Complexity Verification using Guided Theorem Enumeration

Akhilesh Srikanth
Georgia Institute of Technology, USA
akhilesh.srikanth@Gatech.edu

Burak Sahin
Georgia Institute of Technology, USA
buraksahin@Gatech.edu

William R. Harris
Georgia Institute of Technology, USA
wharris@cc.gatech.edu

Abstract
Determining if a given program satisfies a given bound on the amount of resources that it may use is a fundamental problem with critical practical applications. Conventional automatic verifiers for safety properties cannot be applied to address this problem directly because such verifiers target properties expressed in decidable theories; however, many practical bounds are expressed in non-linear theories, which are undecidable.

In this work, we introduce an automatic verification algorithm, CAMPY, that determines if a given program $P$ satisfies a given resource bound $B$, which may be expressed using polynomial, exponential, and logarithmic terms. The key technical contribution behind our verifier is an interpolating theorem prover for non-linear theories that lazily learns a sufficiently accurate approximation of non-linear theories by selectively grounding theorems of the non-linear theory that are relevant to proving that $P$ satisfies $B$. To evaluate CAMPY, we implemented it to target Java Virtual Machine bytecode. We applied CAMPY to verify that over 20 solutions submitted for programming problems hosted on popular online coding platforms satisfy or do not satisfy expected complexity bounds.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: Model Checking, Reliability, Formal Methods

General Terms Languages, Performance, Verification

Keywords Complexity, Interpolation, Theorem Grounding

1. Introduction
In many contexts, programmers must be able to derive strong guarantees about the performance of their program. Such contexts can include both mission-critical systems and high-performance code that executes on servers. If such programs fail to satisfy resource bounds that a programmer expects, there can be dire consequences because such verifiers target properties expressed in decidable theories; however, many practical bounds are expressed in non-linear theories, which are undecidable.

Previous work has introduced bound analyses, which take a program $P$ and always infer a sound bound on the resources used by $P$ (Albert et al. [2012], Alias et al. [2010], Gulwani et al. [2009a,b], Gulwani and Zuleger [2010]). The key contribution of bound analyses is that they can often provide useful information to a programmer with no additional effort required of the programmer. However, the limitation of bound analyses is that, by the nature of the problem that they address, they cannot ensure that the bound that they infer is the tightest one possible, and cannot be used to show that a program does not satisfy an expected bound.

In this work, we introduce a bound-driven automatic complexity verifier, named CAMPY. CAMPY takes from a programmer a program $P$ and an expected bound $B$ on the number of steps of execution that $P$ takes in terms of its inputs. CAMPY synthesizes either (1) a proof that $P$ satisfies $B$ on all inputs or (2) a run of $P$ on which does not satisfy $B$ (CAMPY may also fail to terminate). While CAMPY thus requires a programmer to spend additional effort to construct an expected bound $B$, when successful, it always gives the programmer a concrete guarantee concerning the performance of the program with respect to $B$.

CAMPY casts the problem of verifying that a given program $P$ satisfies a given bound $B$ as verifying that $P$ satisfies a safety property that specifies that a cost counter satisfies a potentially-non-linear constraint over values in the program’s initial state. Like conventional automatic safety verifiers, CAMPY attempts to infer bounds on a counter in terms of state variables by selecting individual paths of execution of $P$, proving that all runs of each individual path satisfy $B$, and combining the proofs for each path to prove that all runs of $P$ satisfy $B$.

However, developing a complexity verifier presents key technical challenges that are not addressed when designing an eager bounds analysis or verifiers for classes of safety properties. In particular, invariants that are used to express proofs of bound satisfaction typically must be expressed in an undecidable theory, such as the theory of non-linear arithmetic. As a result, conventional techniques for automatically verifying safety properties (Ball et al. [2001], Henzinger et al. [2002, 2004], McMillan [2006]), which rely on automatic decision procedures to infer invariants that prove the safety of all runs of a single path, cannot be applied directly. Second, inductive invariants used in proofs of bound satisfaction typically are defined over non-linear terms that are not apparent from the original program or its semantic constraints.

The core technical contribution of CAMPY is its technique for constructing a proof that runs of a program path satisfy a given non-linear bound. To do so, CAMPY uses a novel theorem prover for non-linear arithmetic, which itself uses a decision procedure for the combination of theories of linear arithmetic and uninterpreted functions. To determine satisfiability of a path formula and synthesize invariants for all runs of the path in a non-linear theory, the theorem prover lazily refines an approximation of the theory of non-linear arithmetic. To do so, the theorem prover iteratively attempts to find a model of the path formula under the prover’s maintained approximation of non-linear arithmetic. If it finds a model, it tests the model as a model of the path formula under the standard model of non-linear arithmetic. If the model under the approximation diverges from the standard model such that the divergence changes the final evaluation
public static int BinarySearch(int arr[], int n) {
    int fst = 0;
    int lst = arr.len - 1;
    while (fst <= lst) {
        int mid = (fst + lst) / 2;
        if (arr[mid] < n) {
            lst = mid - 1;
        } else if (arr[mid] == n) {
            break;
        } else lst = mid - 1;
    }
    return mid;
}

Figure 1: BinarySearch: an implementation of binary search that takes a sorted array of integers arr and an integer value n and returns an index at which arr stores n.

of the path formula, then the prover uses the divergence to guide a search for quantifier-free theorems of non-linear arithmetic that are sufficient to prevent similar divergences.

Our key results are that CAMPY is sound, and in practice, it can potentially be applied by programmers to automatically synthesize either proofs that an implementation of a subtle algorithm satisfies an expected bound or a run of the implementation that does not satisfy the bound. In particular, we implemented CAMPY as a verifier for JVM bytecode programs, and ran it to verify expected complexity bounds of programs submitted by students as solutions to challenge problems hosted on several online coding platforms (codechef, leetcode, codeforces). We have used CAMPY to prove or disprove that over 20 programs satisfy or not satisfy complexity bounds.

The remainder of this paper is organized as follows. §2 gives an informal overview of CAMPY by example. §3 reviews previous work on which CAMPY is based. §4 describes CAMPY in technical detail. §5 gives an empirical evaluation of CAMPY. §6 compares CAMPY to related work on complexity analysis.

2. Overview

In this section, we illustrate our approach by example. In §2.1 we present an iterative implementation of binary search, named BinarySearch, as a running example, along with an expected bound on its execution time. In §2.2 we give an expected bound on the number of execution steps taken by BinarySearch and inductive invariants that prove that BinarySearch satisfies its expected bound. In §2.3 we describe how CAMPY synthesizes the inductive invariants given in §2.2 from BinarySearch and its expected bound automatically.

2.1 An Iterative Implementation of Binary Search

Fig. 1 contains source code for an implementation of binary search adapted from a class named BinarySearch posted to the LeetCode online coding platform; the hosted code has been simplified and refactored for the purpose of illustrating our approach, but CAMPY verifies BinarySearch without manual modifications.

BinarySearch takes an array of integers arr and an integer n. If arr is sorted and contains n, then BinarySearch returns an index at which it stores n. BinarySearch executes a loop that maintains the inductive invariant that, under the above assumptions, there is some index at which arr stores n that is greater than or equal to fst and less than or equal to lst. Before BinarySearch executes the loop, it establishes the invariant by initializing fst to 0 (line 2) and initializing lst to len(arr) − 1 (line 3).

In each iteration of the loop, BinarySearch tests that fst \leq lst (line 4). If so, BinarySearch computes the average of fst and lst and stores the result in mid (line 5). BinarySearch then tests if the value of arr at index mid is less than n (line 6); if so, BinarySearch updates fst to store mid incremented by 1 (line 7). Otherwise, BinarySearch tests if the value of arr at mid is equal to n (line 8); if so, BinarySearch immediately exits the loop (line 9). Otherwise, BinarySearch updates lst to store the index in mid decremented by 1 (line 10) and completes the current loop iteration.

2.2 Inductive Step-Count Invariants

Each run of BinarySearch uses an amount of time bounded by a logarithmic function (base 2) of the length of arr (when using log to express bounds, we will treat it as a function over integers that maps each non-negative integer to 0). We can express such a relationship as a property of the state of BinarySearch by interpreting BinarySearch under a semantics that extends the state of each program with a variable cost that models the current cost of an execution. Under the extended semantics, cost is incremented whenever BinarySearch performs a complete iteration of a loop (i.e., whenever BinarySearch steps from line 9 to line 8 in Fig. 1). Proving that BinarySearch executes in time bounded by \log(length(arr)) + 1 can be expressed as requiring all runs of BinarySearch to satisfy the post-condition \( B_{\text{sch}} \equiv \text{cost} \leq \log(length(arr)) + 1 \).

In general, under such a formulation of bound satisfaction, a program with non-terminating executions could potentially satisfy a bound. However, the formulation can easily be adapted to require that the condition on cost be satisfied at each cutpoint in the program. To simplify the presentation of CAMPY, in this paper we only explicitly consider the problem of verifying that a program satisfies a step-count bound on all terminating executions, and thus that a step-count condition need only be checked at the final location of the program.

BinarySearch can be proved to satisfy \( B_{\text{sch}} \) by establishing the loop invariant \( I \equiv \text{cost} + 1 + \log(1st - fst) \leq \log(length(arr)) \) and proving that \( I \) implies that \( B_{\text{sch}} \) is satisfied at the end of execution. Using the rules of Hoare logic, such a problem can be reduced to discharging entailments that express the following conditions. When each initial state steps to the entry point of the loop, the resulting state must satisfy:

\[
|I| = 0 + \log(|1st - length(arr)| - 0) \leq \log(length(arr)) + 1 \tag{1}
\]

Each state that satisfies \( I \) and does not satisfy the loop guard must satisfy \( B_{\text{sch}} \):

\[
\text{cost} + \log(1st - fst) \leq \log(length(arr)) + 1 \tag{2}
\]

Each state that satisfies \( I \) and steps through the loop on the path that contains line 7 must step to a state that satisfies \( I \):

\[
\text{cost} + \log(1st - fst) \leq \log(length(arr)) + 1 \tag{3}
\]

Each state that satisfies \( I \) and steps through the loop on the path that contains line 9 must step to a state that satisfies \( B_{\text{sch}} \):

\[
\text{cost} + \log(1st - fst) \leq \log(length(arr)) + 1 \tag{4}
\]

Each state that satisfies \( I \) and steps through the loop on the path that contains line 9 must step to a state that satisfies \( B_{\text{sch}} \):

\[
\text{cost} + \log(1st - fst) \leq \log(length(arr)) + 1 \tag{5}
\]

Eqn. 3 and Eqn. 4 may be weakened to contain an additional assumption that models information known about arr[mid] at lines 6 and 9, but these assumptions are not required to discharge the entailments, and are thus omitted to simplify the presentation. Eqn. 1 and Eqn. 2 can be proved by reasoning about log as an uninterpreted function and using only the axioms of the theory of linear arithmetic and uninterpreted functions (i.e., EUFLIA) Eqn. 2 can be proved using only the axioms of EUFLIA and the theorem

\[
1st - fst \leq 0 \implies \log(1st - fst) = 0 \tag{6}
\]
which is the axiom \( \forall x. x \leq 0 \implies \log x = 0 \) grounded on the term \( 1st - fst \). Eqn. 3 and Eqn. 5 can be proved using the axioms of EUFLIA and the following quantifier-free theorems of \( \log \):

\[
\log 2 = 1 \tag{7}
\]

\[
\log ((1st - fst)/2) = \log (1st - fst) - \log 2 \tag{8}
\]

Eqn. 8 is the theorem \( \forall x, y. \log x/y = \log x - \log y \) grounded on terms \( 1st - fst \) and 2.

2.3 Synthesizing Step-Count Invariants

In this section, we illustrate how CAMPY automatically synthesizes inductive step-count invariants that prove that BinarySearch satisfies \( B_{\text{Srch}} \). CAMPY attempts to prove that a given program \( P \) satisfies a resource bound \( B \) by synthesizing sufficient inductive invariants of \( P \) under a semantics extended with a cost model. E.g., if CAMPY is given program BinarySearch and expected bound \( B_{\text{Srch}} \), CAMPY attempts to prove that BinarySearch satisfies \( B_{\text{Srch}} \) by synthesizing inductive step-count invariants similar to those given in §2.2.

Similarly to conventional automatic verifiers for safety properties (Henzinger et al. 2002; 2004; McMillan 2000), CAMPY attempts to synthesize inductive step-count invariants for \( P \) by iteratively selecting a control path \( q \) of \( P \), synthesizing path-step-count invariants that prove that all runs of \( q \) satisfy \( B \), and combining the step-count invariants for individual paths to attempt to search for inductive invariants of \( P \). E.g., if CAMPY is run on BinarySearch and bound \( B_{\text{Srch}} \), it may select the control path of BinarySearch that iterates through the loop in Fig. 1 twice, depicted in Fig. 2. CAMPY would synthesize invariants for each point in \( q \), such as the invariants given in Fig. 2 for each point in \( q \) at lines 2 or 4.

The key technical problem addressed in this work is, given a resource bound \( B \) expressed in a non-linear theory and a control path \( q \), to automatically synthesize path invariants for \( q \) that are sufficient to prove that all runs of \( q \) satisfy \( B \). While a similar problem is addressed by automatic verifiers for safety properties, the key distinction between synthesizing path invariants that support a safety property and synthesizing path invariants that support a step-count bound is that step-count bounds are often expressed in non-linear theories. E.g., the bound \( B_{\text{Srch}} \) for BinarySearch includes a term constructed from the logarithm function.

CAMPY synthesizes sufficient path invariants on resource usage expressed in a non-linear theory \( T \) with a standard model by using a novel interpolating theorem prover, named \( T[P] \). There cannot be a complete theorem prover for logics used to express practical complexity bounds due to fundamental results establishing the undecidability of non-linear arithmetic (Gödel et al. 1934). However, the design of \( T[P] \) is motivated by the key observation that often, path invariants that prove that all runs of a path satisfy a bound are supported by the axioms of a decidable theory used to express the semantics of program instructions, combined with a small set quantifier-free theorems of the non-linear theory over small terms. E.g., the validity of the path invariants for path \( q \) of BinarySearch given in §2.3 is supported by EUFLIA extended with only the quantifier-free non-linear theories Eqn. 7 and Eqn. 8.

\( T[P] \) attempts to simultaneously synthesize valid path invariants and quantifier-free \( T \) theorems \( A \) that support their validity using a counterexample-guided loop. In each iteration of the loop, \( T[P] \) queries a decision procedure for EUFLIA, named EUFLIASAT, to determine if a formula whose models correspond to runs of \( q \) that do not satisfy bound \( B \) (i.e., the path formula for \( q \) and \( B \), denoted \( \phi^B_q \) and a maintained candidate set \( A \) are mutually unsatisfiable as EUFLIA formulas. If EUFLIASAT determines that \( \phi^B_q \land A \) is unsatisfiable as an EUFLIA formula, then \( T[P] \) generates path invariants for \( q \) by running an interpolating theorem prover for EUFLIA on \( \phi^B_q \land A \) (interpolating theorem provers are presented in detail in §3.2.2).

E.g., for bound \( B_{\text{Srch}} \) and path \( q \) of BinarySearch §2.3, the path formula \( \phi^B_q \) is

\[
\text{cost}_0 = 0 \land \text{fst}_0 = 0 \land \text{lst}_0 = \text{len} (\text{arr}) - 1 \land \text{fst}_0 \leq \text{lst}_0 \land \text{mid}_0 = (\text{fst}_0 + \text{lst}_0)/2 \land \text{arr} [\text{mid}_0] < n \land \text{fst}_1 = \text{mid}_0 + 1 + \text{cost}_1 = \text{cost}_0 + 1 \land \text{mid}_0 = (\text{fst}_1 + \text{lst}_1)/2 \land \text{arr} [\text{mid}_0] \neq n \land \text{arr} [\text{mid}_0] = n \land \text{cost}_1 = 0 \implies \log (\text{len} (\text{arr})) + 1
\]

While \( \phi^B_q \) does not have a model in non-linear arithmetic, it does have one in EUFLIA, where the function symbol \( \log \) is interpreted. In particular, one satisfying partial model is the following assignment \( m \):

\[
m(\log 4) = 0 \quad m(\text{arr}[1]) = 10 \quad m(\text{arr}[2]) = 20 
\]

\[
m(\text{len}(\text{arr})) = 4 \quad m(n) = 20 \quad m(\text{cost}_0) = 0 
\]

\[
m(\text{fst}_0) = 0 \quad m(\text{lst}_0) = 3 \quad m(\text{mid}_0) = 1 
\]

\[
m(\text{fst}_1) = 3 \quad m(\text{cost}_1) = 1 \quad m(\text{mid}_1) = 2
\]

\( m \) is a model of \( \phi^B_q \) under EUFLIA but not under non-linear arithmetic because it assigns \( \log 8 \) to 0 (the assignment is emphasized above in bold typeface).

If EUFLIASAT returns an EUFLIA model \( \phi^B_q \) of \( \phi^B_q \), then \( T[P] \) evaluates \( \phi^B_q \) on \( m \) using the standard model of all \( T \) functions. If \( \phi^B_q \) is satisfied by \( m \) using the standard model of \( T \), then \( T[P] \) determines that \( \phi^B_q \) is satisfiable as a \( T \) formula, and as a result, that \( q \) does not satisfy the bound \( B \). E.g., for path \( q \) of BinarySearch and bound \( B_{\text{Srch}} \), if \( T[P] \) runs EUFLIASAT on \( \phi^B_q \) and obtains model \( m \), then it evaluates \( \phi^B_q \) under the model \( m \) that binds \( \text{cost}_0, \text{fst}_0, \text{len}_0, \text{lst}_0, \text{mid}_0, \text{arr}_0, \text{mid}_0, \text{cost}_1 \), to their values in \( m \), and interprets \( \log \) under the standard model, \( m \) does not satisfy \( \phi^B_q \).

In general, if \( m \) does not satisfy path formula \( \phi^B_q \), then \( T[P] \) chooses a clause \( C \) of \( \phi^B_q \), not satisfied by \( m \) and generates a set of \( T \) formulas that are not satisfied by any model that evaluates the \( T \) terms in \( C \) to their values under \( m \). E.g., when \( T[P] \) attempts to find path invariants of path \( q \) of BinarySearch, \( T[P] \) determines that \( m \) does not satisfy the clause \( \text{cost}_1 \leq \text{log} (\text{len}(\text{arr})) \) of \( \phi^B_{\text{Srch}} \). \( T[P] \) enumerates theorems of non-linear arithmetic over the variables \( \text{cost}_0, \text{fst}_0, \text{len}_0, \text{lst}_0, \text{mid}_0, \text{arr}_0, \text{mid}_0, \text{cost}_1 \); until it finds a minimal set that has model \( m \) that satisfies \( \phi^B_{\text{Srch}} \).

\[
m(\text{len}(\text{arr})) = m(\text{len}(\text{arr})) = 4 \quad m(\log (4) = 3
\]

One such set are the axioms of non-linear arithmetic Eqn. 7 and Eqn. 8.

The effectiveness of \( T[P] \) iteratively generates models and enumerates theorems until either it finds a \( T \) model of \( \phi^B_q \) or it finds a set of quantifier-free \( T \) theorems \( A \) such that \( \phi^B_q \land A \) is unsatisfiable. E.g., when \( T[P] \) attempts to find path invariants of path \( q \) of BinarySearch that prove that \( q \) satisfies \( B_{\text{Srch}} \), if \( T[P] \) enumerates the theorems Eqn. 9, then \( T[P] \) determines in its next iteration that \( \phi^B_q \land A \) is an unsatisfiable EUFLIA formula. \( T[P] \) runs an interpolating theorem prover for EUFLIA on \( \phi^B_q \land A \) to obtain path invariants for \( q \), such as the path invariants given in Fig. 2.

The effectiveness of \( T[P] \) depends critically on its ability to efficiently enumerate a set of quantifier-free theorems that is sufficient to refute a model with an inaccurate interpretation of theory functions, and ultimately to support valid path and inductive invariants on resource usage. E.g., \( T[P] \), given path \( q \) of BinarySearch, could generate the valid theorem \( \log 8 = 3 \) to prove that \( q \) satisfies \( B_{\text{Srch}} \), but such a theorem could not be used to synthesize inductive
step-count invariants of BinarySearch that prove that all of its runs satisfy $B_{\text{ret}}$.

To select general theorems, $T \cup \Gamma$ enumerates theorems by grounding a small set of universally-quantified axioms on quantifier-free terms, first enumerating small terms that minimize the use of constants. E.g., to prove that a given path satisfies a bound expressed in terms of $\log$, it enumerates instances of the theorems

$$\forall x. x \leq 0 \rightarrow \log x = 0$$

$$\forall x, y. \log(x \cdot y) = \log x + \log y$$

$$\forall x, y. \log(x/y) = \log x - \log y$$

with $x$ and $y$ grounded to small terms over the logical variables used to model the state of $q$, such as $\text{fst}0, \text{lst}0$, and $\text{lst}0 - \text{fst}0$.

To enumerate a set of theorems that result in path invariants that can be used to construct inductive program invariants, and to do so efficiently, $T \cup \Gamma$ only enumerates theorems over sets of logical variables that model state at some common point in the given control path. I.e., $T \cup \Gamma$ uses the locality of variable uses and definitions along a path to guide theorem enumeration similarly to how safety verifiers use the locality of uses and definitions to guide the selection of path invariants (Henzinger et al. 2004). E.g., when enumerating axioms for path $q$ of BinarySearch, $T \cup \Gamma$ enumerates theorems defined over logical variables $\text{fst}0$ and $\text{lst}0$ (which model state at points $b$ and $c$) or $\text{fst}1$ and $\text{lst}0$ (which model state at points $d$ and $e$), but not $\text{lst}0$ and $\text{fst}1$ or $\text{cost}0$ and $\text{cost}1$. The locality-guided enumeration strategy is given in detail in §4.3.2.

### 3. Background

In this section, we define a target language that we use to present our approach (§3.1). We then review previous foundations of formal logic (§3.2).

#### 3.1 Program Structure

A program is a set of statements, each of which tests and updates state, calls, or returns. Let $\text{procs}$ be a space of procedure names; let $\text{Locs}_{\text{br}}, \text{Locs}_{\text{call}}$, and $\text{Locs}_{\beta}$ be disjoint sets of branch, call, and return control locations, and let the set of all control locations be denoted $\text{Locs} = \text{Locs}_{\text{br}} \cup \text{Locs}_{\text{call}} \cup \text{Locs}_{\beta}$, with a distinguished initial location $\text{INIT}$ and final location $\text{FINAL}$. Let proc : $\text{Locs} \rightarrow \text{procs}$ map each control location to the procedure that contains it, with proc($\text{INIT}$) = proc($\text{FINAL}$). Let entry : $\text{procs} \rightarrow \text{Locs}$ map each procedure to its entry control location and exit : $\text{procs} \rightarrow \text{Locs}$ map each procedure to its exit control location. The space of all variables is denoted $\text{Vars}$, and contains the distinguished cost variable cost $\in \text{Vars}$. The space of all instructions is denoted $\text{Instrs}$. For each control location $L \in \text{Locs}$ and sequence of locations $s \in \text{Locs}^*$, $s$ is an immediate suffix of $L :: s$.

A program statement either tests and updates state, calls a procedure, or returns from a call. A pre-location, instruction, and branch-target-location is a branch statement; i.e., the space of branch statements is denoted $\text{BrS} = \text{Locs}_{\beta} \times \text{Instrs} \times \text{Locs}$. For each branch statement $b \in \text{BrS}$, the pre-location, instruction, and post-location of $b$ are denoted $\text{PreLoc}[b], \text{Instr}[b]$, and $\text{BrTgt}[b]$, respectively.

A pre-location, call-target procedure name, and return-target control location are a call statement; i.e., the space of call statements is denoted $\text{Calls} = \text{Locs}_{\text{br}} \times \text{procnms} \times \text{Locs}$. For each call statement $c \in \text{Calls}$, the pre-location, call target, and return target of $c$ are denoted $\text{PreLoc}[c], \text{CallTgt}[c]$, and $\text{RetTgt}[c]$, respectively.

The entry point of $c$ is denoted $\text{entry}(c) = \text{entry}(\text{CallTgt}[c])$.

A return location represents a return statement; i.e., the space of return statements is denoted $\text{Rets} = \text{Locs}_{\text{br}}$.

The space of all statements is denoted $\text{Stmts} = \text{Calls} \cup \text{Rets}$. Each control location is the target of either potentially-many branch statement or exactly one call statement. For each call statement $c \in \text{Calls}$ and return statement $r \in \text{Rets}$ such that $\text{CallTgt}[c] = \text{proc}(\text{PreLoc}[r])$, $r$ returns to $c$. A program $P$ is a set of statements in which for each branch location $L \in \text{Locs}_{\beta}$ and location $L' \in \text{Locs}$, there is at most one branch statement, denoted $\text{BrAt}[P](L, L')$ with $\text{PreLoc}[b] = L$ and $\text{BrTgt}[b] = L'$. The language of programs is denoted $\text{L}$.}

#### 3.2 Program Semantics

A run of a program $P$ is a sequence of stores that are valid along an interprocedural path of $P$. A nesting relation over indices models the matched calls and returns of along a control path (Alur and Madhusudan 2009). For each $n \in \mathbb{N}$, let the space of positive integers less than $n$ be denoted $\mathbb{Z}_n$.

**Definition 1.** For each $n \in \mathbb{N}$ and $j \in \mathbb{Z}_n \times \mathbb{Z}_n$ such that for all indices $i_0, i_1, i_2 \in \mathbb{Z}_n$ with $i_0 < i_1 < i_2$, either $i_0 < i_1 < i_2$, $i_1 < i_0 < i_2$, or $i_1 < i_0 < i_2$, $i_2 < i_0 < i_1$, $\sim$ is a nesting relation over $n$.

For each $n \in \mathbb{N}$, the nesting relations over $\mathbb{Z}_n$ are denoted $\text{Nestings}[n]$. For each $i, j < n$ and nesting relation $\sim \in \text{Nestings}[n]$, we denote $(i,j) \sim$ alternatively as $i \sim j$.

A control path is a sequence of control locations visited by a sequence of branch statements, calls, and matching returns.

**Definition 2.** Let program $P \in \mathcal{L}$ and control locations $L = [L_0, \ldots, L_{n-1}] \in \text{Locs}^*$ be such that the following conditions hold.

1. For each $0 \leq i < n$ such that $L_i \in \text{Locs}_{\text{br}}$, there is a branch statement $b \in P$ such that $\text{PreLoc}[b] = L_i$ and $\text{BrTgt}[b] = L_{i+1}$.

2. There is a nesting relation $\sim \in \text{Nestings}[n]$ such that the domain and range of $\sim$ are exactly the indices of the call and successors of return locations in $L$. For all $0 \leq i < j < n$ such that $i \sim j + 1$, there is some call statement $c \in P$ such that $i+1 = \text{entry}(c)$ and $L_{i+1} = \text{RetTgt}[c]$, and some return statement $r \in P$ such that $L_j = \text{PreLoc}[r]$.

Then $[L_0, \ldots, L_{n-1}]$ is a path of $P$.

For each program $P \in \mathcal{L}$, the space of paths of $P$ is denoted $\text{Paths}[P]$, and the set of all paths is denoted $\text{Paths}$. For each path
An evaluation of all variables in $Vars$ and $Voc$.

The transition relation of an instruction need not be total: thus, $Stores$ of $T$ to a formula $φ$ terms $T$ in $theory$ of equal length, such that adjacent stores in $Σ$ last $i$ let the caller is denoted $ρ$ each calling store, exit store of a callee, and resulting return store in $denotes$ that act as $b$ that operate on only integers in order to simplify the presentation.

We add to integers, but we present $C$ the space of values is $Locs$ $p$ $σ$ $⊆$ $D$ $,$ . . . , $y$ $domain$ $X$ $is$ the restriction of the standard model of $T$ and a map from each $BrAt$ $i < n$ $|$ $Y$ $∈$ $x$ $is$ denoted $σ, σ$ $⊆$ $s$ $P$ $s$ $| Brs$ $∈$ $IA$ $σ, σ$ $ψ$ $if$ and only if $m^σ, m^σ' ⧵ ψ[1]$. There is a formula $ψC ∈ Forms[IA](Vars, Vars')$ such that for all stores $σ, σ'$ $∈$ $Stores$, $(σ, σ') ∈ ρC$ if and only if $m^σ, m^σ' ⧵ ψC$. There is a formula $ψR ∈ Forms[IA](Vars, Vars, Vars2, Vars3) such that for all stores $σ, σ, σ_2 ∈ Stores$, $(σ, σ_1, σ_2) ∈ ρB$ if and only if $m^σ, m^σ, m^σ_2 ⧵ ψR$.

Each program $P$ and logical formula $B$ over the initial state of $P$ and final value of the variable cost defines a bound-satisfaction problem. The problem is to decide if over each run $r$ of $P$, the initial state of $P$ and the value of cost in the final state of $r$ satisfy $B$. For theory $T$, the space of bound constraints is denoted $Bounds[T] = Forms[T](Vars ∪ \{cost\})$.

For each instruction $i$ in Instrs, there is a formula $ψ[i] ∈ Forms[IA](Vars, Vars')$ such that for all stores $σ_1, σ_2 ∈ Stores$. $σ, σ'$ $∈$ $ρC$ if and only if $m^σ, m^σ' ⧵ ψ[i]$. There is a formula $ψC ∈ Forms[IA](Vars, Vars', Vars2, Vars3)$ such that for all stores $σ_1, σ_2 ∈ Stores$, $(σ_1, σ_2) ∈ ρB$ if and only if $m^σ_1, m^σ_2 ⧵ ψC$. There is a formula $ψR ∈ Forms[IA](Vars, Vars, Vars2, Vars3) such that for all stores $σ_1, σ_2 ∈ Stores$, $(σ_1, σ_2) ∈ ρB$ if and only if $m^σ_1, m^σ_2 ⧵ ψR$.

Each program $P$ and formula $B$ over the initial state of $P$ and the final value of the variable cost defines a bound-satisfaction problem. The problem is to decide if over each run $r$ of $P$, the initial state of $P$ and the value of cost in the final state of $r$ satisfy $B$. For theory $T$, the space of bound constraints is denoted $Bounds[T] = Forms[T](Vars ∪ \{cost\})$.

4. Definition. For each extension $T$ of LIA, program $P$ ∈ $Bounds[T]$, and path $q$ ∈ $Paths[P]$, if for each run $r ∈ Runs[q]$, it holds that $n_{head(r)}$ $→ n_{last(r)}(cost) ⧵ B$, then $ψ$ satisfies $B$. For each path $q ∈ Paths[P]$, it holds that $ψ$ satisfies $B$, then $P$ satisfies $B$, denoted $P ⧵ B$. The bound-satisfaction problem $(P, B)$ is to determine if $P ⧵ B$.

While we present our verifier CAMPY for a simple language whose semantics can be modeled using only LIA, practical languages typically provide features that can only be directly modeled using LIA in combination with the theories of uninterpreted functions and the theory of arrays. The complete implementation of CAMPY supports such language features (see §5).

3.2.2 Interpolation

Tree-interpolation problems formulate the problem of finding valid invariants for all runs of a particular program path that contains calls and returns (Heizmann et al. 2010b). The branching structure in the tree-interpolation problem models the dependency of the result of a function call on the effect of the path through the callee combined with the arguments provided to the callee by the caller.
Definition 5. For theory $T$, a $T$-tree-interpolation problem is a triple $(N, E, C)$ in which:

- $N$ is a set of nodes.
- $E \subseteq N \times N$ is a set of edges such that the graph $T = (N, E)$ is a tree with root $r \in N$.
- $C \colon N \to \text{Forms}[T](X)$ assigns each node to an $T$ constraint.

For each tree-interpolation problem $T = (N, E, C)$, an interpolant of $T$ is an assignment $I : N \to \text{Forms}[T](X)$ from each node to a $T$ formula such that:

- The interpolant at the root $r \in N$ of $T$ entails False. I.e., $I(r) \models T$ False. For the interpolant.
- For each node $n \in N$, the interpolants at the children of $n$ and the constraint at $n$ entail the interpolant at $n$, i.e., $I(m) \models (m, n) \in E$, $C(n) \models T$ I(n).
- For each node $n$, the vocabulary at $n$ is the common vocabulary of all descendants for $n$ and all non-descendants of $n$. The vocabulary at the interpolant is contained in the vocabulary of $n$, i.e., $\text{Voc}(n) = \bigcup \text{Voc}(C(m)) \cap \bigcup \text{Voc}(C(m'))$.

For theory $T$ and variables $X$, the space of all tree-interpolation problems whose constraints are $T$ formulas over $X$ is denoted $\text{Irr}[T, X]$. For each tree-interpolation problem $T, X \in \text{Irr}[T, X]$, the nodes, edges, root, and constraints of $P$ are denoted $N[T], E[T], r[T]$, and $C[T]$, respectively. The conjunction of all constraints in $T$ is denoted $\text{Crt}[T] = \bigwedge_{n \in N[T]} C(n)$. A model of $\text{Crt}[T]$ is referred to alternatively as a model of $T$.

For each tree-interpolation problem $T$ and node $n \in N[T]$, the tree-interpolation problem formed by the restriction of $T$ to the sub-tree with root $n$ is denoted $T|_n$. The procedure $\text{TMRG}$ takes two tree-interpolation problems $(N, E, C)$ and $(N', E', C')$ with $(N, E)$ a subtree of $(N, E)$ and constructs a tree-interpolation problem in which the constraint for each node is the conjunction over constraints for all nodes in $N'$. I.e., $\text{TMRG}((N, E, C), (N', E', C')) = (N, E, C'')$ with $C''(n) = C(n)$ for each $n \in N \setminus N'$ and $C''(n) = C(n) \land C'(n)$ for each $n \in N'$.

For theory $T$ and $T$-interpolation problems $U_0$ and $U_1$ containing the same nodes and edges, $U_0$ is as weak as $U_1$ if for each node $n$, the constraint in $U_0$ for $n$ is as weak as the constraint for $n$ in $U_1$ and the vocabulary of the constraint in $U_1$ is contained by the vocabulary of the constraint in $U_0$.

Definition 6. For theory $T$, variables $X$, and $U_0, U_1 \in \text{Irr}[T, X]$, if for $N = N[T] = N[U], E[T] = E[U]$, and for each $n \in N$, ($I$) $\text{Crt}[U_0](n) \models \text{Crt}[U_1](n)$ and ($2$) $\text{Voc}(\text{Crt}[U_0](n)) \subseteq \text{Voc}(\text{Crt}[U_1](n))$, then $U_1$ is as weak as $U_0$.

Because weaker interpolation problems have weaker constraints per node, they admit fewer interpolants.

Lemma 1. For theory $T$, variables $X$, and $U_0, U_1 \in \text{Irr}[T, X]$ with common nodes $N$ such that $U_1$ is as weak as $U_0$, and all $I : N \to \text{Forms}[T](X)$ such that $I$ is an interpolant of $U_1$, $I$ is an interpolant of $U_0$.

For theory $T$, an interpolating theorem prover takes a $T$-interpolation problem $T$ and returns either a $T$-model or a map from the nodes of $T$ to $T$-formulas. An interpolating theorem prover is sound if it only returns a valid model or interpolant of its input.

Definition 7. For theory $T$, variables $X$, let effective procedure $t : \text{Irr}[T, X] \to (X \to \text{Dom}[T]) \cup (N[T] \to \text{Forms}[T](X))$ be such that for each tree-interpolation problem $U \in \text{Irr}[T, X]$, ($I$) if $t(U) : X \to \text{Dom}[T]$, then $t(U)$ is a model of $U$; ($2$) if $t(U) : N[T] \to \text{Forms}[T](X)$, then $t(U)$ are interpolants of $U$. Then $t$ is a sound interpolating theorem prover for $T$.

Previous work has presented an algorithm $\text{EUF-LIA-ITP}$ that solves a given $\text{EUF-LIA}$ tree-interpolation problem $T$ by invoking an interpolating theorem prover for $\text{EUF-LIA}$ a number of times bounded by $|N[T]|$ (Heizmann et al. 2010). In §4 we describe an approach for proving that a program whose semantics are expressed in a theory $\mathcal{T}$ satisfies a bound expressed in an extension $T$. To simplify the presentation of our approach, we fix $\mathcal{T}$ to be LIA, and fix $T$ to be an arbitrary extension of LIA. However, our approach can be applied using any theory for $\mathcal{T}$ that satisfies the above conditions: in particular, our actual implementation of CAMPY uses the combination of the theories of linear arithmetic, uninterpreted functions with equality, and arrays as its base theory.

4. Technical Approach

In this section, we describe a verifier CAMPY for the bound-satisfaction problem. In §4.1, we define the summaries that CAMPY attempts to infer to prove that a given program satisfies a given bound. In §4.2, we give the design of CAMPY’s core verification algorithm. In §4.3, we give the design of a tree-interpolating theorem prover, which is used by CAMPY to prove that individual paths satisfy a given bound. In §4.4, we discuss key properties of CAMPY.

4.1 Summaries

CAMPY, given a program $P$ and bound $B$, attempts to infer summaries of the behavior of $P$ that imply that all paths of $P$ satisfy $B$. A program summary is a map from each control location $L$ to a symbolic summary of the effects of all runs from the entry point of $L$’s procedure to $L$. The space of program summaries is denoted $\text{ProgSums} = \text{Locs} \to \text{Summaries}$. Program summaries are inductive for $P$ and $B$ if they imply that all runs of $P$ satisfy $B$.

Definition 8. For program $P \in \mathcal{L}$ and bound constraint $B \in \text{Bounds}[T]$, let $S \in \text{ProgSums}$, be such that:

($1$) For each procedure $f \in \text{procs}$, $\text{Vars} = \text{Vars}' \models S(\text{entry}(f))$.

($2$) $S(\text{FINAL}) \models \text{Bounds}[T]$.

($3$) For each branch statement $b \in P$, $S(\text{PreLoc}[b])[\text{Var}, 1], \psi[b][\text{Var}_1, \text{Var}_2] = S(\text{BrTgt}[b])[\text{Var}_0, \text{Var}_2]$

($4$) For each call statement $c \in P$ and each return statement $r \in P$ that returns to $c$, $S(\text{PreLoc}[c])[\text{Var}_0, \text{Var}_1], \psi[c][\text{Var}_1, \text{Var}_2], S(r)[\text{Var}_2, \text{Var}_3], \psi[r][\text{Var}_1, \text{Var}_3, \text{Var}_4] = S(\text{RetTgt}(c))[\text{Var}_0, \text{Var}_4]$

Then $S$ are inductive step-count summaries for $P$ and $B$.

The space of inductive summaries for program $P \in \mathcal{L}$ and bound constraint $B \in \text{Bounds}[T]$ is denoted $\text{Ind}[P, B]$.

Example 1. The step-count invariants given in §2.2 directly define inductive step-count summaries for BinarySearch and $B_{\text{sum}}$.

Inductive summaries are evidence of bound satisfaction.

Definition 9. For each path $p$, a visible suffix of $p$ is a sequence of locations in $p$ connected over only branch edges, nesting edges, and return edges.

Definition 9. For each path $p$ in Paths, let $L \in \text{Locs}^*$ be such that $\text{last}[L] = \text{FINAL}$ and there is some function $m : \text{Locs} \to \text{Locs}$ such that for each $0 \leq i < |L|$, $\text{Loc}[i] = p(m(i))$, if $\text{Loc}[i] \in \text{Locs}_B$ or
Let $L[i] \in \text{Locs}_R$, then $L[i + 1] = p[m(i) + 1]$, and if $L[i] \in \text{Locs}_C$, then for $j < |p|$ such that $i \sim_p j$, $L[i + 1] = p[j]$. Then $L$ is a visible suffix of $p$.

For each path $p \in \text{Paths}$, the set of visible suffixes of $p$ be denoted $\text{Suffixes}[p] \subseteq \text{Locs}$. For program $P \in L$, the visible suffixes of all paths of $P$ are denoted $\text{Suffixes}[P] = \bigcup_{p \in \text{Paths}[P]} \text{Suffixes}[p]$.

Path summaries are sets of visible suffixes of a program’s paths, with each visible suffix $s$ mapped to a summary of the effect of all runs of $s$.

**Definition 10.** For program $P \in L$, let $Q \subseteq \text{Suffixes}[P]$ be visible suffixes of paths of $P$, and let $S : Q \rightarrow \text{Summaries}$. Then $(Q, S)$ are visible suffix summaries of $P$.

For program $P \in L$, the space of all visible suffix summaries of $P$ is denoted $\text{VisSum}[P]$. For all visible suffix summaries $S \in \text{VisSum}[P]$, the visible suffixes and summary map of $S$ are denoted $\text{Suffixes}[S]$ and $\text{Summs}[S]$.

If visible suffix summaries soundly model the semantics of the paths of which the summaries are subsequences, then the summaries are valid.

**Definition 11.** For program $P \in L$, let visible-suffix summaries $S \in \text{VisSum}[P]$ be such that for each visible suffix $s \in \text{Suffixes}[S]$, (1) for each procedure $f \in \text{procs}$, if $\text{Head}[s] = \text{entry}(f)$, then
\[
\text{Vars} = \text{Vars}' || \text{Summs}[S](s)
\]
(2) for each branch statement $b \in P$ such that $\text{BrTgt}[b] = \text{Head}[s]$ and $s_0 = \text{PreLoc}[b] :: s \in P$,
\[
\text{Summs}[S](s_0)\text{[Vars, }\text{Vars}_1, \psi[b][\text{Vars}_1, \text{Vars}_2] = \text{Summs}[S](s)[s][\text{Vars}_2]
\]
(3) and each call statement $c \in P$ such that $s_0 = \text{PreLoc}[c] :: s \in \text{Suffixes}[S]$ and return statement $r \in P$ such that $\text{proc}(c) = \text{proc}(r)$ and $s_1 = r :: s \in \text{Suffixes}[S]$,
\[
\text{Summs}[S](s_0)\text{[Vars, }\text{Vars}_1, \psi[c][\text{Vars}_1, \text{Vars}_2] = \text{Summs}[S](s)[s][\text{Vars}_2]
\]

**Example 2.** In §2.3 [Fig. 2] contains path invariants for path $q$ that define valid visible-suffix summaries of BinarySearch over all suffixes of $q$.

Path summaries define inductive summaries of $P$ when they define summaries of all paths of $P$.

**Definition 12.** For each program $P \in L$ and bound constraint $B \in \text{Bounds}[T]$, let $S \in \text{VisSum}[P]$ be valid suffix summaries. Let $S' : \text{ProgSum}$ be such that for each location $L \in \text{Locs}$,
\[
S'(L) = \bigvee \{\text{Summs}[S](t) | t \in \text{Suffixes}[S], \text{Head}[t] = L\}
\]
If $S'$ are inductive summaries for $P$ and $B$ (Defn. 8), then $S$ are inductive visible-suffix summaries for $P$ and $B$.

**CAMPY** attempts to prove that a given program $P$ satisfies a given resource bound $B$ by inferring inductive visible-suffix summaries for $P$ and $B$.

**4.2 Verification Algorithm**

Alg. 1 contains the pseudocode for the core algorithm of **CAMPY**, which takes a program $P$ and bound $B$ and attempts to decide if $P$ satisfies $B$. To do so, **CAMPY** uses a counterexample-guided refinement loop, similar to conventional automatic safety verifiers [Henzinger et al. 2002, 2004; McMillan 2006]. **CAMPY** defines and uses an auxiliary function $\text{Aux}$, which takes visible-suffix summaries $S$ and attempts to construct visible-suffix summaries over an extension of the visible-suffix summaries of $S$ that are inductive (§4.1).

**Algorithm 1:** **CAMPY**: a bound-satisfaction verifier for bounds expressed in an extension of LIAX. **CAMPY** uses procedures $\text{CHKIND}[P, B]$ (summarized in §4.2) and $\text{Tp}[T]$, which is presented in detail in [§4.2.1].
The formulas that label nodes in Fig. 2 correspond to visible-suffix summaries of a tree-interpolation problem that is as weak as $U$ but is not satisfied by $m$ under EUFIA.

\[\begin{align*}
\text{Procedure } \text{TP}[U](V) & : \text{A } T - \text{tree-interpolation problem } U. \\
\text{Output} & : \text{Either interpolants for } U \text{ or HasModel to denote that } U \text{ has a } T - \text{model.}
\end{align*}\]

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure \text{TP}[U](V) \EndProcedure
\end{algorithmic}
\end{algorithm}

\textbf{Algorithm 2:} \text{TP}[U]: an interpolating theorem prover for $T$.

- For each branch statement $b \in \text{Brs}$ such that $\text{BrTgt}[s] = \text{Head}[b]$ and $s' = \text{PreLoc}[b] :: s \in \text{Suffixes}[q]$, let
  \[C'(s) = \psi_b[\text{Vars}_{s'}, \text{Vars}_s] \]
- For each call statement $c \in \text{Calls}$ such that $\text{CallTgt}[c] = \text{Head}[s]$ and $s_0 = \text{PreLoc}[c] :: s \in \text{Suffixes}[q]$ and return statement $r \in \text{Rets}$ such that $s_1 = r :: s \in \text{Suffixes}[q]$ and $s_2 \in \text{Suffixes}[q]$ is the maximal extension of $s_1$ in $\text{Suffixes}[q]$, let
  \[C'(s) = \psi_C[\text{Vars}_{s_0}, \text{Vars}_{s_2}] \]

Then $C([\text{FINAL}]) = C'([\text{FINAL}]) \land B[\text{Vars}_{\text{FINAL}}/\text{Vars}'/\text{Vars}_{s_0}, \text{Vars}_{s_2}]$.

Each interpolant of $C_{\text{Tree}}[P, B, q]$ defines valid visible-suffix summaries of $q$. In particular, for each map $I : N \rightarrow \text{Suffixes}[q]$, let $S_I : \text{Suffixes}[q] \rightarrow \text{Summaries}$ be such that for each $s \in \text{Suffixes}[q]$ with maximal extension $s_0 \in \text{Suffixes}[q]$,
  \[S_I(s) = I(s)[\text{Vars}, \text{Vars}'/\text{Vars}_{s_0}, \text{Vars}_s] \]

\textbf{Lemma 3.} For each program $P \in \mathcal{P}$, path $q \in \mathcal{P} \mathcal{A} \mathcal{T}[P]$, bound $B \in \mathcal{B}[T]$, and constraint map $I : \text{Suffixes}[q] \rightarrow \text{Suffixes}[q]$, if $I$ are interpolants of $C_{\text{Tree}}[P, B, q]$, then $S_I$ are valid visible-suffix summaries of $q$ (Defn. 17).

A proof of Lemma 3 follows immediately from previous work on tree interpolation (Heizmann et al. 2010).

\textbf{Example 3.} In Fig. 2 depicts a linear tree-interpolation problem $T_q$ for which each model defines a ran of path $q$ that breaks bound $B$. Each node of $T_q$ is a point in the exchange path $p$, each edge that $T_q$ an edge in the path. The constraint for each node $n$ is the symbolic transition relation of the instruction with destination $n$. The formulas that label nodes in Fig. 2 correspond to visible-suffix summaries of $q$ defined by interpolants of $T_q$.

\subsection{4.3 Tree Interpolation by Theorem Enumeration}

In this section, we describe the design of a tree-interpolating theorem prover $\text{TP}[T]$ for $T$. In §4.3.1, we describe the core algorithm of $\text{TP}[T]$. In §4.3.2, we describe a key procedure used by $\text{TP}[T]$ to generate $T$ formulas to search for a model or interpolants of a given tree-interpolating problem $T$.

\subsection{4.3.1 The Algorithm-Proving Algorithm}

\textbf{Alg. 2} contains pseudocode for $\text{TP}[T]$, an interpolating theorem prover for $T$. $\text{TP}[T]$ takes a $T$-interpolation problem $U$ and returns either the value HasModel to denote that $U$ has a $T$-model or interpolants for $U$. $\text{TP}[T]$ defines a procedure $\text{TPaux}[T]$ (line 2) that takes a $T$-interpolation problem $V$ that is as weak as $U$ and returns either HasModel or $\text{TPaux}[T]$ of $V$. $\text{TP}[T]$ calls $\text{TPaux}[T]$ on $U$ and returns the result (line 13).

$\text{TPaux}[T]$ determines if $V$ has an EUFIA model by running EUFIA SAT on $V$ (line 3). EUFIA SAT is introduced in §3.2. If EUFIA SAT returns that $V$ has no EUFIA model, then $\text{TPaux}[T]$ runs an interpolating theorem prover EUFIA ITP (introduced in §3.2.2) on $V$ and returns the resulting interpolants (line 4).

Otherwise, if $\text{TPaux}[T]$ returns an EUFIA model $m$ of $V$ (line 5), then $\text{TPaux}[T]$ collects the nodes $N' \subseteq N[V]$ whose clauses are not satisfied by $m$, the restriction of $m$ to the variables that occur in $\text{V}$ combined with the standard model of $T$ (line 6). $N'$ is empty (i.e., $m$ satisfies $U$), then $\text{TPaux}[T]$ returns that $V$ has a model (line 7). Otherwise, $\text{TPaux}[T]$ constructs a tree-interpolation problem $V'$ that is as weak as $V$ but for which $m$ is not an EUFIA model by running the procedure $\text{GRN}[T]$ on $V[\text{Vars}_s/\text{Vars}_s]$ (line 9). $\text{GRN}[T]$ is described in §4.3.2. $\text{TPaux}[T]$ merges $V'$ with $V$, recurses on the result, and returns the result of the recursion (line 10).

\textbf{Example 4.} §3.3 describes a run of $\text{TP}[T]$ on the tree-interpolation problem for path $q$ of program BinarySearch. $\text{TP}[T]$ first obtains a model $m$ of the tree-interpolating problem for $q$, $T_q$, under EUFIA (Algorithm 2, line 3). $\text{TP}[T]$ determines that $m$ does not satisfy the clause for the node of Fig. 2 under the standard model of nonlinear arithmetic (line 6), chooses $q$ as the unique node in $T_q$ that is not satisfied by $m$, and runs $\text{GRN}[T]$ on $T_q$ restricted to $q$ (i.e., $T_q$ itself). $\text{GRN}[T]$ generates a weaker tree-interpolation problem described in §4.3.2, Ex. 6, which $\text{TPaux}[T]$ merges with $T_q$, and recurses on (line 10).
4.3.2 Enumerating Quantifier-Free $T$-Theorems

Alg. 3 contains pseudocode for GRND[$T$] (line 1) [line 13]. GRND[$T$] takes a $T$-interpolation problem $U \in \text{FTP}[T, X]$ over variables $X$ and assignment $m : X \rightarrow \text{Dom}[T]$ and returns a $T$-tree-interpolation problem that is as weak as $U$ but for which $m$ is not a model under EUFLIA. GRND[$T$] uses a procedureEnum[$T$, $X$] that takes quantifier-free $T$ formulas $A$ over variables $X$ and enumerates a quantifier-free $T$ formula over $X$ not in $A$. GRND[$T$] constructs a map $\mathcal{V} : \mathcal{N}[U] \rightarrow \mathcal{P}(X)$ from each node $n \in \mathcal{N}[U]$ to the vocabulary of $n$ (line 2), the vocabulary of a node in an interpolation problem is defined in [§3.2, Defn. 5]. GRND[$T$] then runs a procedure EVALCTR that takes an assignment $m : X \rightarrow \text{Values}$ and returns a constraint that each variable in $X$ equal to its value under $m$ (line 3). I.e.,

$$\text{EVALCTR}(m) \equiv \bigwedge_{x \in X} \{ x = \text{term}[m(x)] \}$$

GRND[$T$] sets $E$ to the result of running EVALCTR on $m$.

Example 5. When $T\!P[T]$ is run to find path invariants of path $q$ of BinarySearch ($§2.3$), it enumerates non-linear theorems until it finds a set that is not satisfied by the model $m$, which assigns cost$_1$ to 2, len(arr) to 4. When $T\!P[T]$ runs GRND[$T$], EVALCTR generates clause $\text{cost}_1 \equiv 2 \land \text{len}(arr) = 4$.

GRND[$T$] defines a procedure GAUX[$T$] (line 4) [line 12] that uses a set of quantifier-free $T$ theorems to find a $T$-tree-interpolation problem that is as weak as $U$ but is not satisfied by $m$ under EUFLIA. GAUX[$T$], given quantifier-free formulas $A$, first collects the set $A_T$ of formulas in $A$ that each have a vocabulary contained by the vocabulary of some set in $V$ (line 3). GAUX[$T$] then checks if there is a model of the constraints of $T$, $X$, and all theorems in $A_T$ (line 6). If so, then GAUX[$T$] recurses on $A$ extended with the next enumerated theorem of $T$ (line 7).

Otherwise, if $\text{Ctr}[U]$, $E$, and $A_T$ are not mutually satisfiable, then GAUX[$T$] constructs a minimal subset $A'_T$ of $A_T$ that is mutually unsatisfiable with $\text{Ct}[U]$ and $E$ (line 9). GAUX[$T$] returns the given tree-interpolation problem $U$, updated with a constraint map $C'$ that maps each node $n$ of $U$ to the conjunction of its constraint in $U$ and the conjunction of all theorems in $A'_T$ whose vocabularies are contained in the set of variables in the vocabulary of $n$ (line 10) [line 12].

Example 6. When $T\!P[T]$ runs GRND[$T$] on the tree-interpolation problem $\text{CTree[BinarySearch, B}_{\text{sch}}, q]$ with variables $X$ for the path $q$ of BinarySearch ($§2.3$), GRND[$T$] constructs the evaluation clause $E$ (Ex. 5) and runs GAUX[$T$]. For $T$ the theory of non-linear arithmetic, GAUX[$T$] iteratively runs Enum [$T$, $X$], which generates $T$ formulas over $X$, such as $\text{Eqn. 9} \text{ Eqn. 11}$ and $\text{log}(\text{fst}_0 \cdot \text{fst}_1) = \text{log} \text{fst}_0 + \text{log} \text{fst}_1$.

T{x \in X} \{ x = \text{term}[m(x)] \}$

Of the enumerated theorems, $A_T$ (defined at [Alg. 3], line 3) contains, e.g., $\text{Eqn. 11}$ because $\text{fst}_0 \cdot \text{fst}_1 = \text{log} \text{fst}_0 + \text{log} \text{fst}_1$ because $\text{fst}_0 \cdot \text{fst}_1$ is not in the vocabulary of any node in $\text{CTree[BinarySearch, B}_{\text{sch}}, q]$. GRND[$T$] enumerates theorems of non-linear arithmetic over $X$ until it enumerates a set that is mutually unsatisfiable with $\text{CTree[BinarySearch, B}_{\text{sch}}, q]$ (given in [§3.2]) and $E$. One such set are the theorems $\text{Eqn. 9} \text{ Eqn. 12}$ given in [§3.2].

When GRND[$T$] returns, it always generates a tree-interpolation problem that is as weak as its input.

Lemma 4. For all variables $X$, each tree-interpolation problems $U \in \text{FTP}[T, X]$, and assignment $m$ over $X$, if GRND[$T$] terminates on $U$ and $m$, then GRND[$T$]($U, m$) is as weak as $U$.

Proof. Proof by induction on the evaluation of GAUX[$T$] on its argument $A$. The claim proved by induction is that each formula in $A$ is a theorem of $T$. For the base step, GAUX[$T$] is initially called on the empty set of theorems (Alg. 3, line 13), and the claim is trivially satisfied. For the inductive step, by the inductive hypothesis, $A$ contains only $T$-theorems. In the next step of evaluation, GAUX[$T$] is called with a set consisting of $A$ and a formula generated by Enum[$T, X$], which by assumption generates only $T$-theorems (§3.2). Thus, in the next step of evaluation of GAUX[$T$]. $A$ contains only $T$-theorems.

GAUX[$T$] returns a tree-interpolation problem with nodes and edges identical to $U$, and a constraint map $C'$ that maps each node $n$ to a conjunction of its constraint in $U$ and formulas in $A$. The fact that each formula in $A$ is a $T$ theorem implies that $n$ is bound to a constraint that is no stronger under $T$ than its constraint in $U$.

GAUX[$T$] includes a formula in $A$ in the constraint $C'(n)$ only if its vocabulary is a subset of the vocabulary for $n$ (line 3) and (line 10). Thus the vocabulary of $C'(n)$ the vocabulary of $\text{Ct}[T](n)$, $U'$ as weak as $U$, by definition (§3.2, Defn. 6).

$T$ is as weak as $T'$ as well, although this fact is both trivial from the construction of $T'$ from $T$, and not required to prove the soundness of CAMPY. Furthermore, GRND[$T$], given tree-interpolation problem $T$ and model $m$, always generates a tree-interpolation problem for which $m$ is not a model. However, this fact is not required in order to prove the soundness of CAMPY (§3.4, Thm. 1), so we withhold a complete statement and proof.

A consequence of Lemma 4 is that $T\!P[T]$ is a sound interpolating theorem prover for $T$.

Lemma 5. $T\!P[T]$ is a sound interpolating theorem prover for $T$ (§3.2, Defn. 7).

Proof. For $T$-tree-interpolation problem $U, proof by induction on the evaluation of $T\!P[T](U)$. The claim established by induction is that $V$ (Alg. 3, line 2) is as weak as $T$. For the base step of the claim’s proof, $U$ is set to $T$ at line 13. For the inductive step of the claim’s proof, by the inductive hypothesis, $U$ is as weak as $T$. By GRND[$T$]($U, m$) is as weak as $U, m$, and by the definition of TMrg ($§3.2.2$), $U' = \text{TMrg}(U, \text{GRND}[T](U, m))$ is as weak as $U$. Thus, in $U'$’s next step of evaluation, $U$ is as weak as $T$.

$T\!P[T]$ only returns a constraint map if it represents interpolants of $U$ under EUFLIA (line 4). Each interpolant of $U$ under EUFLIA is an interpolant of $U$ under $T$. The fact that $U$ is weaker than $T$. Combined with Lemma 1 (see §3.2), implies that if $T\!P[T]$ returns a constraint map $T, then T$ are valid interpolants of $T$.

Because $T\!P[T]$ is a valid decision procedure for theories that may not be decidable, $T\!P[T]$ is not total: i.e., there are tree-interpolation problems on which it may not terminate.

4.4 Properties

CAMPY is a sound verifier for the bound-satisfaction problem.

Theorem 1. For each program $P \in \mathcal{L}$ and bound $B \in \text{Bounds}[T]$, if CAMPY($P, B$) = True, then $P \vdash B$.

Proof. Proof by induction on the evaluation of CAMPY on $P$ and $B$. The claim proved by induction is that at each step of the evaluation of CAMPY, the visible suffix summaries $S$ are valid (§4.1, Defn. 11). The base step of the proof of the claim follows from the definition of valid summaries and the fact that AUX is called initially with summaries over an empty set of suffixes (§4.2, Alg. 1, line 12).
We performed an empirical evaluation of CAMPY from our results on the effectiveness of CAMPY on selected benchmarks from the LeetCode (leetcode), CodeChef (codechef), and CodeForce (codeforces) coding platforms. For each benchmark, we used a cost model in which each backward branch in a depth-first traversal of the control-flow graph costs a unit of time and every other instruction incurs no cost. The resulting bounds describe a relationship between a program’s input and the number of times that it executes an iterative computation.

We will now illustrate the features and current limitations of CAMPY by discussing three of the benchmark programs: `length0fLIS`, `SmartSieve`, and `Cycle`. CAMPY correctly verified the programs `length0fLIS` and `SmartSieve`, but failed to prove or disprove that `Cycle` satisfied an expected bound.

### 5.1 Benchmarks

In this section, we illustrate the operation of CAMPY on selected benchmarks from the LeetCode (leetcode), CodeChef (codechef), and CodeForce (codeforces) coding platforms. For each benchmark, we used a cost model in which each backward branch in a depth-first traversal of the control-flow graph costs a unit of time and every other instruction incurs no cost. The resulting bounds describe a relationship between a program’s input and the number of times that it executes an iterative computation.

We will now illustrate the features and current limitations of CAMPY by discussing three of the benchmark programs: `length0fLIS`, `SmartSieve`, and `Cycle`. CAMPY correctly verified the programs `length0fLIS` and `SmartSieve`, but failed to prove or disprove that `Cycle` satisfied an expected bound.

#### Longest Increasing Subsequence

The Longest Increasing Subsequence (LIS) Problem is to take an array of integers and return the length of a longest increasing subsequence of the array. `length0fLIS`, given in Fig. 3 was posed to LeetCode as a solution to the LIS problem. `length0fLIS` iterates from 1 to n, and at each intermediate value i, maintains in array res the longest increasing subsequence of nums up to i, and maintains in len the length of the longest increasing subsequence (lines 7—18). If the value in nums at i is greater than the current last element in res, then `length0fLIS` extends res to contain nums[i] (lines 9—11). Otherwise, if the value in nums at i is less than or equal to the last value in res, then `length0fLIS` updates the index in i and an entry of res by performing a binary search over the values in res and nums between i and the length of nums (lines 12—18).

To derive a verified tight bound for `length0fLIS`, we first provided to CAMPY `length0fLIS` and the bound $n^2$, derived from identifying the nested loops of `length0fLIS`; CAMPY verified that `length0fLIS` satisfies the $n^2$ bound. We then provided the bound $n \cdot \log n$, inspired by the observation that the nested loop in `length0fLIS` at lines 14—18 updates a value used in the loop guard with division by a constant; CAMPY verified that `length0fLIS` satisfies the bound $n \cdot \log n$ as well. To determine if `length0fLIS` actually executes each loop in constant time, we provided a bound of n to CAMPY. CAMPY returned an execution of `length0fLIS` on which it does not satisfy the bound n.

#### SmartSieve

The Smart Sieve Problem hosted on CodeChef is to take two integers $\text{MAX}$ and k and return an integer array containing the prime factors of k up to $\text{MAX}$. One solution submitted for the Smart Sieve Problem, `SmartSieve`, is given in Fig. 4. `SmartSieve` first stores the smallest prime factor of every even number up to

```java
public int length0fLIS(int[] nums, int n) {
    if (nums == null || n == 0) return 0;
    int[] res = new int[n];
    int len = 0;
    res[len] = nums[0];
    len++;
    for (int i = 1; i < n; i++) {
        if (nums[i] > res[len - 1]) {
            int mid = (i + j) / 2;
            if (res[mid] < nums[i]) i = mid + 1;
            else j = mid;
        }
        res[j] = nums[i];
    }
    return len;
}
```

Figure 3: `length0fLIS`: a solution to the LIS Problem.
public static int[] SmartSieve(int MAX, int k) {
    boolean[] v = new boolean[MAX];
    int[] sp = new int[MAX];
    for (int i = 2; i < MAX; i += 2) {
        if (!v[i]) {
            sp[i] = 2;
            for (int j = i * 2; j < MAX; j += 2) {
                if (!v[j]) {
                    v[j] = true;
                    sp[j] = i;
                }
            }
        }
    }
    int[] ans = new int[sp.length];
    int j = 0;
    while (k > 1) {
        ans[j++] = sp[k];
        k /= sp[k];
    }
    return ans;
}

Figure 4: SmartSieve: a solution to the Smart Sieve problem.

public static boolean done = false;
public static void Cycle(int v, int x, int k, int[] d, int[] a, List<int[]> g) {
    if (done) return;
    if (d[v] > 0) {
        done = true;
        return;
    }
    d[v] = x;
    for (int i = 0; i < g[v].size(); i++) {
        int cur = (Integer) g[v].get(i);
        a[x + i] = cur;
        Cycle(cur, x + 1, k);
    }
    return;
}

Figure 5: Cycle: a solution to the Cycle of a Graph Problem.

MAX (namely, 2) at its index in sp and then stores the smallest prime factor of every odd number up to MAX at its index in sp (lines 2—12). SmartSieve then computes the prime factors of k by iteratively collecting the smallest prime of k and then dividing k by its smallest prime (lines 15—17).

It is somewhat immediate for an automatic verifier to determine that SmartSieve up to line 12 executes in n^2 steps, but less immediately to prove that a complete run of SmartSieve executes in n^2 + \log n steps. Proving that SmartSieve satisfies such a bound relies on proving that the loop in lines 15—17 executes in \log n steps. CAMPY proves that SmartSieve satisfies such a bound by synthesizing the supporting invariant that at each index k, sp[k] \geq 2.

Cycle of a Graph

An instance of the Cycle of a Graph problem is an undirected graph G consisting of n nodes, where each node is connected to at least k other nodes. A solution is a cycle with at least k + 1 nodes. Figure 5 contains one solution to the problem given on Codeforces. Cycle takes as input an integer v storing the node currently being traversed, x the current recursion depth, k the lower bound on cycle size, d maintains the known depth of each node, a store the list of nodes in the result cycle, and g stores the adjacency list for all nodes in the graph. Cycle maintains a global variable done that stores whether or not it has found a cycle. Cycle outputs all nodes in the cycle to stdout.

Cycle first checks if it has already found a cycle (line 4). If not, then Cycle checks if the depth of v in d is greater than 0 (line 5). If so, then Cycle checks if the difference between the current recursion depth x and the depth of v is greater than k (line 6). If so, Cycle prints the sequence of nodes visited between the depth of v and the current recursion depth (lines 7—8), stores that it has found a cycle (line 9). In either case, Cycle returns (line 10). If the depth of v is not greater than 0, then Cycle stores x as the depth of v (line 11). For each successor cur of v (line 13), it stores cur as the next node visited (line 14) and recurses (line 15).

When we ran CAMPY on Cycle and the expected bound n^2 + k, it failed to prove that Cycle satisfies the bound. CAMPY failed to prove that Cycle satisfied the bound because the execution time of Cycle is determined partly by it’s traversal of linked lists as values in an array, but CAMPY cannot use assumptions about, or synthesize sophisticated invariants describing the size of linked data structures.

5.2 Results and Conclusions

The results of our evaluation are contained in Table 1. The names for most benchmarks given in Table 1 were derived from information in the program source, such as the name of a class or critical method. In cases where a unique name was not apparent from the source, a name of the form Testi was assigned.

Overall, our experience using CAMPY indicates that it can be applied to verify that programs that implement intricate algorithms satisfy or do not satisfy expected resource bounds. As illustrated in [5,11], it was typically straightforward to use an informal understanding of a program module to derive a tight expected bound that CAMPY could verify. Moreover, in cases where an expected bound was not apparent, we were able to use both CAMPY’s ability to verify that a program satisfies a bound and to determine when it does not to find a reasonably tight bound after a few uses. Of the bounds listed in Table 1, the bound for Jewe12 contains a surprising coefficient of 350. This bound is immediate from the fact that Jewe12 contains loops that iterate up to 150 and 200. When run on the benchmark Checkpost, CAMPY timed out as in the case of Cycle, because it was unable to infer sufficiently rich step-counting invariants in terms of the dynamic data structure that it allocated.

In general, the performance of CAMPY strongly indicates that it could be integrated into an automated educational aid for providing immediate feedback to students about the efficiency of the programs. Extending CAMPY to infer invariants over richer classes of data structures could further improve its performance in practice.

6. Related Work

Automated Complexity Analysis

The problem of bound analysis, i.e., taking a program P and inferring a sound bound on the resources used by P, has been the subject of significant previous work. SPEED infers a symbolic bound on the execution time of a program P by (1) instrumenting P to form a new program P’ that stores in counter variables how often it has executed particular control locations, (2) generates numerical invariants over program and counter variables, and (3) from the generated invariants, derives bounds on the execution time of P (Gulwani and Zuleger 2010). Another approach analyzes Java bytecode under a given control model, and reduces the problem of inferring a symbolic bound to solving a set of recursive equations defined from an abstraction interpretation that models the change in size of data objects as a result of each program instruction (Albert et al. 2012). Another approach derives a complexity bound for a given program P by modeling P as a lossy vector addition system (VASS) and constructing a lexicographic linear ranking function that proves that the VASS terminates (Sinn et al. 2016). Another approach infers ranking functions for a program via linear programming and derives a bound from the ranking functions (Alias et al. 2010).

Several approaches derive bounds by extending type systems to contain information about the resources used to perform a
<table>
<thead>
<tr>
<th>Program Structure</th>
<th>Bound</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>LoC</td>
<td>Loops</td>
</tr>
<tr>
<td>Array2</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td>BirthdayCandles</td>
<td>43</td>
<td>4</td>
</tr>
<tr>
<td>Bit</td>
<td>61</td>
<td>2</td>
</tr>
<tr>
<td>CodeChefJava</td>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>DRGNBOOL</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>FibonacciIterative</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>Ideone</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>JewelAndStone</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>Jewel2</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>LIS</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>Loops_1</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>Pie</td>
<td>34</td>
<td>3</td>
</tr>
<tr>
<td>Scroll</td>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>SmartSieve</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>Sweet</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>Test0</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>Test1</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>Test2</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>Test3</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>Test4</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>Test5</td>
<td>77</td>
<td>6</td>
</tr>
<tr>
<td>Test6</td>
<td>36</td>
<td>4</td>
</tr>
</tbody>
</table>

| Table 1: The results of evaluating CAMPY. Each benchmark program is associated with two rows: the first row contains data for verifying that the program satisfies the tightest bound found; the second row contains data for verifying that the program does not satisfy the loosest bound found. The column titled “Name” contains the benchmark’s name; the column titled “LoC” contains the number of lines of source code; the column titled “Loops” contains the number of loops in the benchmark; the column titled “Nesting” contains the maximum nesting depth of loops in the benchmark. The column titled “Bound” contains the bound provided to CAMPY. Under heading “Performance”, the column titled “Time” contains the time used by CAMPY; the column titled “Memory” contains the peak amount of memory used by CAMPY. “-” indicates that CAMPY timed out on the benchmark and did not return a definite result.

One approach presents a type system and inference algorithm that infers polynomial bounds, given the maximal degree of the polynomial of all bound functions, with the restriction that each term in each polynomial contains a single variable (Hoffmann and Hofmann 2010). Further work extends the approach to infer bounds over multi-variate polynomials (Hoffmann et al. 2011). Such approaches have been formulated as compositional rules for deriving bounds of programs, which generate bounds with proofs that can be independently certified efficiently (Carbonneaux et al. 2014, 2015).

CAMPY addresses a problem that is related to, but distinct from, bound analysis, namely bound verification. I.e., CAMPY takes a program \( P \) and an expected bound \( B \) and attempts to decide if \( P \) satisfies \( B \). A key payoff of bound analyses is that their results can potentially provide useful information to a programmer about the resources used by their program, while requiring no additional computation. One approach presents a type system and inference algorithm that infers polynomial bounds, given the maximal degree of the polynomial of all bound functions, with the restriction that each term in each polynomial contains a single variable (Hoffmann and Hofmann 2010). Further work extends the approach to infer bounds over multi-variate polynomials (Hoffmann et al. 2011). Such approaches have been formulated as compositional rules for deriving bounds of programs, which generate bounds with proofs that can be independently certified efficiently (Carbonneaux et al. 2014, 2015).
effort from the programmer. A key payoff of bound verification is that, when successful, it can provide definite information about a critical bound, including concrete counterexamples that prove that a program does not satisfy an expected bound.

We believe that because bound analyses and CAMPY were designed to address distinct problems and are based on complementary techniques (namely, abstract interpretation and type-checking, compared to model checking), further work could improve the state of the art of automated reasoning about program complexity by combining bound analyses with bound verifiers, such as CAMPY. In particular, a bound analysis could be used to find sound initial bounds for a program module, which a bound verifier could further strengthen by enumerating tighter bounds and either verifying them or refuting them. Techniques used in CAMPY for selecting path invariants, in particular the grounding algorithm at the core of CAMPY’s automatic theorem prover \((\text{CAMPY})\), could be used to select relevant non-linear terms for synthesizing non-linear numerical domains, a critical problem in developing effective bound analyses [Gulwani and Gulwani 2008].

\textbf{Interpolation-Based Verification} Several model checkers have been proposed that use Craig interpolants to select an abstraction under which to verify a program. BLAST uses interpolants to select the predicates that it collects to define an abstraction in a predicate abstraction of the state space of a program [Heizmann et al. 2010]. IMPACT uses conjunctions of the actual interpolants used to refute a program path as the abstraction of a program [McMillan 2006]. Such interpolants are found by solving graph-interpolation problems defined by \textit{linear} dependency graphs. An approach that maintains an automaton of nested words annotated with interpolants [Heizmann et al. 2010] uses interpolants to refute an interprocedural trace with facts, that at each point in the trace, refer only to variables that are local to the procedure; such interpolants are found by solving graph-interpolation problems defined by \textit{tree} dependency graphs. CAMPY proves that a given program path satisfies a given resource bound by reducing the search for inductive step-count invariants to finding a tree interpolant [Heizmann et al. 2010] over a theory that is potentially richer than standard theories that allow interpolation. The key technique employed by CAMPY is to lazily ground axioms of such theories, using the structure of associated variables in the original formula.

Similar to CAMPY, verifiers of shape properties that implement instantiation-based interpolation [Sofronie-Stokkermans 2008; Tota and Wies 2013] instantiate universally-quantified axioms of an extension theory for describing program heaps until a given path formula becomes unsatisfiable a base theory that supports interpolation. The key advantage of such approaches is that by using local extensions, they typically satisfy strong completeness guarantees. While the theorem prover used by CAMPY is only a semi-decision procedure for theory extensions, it can be applied to non-local extensions, allowing it to potentially prove satisfaction of bounds expressed in non-linear integer arithmetic.

\textbf{Automated Reasoning} Decision procedures for solving non-linear constraints are a topic of extensive study in the foundations of mathematics, and have been applied to various problems in program and system verification. Classic and modern results provide decision procedures for solving quantified formulas with polynomials over real-closed fields [Collins 1998; Dai et al. 2015; Jovanovic and De Moura 2012; Tarski 1951]. These techniques likely cannot be directly applied to discharge verification conditions of programs or program paths, where the model of each formula must correspond to a run of a program that computes integer values.

While solving non-linear integer formulas is undesirable in general, recent approaches have been proposed for solving such formulas using SAT solvers [Borralleras et al. 2009]. While such a solver could potentially be used to decide if a particular path of a program in a language with a restricted set of features satisfies a bound, it is not clear how such a solver can be used to solve formulas in theories that are a combination of non-linear arithmetic with other theories used to model practical language features, such as \textit{EUF} of the theory of arrays. Furthermore, it is not clear if such approaches can be extended to provide a proof of bound satisfaction that can be used to prove bound satisfaction of over related program paths.

Several synthesis techniques [Udupa et al. 2013; Gulwani 2011; Harris and Gulwani 2011; Reynolds et al. 2015] have been unified as instances of syntax-guided synthesis [Alur et al. 2015], which is a meta-technique for synthesizing programs and program invariants as guided by a syntactic restriction. While CAMPY can likely be expressed as an instance of \textit{SYGUS}, CAMPY’s technique for enumerating terms as restricted by vocabularies of subformulas is distinct from search strategies proposed in all prior work on \textit{SYGUS}, and is critical to ensuring that path invariants can be extracted as interpolants from the resulting unsatisfiable formula. Other \textit{SYGUS} techniques often implement a search for program terms by solving a suitable system of symbolic constraints [Gulwani et al. 2011; Solar-Lezama et al. 2005]. CAMPY’s search for grounding terms could potentially be implemented using symbolic techniques. We leave the development of such techniques as future work.

7. Conclusion

We have presented an automatic verifier, CAMPY, that attempts to prove that a given program satisfies a given resource bound. The key technical challenge addressed by CAMPY is to synthesize invariants that prove that all runs of a given program path satisfy a bound, which may be expressed in an undecidable logic, such as non-linear arithmetic. The key technical contribution behind CAMPY that addresses this challenge is a theorem prover, \(T\), an extension of \(T\) of a decidable theory, \(T(\varphi)\) attempts to prove or disprove that a given \(T\)-formula \(\varphi\) is unsatisfiable by selectively refining a weakening of \(T\) by enumerating quantifier-free \(T\)-formulas, guided by the structure of \(\varphi\).

We have implemented CAMPY for JVM bytecode and applied it to verify that over 20 solutions submitted to online coding platforms satisfy or do not satisfy expected complexity bounds.

\textbf{Acknowledgments} We thank Aws AlBarghouthi, Somesh Jha, Boris Köpf, Wenke Lee, and Mayur Naik for insightful discussions concerning the
context for this work. We thank the reviewers for detailed and constructive readings of a preliminary version of this paper. This work is supported by NSF Award 1526211 and DARPA Award HR0011-16-C-0059.

References


