undefined, attempts to synthesize invariants for $p_0$ and $p_1$, and if it finds such invariants, merges them with the maintained set of path-pair invariants to complete its current iteration. PEQUOD’s algorithm is described in detail in §4.2.

E.g., PEQUOD, given climbStairs$\emptyset$ and climbStairs$1$, synthesizes inductive path-pair invariants for the programs over the following steps. PEQUOD chooses as initial path-pair invariants the empty map. PEQUOD then determines that $\emptyset$ does not define inductive path-pair invariants, using a procedure discussed informally below and given in detail in §4.2.2. As a result, PEQUOD chooses $p_0$ as a path of climbStairs$\emptyset$ and $p_1$ as a path of climbStairs$1$ that have no path-pair invariant in $\emptyset$. PEQUOD then attempts to synthesize path-pair invariants for $p_0$ and $p_1$.

PEQUOD could be adapted to use different path-selection algorithms, causing it to choose different pairs of complete paths. We will consider a scenario in which PEQUOD chooses $p_0$ and $p_1$ in particular, because those paths most clearly illustrate the operation of PEQUOD.

Proving equivalence of pairs of paths. After PEQUOD selects a pair of paths $p_0$ and $p_1$ that are undefined in its maintained set of path-pair invariants, PEQUOD determines if $p_0$ and $p_1$ are equivalent, and synthesizes path-pair invariants for $p_0$ and $p_1$ by issuing repeated queries to an interpolating theorem prover. PEQUOD synthesizes path-pair invariants for $p_0$ and $p_1$ that contain path-pair invariants for each prefix of $p_0$ paired with each prefix of $p_1$. Each path-pair invariant for a pair of path prefixes is synthesized from a logical interpolant, generated by a query to an interpolating theorem prover. The definition of interpolants is reviewed in §3.2, Defn. 4; the reduction from synthesizing path-pair invariants to finding interpolants is given in §4.2.1.

E.g., to verify that path $p_0$ of climbStairs$\emptyset$ and path $p_1$ climbStairs$1$ are equivalent, PEQUOD synthesizes invariants for each pair of a prefix of $p_0$ with a prefix of $p_1$. One such collection of invariants over pairs of prefixes is depicted in Figure 2.

From path-pair invariants to a product program and its invariants. After PEQUOD extends its maintained path-pair invariants to include path-pair invariants for chosen paths $p_0$ and $p_1$, it inspects the extended path-pair invariants to determine if some restriction are inductive, using a novel algorithm described in §4.2.2.

E.g., for paths $p_0$ of climbStairs$\emptyset$ and $p_1$ of climbStairs$1$, some restriction of the path pair invariants $I_p$ defines inductive path-pair invariants for a sub-program of climbStairs$\emptyset$ paired with a subprogram of climbStairs$1$. In particular, let climbStairs$\emptyset$else be climbStairs$\emptyset$, transformed so that the then branch is replaced with an instruction that halts without returning, and similarly for climbStairs1else and climbStairs$1$. Let climbStairs$\prime$else be the product program of climbStairs$\emptyset$else and climbStairs1else, defined similarly to climbStairs$\prime$ for climbStairs$\emptyset$ and climbStairs$1$. Let $I_p$ be the restriction of $I_p$ to the invariants for the pairs of paths $\{(2, 5), (2, 9, 9)\}$, $\{(2, 9, 9), (2, 9, 9)\}$, $\{(2, 9, 9), (2, 9, 9, 15)\}$, $\{(2, 9, 9, 9), (2, 9, 9, 15)\}$. Then LocRel$[I_p]$ are inductive invariants of climbStairs$\prime$else and thus prove the equivalence of climbStairs$\emptyset$else and climbStairs1else. PEQUOD, given climbStairs$\emptyset$else and climbStairs1else would automatically synthesize from $I_p$ both climbStairs$\prime$else and its inductive invariants LocRel$[I_p]$ as a proof of equivalence.

However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$. However, $I_p$ are not inductive invariants for product program climbStairs$\prime$, because they map the pair of paths $\{(2, 2)\}$ to $n_0 = n_0$, and are not defined for any pair of paths that contain line 5 in climbStairs$\emptyset$ and line 5 in climbStairs$1$.
Thus, \( \text{LocRels}[\text{Lsel}] (2, 2) = n_0 = n_1 \) and \( \text{LocRels}[\text{Lsel}] (15, 15) = \text{False} \); as a result, \( \text{LocRels}[\text{Lsel}] \) are not inductive invariants of \( \text{climbStairs'} \).

\text{PEQUOD}, given \( \text{climbStairs} \) and \( \text{climbStairs} \), determines that in fact no restriction of \( I_p \) are inductive path-pair invariants. \text{PEQUOD} continues to determine the equivalence of \( \text{climbStairs} \) and \( \text{climbStairs} \) by choosing a path of paths \( p_0' \) of \( \text{climbStairs} \) and \( p_1' \) of \( \text{climbStairs} \) that reach each line 5. \text{PEQUOD} then synthesizes path-pair invariants \( I_p' \) for \( p_0' \) and \( p_1' \). \text{PEQUOD} then uses \( I_p \) and \( I_p' \) to synthesize path-pair invariants \( I_p'' \) for both \( (p_0, p_1) \) and \( (p_0', p_1') \), determines that some restriction of \( I_p'' \) are inductive invariants for the product program \( \text{climbStairs}' \), and thus determines that \( \text{climbStairs} \) is equivalent to \( \text{climbStairs} \).

3 BACKGROUND

In this section, we review technical concepts on which our approach is based. In §3.1, we define a target language of imperative programs. In §3.2, we review concepts from formal logic.

3.1 Target language

In this section, we define the structure (§3.1.1) and semantics (§3.1.2) of a language of imperative programs.

3.1.1 Program structure. A program is a set of instructions that bind the results of computations to variables. Let \( \text{Locs} \) be a space of control locations that contain a distinguished initial location \( \text{INIT} \) and final location \( \text{FINAL} \). Let \( \text{Vars} \) be a space of program variables, which contains parameter variables \( \text{Params} \) and a return variable \( \text{ret} \). Let \( \text{Instrs} \) be a space of program instructions.

A program instruction tests and updates variables and then transfers a current control location to a target control location. A pre-location, instruction, and branch-target-location is a labeled instruction; i.e., the labeled instructions are \( \text{LInstrs} = \text{Locs} \times \text{Instrs} \times \text{Locs} \). For each labeled instruction \( i \in \text{LInstrs} \), the pre-location, instruction, and post-location of \( i \) are denoted \( \text{Pre}[i], \text{Instr}[i], \) and \( \text{BrTgt}[i] \), respectively.

A program \( P \) is a set of labeled instructions such that for all \( i_0, i_1 \in P \), if \( \text{Pre}[i_0] = \text{Pre}[i_1] \) and \( \text{BrTgt}[i_0] = \text{BrTgt}[i_1] \), then \( i_0 = i_1 \). We denote each \( i \in P \) alternatively as \( \text{Instr}[P][\text{Pre}[i], \text{BrTgt}[i]] \). There is no labeled instruction \( i \in P \) for which \( \text{Pre}[i] = \text{FINAL} \). The space of programs is denoted \( \text{Lang} \). For the remainder of this section, let \( P \in \text{Lang} \) denote a fixed, arbitrary program.

3.1.2 Program semantics. A run of \( P \) is a sequence of states generated by a sequence of labeled instructions in which adjacent instructions have matching target and pre locations. Let the space of program values be the space of integers; i.e., the space of values is \( \text{Vals} = \mathbb{Z} \). An evaluation of all variables in \( \text{Vars} \) is a store; i.e., the space of stores is \( \text{Stores} = \text{Vars} \rightarrow \text{Vals} \). The practical implementation of \text{PEQUOD} verifies partial equivalence of programs that operate on objects and arrays combined with integers. In this paper, we primarily consider programs that operate over only integers, and describe how our implementation handles practical language features in §4.3.3.

For each \( i \in \text{Instrs} \), there is a transition relation \( \rho[i] \subseteq \text{Stores} \times \text{Stores} \). For each \( i \in \text{LInstrs} \), the transition relation of the instruction in \( i \) is denoted \( \rho[i] = \rho[\text{Instr}[i]] \). The transition relation of an instruction need not be total: thus, labeled instructions can implement control branches using instructions that act as assume instructions.

A path of \( P \) is a sequence of control locations that are in adjacent labeled instructions of \( P \).

Definition 1. Let \( i_0, \ldots, i_n \in P \) be such that (1) \( \text{Pre}[i_0] = \text{INIT} \) and (2) for each \( 0 \leq j < n \), \( \text{BrTgt}[i_j] = \text{Pre}[i_{j+1}] \). Then \( \text{Pre}[i_0], \ldots, \text{Pre}[i_n], \text{BrTgt}[i_n] \) is a path of \( P \).
The space of paths of \( P \) is denoted \( \text{Paths}[P] \). The last location in \( p \) is denoted \( \text{tl}[p] \). If \( \text{tl}[p] = \text{FINAL} \), then \( p \) is a complete path. For each \( p \in \text{Paths}[P] \), the non-empty prefixes of \( p \) are denoted \( \text{Prefixes}[p] \). For all \( p, p' \in \text{Paths}[P] \), the set of paths \( p'' \in \text{Paths}[P] \) such that \( p \) is a prefix of \( p'' \) and \( p'' \) is a prefix of \( p' \) is denoted \( \text{Subrange}(p, p') \).

A run of a program \( P \) is a path and a sequence of stores \( \Sigma \) of equal length, such that adjacent stores in \( \Sigma \) satisfy transition relations of instructions at their corresponding locations in \( p \).

**Definition 2.** Let \( \Sigma = \sigma_0, \ldots, \sigma_{n-1} \in \text{Stores} \) and \( L_0, \ldots, L_{n-1} \in \text{Paths}[P] \) be such that for each \( 0 \leq i < n - 1 \), \((\sigma_i, \sigma_{i+1}) \in \rho[\text{Instr}[P](L_i, L_{i+1})]\). Then \((p, \Sigma)\) is a run of \( P \).

The space of runs of \( P \) is denoted \( \text{Runs}[P] \). For each path \( p \in \text{Paths}[P] \), the runs \( r \in \text{Runs}[P] \) such that \( p \) is the path of \( r \) are the runs of \( p \).

\( p_0, p_1 \in \text{Lang} \) are partially equivalent if all complete runs of \( p_0 \) and \( p_1 \) that begin from stores in which parameters have equal values end in stores in which the return variables store equal values.

**Definition 3.** For all \( p_0, p_1 \in \text{Lang} \) and complete \( p_0 \in \text{Paths}[P_0] \) and \( p_1 \in \text{Paths}[P_1] \), if for all \( \sigma_0^0, \ldots, \sigma_m^0, \sigma_0^1, \ldots, \sigma_n^1 \in \text{Stores} \) such that \((p_0, [\sigma_0^0, \ldots, \sigma_m^0]) \in \text{Runs}[P_0] \) and \((p_1, [\sigma_0^1, \ldots, \sigma_n^1]) \in \text{Runs}[P_1] \) and \( \sigma_0^0(\text{Params}) = \sigma_0^1(\text{Params}) \), it holds that \( \sigma_0^0(\text{ret}) = \sigma_0^1(\text{ret}) \), then \( p_0 \) is equivalent to \( p_1 \) under \( p_0 \) and \( p_1 \), denoted \( p_0 \equiv p_1 \).

If for all complete \( p_0 \in \text{Paths}[P_0] \) and \( p_1 \in \text{Paths}[P_1] \) it holds that \( p_0 \equiv p_1 \), then \( p_0 \) is equivalent to \( p_1 \), denoted \( p_0 \equiv p_1 \).

In order to simplify the presentation of our approach, we have given a definition of equivalence in terms of equality over identical parameter and return variables. However, our approach can be immediately generalized to take as a specification of equivalence any equivalence relation over input and final states of two programs. Because Defn. 3 defines equivalence in terms of equal input states and equal resulting output states, it can describe pairs of programs with different control structures and variables used for internal computation, such as \text{climbStairs}0 and \text{climbStairs}1 (introduced in §2.1).

### 3.2 Formal logic

\textsc{PEQUOD} uses formal logic to model the semantics of programs and represent invariants that relate their states. The quantifier-free fragment of the theory of linear arithmetic is denoted \textsc{LIA}. For each space of logical variables \( X \), the space of \textsc{LIA} formulas over \( X \) is denoted \textsc{Forms}[X]. For each formula \( \varphi \in \textsc{Forms}[X] \), the set of variables that occur in \( \varphi \) (i.e., the vocabulary of \( \varphi \)) is denoted \( \text{Var}(\varphi) \). A \textsc{LIA} model \( m \) over \( X \) is an assignment from each variable in \( X \) to an integer. The fact that model \( m \) satisfies a formula \( \varphi \) is denoted \( m \vDash \varphi \).

For formulas \( \varphi_0, \ldots, \varphi_n, \varphi \in \textsc{Forms}[X] \), the fact that \( \varphi_0, \ldots, \varphi_n \) entail \( \varphi \) is denoted \( \varphi_0, \ldots, \varphi_n \models \varphi \).

For all vectors of variables \( X = [x_0, \ldots, x_n] \) and \( Y = [y_0, \ldots, y_n] \), the \textsc{LIA} formula constraining the equality of each element in \( X \) with its corresponding element in \( Y \), i.e., the formula \( \bigwedge_{0 \leq i \leq n} x_i = y_i \), is denoted \( X = Y \). The repeated replacement of variables \( \varphi[y_0/x_0, \ldots, y_{n-1}/x_{n-1}] \) is denoted \( \varphi[X/Y] \). For each formula \( \varphi \) defined over free variables \( X \), \( \varphi[Y/X] \) is denoted alternatively as \( \varphi[Y] \).

Although determining the satisfiability of a \textsc{LIA} formula is NP-complete in general, decision procedures for \textsc{LIA} have been proposed that often determine the satisfiability of formulas that arise from practical verification problems efficiently [14]. \textsc{PEQUOD} assumes access to a decision procedure for \textsc{LIA}, denoted \textsc{ISSAT}.

An interpolant of mutually inconsistent formulas \( \varphi_0 \) and \( \varphi_1 \) is a \textsc{LIA} formula that explains their inconsistency using their common vocabulary.

**Definition 4.** For spaces of logical variables \( X \) and \( Y \), \( \varphi_0 \in \textsc{Forms}[X] \) and \( \varphi_1 \in \textsc{Forms}[Y] \), if \( I \in \textsc{Forms}[X \cap Y] \) is such that (1) \( \varphi_0 \models I \) and (2) \( I, \varphi_1 \models \text{False} \), then \( I \) is an interpolant of \( \varphi_0 \) and \( \varphi_1 \).
Previous work has introduced interpolating theorem provers that synthesize interpolants of pairs of mutually-unsatisfiable formulas in extensions theories used to model program semantics and specifications [31]. To present PEQUOD, we assume access to a procedure ITP that, given mutually unsatisfiable LIA formulas \( \varphi_0, \varphi_1 \), returns an interpolant of \( \varphi_0 \) and \( \varphi_1 \).

### 3.2.1 Symbolic representation of program semantics

The semantics of \( \text{Lang} \) can be represented symbolically using LIA formulas. In particular, each program store \( \sigma \in \text{Stores} \) corresponds to a LIA model over the vocabulary \( \text{Vars} \), denoted \( m^\sigma \). For each space of variables \( X \), space of indices \( I \) and index \( i \in I \), the space of variables \( X_i \) denotes a distinct copy of the variables in \( X \). \( X' \) denotes primed copies of \( X \), which will typically be used to model the post-state resulting from an instruction.

For each instruction \( i \in \text{Instrs} \), there is a formula \( \text{Sem}[i] \in \text{Forms}[\text{Vars}, \text{Vars}'] \) such that for all stores \( \sigma, \sigma' \in \text{Stores} \), \( (\sigma, \sigma') \in \rho[i] \) if and only if \( m^\sigma, m^{\sigma'} \vDash \text{Sem}[i] \). A symbolic relation is a formula whose models define pairs of states from distinct programs. The space of symbolic relations is denoted \( \text{SymRels} = \text{Forms}[\text{Vars}_0, \text{Vars}_1] \).

### 4 TECHNICAL APPROACH

In this section, we describe our approach in technical detail. In §4.1, we define a class of proof structures that each represent a product program paired with its inductive invariants. In §4.2, we describe PEQUOD, which given two programs, attempts to prove or falsify their equivalence by synthesizing such a proof structure. In §4.3, we state and prove the correctness of PEQUOD, and compare it to related approaches for proving program equivalence. Proofs for each lemma and theorem stated in this section are given in Appendix A.

### 4.1 Proof structures

For fixed \( P_0, P_1 \in \text{Lang} \), location-pair invariants of \( P_0 \) and \( P_1 \) describe each pair of runs of \( P_0 \) and \( P_1 \). Location-pair invariants are represented as a map from each pair of control locations to a symbolic relation that describes pairs of states of \( P_0 \) and \( P_1 \) at the mapped pair of locations. Let the space of location-pair relations be denoted \( \text{LocRel} = \text{Locs} \times \text{Locs} \rightarrow \text{SymRels} \).

**Definition 5.** Let \( I_0, I_1 \in \text{LocRel} \) be such that (1) \( \text{Params}_0 = \text{Params}_1 \models I_0(\text{INIT}, \text{INIT}) \lor I_1(\text{INIT}, \text{INIT}) \), (2) for each \( i \in P_0 \) and \( L \in \text{Locs} \),

\[
i_0(\text{Pre}[i], L), \text{Sem}[i][\text{Vars}_0, \text{Vars}_0'] \models (I_0(\text{BrTgt}[i], L) \lor I_1(\text{BrTgt}[i], L))[\text{Vars}_0'/\text{Vars}_0]
\]

(3) for each \( L \in \text{Locs} \) and \( i \in P_1 \),

\[
i_1(L, \text{Pre}[i]), \text{Sem}[i][\text{Vars}_1, \text{Vars}_1'] \models (I_0(L, \text{BrTgt}[i]) \lor I_1(L, \text{BrTgt}[i]))[\text{Vars}_1'/\text{Vars}_1]
\]

(4) \( I_0(\text{FINAL}, \text{FINAL}) \models \text{ret}_0 = \text{ret}_1 \) and \( I_1(\text{FINAL}, \text{FINAL}) \models \text{ret}_0 = \text{ret}_1 \).

Then \( (I_0, I_1) \) are location-pair invariants of \( P_0 \) and \( P_1 \). The space of location-pair invariants for \( P_0 \) and \( P_1 \) is denoted \( \text{LocInv} = \{ P_0, P_1 \} \).

Location-pair invariants for \( P_0 \) and \( P_1 \) define both a product program \( P' \) for \( P_0 \) and \( P_1 \), along with inductive invariants of \( P' \) that imply that \( P_0 \equiv P_1 \), as described in §2.2. Let \( (I_0, I_1) \) be location-pair invariants for \( P_0 \) and \( P_1 \); the product program \( P' \) defined by \( (I_0, I_1) \) is as follows. For all \( I_0, I_1 \in \text{Locs} \), if \( P' \) is in a state that satisfies \( I_0(\text{I}_0, \text{I}_1) \), then \( P' \) may choose \( P_0 \) as
its stepping program; if \( P' \) is in a state that satisfies \( I_1(L_0, L_1) \), then \( P' \) may choose \( P_1 \) as its stepping program. Otherwise, the next step \( P' \) is undefined in its current state.

The inductive invariants of \( P' \) are, for all \( L_0, L_1 \in \text{Locs}, I_0(L_0, L_1) \lor I_1(L_0, L_1) \).

**Example 1.** \( \text{climbStairs} \) and \( \text{climbStairs}! \) have location-pair invariants \( I_0, I_1 \in \text{LocRel} \) that correspond to the product program \( \text{climbStairs}' \) and its inductive invariants given in \$2.2. Key entries in \( I_0 \) include

\[
\begin{align*}
I_0(2, 2) & \equiv n_0 = n_1 & & I_0(5, 5) \equiv \text{True} \\
I_0(9, 9) & \equiv i_0 = i_1 + 1 \land \text{sum}_0 = \text{count}_1 \land n_0 = n_1 & & I_0(15, 15) \equiv \text{result}_0 = \text{result}_1
\end{align*}
\]

\( I_1 \) at each of the location pairs given above is False. At pairs of locations that are not the same line numbers in \( \text{climbStairs} \) and \( \text{climbStairs}! \), \( I_0 \) is False and \( I_1 \) is a suitable symbolic relation.

Location-pair invariants for \( P_0 \) and \( P_1 \) are evidence of the partial equivalence of \( P_0 \) and \( P_1 \).

**Lemma 1.** If there are \( I_0, I_1 \in \text{LocInv}[P_0, P_1] \), then \( P_0 \equiv P_1 \).

PEQUOD attempts to synthesize location-pair invariants from maps from pairs of paths to symbolic relations. Let a path-pair relation be a partial map from pairs of paths to symbolic relations; i.e., the space of path-pair relations is \( \text{PathRel}[P_0, P_1] = \text{Paths}[P_0] \times \text{Paths}[P_1] \rightarrow \text{SymRel} \). Path-pair relations that (1) are supported by the assumption that runs of \( P_0 \) and \( P_1 \) begin with equal arguments, (2) soundly model steps of execution of \( P_0 \), (3) soundly model steps of execution of \( P_1 \), and (4) support the conclusion that all modeled pairs of complete paths end in states with equal return values are path-pair invariants.

**Definition 6.** Let \( I \in \text{PathRel}[P_0, P_1] \) be such that (1) \( \text{Params}_0 = \text{Params}_1 \models I(\text{INIT}, \text{INIT}); \) (2) for each \( p_0 \in \text{Paths}[P_0] \), \( i \in P_0 \) and \( p_1 \in \text{Paths}[P_1] \) such that \( (p_0 \cdot \text{Pre}[i] \cdot \text{BrTgt}[i], p_1) \in \text{Dom}(I) \) (where for function \( f \), \( \text{Dom}(f) \) denotes the domain of \( f \)),

\[
I(p_0 \cdot \text{Pre}[i], p_1), \text{Sem}[i][\text{Vars}_0, \text{Vars}_1'] \models I(p_0 \cdot \text{Pre}[i] \cdot \text{BrTgt}[i], p_1)[\text{Vars}_0', \text{Vars}_1']
\]

(3) for each \( p_0 \in \text{Paths}[P_0] \), \( p_1 \in \text{Paths}[P_1] \), and \( i \in P_1 \) such that \( (p_0, p_1 \cdot \text{Pre}[i] \cdot \text{BrTgt}[i]) \in \text{Dom}(I) \),

\[
I(p_0, p_1 \cdot \text{Pre}[i]), \text{Sem}[i][\text{Vars}_1, \text{Vars}_1'] \models I(p_0, p_1 \cdot \text{Pre}[i] \cdot \text{BrTgt}[i])[\text{Vars}_0, \text{Vars}_1']
\]

(4) for all complete paths \( p_0 \in \text{Paths}[P_0] \) and \( p_1 \in \text{Paths}[P_1] \), \( I(p_0, p_1) \models \text{ret}_0 = \text{ret}_1 \).

Then \( I \) are path-pair invariants of \( P_0 \) and \( P_1 \).

The space of path-pair invariants for \( P_0 \) and \( P_1 \) is denoted \( \text{PathInv}[P_0, P_1] \). For \( p_0 \in \text{Paths}[P_0] \) and \( p_1 \in \text{Paths}[P_1] \), the space of path-pair invariants in which \( (p_0, p_1) \) is defined is denoted \( \text{PathInv}[p_0, p_1] \).

If path-pair invariants \( I \) define a product program and inductive invariants that prove \( P_0 \equiv P_1 \), then \( I \) are inductive for \( P_0 \) and \( P_1 \). For \( R \in \text{PathRel}[P_0, P_1] \), let \( \text{LocRel}[R] \in \text{LocRel} \) be such that for all \( L_0, L_1 \in \text{Locs}, \)

\[
\text{LocRel}[R](L_0, L_1) = \lor \{ R(p_0 \cdot L_0, p_1 \cdot L_1) | p_0 \in \text{Paths}[P_0], p_1 \in \text{Paths}[P_1], L_0, L_1 \in \text{Locs}, (p_0 \cdot L_0, p_1 \cdot L_1) \in \text{Dom}(R) \}
\]

**Definition 7.** For \( I \in \text{PathInv}[P_0, P_1] \), if there are \( R_0, R_1 \in \text{PathRel}[P_0, P_1] \) such that \( I = R_0 \cup R_1 \) and \( (\text{LocRel}[R_0], \text{LocRel}[R_1]) \) are location-pair invariants of \( P_0 \) and \( P_1 \) (Defin. 5), then \( I \) are inductive path-pair invariants for \( P_0 \) and \( P_1 \).

Inductive path-pair invariants for \( P_0 \) and \( P_1 \) are evidence of partial equivalence, by Lemma 1. PEQUOD, given \( P_0 \) and \( P_1 \), attempts to prove \( P_0 \equiv P_1 \) by synthesizing inductive path-pair invariants of \( P_0 \) and \( P_1 \).
Figure 2) prove their partial equivalence.

Input \( P_0, P_1 \in \text{Lang} \)

Output: A decision as to whether \( P_0 \equiv P_1 \)

```
Procedure Pequod\((P_0, P_1)\)
switch ChkInd\((P_0, P_1, I)\) do
  case HasInd do return True;
  case \( P_0 \in \text{Paths}[P_0], P_1 \in \text{Paths}[P_1] \) do
    switch PathInv\(s(P_0, P_1, p_0, p_1)\) do
      case NonEq do return False;
      case \( I' \in \text{PathInvVs}[P_0, P_1] \) do
        return Peq'\((\text{Mrg}(I, I'))\)
end
end
return Peq'\((0)\)
```

Algorithm 1: PEQUOD: given \( P_0 \) and \( P_1 \), determines if \( P_0 \equiv P_1 \), using procedures CHKIND and PATHINVVS, which are discussed in §4.2.

Example 2. The path-pair invariants \( I_p \) relating path \( p_0 \) of \text{climbStairs}@ and path \( p_1 \) of \text{climbStairs}1 (given in §2.3, Figure 2) prove their partial equivalence. \( I_p \) cannot be expressed as the union of any two path-pair relations \( R_0 \) and \( R_1 \) such that \( \text{LocRel}(R_0), \text{LocRel}(R_1) \) are location-pair invariants, as discussed in §2.3. Thus, \( I_p \) are not inductive path-pair invariants of \text{climbStairs}@ and \text{climbStairs}1.

4.2 Verification algorithm

Pseudocode for the core algorithm implemented by PEQUOD is given in Alg. 1. The core algorithm is structured as a counterexample-guided refinement loop analogous to conventional automatic verifiers of safety properties [11, 32]. PEQUOD takes \( P_0, P_1 \in \text{Lang} \) as input (line 1). PEQUOD defines a procedure \( \text{PEQ}' \) that, given \( I \in \text{PathInvVs}[P_0, P_1] \), attempts to determine if \( P_0 \equiv P_1 \) by constructing inductive path-pair invariants from \( I \) (line 2—line 13). PEQUOD runs \( \text{PEQ}' \) on the empty path-pair relation and returns the result (line 14).

\( \text{PEQ}' \), given path-pair invariants \( I \) (line 2), first runs a procedure \text{CHKIND} on \( P_0, P_1, \) and \( I \) (line 3). If \text{CHKIND} returns value HasInd to denote that some restriction of \( I \) are inductive path-pair invariants of \( P_0 \) and \( P_1 \), then \( \text{PEQ}' \) returns True, to denote \( P_0 \equiv P_1 \) (line 4). Otherwise, if \text{CHKIND} returns a pair of paths \( p_0 \) and \( p_1 \) that are not defined in \( I \) (line 5), then \( \text{PEQ}' \) runs a procedure \text{PATHINVVS} on \( P_0, P_1, p_0 \) and \( p_1 \) (line 6). If \text{PATHINVVS} returns that \( p_0 \not\equiv p_1 \), then \( \text{PEQ}' \) returns False, to denote \( P_0 \not\equiv P_1 \) (line 7).

Otherwise, if \text{PATHINVVS} returns \( I' \in \text{PathInvVs}[p_0, p_1] \), then \( \text{PEQ}' \) runs \text{MRG} on \( I \) and \( I' \) to obtain path-pair invariants defined over all pairs of paths defined in \( I \) or \( I' \), recurses on the result, and returns the result of the recursion (line 9). \text{MRG} returns \( I'' \in \text{PathInvVs}[p_0, p_1] \) such that for each \( p_0 \in \text{Paths}[P_0] \) and \( p_1 \in \text{Paths}[P_1] \), if \( (p_0, p_1) \in \text{Dom}(I) \setminus \text{Dom}(I') \).

Input \( : P_0, P_1 \) and \( I \in \text{PathInvVs}[P_0, P_1] \).
Output: HasInd to denote that some restriction of \( I \) are inductive path-pair invariants or a pair of paths not defined in \( I \).

```
Procedure C'\((\text{obs}, \text{dis})\)
  if \( \text{obs} = \emptyset \) then return HasInd;
  \((p_0, p_1), \text{obs}') := \text{REM}(\text{obs})\);
  if \( (p_0, p_1) \not\in \text{Dom}(I) \) then
    \( \text{return } (\text{Cmpl}(P_0, p_0), \text{Cmpl}(P_1, p_1)) \)
  end
  dis := \text{dis} \cup \{(p_0, p_1)\};
  r := C'(\text{obs}', \text{dis}');
  r_0 := C'(\text{obs}' \cup \text{Ext}(P_0, p_0) \times \{p_1\}, \text{dis}');
  r_1 := C'(\text{obs}' \cup \{p_1\} \times \text{Ext}(P_1, p_1), \text{dis}');
  if IsDis(I, p_0, p_1, \text{dis}) then return r;
else if \( \text{tl}[p_0] = \text{FINAL} \) then return r_1;
else if \( \text{tl}[p_1] = \text{FINAL} \) then return r_0;
else return \text{Choose}(r_0, r_1);
end
return C'([([\text{INIT}], [\text{INIT}]), \emptyset]);
```

Algorithm 2: CHKIND: given \( P_0, P_1 \in \text{Lang} \) and \( I \in \text{PathInvVs}[P_0, P_1] \), returns HasInd to denote that some restriction of \( I \) are inductive or a pair of paths not defined in \( I \).
4.2.1 Finding path-pair invariants using PathInvns. PathInvns, given $p_0, p_1 \in \text{Lang}$, $p_0 \in \text{Paths}[p_0]$, and $p_1 \in \text{Paths}[p_1]$, either returns path-pair invariants of $p_0$ and $p_1$ or determines that $p_0 \not\equiv p_1$. PathInvns attempts to find invariants of each $p'_0 \in \text{Prefixes}[p_0]$ paired with each $p'_1 \in \text{Prefixes}[p_1]$ as the interpolant of (1) the disjunction of path-pair invariants describing all pairs of states immediately before $p_0$ and $p_1$ take a final step to complete $p'_0$ and $p'_1$ and (2) a formula describing all pairs of states at $p'_0$ and $p'_1$ from which the remainder of $p_0$ and $p_1$ result in states with non-equal return values.

PathInvns performs the following procedure. For each $p'_0 \in \text{Prefixes}[p_0]$, let there be a distinct copy of $\forall$ $\text{Vars}$ denoted $\text{Vars}[p'_0]$. Let $\text{RemainCtr}[p_0, p'_0] \in \text{Forms}[igcup_{p \in \text{Prefixes}[p_0]} \text{Vars}[p]]$ be the conjunction of semantic constraints from all steps following in $p_0$ following $p'_0$: $\bigwedge_{p''_0 \in \text{Prefixes}[p_0], \ L' \in \text{Locs}, \ p'_0 \cdot L' \in \text{Subrange}(p'_0, p_0)} \text{Sem}[\text{Instr}[p_0](L, L')](\forall \text{Vars}[p''_0 \cdot L], \text{Vars}[p''_0 \cdot L' \cdot L'])$

For each $p'_1 \in \text{Prefixes}[p_1]$, $\text{RemainCtr}[p_1, p'_1]$ is defined similarly.

PathInvns first determines if $p_0$ and $p_1$ are equivalent by running ISSAT on a formula $\text{NoEq}[p_0, p_1]$ for which each model corresponds to a run of $p_0$ paired with a run of $p_1$ that start with equal parameter values and complete with unequal return values. I.e., $\text{NoEq}[p_0, p_1]$ is:

$\text{Params}_0[[\text{INIT}]] = \text{Params}_1[[\text{INIT}]] \land \text{RemainCtr}[p_0, [\text{INIT}]] \land \text{RemainCtr}[p_1, [\text{INIT}]] \land \text{ret}_0[p_0] \neq \text{ret}_1[p_1]$

If $\text{NoEq}[p_0, p_1]$ is satisfiable, then PathInvns returns NonEq.

Example 3. To determine if $p_0$ from climbStairs0 and $p_1$ from climbStairs1 (see §2.3) are partially equivalent, Peqoud determines the satisfiability of the following formula:

$r_0 = n_1 \land \text{RemainCtr}[\text{climbStairs0}, [2]] \land \text{RemainCtr}[\text{climbStairs1}, [2]] \land \text{result}_0 \neq \text{result}_1$

P eqoud uses IsSat to determine that the above formula is unsatisfiable, and thus that the $p_0 \equiv p_1$.

If $\text{NoEq}[p_0, p_1]$ is unsatisfiable, then $p_0 \equiv p_1$. In such a case, PathInvns computes, for each $p'_0 \in \text{Prefixes}[p_0]$ paired with each $p'_1 \in \text{Prefixes}[p_1]$, a path-pair invariant $I(p'_0, p'_1)$ as an interpolant of two formulas. The first formula, referred to as the PreCtr($p'_0, p'_1$), is determined by the form of $p'_0$ and $p'_1$. PreCtr([[INIT]], [INIT]) is

$\text{Params}_0[[\text{INIT}]] = \text{Params}_1[[\text{INIT}]]$

For $p'_0 \in \text{Prefixes}[p_0]$ and $L \in \text{Locs}$ such that $p'_0 \cdot L \in \text{Prefixes}[p_0]$, PreCtr($p'_0 \cdot L$, [INIT]) is

$I(p'_0, [\text{INIT}]) \land \text{Sem}[\text{Instr}[p_0](\text{Vars}_0[p'_0], \text{Vars}_0[p'_0 \cdot L])$

For $p'_1 \in \text{Prefixes}[p_1]$ and $L \in \text{Locs}$ such that $p'_1 \cdot L \in \text{Prefixes}[p_1]$, PreCtr([[INIT]], $p'_1 \cdot L$) is

$I([\text{INIT}], p'_1) \land \text{Sem}[\text{Instr}[p_1](\text{Vars}_1[p'_1], \text{Vars}_1[p'_1 \cdot L])]$
Figure 2.

E.g., in order to synthesize the pair-pair invariant that relates prefix (only \(\text{obs}\) path-pair invariants defined by \(I\) pairs) \(I\) that some restriction of \(\text{given}\) it is given a pair of paths that are equivalent. For all \(\text{climbStairs0}\) and \(\text{climbStairs1}\) previous interpolation queries. The correctness of \(P\) \(\text{Pequod}\) constructs a post-constraint consisting of the conjunction of \(\text{climbStairs0}\) stepping from 2 to 9 and \(\text{climbStairs1}\) stepping from 9 to 9. Pequod constructs a post-constraint consisting of the conjunction of (1) \(\text{RemainCrit}[p_0, [2, 9]]\), which models \(\text{climbStairs0}\) stepping from 9 to 9 and then from 9 to 15, and (2) \(\text{RemainCrit}[p_1, [2, 9, 9]]\), which models \(\text{climbStairs1}\) stepping from 9 to 9 and then from 9 to 15, and (3) \(\text{result}_0 \neq \text{result}_1\).

One interpolant of the pre-constraint and post-constraint given above is \(I(9, 9), \text{the invariant for location 9 in} \text{climbStairs0} \text{and 9 in}\) \(\text{climbStairs1}\) \text{that is also a path-pair invariant for paths} \([2, 9] \text{and} [2, 9, 9]\), \text{as depicted in Figure 2.}

For each \(p_0 \in \text{Prefixes}[p_0]\), \(p_1 \in \text{Prefixes}[p_1]\), \(I(p_0', p_1')\) is the interpolant of \(\text{PreCrit}[p_0', p_1']\) and \(\text{PostCrit}[p_0', p_1']\). The entries of \(I\) can be computed in any ordering of the pairs of prefixes of \(p_0\) and \(p_1\) that respects the prefix ordering of both \(p_0\) and \(p_1\). \(\text{PATHINVS}\) returns the path-pair relations \(I' \in \text{PathInvs}[p_0, p_1]\) such that for each \(p_0' \in \text{Prefixes}[p_0]\) and \(p_1' \in \text{Prefixes}[p_1]\), \(I'(p_0', p_1') = I(p_0', p_1')[\text{Vars}_0, \text{Vars}_1]\).

The correctness of \(\text{Pequod}\) is partially established by the fact that \(\text{PATHINVS}\) returns path-pair relations exactly when it is given a pair of paths that are equivalent.

Lemma 2. For all \(p_0 \in \text{Paths}[p_0]\) and \(p_1 \in \text{Paths}[p_1]\), if \(p_0 \equiv p_1\), then \(\text{PathInvs}(p_0, p_1, p_0, p_1) \in \text{PathInvs}[p_0, p_1]\). Otherwise, \(\text{PathInvs}[p_0, p_1] = \text{NonEq}\).

4.2.2 Finding inductive path-pair invariants using ChkInd. Alg. 2 contains pseudocode for CHKIND. CHKIND, given \(p_0, p_1 \in \text{Lang}\) and path-pair invariants \(I \in \text{PathRel}s[p_0, p_1]\) (line 1), returns either (1) the value HasInd to denote that some restriction of \(I\) is inductive pair-pair invariants of \(p_0\) and \(p_1\), or (2) a pair of paths of \(p_0\) and \(p_1\) that have no invariant in \(I\). CHKIND defines a procedure \(C'\) (line 2—line 15) that takes two sets of pairs of paths: (1) obligation pairs obs and (2) discharged pairs dis. \(C'\) returns either (1) the value HasInd to denote that \(p_0\) and \(p_1\) have inductive path-pair invariants defined by \(I\) restricted to some set of path-pairs that contains dis \(\cup\) obs or (2) a pair of paths that are an extension of some pair in obs that have no invariant in \(I\). CHKIND runs \(C'\) on an initial set of obligations that contains only ([INIT], [INIT]) and an empty set of discharged path pairs, and returns the result (line 16).

\(C'\) first tests if obs is empty, and if so returns HasInd (line 3). Otherwise, if obs is not empty, then \(C'\) chooses and removes a path-pair \((p_0, p_1)\) from obs (line 4). \(C'\) then tests if \((p_0, p_1)\) is undefined in \(I\) (line 5) and, if so, returns a pair of a minimum-length complete extensions of \(p_0\) and \(p_1\) (line 6).
Otherwise, C’ extends dis to contain \((p_0, p_1)\) to form \(\text{dis}'\) (line 8), and computes the result of recursing on \(\text{dis}'\) on three distinct sets of obligations: (1) \(\text{obs}'\), the result of which is stored in \(\text{r}\) (line 9); (2) \(\text{obs}'\) extended with all control successors in \(P_0\) of \((p_0, p_1)\) (denoted \(\text{Ext}(P_0, p_0)\)), the result of which is stored in \(r_0\) (line 10); (3) \(\text{obs}'\) extended with all control successors in \(P_1\) of \((p_0, p_1)\) (denoted \(\text{Ext}(P_1, p_1)\)), the result of which is stored in \(r_1\) (line 11).

C’ tests if \(I(p_0, p_1)\) entails the invariant in \(I\) for some discharged pair of paths with the same final locations by computing:

\[
\text{IsDis}(\text{I}, p_0, p_1, \text{dis}) = \bigvee \{ I(p'_0, p'_1) \mid I(p_0, p_1) \land (p'_0, p'_1) \in \text{dis}, \text{tl}[p_0] = \text{tl}[p'_0], \text{tl}[p_1] = \text{tl}[p'_1] \}
\]

If \(\text{IsDis}(\text{I}, p_0, p_1, \text{dis})\) holds, then \(C'\) returns \(r\) (line 12). Otherwise, if only \(p_0\) is a complete path, then \(C'\) returns \(r_0\) (line 14). Otherwise, \(C'\) runs a procedure \(\text{CHOOSE}\) on \(r_0\) and \(r_1\) (line 15). If either \(r_0 = \text{IsDis}\) or \(r_1 = \text{IsDis}\), then \(\text{CHOOSE}\) returns \(\text{IsDis}\); otherwise, \(\text{CHOOSE}\) returns either result as a complete pair of paths undefined in \(I\) (line 15).

**Example 5.** The path-pair invariants \(I_p\) described in §2.3 are path-pair invariants of \(p_0\) of \(\text{climbPaths}^\emptyset\) and \(p_1\) of \(\text{climbPaths}^I\). However, no restriction of \(I_p\) are inductive path-pair invariants of \(\text{climbStairs}^\emptyset\) and \(\text{climbStairs}^I\). When Pequod inspects \(I_p\) to determine if some restriction of \(I_p\) are inductive path-pair invariants, it determines that they are not inductive.

In particular, when Pequod first considers the pair of paths consisting of only the entry locations \([2]\) and \([2]\), it does not contain any pair of paths in the set \(I\). Therefore, Pequod only determines that \(I_p\) have an inductive restriction its recursive call succeeds on either all extensions of the pair in \(\text{climbStairs}^\emptyset\) or \(\text{climbStairs}^I\). However, the extensions of the pair in \(\text{climbStairs}^\emptyset\) include the pair of paths \(([2],[5],[2])\), and the extensions of the pair in \(\text{climbStairs}^I\) include the pair of paths \(([2],[2],[5])\). \(I_p\) does not define path-pair invariants for either pair of paths.

Pequod therefore returns a pair of complete paths \(p_0\) and \(p_1\) that includes \(5\) in \(\text{climbStairs}^\emptyset\) or line \(5\) in \(\text{climbStairs}^I\). Pequod then synthesizes path-pair invariants \(I_p''\) for \((p_0, p_1)\) and \((p'_0, p'_1)\), as described in §2.3. When Pequod calls \(\text{ChkInd}\) on \(I_p''\), \(\text{ChkInd}\) determines that some restriction of \(I_p''\) are inductive, and thus that \(\text{climbStairs}^\emptyset \equiv \text{climbStairs}^I\).

The correctness of PEQUOD is partially established partially by the fact that \(\text{CHKIND}\) returns \(\text{HasInd}\) only when given path-pair invariants that for which some restriction is inductive.

**Lemma 3.** For \(I \in \text{PathInvs}[P_0, P_1]\), if \(\text{ChkInd}(P_0, P_1, I) = \text{HasInd}\), then some restriction of \(I\) are inductive path pair invariants of \(P_0\) and \(P_1\).

### 4.3 Discussion

In this section, we discuss several key properties of PEQUOD. In §4.3.1, we establish PEQUOD’s correctness. In §4.3.2, we compare to PEQUOD a technique for proving partial program equivalence given in previous work, self-composition. In §4.3.3, we describe challenges to designing a practical implementation of PEQUOD.

#### 4.3.1 Correctness

Whenever PEQUOD returns a definite result, the result is correct.

**Theorem 1.** For all \(P_0, P_1 \in \text{Lang}\), if \(\text{Pequod}(P_0, P_1)\) is defined, then \(P_0 \equiv P_1\) if and only if \(\text{Pequod}(P_0, P_1) = \text{True}\).

Because determining partial program equivalence is, in general, undecidable, PEQUOD is not total: i.e., there are pairs of programs on which PEQUOD will not terminate.
PEQUOD as presented in Alg. 1, given $P_0, P_1 \in \text{Lang}$, returns a Boolean decision as to whether $P_0 \equiv P_1$. PEQUOD can be directly extended so that if it determines that $P_0 \equiv P_1$, then it returns inductive path-pair invariants of $P_0$ and $P_1$. In particular, CHKIND (Alg. 2) is extended so that given path-pair invariants $I$, if it determines that some restriction of $I$ are inductive path-pair invariants of $P_0$ and $P_1$, then it returns the restrictions of $I$ that define location-pair invariants of $P_0$ and $P_1$. In such a case, PEQUOD directly returns restrictions obtained from CHKIND.

In order to return such restrictions of $I$, CHKIND maintains, in addition to the set of obligation path-pair obs, two sets of discharged pairs of paths, denoted $\text{dis}_0$ and $\text{dis}_0$. When CHKIND calls itself on pairs of paths constructed from extensions of $p_0$ in $P_0$ (line 10), it extends $\text{dis}_0$ to contain $(p_0, p_1)$. When CHKIND calls itself on pairs of paths constructed from extensions of $p_1$ in $P_1$ (line 11), it extends $\text{dis}_1$ to contain $(p_0, p_1)$. When CHKIND determines if a given pair of paths $(p_0, p_1)$ has an invariant that is entailed by an invariant that has been previously discharged by computing the predicate $\text{IsDis}$, it enumerates over $\text{dis}_0 \cup \text{dis}_1$.

PEQUOD can also be directly extended so that if it determines that $P_0 \neq P_1$, then it returns a common input on which $P_0$ and $P_1$ generate different final values. To do so, PATHINVS is extended so that when it is given paths $p_0$ and $p_1$ such that $\text{NoEq}[p_0, p_1]$ (§4.2.1) is satisfiable, PATHINVS returns one of its models, which is then returned directly by PEQUOD as a pair of runs from a common input that results in unequal return values.

### 4.3.2 Comparison to sequential composition

Previous work has proposed several approaches for automatically determining the partial equivalence of programs. One approach that, given programs $P_0$ and $P_1$, constructs the self-composition of $P_0$ and $P_1$, which is a program that passes the same inputs to $P_0$ and $P_1$, stores their results, and asserts that the results are equal [9, 45]. Such an approach has potential applications for verifying that a program satisfies a desired information-flow property, can be formulated as proving that when a program is given two inputs with equivalent publicly-visible components, it generates outputs with equivalent publicly-visible components. However, such an approach typically cannot be applied to prove that two programs are partially equivalent, because it requires a safety prover to infer a summary for each of $P$ and $Q$ that precisely describes their functionality. Most model checkers use logics that are combinations of the quantifier-free fragments of linear arithmetic, uninterpreted functions, and arrays, which cannot express such summaries. In particular, neither climbStairs solutions given in §2, nor the solutions that we describe in §5 can be precisely summarized in such theories.

### 4.3.3 Practical design

In §3.1.2, we defined the state space of a Lang program to be a map from program variables to integer values. Our prototype implementation of PEQUOD can take as input programs represented in JVM bytecode, which use instructions that also dynamically allocate, load from, and store to dynamic memory and arrays. In order to support programs that execute such instructions, PEQUOD uses formulas that axiomatize the semantics of each instruction in the combination of the theory of linear arithmetic with the theory of arrays. Formulas in such theories can also be used to define equivalent initial or final states that contain linked data structures and arrays.

The key properties that must be satisfied by a theory $T$ used by PEQUOD to axiomatize instructions are that (1) PEQUOD must have access to an interpolating theorem prover for $T$, which it uses to generate path-pair invariants (§4.2.1); (2) PEQUOD must have access to an automatic decision procedure for $T$, used by PEQUOD to check entailments between pair-pair invariants of different path pairs in Alg. 2, line 12.

In §4.2, we described PEQUOD as using several procedures that were described only at the level of their interface, not their implementation. In particular, for fixed $P \in \text{Lang}$, the procedure CMPL, given $p \in \text{Paths}[P]$, returns a complete
extension of \( p \). In general, a control path may have infinitely many complete extensions. Our prototype implementation of \textsc{Pequod} chooses a complete extension of minimum length, using breadth-first search.

The procedure \textsc{Choose}, used in \( C' \) (§4.2.2), given two results of recursive calls to \( C' \)—each of which may be either \textsc{HasInd} or a pair of control paths—returns a final result for the \( C' \). Our prototype implementation of \textsc{Pequod}, given \textsc{HasInd} as either one of its arguments, always returns \textsc{HasInd}. Given two pairs of paths, it always returns the pair with the shortest combined length. Other feasible implementations of \textsc{Pequod} could be defined by alternative implementations of \textsc{Cmpl} and \textsc{Choose} that choose paths using alternative criteria explored by software model checkers for safety properties.

\textsc{Chkind}, given path-pair invariants \( I \), can in general execute in time exponential in the length of the minimal pair of paths not defined by \( I \), as a result of the fact that in each iteration, it may attempt to find inductive path-pair invariants by extending a path in \( P_0 \) or \( P_1 \). Our prototype implementation of \textsc{Chkind} lazily calls itself recursively, based on the results of evaluating the predicate \textsc{IsDis} and recursive calls. The prototype also memoizes sets of obligations and discharged pairs considered. While this optimization does not improve \textsc{Chkind}'s performance in the worst case, in practice, it causes \textsc{Chkind} to perform significantly more effectively than a conventional inductiveness check on practical pairs of programs (see §5).

5 EVALUATION

We performed an empirical evaluation of \textsc{Pequod} to answer the following questions: (1) Can \textsc{Pequod} verify the partial equivalence of programs written independently that implement distinct, subtle algorithms? (2) Can \textsc{Pequod} verify the partial equivalence of programs written by a wide set of independent programmers? (3) Can \textsc{Pequod} verify equivalence of programs more effectively than self-composition technical that using generic solver?

To answer the above experimental questions, we implemented \textsc{Pequod} as a partial-equivalence verifier for programs represented in JVM bytecode. While we presented \textsc{Pequod} in §4 as a verifier for programs whose instructions are defined in the theory of linear arithmetic, the actual implementation models core JVM language features, including arrays and objects, using the combined theory of linear arithmetic, arrays, and uninterpreted functions (\textsc{Auflia}). The only requirement imposed by \textsc{Pequod} on the logic for expressing program semantics is that the logic has (1) an effective decision procedure, which \textsc{Pequod} uses to check entailment over unknown predicates (§4.2.2), and (2) an effective procedure that constructs interpolants, which \textsc{Pequod} uses to synthesize path-pair invariants (§4.2.1). Both operations are supported by the Z3 interpolating theorem prover [48], which is used in our implementation. We applied \textsc{Pequod} to attempt to prove partial equivalence of 369 pairs of programs submitted by independent programmers as solutions to problems hosted on the online coding platforms Leetcode [27] and CodeChef [13].

In short, our experiments answer the above questions positively: \textsc{Pequod} was able to prove the partial equivalence of an overwhelming majority of pairs of programs to which it was applied. \textsc{Pequod} consistently proved the partial equivalence of programs more efficiently than self-composition technical that using generic solver. The results indicate that \textsc{Pequod} can synthesize proofs of partial equivalence effectively enough to be used as an educational aid, or as an underlying engine for other educational aids, such as autograders [42].

5.1 Experimental procedure

\textsc{Pequod} takes as input (1) two programs \( P_0 \) and \( P_1 \), each represented as a JVM bytecode module. If \textsc{Pequod} determines \( P_0 \equiv P_1 \), then it outputs the relational invariants of \( P_0 \) and \( P_1 \) as the proof. If \textsc{Pequod} determines that \( P_0 \not\equiv P_1 \), it generates a pair of runs from \( P_0 \) and \( P_1 \) from a common input that result in outputs that are not equivalent. \textsc{Pequod}
is implemented in 4,932 lines of Java source code. PEQUOD uses the Soot analysis framework [43] to construct the control-flow graph of given programs, and uses the Z3 interpolating theorem prover [48] to synthesize path-pair invariants (see §4.2.1).

We collected as benchmarks programs submitted as solutions to problems posted on the coding platforms LeetCode and CodeChef. Each problem has over 200 posts in its discussion thread. To determine if PEQUOD can synthesize proofs of equivalence for many programs written independently by programmers with a variety of backgrounds, we collected 369 pairs of solutions of 14 different programming exercises on LeetCode and CodeChef. We ran PEQUOD to determine the equivalence of each pairs of solutions. To show the ability of PEQUOD can synthesize proofs of equivalence across programs that implement subtle algorithms, we presents four pairs of solutions submitted for five challenge problems hosted on LeetCode and CodeChef, in addition to the pair of solutions to climbStairs presented in §2. The results of running PEQUOD on these benchmarks are described in detail in §5.2.

The current version of PEQUOD cannot prove equivalence of the vast majority of solutions on such sites, as proofs of their equivalence require either quantified invariants over arrays or expressive heap invariants. While PEQUOD can model the semantics of such programs accurately, inferring sufficient invariants over data with such structure is itself an ongoing topic of research. We believe that combining PEQUOD with such approaches is an encouraging direction for future research.

In order to evaluate the ability of PEQUOD to prove equivalence compared to previous completely-automatic approaches, we implemented an equivalence verifier, named BASELINE, that uses self-composition (described in §4.3.2), to reduce equivalence verification to safety verification, and apply the best known techniques for safety verification. BASELINE, given programs $P_0$ and $P_1$ constructs systems of constrained Horn clauses [11] $S_0$ and $S_1$ that model all executions of $P_0$ and $P_1$. BASELINE extends $S_0$ and $S_1$ to form a CHC system $S'$ for which each solution corresponds to invariants of the self-composition of $P_0$ and $P_1$ that prove their equivalence. BASELINE then gives $S'$ to DUALITY, a competitive CHC solver implemented within the z3 automated theorem prover.

Verifying equivalence of programs $P_0$ and $P_1$ can be reduced to verifying safety only if the $P_0$ and $P_1$ read input and write output to vectors of scalar data, not streams. As a result, we applied BASELINE to attempt to verify the equivalence of only programs that operate on scalar data. Such programs coincided exactly with the programs that we found on LeetCode.

Both PEQUOD and BASELINE were run on a machine with 16 1.4 GHz processors and 128 GB of RAM. The current implementation of PEQUOD uses a single thread. The implementation is publicly available [35]. All benchmarks are publicly available at references provided in this paper. All benchmarks were posted publicly by their programmers, and we have anonymized the sources of individual programs when referring to them in our results. We are working with the administrators of the coding platforms to potentially redistribute the collected solutions as a standard set of benchmarks for the verification community.

5.2 Equivalent solutions of challenge problems

In this section, we use example solutions from four challenge problems on LeetCode and CodeChef to illustrate PEQUOD’s ability to synthesize proofs of equivalence of subtle implementations. In the relational invariants given for each pair of programs discussed, variables from the first programs (whose name ends with 0) are subscripted 0 and variables from the second program (whose name ends with 1) are subscripted 1.
The Add Digits Problem [1] is to take a non-negative integer in variable num and return sum of all of the digits in num modulo 9. PEQUOD proves that solutions addDigits1 (Figure 4) and addDigits0 (Figure 3) are partially equivalent by synthesizing the following relational invariant the head of the loop of addDigits0 and the end of addDigits1:

\[
\text{result}_0 = \text{sum}_1 - 9 (\text{sum}_1 - 1)/9
\]

The Trailing Zeroses Problem [46] is, given a non-negative integer \( n \), to return the number of zero digits that occur before the least-significant non-zero digit in \( n \). PEQUOD proves that solutions trailingZeroses1 (Figure 5) and trailingZeroses0 (Figure 6) are equivalent by synthesizing the following relational invariant over their loop heads:

\[
\text{sum}_0 = x_1 \land n_0 = y_1/5 \land (n_0 \geq 0 \lor y_1 \geq 0)
\]

The Reverse Integer Problem [39] is to take a non-negative integer \( n \) and return an integer that consists of the digits in \( n \) in reversed order. PEQUOD proves that solutions reverse0 (Figure 7) and reverse1 (Figure 8) are partially equivalent by synthesizing the following relational invariant over their loop heads:

\[
x_0 = x_1 \land (x_0 \geq 0 \lor x_1 \geq 0) \land \text{res}_0 = \text{rev}_1
\]

The Flow-001 Problem [16] is to take a non-negative integer \( T \), then read \( T \) pairs of integers, printing the sum of each pair of integers. PEQUOD proves that two solutions given for the Flow-001 problem, FLOW001_9 (Figure 9) and FLOW001_1 (Figure 10), are equivalent by synthesizing the following relational invariant over their loop heads:

\[
T_0 - x_0 + T_1
\]
| Name         | Pairs | LoC  | Eq.    | Time   | Ineq. | Time   | TO | Eq. | Time | Ineq. | Time |
|--------------|-------|------|--------|--------|-------|--------|----|-----|------|-------|------|
| addDigits    | 1     | 5    | 1      | 21.65s | -     | -      | 0  | 1   | 13.23s | 0     | -    |
| ClimbStairs  | 3     | 10   | 3      | 3m58s  | -     | -      | 3  | 0   | -    | 0     | -    |
| ReverseInteger | 1       | 10   | 1      | 1m43s  | -     | -      | 1  | 0   | -    | 0     | -    |
| trailingZero | 4     | 6.7  | 4      | 1m34s  | -     | -      | 4  | 0   | -    | 0     | -    |
| EX           | 1     | 7    | 1      | 0.21s  | -     | -      | -  | -   | -    | -     | -    |
| LWS          | 2     | 57   | 2      | 1.81s  | -     | -      | -  | -   | -    | -     | -    |
| DIVIDING     | 5     | 24.6 | 5      | 4m1s   | -     | -      | -  | -   | -    | -     | -    |
| ANUTHM       | 30    | 30.3 | 30     | 2m40s  | -     | -      | -  | -   | -    | -     | -    |
| AMIFIB       | 10    | 34.8 | 10     | 28s    | -     | -      | -  | -   | -    | -     | -    |
| FLOW002      | 58    | 19   | 51     | 2m14s  | 7     | 3.32s  | -  | -   | -    | -     | -    |
| FLOW001      | 51    | 19   | 51     | 2m1s   | 0     | -      | -  | -   | -    | -     | -    |
| START01      | 59    | 11.6 | 51     | 0.26s  | 8     | 0.04s  | -  | -   | -    | -     | -    |
| MUFFINS3     | 61    | 19.3 | 51     | 2m24s  | 10    | 2.54s  | -  | -   | -    | -     | -    |
| CIELAB       | 83    | 24.5 | 51     | 22.13s | 32    | 5.54s  | -  | -   | -    | -     | -    |

Table 1. The results of our evaluation of PEQUOD. “Benchmarks Features” contains features of the subject program pairs, in particular the name of the problem that the programs solve ("Name"), the numbers of pairs of solutions ("Pairs") checked, and the average lines of code ("LoC") in each solution. "PEQUOD" contains features of the performance of PEQUOD, in particular the number of pairs of solutions check ("Eq."), the average time taken to prove equivalence ("Time"), the number of pairs of solutions proved, inequivalent ("Ineq.") and the average time of proving inequivalence ("Time"). "BASELINE" contains features of the performance of BASELINE, in particular the number of pairs of solutions that timed out (the timeout limit was 500s.), the number of pairs of solutions proved equivalent ("Eq."), and the average time of proving equivalent ("Time"), over only pairs that did not timeout. the number of pairs of solutions proved inequivalent ("Ineq."), and the average time of proving inequivalent ("Time"), over only pairs that did not timeout.

### 5.3 Results and conclusions

We ran PEQUOD to determine partial equivalence of the 369 program pairs collected. We also ran BASELINE to determine partial equivalence of the nine program pairs collected that did not operate on input and output streams. Because the rest of program pairs we collected has stream I/O that hard to express the assertion in self-composition technical. The results are contained in Table 1. In Table 1, the first four problems are hosted on Leetcode [27] and the rest of the problems are hosted on CodeChef [13].

The only pair of programs that BASELINE can prove equivalent is the pair of solutions to the Add Digits problem. Both solutions to this problem have an input output relation that can be described precisely by a formula in linear arithmetic. BASELINE is able to infer such a formula automatically. PEQUOD requires more time to infer such a solution for the solutions to Add Digits. However, the additional time required by PEQUOD to prove equivalence of a relatively simple pair of programs seems to be an acceptable cost to pay in many contexts in order to obtain the added power of PEQUOD for proving equivalence of more complex pairs of programs.

In summary, our results indicate that PEQUOD significantly improves the state of the art in verifying equivalence of concise, but subtle alternative implementations.
Verifying the equivalence of two programs can also be reduced to synthesizing and proving the correctness of a suitable product program [6, 8]. Previous approaches construct the product program depending partly on matching control structures between the pairs of programs and establishing the logical equivalence of program conditions of matched structures. Previous work has also explored constructing asymmetric product programs [7] which can express proofs of equivalence between programs with loops. Such work does not address the problem of automatically inferring loop invariants of the synthesized product program, which may be viewed alternatively as relational invariants between loops of the original programs. This problem is directly addressed by PEQUOD.

For programs $P_0$ and $P_1$, a special instance of the product programs of $P_0$ and $P_1$ is the sequential composition of $P_0$ and $P_1$. Previous work has explored reducing verifying equivalence to constructing the self-composition of given programs and proving that it satisfies a suitable derived safety property [9, 30, 45] or synthesizing sequential summaries of the program by reduction to solving a system of constrained Horn clauses (CHCs) [15]. A key limitation of such approaches is that they can only infer proofs of correctness that can be expressed using summaries of each program’s behavior in logic used by the verifier. Such logics typically are not sufficiently strong to express summaries required to prove the equivalence of non-trivial programs [6], including the solutions to programming problems that we encountered on online coding platforms [13, 27]. PEQUOD attempts to synthesize relational invariants over internal control locations of two programs. Such a strategy enables PEQUOD to prove partial equivalence of a larger class of pairs of programs, both in principle (as discussed in §4.3.2) and in practice (as discussed in §5.3).

Previous work has proposed automatic verifiers of concurrent programs [18] that synthesize relational invariants by generating a CHC system that is discharged with a generic CHC solver [11, 40]. PEQUOD is similar to such approaches in that it attempts to construct a proof of correctness from relational invariants over pairs of paths. PEQUOD is distinct from such approaches in that it uses a novel construction of relational invariants that can be used to prove partial equivalence of paths of independent programs (given in §4.2.1), and uses a novel algorithm that constructs pairs of relational invariants over locations based on relational invariants for pairs of paths (given in §4.2).

Several automatic equivalence checkers have been proposed for verifying the equivalence of affine [47] and numerical programs [34]. PEQUOD can be applied to programs that use any language features that can be axiomatized in a logical theory with interpolation, such as objects and arrays. PEQUOD does not require widening operations carefully tuned to particular numerical domains in order to converge.

Several proof systems have been proposed in both foundational [23] and modern work [19, 44] for proving total program equivalence, simulation, and $k$-safety. For given programs $P_0$ and $P_1$, such systems can express proofs of equivalence by establishing the validity of semantic summaries that relate the behavior of functions in $P_0$ and $P_1$. Regression-verification techniques [17] match substructures of a pair of programs based on a traversal of the programs’ syntactic structure and attempt to prove that matched substructures are equivalent, using provided candidate relational invariants. Regression verification can be optimized, using symbolic execution to only analyze slices of two given versions of a program that are changed [4]. Regression verification can also be applied to partitions of the given programs’ input space, defined by path formulas of individual program paths, enabling programs to be proved equivalent gradually [12].

Recent work has provided logic systems for reasoning about relational properties of higher-order programs [2]. However, these systems have not yet been used to automatically synthesize proofs of program equivalence. PEQUOD can only infer proofs in a space of structures that is less expressive than the proof structures proposed in such work.
in particular, the proofs inferred by PEQUOD are evidence of only partial equivalence. However, PEQUOD attempts to synthesize such proofs automatically.

Several approaches have been proposed that attempt to verify the equivalence of programs $P_0$ and $P_1$ by symbolically executing the paths of $P_0$ and $P_1$. SYMDIFF verifies that given programs that are loop-free [24] or that are annotated with synchronization points [25] satisfy expected relational summaries. Unlike SYMDIFF, PEQUOD may not always terminate, but PEQUOD can be applied to potentially prove the partial equivalence of programs with loops. UCKLEE, similar to PEQUOD, symbolically executes both programs and inspects pairs of path formulas for control paths of $P_0$ and $P_1$ to determine if they are paths on which $P_0$ and $P_1$ are not equivalent [38]. However, PEQUOD can also potentially use the proofs of equivalence of a pair of paths to prove that given programs are equivalent.

A differential symbolic execution engine [36] symbolically executes given programs $P_0$ and $P_1$, and can optionally construct a formula for each program that over-approximates the effect of each. The engine then compares the relational formulas for each program to determine if the programs may be equivalent. Such an engine is similar to PEQUOD, in that it uses symbolic reasoning to attempt to automatically synthesize a sound over-approximation of the effect of each program. However, a key distinction between such an engine and PEQUOD is that PEQUOD infers relational invariants between programs by iteratively selecting and analyzing particular paths, rather than computing a fixed over-approximation of each program and then comparing the approximations.

Analyses for rootcausing failures of program equivalence [26] take a pair of control paths that prove the non-equivalence of two programs and generate a minimal-cost change to the programs that removes the feasibility of the counterexample. Similarly to rootcausing analyses, PEQUOD applies a precise symbolic analysis to pairs of control paths from $P_0$ and $P_1$. Unlike rootcausing analyses, PEQUOD analyzes control paths either to determine that the paths are a true counterexample to equivalence or to synthesize path invariants that prove that the control paths are equivalent.

Several techniques have been proposed that improve the effectiveness of static program analyses by analyzing multiple versions of a program. The differential-assertion-checking problem [25] is to determine if one version of a given program satisfies all assertions satisfied by a previous version of the program. Verification modulo versions [29] filters warnings generated by applying a static analysis to a new version of a program to only the warnings that are novel to the new version. Optimizations to static analysis have been proposed that compute function summaries using an interpolating theorem prover [41]; when analyzing a new version of the program, the optimized analysis first checks if the summaries computed for functions in the original version of the program are valid summaries for functions in the new version of the program. All of the above approaches use multiple versions of a program to optimize the behavior of a safety analysis; these problems are distinct from the problem addressed by PEQUOD, which is to determine if two programs are partially equivalent. In particular, while PEQUOD also synthesizes an abstraction of given programs from interpolants, the interpolants are synthesized from proofs that pairs of paths from multiple programs are partially equivalent.

Some software model checkers select a program abstraction by constructing Craig interpolants [3, 20, 31, 32, 40] of sub-formulas of formulas that characterize runs of individual paths. However, unlike the above techniques PEQUOD uses interpolants to prove the equivalence of paths selected from distinct programs.

Previous work has identified equivalence verification as a problem with critical applications in programming education, and has proposed autograding techniques for automatically editing a student solution so that it is equivalent to a reference solution [42]. Existing work on autograding relies on a bounded model checker to determine if programs may be equivalent. An autograder that uses an improved equivalence verifier would enjoy a stronger soundness guarantee for determining when a student’s solution is correct. An autograder designed to use not just counterexamples to equivalence
but also relational invariants for equivalence could potentially suggest edits to student solutions that are functionally correct but could be simplified or optimized.

7 CONCLUSION

We have presented a novel algorithm that attempts to prove the partial equivalence of given programs. A key challenge in proving the partial equivalence of given programs \( P_0 \) and \( P_1 \) is to both synthesize a suitable product program \( P' \) of \( P_0 \) and \( P_1 \), and to synthesize inductive invariants of \( P' \) that prove the equivalence of \( P_0 \) and \( P_1 \). Previous approaches address this problem by first choosing a product program either by choosing one from a heavily restricted class of product programs, requiring a product program to be given manually, or choosing one based on fixed heuristics. After choosing a candidate product program, such approaches then attempt to synthesize its inductive invariants.

We have presented a novel equivalence verifier, named PEQUOD, that does not operate under any of the above limitations. The key feature of PEQUOD is that it attempts to synthesize a product program and its invariants simultaneously. To do so, PEQUOD iteratively collects proofs of equivalence of pairs of paths of given programs, and attempts to extract a product program and its inductive invariants from the invariants defined per pair of paths. We have implemented a prototype version of PEQUOD that targets JVM bytecode, and used it to verify hundreds of alternate solutions submitted by students to online coding problems.

REFERENCES

[1] AddDigits 2016. LeetCode Problem: Add Digits. https://leetcode.com/problems/add-digits/. (2016). Accessed: 2016 July 5.
[2] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A Relational Logic for Higher-Order Programs. In ICFP.
[3] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. Craig Interpretation. In SAS.
[4] John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In SPIN.
[5] Thomas Ball and Sriram Rajamani. 2002. The SLAM project: debugging system software via static analysis. In POPL.
[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In FM.
[7] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In LNCS.
[8] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product Programs and Relational Program Logics. In JILAMP.
[9] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In CSFW-17.
[10] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In POPL.
[11] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In SAS.
[12] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based regression verification. In ICSE.
[13] codechef 2016. Programming competition, programming contest, online computer programming. https://www.codechef.com/. (2016). Accessed: 2016 June 14.
[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In TACAS.
[15] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In ASE.
[16] FLOW001 2016. CodeChef Problem: FLOW001. https://www.codechef.com/submit/FLOW001. (2016). Accessed: 2016 July 5.
[17] Benny Godlin and Ofer Strichman. 2009. Regression verification. In DAC.
[18] Ashutosh Gupta, Cornelius Popeea, and Andrey Rybalchenko. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In POPL.
[19] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In CADE-24.
[20] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested interpolants. In POPL.
A PROOF OF CORRECTNESS

Here, we give a formal proof for Thm. 1 and its lemmas. First we provide the proof of the underlying lemmas, then we provide the proof of the theorem.

The following is a proof of Lemma 1:

**Proof.** If there exists $I_0, I_1 \in \text{LocInvs}[P_0, P_1]$, then the definition of location-pair invariants (Defn. 5) implies that for each pair of runs of $P_0$ and $P_1$ under the same input, $P_0$ and $P_1$ do not produce different output. The definition of path (Defn. 1), of run (Defn. 2), and of partial equivalence (Defn. 3) prove that $P_0 \equiv P_1$. □

The following is a proof of Lemma 2:
There are two cases for this proof.

For all \( p_0 \in \text{Paths}[p_0] \) and \( p_1 \in \text{Paths}[p_1] \), if \( p_0 \equiv p_1 \), then \( \text{PathInvs}(p_0, p_1, p_0, p_1) \in \text{PathInvs}[p_0, p_1] \). If \( p_0 \not\equiv p_1 \), then by the definition of partial equivalence (Defn. 3), for all pairs of paths \( p_0 \in \text{Paths}[p_0] \) and \( p_1 \in \text{Paths}[p_1] \), \( \text{NoEq}(p_0, p_1) \) is not satisfiable. By the core algorithm of \( \text{PEQUOD} \) (Alg. 1, line 4), \( \text{PathInvs}(p_0, p_1, p_0, p_1) \in \text{PathInvs}[p_0, p_1] \).

For all \( p_0 \in \text{Paths}[p_0] \) and \( p_1 \in \text{Paths}[p_1] \), if \( p_0 \not\equiv p_1 \), then \( \text{PathInvs}[p_0, p_1] = \text{NonEq} \). If \( p_0 \not\equiv p_1 \), then by the definition of partial equivalence (Defn. 3), there exists a pair of paths \( p_0 \in \text{Paths}[p_0] \) and \( p_1 \in \text{Paths}[p_1] \), such that \( \text{NoEq}(p_0, p_1) \) is satisfiable. By the core algorithm of \( \text{PEQUOD} \) (Alg. 1, line 7), \( \text{PathInvs}[p_0, p_1] = \text{NonEq} \).

The following is a proof of Lemma 3:

**Proof.** We construct this proof by induction on the evaluation of \( \text{CHKIND} \) run over obs and dis:

The inductive claim is that if the path-pair invariants in obs are inductive, then there exists a restriction on \( I \) which is a set of inductive path-pair invariants that contains all elements of dis.

For the base case, \( \text{CHKIND} \) is called on \( ([\text{INIT}], [\text{INIT}]) \) and \( \emptyset \) (Alg. 2), which combined with the definition of inductive path-pair invariants (Defn. 6), implies the claim.

For the inductive case, when obs is non-empty, a path-pair invariant \( I(p_0, p_1) \in \text{obs} \) is inspected. obs is constructed by removing \( I(p_0, p_1) \) from \( \text{obs} \), and \( \text{dis'} \) is constructed by adding \( I(p_0, p_1) \) to \( \text{dis} \). From here, there are two possibilities:

1. If \( I(p_0, p_1) \) is entailed by \( I(p'_0, p'_1) \), \( p_0 \) and \( p'_0 \) end with the same control location, \( p_1 \) and \( p'_1 \) end with the same control location, and \( I(p'_0, p'_1) \in \text{dis} \) then \( \text{CHKIND} \) calls itself recursively with obs and dis'. (Alg. 2, line 9). This step maintains the inductive claim.

Location-pair invariants are constructed by taking the disjunction of all path-pair invariants that end with the same control location. This fact, the fact that \( p'_0 \) and \( p_0 \) end with the same control location as \( p_1 \) and \( p'_1 \), and the fact \( I(p'_0, p'_1) \) entails \( I(p_0, p_1) \) together indicate that \( I(p_0, p_1) \lor I(p'_0, p'_1) \) still entails \( I(p_0, p_1) \). Because \( I(p_0, p_1) \lor I(p'_0, p'_1) \) hold for all clauses in the location-pair invariant system, the claim is established by definition of inductive path-pair invariants.

Otherwise, \( \text{CHKIND} \) calls itself recursively on obs' extended with the path-pair invariant from taking a step in the left program (Alg. 2, line 10) or in the right program (Alg. 2, line 11) together with dis'. In these cases, the claim is established by the definition of inductive invariants (Defn. 7) and the definition of location-pair invariants rules 2 and 3 respectively (Defn. 5).

When obs is empty, \( \text{CHKIND} \) returns HasInd, by Alg. 2. This fact, together with the inductive claim, implies that \( I \) is a set of inductive path-pair invariants for \( p_0 \) and \( p_1 \).

As stated by Thm. 1, whenever \( \text{PEQUOD} \) returns a definite result, the result is correct.

**Proof.** First we prove: For all \( p_0, p_1 \in \text{Lang} \), if \( \text{PEQUOD}(p_0, p_1) \) is defined and \( p_0 \equiv p_1 \), then \( \text{PEQUOD}(p_0, p_1) = \text{True} \). This can be restated as: for all \( p_0, p_1 \in \text{Lang} \), if \( \text{PEQUOD}(p_0, p_1) \) is defined and \( \text{PEQUOD}(p_0, p_1) = \text{False} \), then \( p_0 \not\equiv p_1 \). If \( \text{PEQUOD}(p_0, p_1) = \text{False} \), then the core algorithm \( \text{PEQUOD} \) (Alg. 1, line 7), implies that there exists \( p_0 \in \text{Paths}[p_0] \) and \( p_1 \in \text{Paths}[p_1] \) such that \( \text{PathInvs}[p_0, p_1] \) is unsatisfiable. Therefore, by Lemma 2, \( p_0 \not\equiv p_1 \).

Next we prove: For all \( p_0, p_1 \in \text{Lang} \), if \( \text{PEQUOD}(p_0, p_1) \) is defined and \( \text{PEQUOD}(p_0, p_1) = \text{True} \), then \( p_0 \equiv p_1 \). If \( \text{PEQUOD}(p_0, p_1) = \text{True} \), then the core algorithm \( \text{PEQUOD} \) (Alg. 1, line 4), implies that \( \text{CHKIND}(p_0, p_1, I) = \text{HasInd} \). By Lemma 3, there exists some restriction of \( I \) which is a set of inductive path-pair invariants of \( p_0 \) and \( p_1 \). By the definition of inductive path-pair invariants (Defn. 7) and of location-pair invariants (Defn. 5), there exists \( I_0, I_1 \in \text{LocInvs}[p_0, p_1] \). Therefore, by Lemma 1, \( p_0 \equiv p_1 \).