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 for both the performance, and in some cases security (cve-2011-3191), of the application and its system. Providing programming tools that enable programmers to understand when and why their programs use resources as expected is a critical open problem.

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; Sinn et al. 2014). 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

```
1  public static int BinarySearch(int arr[], int n) {
2    int fst = 0;
3    int lst = arr.len - 1;
4    while (fst <= lst) {
5      int mid = (fst + lst) / 2;
6      if (arr[mid] < n)
7        fst = mid + 1;
8      else if (arr[mid] == n)
9        break;
10     else lst = mid - 1; }
11   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 $\texttt{len}(\texttt{arr}) - 1$ (line 3).

In each iteration of the loop, `BinarySearch` tests that $\texttt{fst} \leq \texttt{lst}$ (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-positive 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 $\log \mathsf{length}(\texttt{arr}) + 1$ can be expressed as requiring all runs of `BinarySearch` to satisfy the post-condition $B_{\mathsf{Srch}} \equiv \mathsf{cost} \leq \log(\mathsf{length}(\texttt{arr})) + 1$.

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_{Srch} by establishing the loop invariant $I \equiv \mathsf{cost} + 1 + \log(\texttt{lst} - \texttt{fst}) \leq \log \texttt{len}(\texttt{arr})$ and proving that I implies that B_{Srch} 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:

$$\models 0 + \log(\texttt{len}(\texttt{arr}) - 0) \leq \log(\texttt{len}(\texttt{arr})) + 1 \tag{1}$$

Each state that satisfies I and does not satisfy the loop guard must satisfy B_{Srch}:

$$\mathsf{cost} + \log(\texttt{lst} - \texttt{fst}) \leq \log(\texttt{len}(\texttt{arr})) + 1, \texttt{fst} \not\leq \texttt{lst} \models \mathsf{cost} \leq \log(\texttt{len}(\texttt{arr})) + 1 \tag{2}$$

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:

$$\mathsf{cost} + \log(\texttt{lst} - \texttt{fst}) \leq \log(\texttt{len}(\texttt{arr})) + 1 \models \mathsf{cost} + \log(\texttt{lst} - (\texttt{fst} + \texttt{lst})/2) \leq \log(\texttt{len}(\texttt{arr})) + 1 \tag{3}$$

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_{Srch}:

$$\mathsf{cost} + \log(\texttt{lst} - \texttt{fst}) \leq \log(\texttt{len}(\texttt{arr})) + 1 \models \mathsf{cost} + \log(\texttt{lst} - \texttt{fst}) \leq \log(\texttt{len}(\texttt{arr})) + 1 \tag{4}$$

Each state that satisfies I and steps through the loop on the path that contains line 9 must step to a state that satisfies I:

$$\mathsf{cost} + \log(\texttt{lst} - \texttt{fst}) \leq \log(\texttt{len}(\texttt{arr})) + 1 \models \mathsf{cost} + \log(\texttt{lst} - (\texttt{fst} + \texttt{lst})/2) \leq \log(\texttt{len}(\texttt{arr})) + 1 \tag{5}$$

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. 4 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

$$\texttt{lst} - \texttt{fst} \leq 0 \implies \log(\texttt{lst} - \texttt{fst}) = 0 \tag{6}$$

which is the axiom $\forall x.x \leq 0 \implies \log x = 0$ grounded on the term $\texttt{lst} - \texttt{fst}$. Eqn. 3 and Eqn. 5 can be proved using the axioms of EUFLIA and the following quantifier-free theorems of log :

$$\log 2 = 1 \tag{7}$$
$$\log((\texttt{lst} - \texttt{fst})/2) = \log(\texttt{lst} - \texttt{fst}) - \log 2 \tag{8}$$

Eqn. 8 is the theorem $\forall x, y.\ \log x/y = \log x - \log y$ grounded on terms $\texttt{lst} - \texttt{fst}$ 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_{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_{Srch}, CAMPY attempts to prove that `BinarySearch` satisfies B_{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 2006), 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_{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_{Srch} for `BinarySearch` includes a term constructed from the log function.

CAMPY synthesizes sufficient path invariants on resource usage expressed in a non-linear theory $\mathcal{T}$ with a standard model by using a novel interpolating theorem prover, named TP$[\mathcal{T}]$. 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 TP$[\mathcal{T}]$ 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 theorems Eqn. 7 and Eqn. 8.

TP$[\mathcal{T}]$ attempts to simultaneously synthesize valid path invariants and quantifier-free $\mathcal{T}$ theorems A that support their validity using a counterexample-guided loop. In each iteration of the loop, TP$[\mathcal{T}]$ 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 φ_q^B) and a maintained candidate set A are mutually unsatisfiable as EUFLIA formulas. If EUFLIASAT determines that $\varphi_q^B \wedge A$ is unsatisfiable as an EUFLIA formula, then TP$[\mathcal{T}]$ generates path invariants for q by running an *interpolating theorem prover* for EUFLIA on $\varphi_q^B \wedge A$ (interpolating theorem provers are presented in detail in §3.2.2).

E.g., for bound B_{Srch} and path q of `BinarySearch` (§2.3), the path formula φ_q^B is

$$\begin{array}{r}\mathsf{cost}_0 = 0 \wedge \texttt{fst}_0 = 0 \wedge \texttt{lst}_0 = \texttt{len}(\texttt{arr}) - 1 \wedge \texttt{fst}_0 \leq \texttt{lst}_0 \wedge \\ \texttt{mid}_0 = (\texttt{fst}_0 + \texttt{lst}_0)/2 \wedge \texttt{arr}[\texttt{mid}_0] < \texttt{n} \wedge \texttt{fst}_1 = \texttt{mid}_0 + 1 \wedge \\ \mathsf{cost}_1 = \mathsf{cost}_0 + 1 \wedge \texttt{mid}_1 = (\texttt{fst}_1 + \texttt{lst}_0)/2 \wedge \\ \texttt{arr}[\texttt{mid}_0] \not< \texttt{n} \wedge \texttt{arr}[\texttt{mid}_0] = \texttt{n} \wedge \mathsf{cost}_1 \not\leq \log(\texttt{len}(\texttt{arr})) + 1\end{array}$$

While φ_q^B does not have a model in non-linear arithmetic, it does have one in EUFLIA, where the function symbol `log` is uninterpreted. In particular, one satisfying partial model is the following assignment m:

$$\begin{array}{rrr}\mathbf{m}(\log \mathbf{4}) = \mathbf{0} & m(\texttt{arr}[1]) = 10 & m(\texttt{arr}[2]) = 20 \\ m(\texttt{len}(\texttt{arr})) = 4 & m(\texttt{n}) = 20 & m(\mathsf{cost}_0) = 0 \\ m(\texttt{fst}_0) = 0 & m(\texttt{lst}_0) = 3 & m(\texttt{mid}_0) = 1 \\ m(\texttt{fst}_1) = 3 & m(\mathsf{cost}_1) = 1 & m(\texttt{mid}_1) = 2\end{array}$$

m is a model of φ_q^B 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 m of φ_q^B, then TP$[\mathcal{T}]$ evaluates φ_q^B on m using the standard model of all $\mathcal{T}$ functions. If φ_q^B is satisfied by m using the standard model of $\mathcal{T}$, then TP$[\mathcal{T}]$ determines that φ_q^B is satisfiable as a $\mathcal{T}$ formula, and as a result, that q does not satisfy the $\mathcal{T}$ bound B. E.g., for path q of `BinarySearch` and bound B_{Srch}, if TP$[\mathcal{T}]$ runs EUFLIASAT on $\varphi_q^{B_{\mathsf{Srch}}}$ and obtains model m, then it evaluates φ_q^B under the model m' that binds cost_0, $\texttt{fst}_0$, `len`, $\texttt{lst}_0$, $\texttt{mid}_0$, $\texttt{arr}[\texttt{mid}]$, and cost_1 to their values in m, and interprets log under the standard model. m' does not satisfy φ_q^B.

In general, if m' does not satisfy path formula φ_q^B, then TP$[\mathcal{T}]$ chooses a clause C of φ_q^B not satisfied by m' and generates a set of $\mathcal{T}$ formulas that are not satisfied by any model that evaluates the $\mathcal{T}$ terms in C to their values under m. E.g., when TP$[\mathcal{T}]$ attempts to find path invariants of path q of `BinarySearch`, TP$[\mathcal{T}]$ determines that m' does not satisfy the clause $\mathsf{cost}_1 \leq \log \texttt{len}(\texttt{arr})$ of $\varphi_q^{B_{\mathsf{Srch}}}$. TP$[\mathcal{T}]$ enumerates theorems of non-linear arithmetic over the variables cost_0, $\texttt{fst}_0$, `len`, $\texttt{lst}_0$, $\texttt{mid}_0$, $\texttt{arr}[\texttt{mid}]$, and cost_1 until it finds a minimal set that has no model m' under which $m'(\texttt{len}(\texttt{arr})) = m(\texttt{len}(\texttt{arr})) = 4$ and $m'(\log)(4) \neq 3$. One such set are the axioms of non-linear arithmetic Eqn. 7 and Eqn. 8 grounded on logical variables that model distinct definitions of `fst` and `lst` occur in q. I.e., the following set of theorems is minimal and sufficient:

$$\log 2 = 1 \tag{9}$$
$$\log((\texttt{lst}_0 - \texttt{fst}_0)/2) = \log(\texttt{lst}_0 - \texttt{fst}_0) - \log 2 \tag{10}$$
$$\log((\texttt{lst}_0 - \texttt{fst}_1)/2) = \log(\texttt{lst}_0 - \texttt{fst}_1) - \log 2 \tag{11}$$
$$\log((\texttt{lst}_1 - \texttt{fst}_1)/2) = \log(\texttt{lst}_1 - \texttt{fst}_1) - \log 2 \tag{12}$$

TP$[\mathcal{T}]$ iteratively generates models and enumerates theorems until either it finds a $\mathcal{T}$ model of φ_q^B or it finds a set of quantifier-free $\mathcal{T}$ theorems A such that $\varphi_q^B \wedge A$ is unsatisfiable. E.g., when TP$[\mathcal{T}]$ attempts to find path invariants of path q of `BinarySearch` that prove that q satisfies B_{Srch}, if TP$[\mathcal{T}]$ enumerates the theorems Eqn. 9—Eqn. 12, then TP$[\mathcal{T}]$ determines in its next iteration that $\varphi_q^B \wedge A$ is an unsatisfiable EUFLIA formula. TP$[\mathcal{T}]$ runs an interpolating theorem prover for EUFLIA on $\varphi_q^B \wedge A$ to obtain path invariants for q, such as the path invariants given in Fig. 2.

The effectiveness of TP$[\mathcal{T}]$ 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., TP$[\mathcal{T}]$, given path q of `BinarySearch`, could generate the valid theorem $\log 8 = 3$ to prove that q satisfies B_{Srch}, but such a theorem could not be used to synthesize inductive

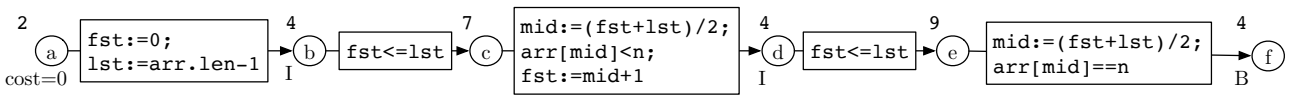

Figure 2: A control path q of `BinarySearch`. Each node n depicts a point in q and is annotated with its line number (i.e., control location); points with path critical invariants are also annotated with their path invariant. Each edge from n to n' is annotated with the instructions of `BinarySearch` taken when `BinarySearch` steps from the location of n to the location of n'.

step-count invariants of `BinarySearch` that prove that all of its runs satisfy B_{Srch}.

To select general theorems, $\textsc{Tp}[\mathcal{T}]$ 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

$$\begin{aligned} \forall x.\ x \leq 0 &\implies \log x = 0 \\ \forall x, y.\ \log(x \cdot y) &= \log x + \log y \\ \forall x, y.\ \log(x/y) &= \log x - \log y \end{aligned}$$

with x and y grounded to small terms over the logical variables used to model the state of q, such as $\mathtt{fst}_0$, $\mathtt{lst}_0$, and $\mathtt{lst}_0 - \mathtt{fst}_0$.

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, $\textsc{Tp}[\mathcal{T}]$ only enumerates theorems over sets of logical variables that model state at some common point in the given control path. I.e., $\textsc{Tp}[\mathcal{T}]$ 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; McMillan 2006). E.g., when enumerating axioms for path q of `BinarySearch`, $\textsc{Tp}[\mathcal{T}]$ enumerates theorems defined over logical variables $\mathtt{fst}_0$ and $\mathtt{lst}_0$ (which model state at points b and c) or $\mathtt{fst}_1$ and $\mathtt{lst}_0$ (which model state at points d and e), but not $\mathtt{fst}_0$ and $\mathtt{fst}_1$ or $\mathtt{cost}_0$ and $\mathtt{fst}_1$. 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 Target Language

In this section, we define the structure (§3.1.1) and semantics (§3.1.2) of the language of programs that CAMPY takes as input.

3.1.1 Program Structure

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

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 $\mathsf{Brs} = \mathsf{Locs}_B \times \mathsf{Instrs} \times \mathtt{Locs}$. For each branch statement $b \in \mathsf{Brs}$, the pre-location, instruction, and post-location of b are denoted $\mathsf{PreLoc}[b]$, $\mathtt{Instr}[b]$, and $\mathsf{BrTgt}[b]$, 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 $\mathsf{Calls} = \mathsf{Locs}_C \times \mathtt{procnms} \times \mathtt{Locs}$. For each call statement $c \in \mathsf{Calls}$, the pre-location, call target, and return target of c are denoted $\mathsf{PreLoc}[c]$, $\mathsf{CallTgt}[c]$, and $\mathsf{RetTgt}[c]$, respectively. The call entry point of c is denoted $\mathsf{entry}(c) = \mathsf{entry}(\mathsf{CallTgt}[c])$.

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

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

3.1.2 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 $\rightsquigarrow \subseteq \mathbb{Z}_n \times \mathbb{Z}_n$ such that for all indices $i_0, i_0', i_1, i_1' \in \mathbb{Z}_n$ with $i_0 \rightsquigarrow i_0'$ and $i_1 \rightsquigarrow i_1'$, either $i_0 < i_1 < i_1' < i_0'$, $i_1 < i_1' < i_0 < i_0'$, or $i_1 < i_0 < i_0' < i_1'$, $\rightsquigarrow$ is a* nesting *relation over n.*

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

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 $L = [\mathtt{L}_0, \ldots, \mathtt{L}_{n-1}] \in \mathtt{Locs}^*$ be such that the following conditions hold.*

(1) For each $0 \leq i < n$ such that $\mathtt{L}_i \in \mathsf{Locs}_B$, there is a branch statement $b \in P$ such that $\mathsf{PreLoc}[b] = \mathtt{L}_i$ and $\mathsf{BrTgt}[b] = \mathtt{L}_{i+1}$.

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

Then $[\mathtt{L}_0, \ldots, \mathtt{L}_{n-1}]$ is a path *of P.*

For each program $P \in \mathcal{L}$, the space of paths of P is denoted $\textsc{Paths}[P]$, and the set of all paths is denoted Paths. For each path

$p \in \mathsf{Paths}$, we denote the locations and nesting relation of p as $\mathtt{Locs}[p]$ and $\rightsquigarrow_p$, respectively.

Let the space of program values be the space of integers; i.e., the space of values is $\mathsf{Values} = \mathbb{Z}$. Our actual implementation of CAMPY can verify programs that operate on objects and arrays in addition to integers, but we present CAMPY as verifying programs that operate on only integers in order to simplify the presentation. An evaluation of all variables in Vars is a store; i.e., the space of stores is $\mathsf{Stores} = \mathsf{Vars} \to \mathsf{Values}$.

For each instruction $\mathtt{i} \in \mathsf{Instrs}$, there is a transition relation $\rho_{\mathtt{i}} \subseteq \mathsf{Stores} \times \mathsf{Stores}$. For each branch statement $b \in \mathsf{Brs}$, the transition relation of the instruction in b is denoted $\rho_b = \rho_{\mathtt{Instr}[b]}$. The transition relation of an instruction need not be total: thus, branch statements can implement control branches using instructions that act as `assume` instructions. The transition relation that relates each store at a callsite to the resulting entry store in a callee is denotes $\rho_C \subseteq \mathsf{Stores} \times \mathsf{Stores}$. The transition relation that relates each calling store, exit store of a callee, and resulting return store in the caller is denoted $\rho_R \subseteq \mathsf{Stores} \times \mathsf{Stores} \times \mathsf{Stores}$.

For each space X, sequence $s \in X^*$, and all $0 \leq i < |X|$, let the ith element in X be denoted $s[i] \in X$. Let the first and last elements of s in particular be denoted $\mathsf{Head}[s] = s[0]$ and $\mathsf{last}[s] = s[|s| - 1]$.

A run of a program P is a sequence of stores Σ and a path p of equal length, such that adjacent stores in Σ satisfy transition relations of statements of P at their corresponding locations in p.

Definition 3. *Let $P \in \mathcal{L}$ be a program, let $\Sigma = \sigma_0, \ldots, \sigma_{n-1} \in \mathsf{Stores}$ be a sequence of stores, and let $q \in \textsc{Paths}[P]$ be such that $|\mathsf{Locs}[q]| = n$, such that the following conditions hold:*

***(1)** For each $i < n - 1$, $(\sigma_i, \sigma_{i+1}) \in \rho_{\mathsf{BrAt}[P](L_i, L_{i+1})}$.*

***(2)** For each $i < j < n-1$ such that $i \rightsquigarrow j+1$, $(\sigma_i, \sigma_{i+1}) \in \rho_C$, $(\sigma_i, \sigma_j, \sigma_{j+1}) \in \rho_R$.*

Then Σ is a run *of q in P.*

For each path $p \in \mathsf{Paths}$, the space of runs of p is denoted $\mathsf{Runs}[p] \subseteq \mathsf{Stores}^*$.

3.2 Formal Logic

Our approach uses formal logic to model the semantics of programs and prove that a given program satisfies a bound. A *theory* is a vocabulary of function symbols and a standard model. For each theory $\mathcal{T}$ and space of logical variables X, let the spaces of $\mathcal{T}$ terms and formulas over X be denoted $\mathsf{Terms}[\mathcal{T}](X)$ and $\mathsf{Forms}[\mathcal{T}](X)$, respectively. For each formula $\varphi \in \mathsf{Forms}[\mathcal{T}](X)$, the set of variable symbols that occur in φ (i.e., the *vocabulary* of φ) is denoted $\mathsf{Voc}(\varphi)$. Each term constructed by applying only function symbols in $\mathcal{T}$ is a *ground term* of $\mathcal{T}$; i.e., the space of ground terms of $\mathcal{T}$ is $\mathsf{GTerms}[\mathcal{T}] = \mathsf{Terms}[\mathcal{T}](\emptyset)$.

For all vectors of variables $X = [x_0, \ldots, x_n]$ and $Y = [y_0, \ldots, y_n]$, the formula constraining the equality of each element in X with its corresponding element in Y, i.e., the formula $\bigwedge_{0 \leq i \leq n} \mathtt{x}_i = \mathtt{y}_i$, is denoted $X = Y$. For each vector of terms $T_Y = [t_0, \ldots, t_n]$, the repeated replacement of variables $\varphi[\ldots[t_0/x_0]\ldots t_{n-1}/x_{n-1}]$ is denoted $\varphi[X/T_Y]$. For each formula φ defined over free variables X, the substitution of Y in φ is denoted $\varphi[Y] \equiv \varphi[Y/X]$.

A *domain* is a finite set of values. For each theory $\mathcal{T}$, a *model* of $\mathcal{T}$ is a domain D and a map from each k-ary function symbol in $\mathcal{T}$ to a k-ary function over D. The standard model of a theory $\mathcal{T}$ is a distinguished model of $\mathcal{T}$. The domain of the standard model of $\mathcal{T}$ is denoted $\mathsf{Dom}[\mathcal{T}]$.

For theories $\mathcal{T}_0$ and $\mathcal{T}_1$, $\mathcal{T}_1$ is an *extension* of $\mathcal{T}_0$ if the vocabulary of $\mathcal{T}_0$ is contained by the vocabulary of $\mathcal{T}_1$ and the standard model of $\mathcal{T}_0$ is the restriction of the standard model of $\mathcal{T}_1$ to the vocabulary of $\mathcal{T}_0$. For all theories $\mathcal{T}_0$ and $\mathcal{T}_1$ whose standard models are equal on all symbols in the common vocabulary of $\mathcal{T}_0$ and $\mathcal{T}_1$, the combination (Nelson and Oppen 1979) of theories $\mathcal{T}_0$ and $\mathcal{T}_1$ is denoted $\mathcal{T}_0 \cup \mathcal{T}_1$. We only consider theories $\mathcal{T}$ with standard model m that maps to domain D such that for each element $d \in D$, there is a ground term $\mathsf{term}[d] \in \mathsf{GTerms}[T]$ such that $m(\mathsf{term}[d]) = d$ (e.g., theories of arithmetic), along with their combinations with the theory of uninterpreted functions (EUFLIA).

For each theory $\mathcal{T}$, formula $\varphi \in \mathsf{Forms}[\mathcal{T}](X)$, and assignment m of X to the domain of $\mathcal{T}$, m *satisfies* φ if φ evaluates to True under m combined with the standard model of $\mathcal{T}$ (denoted $m \vdash_{\mathcal{T}} \varphi$). For all $\mathcal{T}$ formulas $\varphi_0, \ldots, \varphi_n, \varphi \in \mathsf{Forms}[\mathcal{T}]$, we denote that $\varphi_0, \ldots, \varphi_n$ *entail* φ_n as $\varphi_0, \ldots, \varphi_n \models_{\mathcal{T}} \varphi$. A $\mathcal{T}$-formula φ is a *theorem* of $\mathcal{T}$ if $\models_{\mathcal{T}} \varphi$.

Although determining the satisfiability of formulas in theories required to model the semantics of practical languages, such as LIA, is NP-complete in general, solvers have been proposed that often efficiently determine the satisfiability of formulas that arise from practical verification problems (de Moura and Bjørner 2008). Our approach assumes access to a decision procedure for EUFLIA, named EUFLIASAT.

3.2.1 Symbolic Representation of Program Semantics

The semantics of $\mathcal{L}$ can be represented symbolically using LIA formulas. In particular, each program store $\sigma \in \mathsf{Stores}$ corresponds to a LIA model over the vocabulary Vars, denoted m^σ. For each space of indices I and index $i \in I$, the space of variables Vars_i denotes a distinct copy of the variables in Vars, as does Vars', which will typically be used to represent the post-state of a sequence of transitions. For theory $\mathcal{T}$, the space of program *summaries* is $\mathsf{Summaries} = \mathsf{Forms}[\mathcal{T}](\mathsf{Vars}, \mathsf{Vars}')$.

For each instruction $\mathtt{i} \in \mathsf{Instrs}$, there is a formula $\psi[\mathtt{i}] \in \mathsf{Forms}[\textsc{Lia}](\mathsf{Vars}, \mathsf{Vars}')$ such that for all stores $\sigma, \sigma' \in \mathsf{Stores}$, $(\sigma, \sigma') \in \rho_{\mathtt{i}}$ if and only if $m^\sigma, m^{\sigma'} \vdash \psi[\mathtt{i}]$. There is a formula $\psi_C \in \mathsf{Forms}[\textsc{Lia}](\mathsf{Vars}, \mathsf{Vars}')$ such that for all stores $\sigma, \sigma \in \mathsf{Stores}$, $(\sigma, \sigma') \in \rho_C$ if and only if $m^\sigma, m^{\sigma'} \vdash \psi_C$. There is a formula $\psi_R \in \mathsf{Forms}[\textsc{Lia}](\mathsf{Vars}_0, \mathsf{Vars}_1, \mathsf{Vars}_2)$ such that for all stores $\sigma_0, \sigma_1, \sigma_2 \in \mathsf{Stores}$, $(\sigma_0, \sigma_1, \sigma_2) \in \rho_R$ if and only if $m^{\sigma_0}, m^{\sigma_1}, m^{\sigma_2} \vdash \psi_R$.

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 $\mathcal{T}$, the space of bound constraints is denoted $\mathsf{Bounds}[\mathcal{T}] = \mathsf{Forms}[\mathcal{T}](\mathsf{Vars} \cup \{\mathsf{cost}'\})$.

Definition 4. *For each extension $\mathcal{T}$ of* LIA*, program $P \in \mathcal{L}$, bound constraint $B \in \mathsf{Bounds}[\mathcal{T}]$, and path $q \in \textsc{Paths}[P]$, if for each run $r \in \mathsf{Runs}[q]$, it holds that $m^{\mathsf{Head}[r]}, \mathsf{cost}' \mapsto m^{\mathsf{last}[r]}(\mathsf{cost}) \vdash B$, then q* satisfies *B. For each path $q \in \textsc{Paths}[P]$ it holds that q satisfies B, then P satisfies B, denoted $P \vdash B$. The bound-satisfaction problem (P, B) is to determine if $P \vdash B$.*

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. 2010). 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 $\mathcal{T}$, a $\mathcal{T}$-*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 $r \in N$.*
- *$C : N \to \mathsf{Forms}[\mathcal{T}](X)$ assigns each node to an $\mathcal{T}$* constraint*.*

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

- *The interpolant at the root $r \in N$ of T entails* False*. I.e., $I(r) \models_{\mathcal{T}} \mathsf{False}$.*
- *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., $\{I(m)\}_{(m,n)\in E}, C(n) \models_{\mathcal{T}} I(n)$.*
- *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 of the interpolant at n is contained in the vocabulary of n. I.e.,*

$$\mathsf{Voc}(n) = \bigcup_{(m,n)\in E^*} \mathsf{Voc}(C(m)) \cap \bigcup_{(m',n)\notin E^*} \mathsf{Voc}(C(m'))$$
$$\mathsf{Voc}(I(n)) \subseteq \mathsf{Voc}(n)$$

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

For each tree-interpolation problem T and node $n \in \mathsf{N}[T]$, the tree-interpolation problem formed by the restriction of T to the subtree with root n is denoted $T|_n$. The procedure $\textsc{TMrg}$ 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., $\textsc{TMrg}((N, E, C), (N', E', C')) = (N, E, C'')$ with $C''(n) = C(n)$ for each $n \in N \setminus N'$ and $C''(n) = C(n) \wedge C'(n)$ for each $n \in N'$.

For theory $\mathcal{T}$ and $\mathcal{T}$-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 $\mathcal{T}$, variables X, and $U_0, U_1 \in \textsc{Itp}[\mathcal{T}, X]$, if for $N = \mathsf{N}[T] = \mathsf{N}[U]$, $E[T] = E[U]$, and for each $n \in N$, **(1)** $\mathsf{Ctr}[U_0](n) \models \mathsf{Ctr}[U_1](n)$ and **(2)** $\mathsf{Voc}(\mathsf{Ctr}[U_0](n)) \subseteq \mathsf{Voc}(\mathsf{Ctr}[U_1](n))$, 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 $\mathcal{T}$, variables X, all $U_0, U_1 \in \textsc{Itp}[T, X]$ with common nodes N such that U_1 as weak as U_0, and all $I : N \to \mathsf{Forms}[\mathcal{T}](X)$ such that I is an interpolant of U_1, I is an interpolant of U_0.*

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

Definition 7. *For theory $\mathcal{T}$, variables X, let effective procedure $t : \textsc{Itp}[\mathcal{T}, X] \to (X \to \mathsf{Dom}[\mathcal{T}]) \cup (\mathsf{N}[T] \to \mathsf{Forms}[\mathcal{T}](X))$ be such that for each tree-interpolation problem $U \in \textsc{Itp}[\mathcal{T}, X]$, **(1)** if $t(U) : X \to \mathsf{Dom}[\mathcal{T}]$, then $t(U)$ is a model of U; **(2)** if $t(U) : \mathsf{N}[T] \to \mathsf{Forms}[\mathcal{T}](X)$, then $t(U)$ are interpolants of U. Then t is a* sound *interpolating theorem prover for $\mathcal{T}$.*

Previous work has presented an algorithm $\textsc{EufLiaItp}$ that solves a given $\textsc{EufLia}$ tree-interpolation problem T by invoking an interpolating theorem prover for $\textsc{EufLia}$ a number of times bounded by $|\mathsf{N}[T]|$ (Heizmann et al. 2010).

In §4, we describe an approach for proving that a program whose semantics are expressed in a theory $\mathcal{T}_0$ satisfies a bound expressed in an extension $\mathcal{T}$. To simplify the presentation of our approach, we fix $\mathcal{T}_0$ to be $\textsc{Lia}$, and fix $\mathcal{T}$ to be an arbitrary extension of $\textsc{Lia}$. However, our approach can be applied using any theory for $\mathcal{T}_0$ that satisfies the above conditions: in particular, our actual implementation of $\textsc{Campy}$ 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 $\textsc{Campy}$ for the bound-satisfaction problem. In §4.1, we define the summaries that $\textsc{Campy}$ attempts to infer to prove that a given program satisfies a given bound. In §4.2, we give the design of $\textsc{Campy}$'s core verification algorithm. In §4.3, we give the design of a tree-interpolating theorem prover, which is used by $\textsc{Campy}$ to prove that individual paths satisfy a given bound. In §4.4, we discuss key properties of $\textsc{Campy}$.

4.1 Summaries

$\textsc{Campy}$, 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 $\texttt{L}$ to a symbolic summary of the effects of all runs from the entry point of $\texttt{L}$'s procedure to $\texttt{L}$. The space of program summaries is denoted $\mathsf{ProgSums} = \mathsf{Locs} \to \mathsf{Summaries}$. 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 $B \in \mathsf{Bounds}[\mathcal{T}]$, let $S \in \mathsf{ProgSums}$, be such that:*

***(1)** for each procedure $f \in$ `procnms`,*

$$\mathsf{Vars} = \mathsf{Vars}' \models S(\mathsf{entry}(f))$$

***(2)** $S(\texttt{FINAL}) \models \mathsf{Bounds}[\mathcal{T}]$;*

***(3)** For each branch statement $b \in P$,*

$$S(\mathsf{PreLoc}[b])[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi[b][\mathsf{Vars}_1, \mathsf{Vars}_2] \models S(\mathsf{BrTgt}[b])[\mathsf{Vars}_0, \mathsf{Vars}_2]$$

***(4)** For each call statement $c \in P$ and each return statement $r \in P$ that returns to c,*

$$S(\mathsf{PreLoc}[c])[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi_C[\mathsf{Vars}_1, \mathsf{Vars}_2], S(r)[\mathsf{Vars}_2, \mathsf{Vars}_3], \psi_R[\mathsf{Vars}_1, \mathsf{Vars}_3, \mathsf{Vars}_4] \models S(\mathsf{RetTgt}[c])[\mathsf{Vars}_0, \mathsf{Vars}_4]$$

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 $B \in \mathsf{Bounds}[\mathcal{T}]$ is denoted $\mathsf{Ind}[P, B]$.

Example 1. *The step-count invariants given in §2.2 directly define inductive step-count summaries for* `BinarySearch` *and B_{Srch}.*

Inductive summaries are evidence of bound satisfaction.

Lemma 2. *For each program $P \in \mathcal{L}$ and bound $B \in \mathsf{Bounds}[\mathcal{T}]$, if there are inductive summaries $S \in \mathsf{Ind}[P, B]$, then $P \vdash B$.*

For 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 $p \in \mathsf{Paths}$, let $L \in \mathsf{Locs}^*$ be such that $\mathsf{last}[L] =$ `FINAL` and there is some function $m : \mathbb{Z}_{|L|} \to \mathbb{Z}_{|p|}$ such that for each $0 \le i < |L|$, $L[i] = p[m(i)]$, if $L[i] \in \mathsf{Locs}_B$ or*

$L[i] \in \mathsf{Locs}_R$, *then* $L[i+1] = p[m(i)+1]$, *and if* $L[i] \in \mathsf{Locs}_C$, *then for* $j < |p|$ *such that* $i \rightsquigarrow_p j$, $L[i+1] = p[j]$. *Then* L *is a* visible suffix *of* p.

For each path $p \in \mathsf{Paths}$, let the space of visible suffixes of p be denoted $\mathsf{Suffixes}[p] \subseteq \mathtt{Locs}^*$. For program $P \in \mathcal{L}$, the visible suffixes of all paths of P are denoted $\mathsf{Suffixes}[P] = \bigcup_{p \in \textsc{Paths}[P]} \mathsf{Suffixes}[p]$.

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* $P \in \mathcal{L}$, *let* $Q \subseteq \mathsf{Suffixes}[P]$ *be visible suffixes of paths of* P, *and let* $S : Q \to \mathsf{Summaries}$. *Then* (Q, S) *are* visible suffix summaries *of* P.

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

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* $P \in \mathcal{L}$, *let visible-suffix summaries* $S \in \mathsf{VisSums}[P]$ *be such that for each visible suffix* $s \in \mathsf{Suffixes}[S]$, ***(1)*** *for each procedure* $f \in \mathtt{procnms}$, *if* $\mathsf{Head}[s] = \mathsf{entry}(f)$, *then*

$$\mathsf{Vars} = \mathsf{Vars}' \models \mathsf{Sums}[S](s)$$

(2) *for each branch statement* $b \in P$ *such that* $\mathsf{BrTgt}[b] = \mathsf{Head}[s]$ *and* $s_0 = \mathsf{PreLoc}[b] :: s \in P$,

$$\mathsf{Sums}[S](s_0)[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi[b][\mathsf{Vars}_1, \mathsf{Vars}_2] \models \mathsf{Sums}[S](s)[\mathsf{Vars}_2/\mathsf{Vars}_0]$$

(3) *and each call statement* $c \in P$ *such that* $s_0 = \mathsf{PreLoc}[c] :: s \in \mathsf{Suffixes}[S]$ *and return statement* $r \in P$ *such that* $\mathsf{proc}(c) = \mathsf{proc}(r)$ *and* $s_1 = r :: s \in \mathsf{Suffixes}[S]$,

$$\mathsf{Sums}[S](s_0)[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi_C[\mathsf{Vars}_1, \mathsf{Vars}_2] \\ \mathsf{Sums}[S](s_1)[\mathsf{Vars}_2, \mathsf{Vars}_3], \psi_R[\mathsf{Vars}_1, \mathsf{Vars}_3, \mathsf{Vars}_4] \models \mathsf{Sums}[S](s)[\mathsf{Vars}_0, \mathsf{Vars}_4]$$

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* $P \in \mathcal{L}$ *and bound constraint* $B \in \mathsf{Bounds}[\mathcal{T}]$, *let* $S \in \mathsf{VisSums}[P]$ *be valid suffix summaries. Let* $S' \in \mathsf{ProgSums}$ *be such that for each location* $\mathtt{L} \in \mathtt{Locs}$,

$$S'(\mathtt{L}) = \bigvee \{\mathsf{Sums}[S](t) \mid t \in \mathsf{Suffixes}[S], \mathsf{Head}[t] = \mathtt{L}\}$$

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 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, Defn. 12) for P and B (line 2—line 11). CAMPY runs AUX on the empty set of visible-suffix summaries paired with an empty map to summaries and returns the result (line 12).

Input : A program $P \in \mathcal{L}$ and resource bound $B \in \mathsf{Bounds}[\mathcal{T}]$.
Output : A decision as to whether P satisfies B.

```
1  Procedure CAMPY(P, B)
2      Procedure AUX(S)
3          switch CHKIND[P, B](S) do
4              case True: return True ;
5              case q:
6                  switch TP[T](BRKS[P, B](q)) do
7                      case HasModel: return False ;
8                      case S_q: return AUX(MRG(S, S_q)) ;
9                  endsw
10             end
11         endsw
12     return AUX((∅, ∅)) ;
```

Algorithm 1: CAMPY: a bound-satisfaction verifier for bounds expressed in an extension of LIA. CAMPY uses procedures $\textsc{ChkInd}[P, B]$ (summarized in §4.2) and $\textsc{Tp}[\mathcal{T}]$, which is presented in detail in §4.2.1.

AUX runs a procedure $\textsc{ChkInd}[P, B]$ that determines if visible-suffix summaries S are inductive. If so, $\textsc{ChkInd}[P, B]$ returns True, and otherwise returns a path of P not summarized by S (line 3; the design of $\textsc{ChkInd}[P, B]$ is as described in previous work on automatic verification of safety properties (McMillan 2006), and we omit a description here). If $\textsc{ChkInd}[P, B]$ determines that S are inductive summaries, then AUX returns that P satisfies B (line 4).

Otherwise, if $\textsc{ChkInd}[P, B]$ returns a path q not summarized by S, then AUX runs a procedure $\textsc{Brks}[P, B]$ on q. $\textsc{Brks}[P, B]$ returns a tree-interpolation problem for which each model corresponds to a run of q in P that does not satisfy B, and all tree interpolants correspond to path invariants of q that prove that q satisfies B; the design of $\textsc{Brks}[P, B]$ is described in detail in §4.2.1. AUX runs an interpolating theorem prover $\textsc{Tp}[\mathcal{T}]$ for $\mathcal{T}$ on the resulting tree-interpolation problem (line 6). If $\textsc{Tp}[\mathcal{T}]$ returns that the interpolation problem has a model, then AUX returns that P does not satisfy B (line 7).

Otherwise, if $\textsc{Tp}[\mathcal{T}]$ returns summaries S_q of q that prove that q satisfies B, then AUX runs a procedure MRG on S and S_q to construct visible-suffix summaries that prove that all paths of S and q satisfy B, recurses on the result, and returns the result of the recursive call (line 8; the implementation of MRG is immediate from previous approaches for automatically verifying safety properties (McMillan 2006)).

4.2.1 Per-Path Bound Satisfaction as Tree-Interpolation

For program $P \in \mathcal{L}$ and bound $B \in \mathsf{Bounds}[\mathcal{T}]$, $\textsc{Brks}[P, B]$, given path $q \in \textsc{Paths}[P]$, constructs a $\mathcal{T}$-interpolation problem for which each model corresponds to a run of q that does not satisfy B. To do so, $\textsc{Brks}[P, B]$ constructs a tree-interpolation problem (§3.2.2, Defn. 5) $\mathsf{CTree}[P, B, q] = (N, E, C)$, defined as follows. **(1)** The nodes N are the visible suffixes of q. I.e., $N = \mathsf{Suffixes}[q]$. **(2)** The edges $E \subseteq N \times N$ are the immediate-suffix relation over visible suffixes of q. I.e.,

$$E = \{(\mathtt{L} :: q', q') \mid \mathtt{L} \in \mathtt{Locs}, q', \mathtt{L} :: q' \in \mathsf{Suffixes}[q]\}$$

(3) The constraints C model the semantics of statements in q and the condition that B must be satisfied. I.e., for each visible suffix $s \in \mathsf{Suffixes}[q]$, let there be a distinct copy of program variables Vars_s. Then:

Input : A $\mathcal{T}$-tree-interpolation problem U.
Output : Either interpolants for U or $\mathsf{HasModel}$ to denote that U has a $\mathcal{T}$-model.

1 **Procedure** $\textsc{Tp}[\mathcal{T}](U)$
2 **Procedure** $\textsc{TpAux}[\mathcal{T}](V)$
3 **switch** $\textsc{EufLiaSat}(V)$ **do**
4 **case** None: **return** $\textsc{EufLiaItp}(V)$;
5 **case** m:
6 $N' := \{n \mid n \in \mathsf{N}[U], m_{\mathcal{T}} \not\models_{\mathcal{T}} C(n')\}$;
7 **if** $N' = \emptyset$ **then return** $\mathsf{HasModel}$;
8 $n' := \mathsf{Elt}(N')$;
9 $V' = \textsc{Grnd}[\mathcal{T}](V|_{n'}, m)$;
10 **return** $\textsc{TpAux}[\mathcal{T}](\textsc{TMrg}(V, V'))$;
11 **end**
12 **endsw**
13 **return** $\textsc{TpAux}[\mathcal{T}](U)$

Algorithm 2: $\textsc{Tp}[\mathcal{T}]$: an interpolating theorem prover for $\mathcal{T}$.

- For each branch statement $b \in \mathsf{Brs}$ such that $\mathsf{BrTgt}[s] = \mathsf{Head}[b]$ and $s' = \mathsf{PreLoc}[b] :: s \in \mathsf{Suffixes}[q]$, let

$$C'(s) = \psi[b][\mathsf{Vars}_{s'}, \mathsf{Vars}_s]$$

- For call statement $c \in \mathsf{Calls}$ such that $\mathsf{CallTgt}[c] = \mathsf{Head}[s]$ and $s_0 = \mathsf{PreLoc}[c] :: s \in \mathsf{Suffixes}[q]$ and return statement $r \in \mathsf{Rets}$ such that $s_1 = r :: s \in \mathsf{Suffixes}[q]$ and $s_2 \in \mathsf{Suffixes}[q]$ is the maximal extension of s_1 in $\mathsf{Suffixes}[q]$, let

$$C'(s) = \psi_C[\mathsf{Vars}_{s_0}, \mathsf{Vars}_{s_2}] \wedge \psi_R[\mathsf{Vars}_{s_0}, \mathsf{Vars}_{s_1}, \mathsf{Vars}_s]$$

Then $C([\texttt{FINAL}]) = C'([\texttt{FINAL}]) \wedge B[\mathsf{Vars}_{[\texttt{FINAL}]}/\mathsf{Vars}']$, and for each $s \neq [\texttt{FINAL}] \in \mathsf{Suffixes}[q]$, $C(s) = C'(s)$. Let the space of all summaries over post-state variables defined per visible suffixes of q be denoted $\mathsf{SuffixSums}[q] = \mathsf{Forms}[\bigcup_{s \in \mathsf{Suffixes}[q]} \mathsf{Vars}_s]$.

Each interpolant of $\mathsf{CTree}[P, B, q]$ defines valid visible-suffix summaries of q. In particular, for each map $I : N \to \mathsf{SuffixSums}[q]$, let $S_I : \mathsf{Suffixes}[q] \to \mathsf{Summaries}$ be such that for each $s \in \mathsf{Suffixes}[q]$ with maximal extension $s_0 \in \mathsf{Suffixes}[q]$,

$$S_I(s) = I(s)[\mathsf{Vars}, \mathsf{Vars}'/\mathsf{Vars}_{s_0}, \mathsf{Vars}_s]$$

Lemma 3. *For each program $P \in \mathcal{L}$, path $q \in \textsc{Paths}[P]$, bound $B \in \mathsf{Bounds}[\mathcal{T}]$, and constraint map $I : \mathsf{Suffixes}[q] \to \mathsf{SuffixSums}[q]$, if I are interpolants of $\mathsf{CTree}[P, B, q]$, then S_I are valid visible-suffix summaries of q (Defn. 11).*

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

Example 3. *In §2.3, Fig. 2 depicts a linear tree-interpolation problem T_q for which each model defines a run of path q that breaks bound B. Each node of T is a point in the control path p, and each edge of T 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.*

4.3 Tree Interpolation by Theorem Enumeration

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

Input : A $\mathcal{T}$-interpolation problem U over variables X and model m that satisfies U under $\textsc{EufLia}$.
Output : A $\mathcal{T}$-interpolation problem that is as weak as U but is not satisfied by m under $\textsc{EufLia}$.

1 **Procedure** $\textsc{Grnd}[\mathcal{T}](U, m)$
2 $\mathcal{V} := \textsc{ColocItp}(U)$;
3 $E := \textsc{EvalCtr}(\mathsf{Ctrs}[T](\mathsf{Head}[U]))$;
4 **Procedure** $\textsc{GAux}[\mathcal{T}](A)$
5 $A_{\mathcal{V}} = \{a \mid n \in \mathsf{N}[U], a \in A, \mathsf{Voc}(a) \subseteq \mathcal{V}(n)\}$;
6 **if** $\textsc{IsSat}(\mathsf{Ctr}[U] \wedge E \wedge \bigwedge A_{\mathcal{V}})$ **then**
7 **return** $\textsc{GAux}[\mathcal{T}](A \cup \{\textsc{Enum}[\mathcal{T}, X](A)\})$
8 **else**
9 $A'_{\mathcal{V}} := \mathsf{MinUnsat}(A_{\mathcal{V}}, \mathsf{Ctr}[U] \wedge B)$;
10 $L(n) := \bigwedge\{a \mid a \in A'_{\mathcal{V}}, \mathsf{Voc}(a) \subseteq \mathcal{V}(n)\}$;
11 **return** U with $C(n) := \mathsf{Ctr}[U](n) \wedge L(n)$;
12 **end**
13 **return** $\textsc{GAux}[\mathcal{T}](\emptyset)$;

Algorithm 3: $\textsc{Grnd}[\mathcal{T}]$: takes a $\mathcal{T}$-tree-interpolation problem U and assignment m of the variables of U that satisfies U under $\textsc{EufLia}$, and generates a tree-interpolation problem as weak as U that m does not satisfy under $\textsc{EufLia}$.

4.3.1 The Theorem-Proving Algorithm

Alg. 2 contains pseudocode for $\textsc{Tp}[\mathcal{T}]$, an interpolating theorem prover for $\mathcal{T}$. $\textsc{Tp}[\mathcal{T}]$ takes a $\mathcal{T}$-interpolation problem U and returns either the value $\mathsf{HasModel}$ to denote that U has a $\mathcal{T}$-model or interpolants for U. $\textsc{Tp}[\mathcal{T}]$ defines a procedure $\textsc{TpAux}[\mathcal{T}]$ (line 2—line 12) that takes a $\mathcal{T}$-tree-interpolation problem V that is as weak as U and returns either $\mathsf{HasModel}$ if V has a model or $\mathcal{T}$-interpolants of V. $\textsc{Tp}[\mathcal{T}]$ calls $\textsc{TpAux}[\mathcal{T}]$ on U and returns the result (line 13).

$\textsc{TpAux}[\mathcal{T}]$ determines if V has an $\textsc{EufLia}$ model by running $\textsc{EufLiaSat}$ on V (line 3; $\textsc{EufLiaSat}$ is introduced in §3.2). If $\textsc{EufLiaSat}$ returns that V has no $\textsc{EufLia}$ model, then $\textsc{TpAux}[\mathcal{T}]$ runs an interpolating theorem prover $\textsc{EufLiaItp}$ (introduced in §3.2.2) on V and returns the resulting interpolants (line 4).

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

Example 4. *§2.3 describes a run of $\textsc{Tp}[\mathcal{T}]$ on the tree-interpolation problem for path q of program* `BinarySearch`*. $\textsc{Tp}[\mathcal{T}]$ first obtains a model m of the tree-interpolation problem for q, T_q, under $\textsc{EufLia}$ (Alg. 2, line 3). $\textsc{Tp}[\mathcal{T}]$ determines that $m_{\mathcal{T}}$ does not satisfy the clause for the node g of Fig. 2 under the standard model of non-linear arithmetic (line 6), chooses g as the unique node in T_q that is not satisfied by $m_{\mathcal{T}}$, and runs $\textsc{Grnd}[\mathcal{T}]$ on T_q restricted to g (i.e., T_q itself). $\textsc{Grnd}[\mathcal{T}]$ generates a weaker tree-interpolation problem described in §4.3.2, Ex. 6, which $\textsc{TpAux}[\mathcal{T}]$ merges with T_p, and recurses on (line 10).*

The tree-interpolation problem generated by $\textsc{Grnd}[\mathcal{T}]$ is sufficiently weak that it is unsatisfiable under $\textsc{EufLia}$. Thus, $\textsc{TpAux}[\mathcal{T}]$ returns interpolants for it found by $\textsc{EufLiaItp}$.

4.3.2 Enumerating Quantifier-Free $\mathcal{T}$-Theorems

Alg. 3 contains pseudocode for GRND$[\mathcal{T}]$ (line 1—line 13). GRND$[\mathcal{T}]$ takes a $\mathcal{T}$-interpolation problem $U \in$ ITP$[\mathcal{T}, X]$ over variables X and assignment $m : X \to \mathsf{Dom}[\mathcal{T}]$ and returns a $\mathcal{T}$ tree-interpolation problem that is as weak as U but for which m is not a model under EUFLIA. GRND$[\mathcal{T}]$ uses a procedure ENUM$[\mathcal{T}, X]$ that takes quantifier-free $\mathcal{T}$ formulas A over variables X and enumerates a quantifier-free $\mathcal{T}$ formula over X not in A.

GRND$[\mathcal{T}]$ constructs a map $\mathcal{V} : \mathsf{N}[U] \to \mathcal{P}(X)$ from each node $n \in \mathsf{N}[U]$ to the vocabulary of n (line 2; the vocabulary of a node in an interpolation problem is defined in §3.2.2, Defn. 5). GRND$[\mathcal{T}]$ then runs a procedure EVALCTR that takes an assignment $m : X \to \mathsf{Values}$ and returns a constraint that each variable in X equal to its value under m (line 3). I.e.,

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

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

Example 5. *When* TP$[\mathcal{T}]$ *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* $\texttt{cost}_1$ *to* 2*,* $\texttt{len}(\texttt{arr})$ *to* 4*. When* TP$[\mathcal{T}]$ *runs* GRND$[\mathcal{T}]$*,* EVALCTR *generates clause* $\texttt{cost}_1 = 2 \wedge \texttt{len}(\texttt{arr}) = 4$*.*

GRND$[\mathcal{T}]$ defines a procedure GAUX$[\mathcal{T}]$ (line 4—line 12) that uses a set of quantifier-free $\mathcal{T}$ theorems to find a $\mathcal{T}$-tree interpolation problem that is as weak as U but is not satisfied by m under EUFLIA. GAUX$[\mathcal{T}]$, given quantifier-free formulas A, first collects the set $A_{\mathcal{V}}$ of formulas in A that each have a vocabulary contained by the vocabulary of some set in $\mathcal{V}$ (line 5). GAUX$[\mathcal{T}]$ then checks if there is a model of the constraints of U, E, and all theorems in $A_{\mathcal{V}}$ (line 6). If so, then GAUX$[\mathcal{T}]$ recurses on A extended with the next enumerated theorem of $\mathcal{T}$ (line 7).

Otherwise, if $\mathsf{Ctr}[U]$, E, and $A_{\mathcal{V}}$ are not mutually satisfiable, then GAUX$[\mathcal{T}]$ constructs a minimal subset $A'_{\mathcal{V}}$ of $A_{\mathcal{V}}$ that is mutually unsatisfiable with $\mathsf{Ctr}[U]$ and E (line 9). GAUX$[\mathcal{T}]$ 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 $A'_{\mathcal{V}}$ whose vocabularies are contained in the set of variables in the vocabulary of n (line 10—line 12).

Example 6. *When* TP$[\mathcal{T}]$ *runs* GRND$[\mathcal{T}]$ *on the tree-interpolation problem* $\mathsf{CTree}[\texttt{BinarySearch}, B_{\mathsf{Srch}}, q]$ *with variables* X *for the path* q *of* `BinarySearch` *(§2.3),* GRND$[\mathcal{T}]$ *constructs the evaluation clause* E *(Ex. 5) and runs* GAUX$[\mathcal{T}]$*. For* $\mathcal{T}$ *the theory of non-linear arithmetic,* GAUX$[\mathcal{T}]$ *iteratively runs* ENUM$[\mathcal{T}, X]$*, which generates* $\mathcal{T}$ *formulas over* X*, such as Eqn. 9, Eqn. 11, and* $\log(\texttt{fst}_0 \cdot \texttt{fst}_1) = \log \texttt{fst}_0 + \log \texttt{fst}_1$*.*

Of the enumerated theorems, $A_{\mathcal{V}}$ *(defined at Alg. 3), line 5) contains, e.g., Eqn. 11 because* $\texttt{lst}_0$ *and* $\texttt{fst}_0$ *are in the vocabulary of some node in* $\mathsf{CTree}[\texttt{BinarySearch}, B_{\mathsf{Srch}}, q]$*, namely the nodes corresponding to points* b *and* c *in* q *(depicted in §2.3, Fig. 2).* $A_{\mathcal{V}}$ *does not contain, e.g.,* $\log(\texttt{fst}_0 \cdot \texttt{fst}_1) = \log \texttt{fst}_0 + \log \texttt{fst}_1$ *because* $\texttt{fst}_0$ *and* $\texttt{fst}_1$ *are not in the vocabulary of any node in* $\mathsf{CTree}[\texttt{BinarySearch}, B_{\mathsf{Srch}}, q]$*.*

GRND$[\mathcal{T}]$ *enumerates theorems of non-linear arithmetic over* X *until it enumerates a set that is mutually unsatisfiable with* $\mathsf{CTree}[\texttt{BinarySearch}, B_{\mathsf{Srch}}, q]$ *(given in §2.3) and* E*. One such set are the theorems Eqn. 9—Eqn. 12, given in §2.3.*

When GRND$[\mathcal{T}]$ 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* $U \in$ ITP$[\mathcal{T}, X]$*, and assignment* m *over* X*, if* GRND$[\mathcal{T}]$ *terminates on* U *and* m*, then* GRND$[\mathcal{T}](U, m)$ *is as weak as* U*.*

Proof. Proof by induction on the evaluation of GAUX$[\mathcal{T}]$ on its argument A. The claim proved by induction is that each formula in A is a theorem of $\mathcal{T}$. For the base step, GAUX$[\mathcal{T}]$ 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 $\mathcal{T}$-theorems. In the next step of evaluation, GAUX$[\mathcal{T}]$ is called with a set consisting of A and a formula generated by ENUM$[\mathcal{T}, X]$, which by assumption generates only $\mathcal{T}$ theorems (§3.2). Thus, in the next step of evaluation of GAUX$[\mathcal{T}]$, A contains only $\mathcal{T}$ theorems.

GAUX$[\mathcal{T}]$ 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 $\mathcal{T}$ theorem implies that n is bound to a constraint that is no stronger under $\mathcal{T}$ then its constraint in U.

GAUX$[\mathcal{T}]$ includes a formula in A in the constraint $C'(n)$ only if its vocabulary is a subset of the vocabulary for n (line 5 and line 10). Thus the vocabulary of $C'(n)$ the vocabulary of $\mathsf{Ctrs}[T](n)$. 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$[\mathcal{T}]$, 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 (§4.4, Thm. 1), so we withhold a complete statement and proof.

A consequence of Lemma 4 is that TP$[\mathcal{T}]$ is a sound interpolating theorem prover for $\mathcal{T}$.

Lemma 5. TP$[\mathcal{T}]$ *is a sound interpolating theorem prover for* $\mathcal{T}$ *(§3.2, Defn. 7).*

Proof. For $\mathcal{T}$-tree-interpolation problem U, proof by induction on the evaluation of TP$[\mathcal{T}](U)$. 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 Lemma 4, GRND$[\mathcal{T}](U|_{n'}, m)$ is as weak as $U|_{n'}$, and by the definition of TMRG (§3.2.2), $U' =$ TMRG$(U,$ GRND$[\mathcal{T}](U|_{n'}, m))$ is as weak as U. Thus, in AUX's next step of evaluation, U is as weak as T.

TP$[\mathcal{T}]$ 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 $\mathcal{T}$. The fact that U is weaker than T, combined with Lemma 1 (see §3.2), implies that if TP$[\mathcal{T}]$ returns a constraint map I, then I are valid interpolants of T. □

Because TP$[\mathcal{T}]$ is a valid decision procedure for theories that may not be decidable, TP$[\mathcal{T}]$ 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* $B \in \mathsf{Bounds}[\mathcal{T}]$*, 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).

The inductive step of the proof of the claim follows from the following argument. By the fact that each interpolant of $\textsc{Brks}[P, B](q)$ is a valid summary that q satisfies B (§4.2.1, Lemma 3) and the fact that if $\textsc{Tp}[\mathcal{T}]$ generates a constraint map then the map is a valid interpolant of its input (§4.3.2, Lemma 5), S_q are valid $\textsc{EufLia} \cup \mathcal{T}$ summaries that prove that q satisfies B. The fact that S_q combined with the assumption that $\textsc{Mrg}$, given valid a pair of valid summaries, generates valid summaries (§4.2), implies that $\textsc{Aux}$ is called on valid suffix summaries in its next iteration.

The claim, combined with the fact that $\textsc{Aux}$ only returns true if $\textsc{ChkInd}[P, B]$ is run on S and returns True, and the assumption that $\textsc{ChkInd}[P, B]$ when given valid path summaries S returns True only if S are inductive path summaries, implies that $\textsc{Campy}(P, B) = \mathsf{True}$ only if P has inductive summaries. The fact that inductive summaries are evidence of bound satisfaction (§4.1, Lemma 2) implies that $P \vdash B$. □

In §4.2, we presented $\textsc{Campy}$ as taking a program P and bound B and returning a Boolean value denoting if P satisfies B. $\textsc{Campy}$ can be directly extended so that if it determines that P does not satisfy B, then it returns a run of P on which it does not satisfy B. In particular, the theorem prover $\textsc{Tp}[\mathcal{T}]$ (§4.3) can be extended so that if it finds an assignment to logical variables that satisfy the path formula of a path q under the standard model of $\mathcal{T}$, then $\textsc{Tp}[\mathcal{T}]$ returns the satisfying model. $\textsc{Campy}$ can be extended to take such a model m from $\textsc{Tp}[\mathcal{T}]$ and from it, synthesize a run of path q that does not satisfy B.

5. Evaluation

We performed an empirical evaluation of $\textsc{Campy}$ in order to answer the following questions: **(1)** Is $\textsc{Campy}$ expressive? I.e., can it prove that programs that implement subtle algorithms satisfy expected bounds, which in practice are often non-linear? **(2)** Is $\textsc{Campy}$ efficient? I.e., when it proves that a program satisfies a bound or finds an input which the program does not satisfy the bound, does it do so efficiently enough to be potentially useful for a programmer?

To answer the above questions, we collected benchmarks from platforms that host solutions to programming exercises (codechef; leetcode; codeforces). For each benchmarks program P, we derived a bound that we expected P to satisfy and a bound that we did not expect P to satisfy by inspecting metadata associated with P, such as the P's signature and information on the site at which it was posted. We implemented $\textsc{Campy}$ as a tool that supports programs represented in JVM bytecode, using the Soot (framework) analysis framework and Z3 (z3) interpolating theorem prover, and applied it to each program paired with its expected bound and unexpected bound, and observed if $\textsc{Campy}$ found a proof of bound satisfaction or counterexample run for each bound.

We performed all experiments on a machine with 16 1.4 GHz processors and 132 GB of RAM. The current implementation of $\textsc{Campy}$ executes using a single thread. $\textsc{Campy}$, along with a container replicating our experimental setup for evaluation, are publicly available (campy). Each benchmark on which we evaluated $\textsc{Campy}$ is publicly available, and we are currently working with online coding platforms to potentially release the programs as a benchmark suite for program analyses and verifiers.

In summary, our evaluation answered the above questions positively. $\textsc{Campy}$ was able to verify that program satisfied or did not satisfy expected bound in at most a few seconds. The rest of this section describes our results in detail. In §5.1, we discuss a selection of solutions on which we ran $\textsc{Campy}$. In §5.2, we draw conclusions from our results on the effectiveness of $\textsc{Campy}$.

```
1  public int lengthOfLIS(int[] nums, int n) {
2    if (nums == null || n == 0) return 0;
3    int[] res = new int[n];
4    int len = 0;
5    res[len] = nums[0];
6    len++;
7    for (int i = 1; i < n; i++) {
8      if (nums[i] < res[0]) res[0] = nums[i]
9      else if (nums[i] > res[len - 1]) {
10       res[len] = nums[i];
11       len++; }
12     else {
13       int j = n - 1;
14       while (i < j) {
15         int mid = (i + j) / 2;
16         if (res[mid] < nums[i]) i = mid + 1;
17         else j = mid; }
18       res[j] = nums[i]; } }
19   return len; }
```

Figure 3: `lengthOfLIS`: a solution to the LIS Problem.

5.1 Benchmarks

In this section, we illustrate the operation of $\textsc{Campy}$ 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 $\textsc{Campy}$ by discussing three of the benchmark programs: `lengthOfLIS`, `SmartSieve`, and `Cycle`. $\textsc{Campy}$ correctly verified the programs `lengthOfLIS` 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 (LIS). `lengthOfLIS`, given in Fig. 3, was posted to LeetCode as a solution to the LIS problem. `lengthOfLIS` iterates from 1 to n, and at each intermediate value i, maintains in array `res` of 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 `lengthOfLIS` 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 `lengthOfLIS` 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 `lengthOfLIS`, we first provided to $\textsc{Campy}$ `lengthOfLIS` and the bound n^2, derived from identifying the nested loops of `lengthOfLIS`; $\textsc{Campy}$ verified that `lengthOfLIS` satisfies the n^2 bound. We then provided the bound $n \cdot \log n$, inspired by the observation that the nested loop in `lengthOfLIS` at lines 14—18 updates a value used in the loop guard with division by a constant; $\textsc{Campy}$ verified that `lengthOfLIS` satisfies the bound $n \cdot \log n$ as well. To determine if `lengthOfLIS` actually executes each loop in constant time, we provided a bound of n to $\textsc{Campy}$. $\textsc{Campy}$ returned an execution of `lengthOfLIS` on which it does not satisfy the bound n.

SmartSieve The Smart Sieve Problem hosted on CodeChef is to take two integers `MAX` and `k` and return an integer array containing the prime factors of `k` up to `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

```
1  public static int[] SmartSieve(int MAX, int k) {
2    boolean[] v = new boolean[MAX];
3    int[] sp = new int[MAX];
4    for (int i = 2; i < MAX; i += 2)
5      sp[i] = 2;
6    for (int i = 3; i < MAX; i += 2) {
7      if (!v[i]) {
8        sp[i] = i;
9        for (int j = i; j * i < MAX; j += 2) {
10         if (!v[j * i]) {
11           v[j * i] = true;
12           sp[j * i] = i; } } } }
13   int[] ans = new int[sp.length];
14   int j = 0;
15   while(k > 1) {
16     ans[j++] = sp[k];
17     k /= sp[k]; }
18   return ans; }
```

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

```
1  public static boolean done = false;
2  public static void
3    Cycle(int v, int x, int k, int[] d, int[] a, List<int>[] g) {
4    if (done) return;
5    if (d[v] > 0) {
6      if (x - d[v] > k) return a;
7        for (int i = d[v]; i < x; i ++) {
8          System.out.print(a[i] + " "); }
9        done = true; }
10     return; }
11   d[v] = x;
12   for (int i = 0; i < g[v].size(); i++) {
13     int cur = (Integer) g[v].get(i);
14     a[x + 1] = cur;
15     Cycle(cur, x + 1, k); }
16   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 immediate 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, $\texttt{sp}[k] \geq 2$.

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. Fig. 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 $n^2 \cdot k$, 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 `Test`i 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 §5.1, 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 `Jewel2` contains a surprising coefficient of 350. This bound is immediate from the fact that `Jewel2` 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 (Gulavani and Gulwani 2008; Gulwani et al. 2009a,b). Another approach generates a transition system as a disjunctive binary relation by selectively expanding the control-flow graph, and then deriving ranking functions by applying rules from a proof system, using pattern matching (Gulwani and Zuleger 2010). Another approach analyzes Java bytecode under a given cost 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. 2014). 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

Program Structure				Bound	Performance	
Name	LoC	Loops	Nesting		Time (s)	Mem (MB)
Array2	39	4	2	n^2	5.3	12.4
				n	5.7	13.3
BirthdayCandles	43	4	2	$t \cdot n$	3.1	15.7
				t	2.9	14.4
Bit	61	2	2	n^2	2.8	13.6
				n	3.1	13.6
CodeChefJava	45	4	2	$3 \cdot t \cdot s$	3.2	46.0
				t	2.9	36.0
DRGNBOOL	51	3	2	$n \cdot (a + b)$	5.1	25.4
				$(a + b)$	3.3	14.5
FibonacciIterative	18	1	1	n	2.7	13.8
				10	3.3	14.1
Ideone	42	2	2	$n \cdot a$	3.5	15.1
				n	2.6	13.4
JewelAndStone	37	3	3	$t \cdot x \cdot y$	4.6	15.2
				$x \cdot y$	3.0	13.8
Jewel2	40	3	2	$350 \cdot t$	3.5	16.8
				10	3.1	13.3
LIS	47	3	2	$n \cdot (\log n)$	3.4	14.6
				n	2.8	13.1
Loops_1	30	2	2	$t \cdot n$	3.4	36.0
				n	2.8	34.0
Pie	34	3	2	$t \cdot 2n$	2.9	14.1
				$t \cdot n$	3.1	13.8
Scroll	58	3	2	$t \cdot (\log a + \log b)$	3.1	13.6
				t	2.9	16.1
SmartSieve	31	2	2	$n^2 + \log n$	4.4	20.2
				10	3.7	18.1
Sweet	125	2	2	$t \cdot n$	3.5	14.2
				n	6.4	20.5
Test0	22	2	1	n	2.7	13.1
				10	2.9	13.1
Test1	45	3	2	$2 \cdot t \cdot n$	3.7	15.2
				n	3.2	12.8
Test2	36	3	2	$n \cdot t$	3.8	13.9
				n	3.2	13.1
Test3	40	4	2	$3 \cdot t \cdot n$	4.3	10.6
				n	3.1	10.8
Test4	37	3	2	$c \cdot n$	3.1	14.7
				c	3.8	16.1
Test5	77	6	3	$t \cdot (n + n^2)$	3.4	54.8
				n^2	2.8	138.0
Test6	36	4	3	$n^2 \log n$	4.1	52.9
				n^2	5.3	47.3

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 looses 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.

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).

CAMPY addresses a problem that is related to, but distinct from, bound analysis, namely bound *verification*. I.e., CAMPY takes a taking 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

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 (§4.3.2) could be used to select relevant non-linear terms for synthesizing non-linear numerical domains, a critical problem in developing effective bound analyses (Gulavani and Gulwani 2008).

CLARITY attempts to find a specific class of performance bugs in programs that maintain collections in data structures (Olivo et al. 2015). In particular, CLARITY uses points-to analysis to determine if a program traverses a collection multiple times without modifying the collection and, if so, determines that the program has a performance bug. CLARITY aggressively applies information about its specific problem context that enable it to analyze far larger codebases than CAMPY; CAMPY can potentially verify or disprove a more general class of step-count bounds than CLARITY.

Worst-Case Execution Time The *worst-case execution time (WCET)* problem is to determine the worst possible execution time of a program (Wilhelm et al. 2008). WCET problems are typically defined for real-time programs and systems, for which all iterations and recursion are explicitly bounded. Analyses that address the WCET problem are primarily concerned with accurately modeling the effects of various complex features of hardware state that may affect performance, such as the cache and branch speculation. While our work also addresses a problem related to predicting the performance of a program, we assume that a model of the cost of each instruction is given, and primarily consider the problem of reasoning accurately about unbounded iterations and recursion.

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 (Henzinger et al. 2004). 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 *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 *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; Totla 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.

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. 2013; Jovanović 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 undecidable 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 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 SYGUS, CAMPY's technique for enumerating terms as restricted by vocabularies of subformulas is distinct from search strategies proposed in all prior work on SYGUS, and is critical to ensuring that path invariants can be extracted as interpolants from the resulting unsatisfiable formula. Other 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, TP$[\mathcal{T}]$, an extension $\mathcal{T}$ of a decidable theory. TP$[\mathcal{T}]$ attempts to prove or disprove that a given $\mathcal{T}$-formula φ is unsatisfiable by selectively refining a weakening of $\mathcal{T}$ by enumerating quantifier-free $\mathcal{T}$-formulas, guided by the structure of φ.

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.

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

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1), 2012.

C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, 2010.

R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56 (3), 2009.

R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2015.

T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In *CADE*, 2009.

campy. campy. `http://www.cc.gatech.edu/~wharris/campy.html`, 2016. Accessed: 2016 Nov 6.

Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI*, 2014.

Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *PLDI*, 2015.

codechef. Programming competition, programming contest, online computer programming. `https://www.codechef.com/`, 2016. Accessed: 2016 June 14.

codeforces. Programming competitions and contests, programming community. `https://www.codeforces.com/`, 2016. Accessed: 2016 June 14.

G. E. Collins. *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*. Springer, 1998.

cve-2011-3191. CVE - CVE-2011-3191. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3192`, 2015. Accessed: 2015 July.

L. Dai, B. Xia, and N. Zhan. Generating non-linear interpolants by semidefinite programming. In *CAV*, 2013.

L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

framework. A framework for analyzing and transforming Java and Android applications. `https://sable.github.io/soot/`, 2016. Accessed: 2016.

K. Gödel, S. C. Kleene, and J. B. Rosser. *On undecidable propositions of formal mathematical systems*. Institute for Advanced Study Princeton, NJ, 1934.

B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, 2008.

S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, 2010.

S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009a.

S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, 2009b.

S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, 2010.

J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, 2011.

D. Jovanović and L. De Moura. Solving non-linear arithmetic. In *Automated Reasoning*, 2012.

leetcode. LeetCode online judge. `https://leetcode.com/`, 2016. Accessed: 2016.

LIS. Problem LIS on LeetCode. `https://leetcode.com/problems/longest-increasing-subsequence/`, 2016. Accessed: 2016.

K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2), 1979.

O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, 2015.

A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *CAV*, 2015.

M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, 2014.

V. Sofronie-Stokkermans. Interpolation in local theory extensions. In *Logical Methods in Computer Science*, 2008.

A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

A. Tarski. A decision method for elementary algebra and geometry. *Texts and Monogrpahs in Symbolic Computation*, 1951.

N. Totla and T. Wies. Complete instantiation based interpolation. In *POPL*, 2013.

A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. In *PLDI*, 2013.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

z3. Z3prover/z3 - github. `https://github.com/Z3Prover/z3`, 2015. Accessed: 2015 Nov 7.