Automated Resource Analysis with Coq Proof Objects[⋆]

Q. Carbonneaux[1], J. Hoffmann[2], T. Reps[3], and Z. Shao[1]

[1] Yale University
[2] Carnegie Mellon University
[3] University of Wisconsin and GrammaTech, Inc.

Abstract. This paper addresses the problem of automatically performing resource-bound analysis, which can help programmers understand the performance characteristics of their programs. We introduce a method for resource-bound inference that (i) is compositional, (ii) produces machine-checkable certificates of the resource bounds obtained, and (iii) features a sound mechanism for user interaction if the inference fails. The technique handles recursive procedures and has the ability to exploit any known program invariants. An experimental evaluation with an implementation in the tool Pastis shows that the new analysis is competitive with state-of-the-art resource-bound tools while also creating Coq certificates.

1 Introduction

To help developers better understand the performance of programs at compile time, the programming-language research community has been developing techniques and tools that can automatically and statically analyze the resource consumption of programs [1, 2, 4, 7, 13, 20, 23, 28, 31]. Most of these techniques derive symbolic worst-case bounds that depend on the sizes of program variables or the arguments of a function. Deriving such bounds for arbitrary programs is an undecidable problem. However, existing tools deliver impressive results for certain classes of programs. Tools deriving bounds on imperative programs include KoAT [10], Rank [5], CoFloCo [16], Loopus [30], and C4B [11]. RAML [17] is able to derive complex bounds on functional programs.

State-of-the-art bound analysis tools suffer from two major shortcomings. First, when the inference of a resource bounds fails, the user has no other choice than to either rewrite the input program or modify the tool itself (the second

[⋆] Supported, in part, by a gift from Rajiv and Ritu Batra; by NSF under grants 1521523 and 1319671; by AFRL under DARPA MUSE award FA8750-14-2-0270, DARPA STAC award FA8750-15-C-0082, and DARPA award FA8750-16-2-0274; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

option being only available to experts). Second, many automated tools are based on sophisticated algorithms relying on subtle invariants for their correctness. Even tools based on time-tested programming-language devices like type systems and program logics have complex implementations that are prone to bugs.

The goal of this paper is to address these shortcomings. We base our work on automatic amortized resource analysis (AARA), a technique that statically derives concrete (non-asymptotic) resource bounds. AARA is implemented in the tools C4B for imperative programs and RAML for functional programs. The benefits of AARA include compositionality—by generating compositional and local constraint systems—and reduction of resource-bound inference to efficient off-the-shelf linear programming (LP).

Contributions. First, we present a new unifying framework for proving the soundness of AARA-based systems for low-level code. The framework applies directly to low-level programs parameterized with abstract base constructs. This parameterized presentation allows, similarly to the theory of abstract interpretation, multiple instantiations depending on the programs to be analyzed. We also introduce *rewrite functions*, a new technical device to encode weakening that is amenable to sound user interaction.

Second, to demonstrate the effectiveness of these new ideas, we have implemented them in a new tool called Pastis.

– Thanks to our new parametric framework, Pastis is the first AARA-based tool to generate polynomial bounds on low-level integer programs with recursive procedures.
– Thanks to rewrite functions, Pastis is the first resource analysis tool to provide a sound mechanism for user interaction when the inference of a bound fails.
– Thanks to the logical foundations of our framework, and to the certificates provided by the LP solver, Pastis is the first resource analysis tool able to automatically generate proof certificates of the validity of its bounds.

Third, to show that Pastis is practical, we have implemented an LLVM frontend to our new analysis and evaluated it on more than 200,000 lines of C code against state-of-the-art tools. Surprisingly for a tool generating proof objects, we are able to report that Pastis is competitive and can successfully generate many polynomial bounds.

Acknowledgments. We thank Vilhelm Sjöberg and Lionel Rieg for their helpful suggestions during the implementation of proof certificates in Coq.

2 Setting

Programs will be represented as standard interprocedural control-flow graphs. This assumption does not tie the presentation of our analysis to a specific set of statements; moreover, it allows the technique to be applied to unstructured input programs, including ones written in bytecode and other low-level languages.

Syntax. A program is represented as a directed graph where nodes are *program points* and edges bear program *actions*. Intuitively the program counter jumps from node to node following edges and updates the program state according to

$$\frac{(\ell, v \leftarrow e, \ell') \in E}{(\ell, \sigma) \to (\ell', \sigma[\llbracket e \rrbracket_\sigma / v])} \qquad \frac{(\ell, \mathsf{weaken}, \ell') \in E}{(\ell, \sigma) \to (\ell', \sigma)} \qquad \frac{(\ell, \mathsf{guard}\ e, \ell') \in E \qquad \llbracket e \rrbracket_\sigma \in \mathsf{OK}}{(\ell, \sigma) \to (\ell', \sigma)}$$

$$\frac{(\ell, \mathsf{call}\ P, \ell') \in E \qquad \begin{array}{c} \Delta(P) = (\_, \ell_e, \ell_x) \\ (\ell_e, \sigma) \to^* (\ell_x, \sigma') \end{array} \qquad \sigma''(v) = \begin{cases} \sigma'(v) & \text{if } v \in G \\ \sigma(v) & \text{otherwise} \end{cases}}{(\ell, \sigma) \to (\ell', \sigma'')}$$

Fig. 1. Mixed-step semantics of a program (E, Δ, G). Non-call steps use a classic small-step transition while calls are atomically executed using $\to^*$.

the actions it encounters. We use the three sets $\mathcal{L}$, $\mathcal{V}$, and $\mathcal{P}$, respectively, for the program points, the variable names, and the procedure names.

A procedure is represented by a tuple (L, ℓ_e, ℓ_x) where $L \subseteq \mathcal{V}$ is the set of local variables for the procedure and $\ell_e, \ell_x \in \mathcal{L}$ are respectively the entry and exit points of the procedure.

$$Act := \; v \leftarrow e \mid \mathsf{call}\ P \mid \mathsf{guard}\ e \mid \mathsf{weaken}$$

An action can be the assignment of a variable $v \leftarrow e$ where $v \in \mathcal{V}$ and e is an expression. We leave the syntax of expressions abstract because it is irrelevant for most of the presentation that follows. An action can also be a call $\mathsf{call}\ P$ to a procedure $P \in \mathcal{P}$. The arguments and return values are passed using global variables. We also include guards as actions. They are used to represent conditional statements and block the execution if their condition is not validated at runtime. Finally, an action can be an explicit weakening hint weaken. Such a hint can be inserted by the user or by a pre-processing heuristic; it does not have any semantic effect but is used by our analysis to perform potential rewrites.

We now define a program as a triple (E, Δ, G) where:
- $E \subseteq (\mathcal{L} \times Act \times \mathcal{L})$ is the set of all edges in the program;
- Δ is a map from procedure names $\mathcal{P}$ to procedures;
- and $G \subseteq \mathcal{V}$ is the set of global variables.

In addition, we require that no two procedures in the program share any local variables, and that the sets of global variables and local variables are disjoint. These properties can be ensured by pre-processing.

Semantics. An execution state is a pair (ℓ, σ) where $\sigma \in \Sigma$ is a program state that maps variables $\mathcal{V}$ to a value domain $\mathcal{D}$, and $\ell \in \mathcal{L}$ is the program counter.

We define single-step execution of the program as a transition relation on execution states. When evaluating an edge of the program, the change made to the program state is determined by the action labelling the edge. For each program state σ we assume that we have an evaluation function $\llbracket \cdot \rrbracket_\sigma$ that maps expressions to the value domain $\mathcal{D}$. We also require the presence of a predicate $\mathsf{OK} \subseteq \mathcal{D}$ on the value domain to check the validity of conditions in guards (e.g., a singleton $\{\mathsf{true}\}$). We use $\sigma[u/v]$ to denote a new program state that extends σ by updating the binding of v to the value u. The complete operational semantics of programs is given in Figure 1. We write $\to^*$ to denote the reflexive transitive closure of the step relation $\to$.

3 Introductory Example

We now illustrate the potential-based technique on the program shown in Figure 2. For presentation purposes, we use a Python-like syntax; the control-flow graph is derived in a standard way from this syntax.

The essence of the potential method is to *annotate* each program point ℓ with a *potential function* $\Gamma_\ell \in (\Sigma \to \mathbb{Q})$, which maps a program state to a rational number. In the example, the potential functions are displayed between curly braces. The potential functions are subject to local constraints that enforce the condition that, during each program execution, the values of the successive potential functions encountered are *non-increasing*. Thus, if the constraints are met, for any execution $(\ell_0, \sigma_0) \to^* (\ell_n, \sigma_n)$, the value $\Gamma_{\ell_0}(\sigma_0)$ of the annotation of the initial program point is an upper bound on the value $\Gamma_{\ell_n}(\sigma_n)$ of the annotation of the final point. The full set of constraints is given in Section 4; annotations matching them are said *admissible*.

```
1  {3.25}
2  k, z = 0, 0
3  {1/4 · (13 − k) + z}
4  while k < 10 and random:
5      {1/4 · (13 − k) + z} =
6      {1/4 · (13 − (k+4)) + 1 + z}
7      k = k + 4
8      {1/4 · (13 − k) + 1 + z}
9      z = z + 1
10     {1/4 · (13 − k) + z}
11 ⩾ {z}
```

Fig. 2. Example.

As we will see, the annotation of the example is admissible, which lets us deduce that 3.25 is an upper bound on the final value of loop counter z. We often refer to the final annotation as a potential *goal*, because that potential function defines the quantity we are looking to bound. In the example, the goal is $\{z\}$.

We now argue that the annotation of the example is admissible: (i) for each assignment "$v \leftarrow e$" in the program, the annotations Γ and Γ', before and after the assignment, respectively, are such that for any state σ, $\Gamma(\sigma) = \Gamma'(\sigma[[\![e]\!]_\sigma/v])$ (that is, they are non-increasing around the assignments); and (ii) for any state σ reachable at line 11, the potential annotations $\Gamma_{10} := \{1/4 \cdot (13 - k) + z\}$ and $\Gamma_{11} := \{z\}$ satisfy $\Gamma_{10}(\sigma) \geqslant \Gamma_{11}(\sigma)$ (indeed, $\sigma(k) \leqslant 13$ at that point). Thus, all steps taken by the program keep the potential non-increasing, and the annotation is admissible.

It is sometimes useful to understand a potential-function annotation "$\{e(\boldsymbol{v})\}$" over $\boldsymbol{v} \subseteq \mathcal{V}$ as an assertion "$\{e(\boldsymbol{v}) \geqslant goal_{\mathit{final}}\}$." From this point of view, a potential-function annotation is a two-vocabulary assertion, where $e(\boldsymbol{v})$ is an expression in the current-state vocabulary (and evaluated in the current state), and $goal_{\mathit{final}}$ is an expression (over $\mathcal{V}$) in the final-state vocabulary (and evaluated in the final state). Thus, line 5 can be read as the assertion "$\{1/4 \cdot (13 - k) + z \geqslant z_{\mathit{final}}\}$," and line 1 can be read as the assertion "$\{3.25 \geqslant z_{\mathit{final}}\}$."

In the remainder of the paper, we explain how to generalize the reasoning we did in this section to programs with multiple procedures, and how to automate it using linear programming. To this end, we formally define admissibility conditions of potential annotations and then encode these conditions into linear programs. As in the example, our framework separates assignment constraints from weakening constraints. To handle the latter we introduce *rewrite functions*.

4 Interprocedural Potential Annotations

In this section, we consider a fixed program (E, Δ, G). A procedure P in the program has entry and exit points $P_e, P_x \in \mathcal{L}$, respectively. A state σ at $\ell \in \mathcal{L}$ is *reachable from* (P_e, σ_0) when there is an *execution trace*

$$(P_e, \sigma_0) \to (\ell_1, \sigma_1) \ldots \to (\ell_n, \sigma_n),$$

such that $\sigma_n = \sigma \wedge \ell_n = \ell$, or $(\ell_n, \mathsf{call}\ Q, \_) \in E$ and σ at ℓ is reachable from (Q_e, σ_n). The latter disjunct is necessary because we use mixed-step semantics: calls are represented by a single step $\to$, and $\to^*$ always relates two points in the same procedure. Finally, we say that σ at ℓ is *reachable (from P_e)*, when there is an initial state σ_0 such that σ at ℓ is reachable from (P_e, σ_0).

A *potential function* is a function that maps program states to rational numbers. A *procedure annotation* Γ associates a potential function Γ_ℓ with each program point ℓ in a procedure. An *interprocedural potential annotation* (IPA) Ψ maps each procedure name P to a set of procedure annotations $\Psi(P)$. Sets of annotations are used because, depending on the context in which a procedure is called, different annotations might be used. A *goal* is a potential function of special interest: it is the quantity at the end of the program that we are seeking to bound in terms of the initial state.

Definition 1 (Admissible IPA). *We say that an IPA Ψ for the program (E, Δ, G) is* ***admissible*** *for a goal g and an entry and exit point S_e and S_x in a procedure S when:*

(A1) $\exists \Gamma \in \Psi(S).\, \Gamma_{S_x} = g$;

and for every procedure P, edge $(\ell, a, \ell') \in E$, annotation $\Gamma \in \Psi(P)$, and state σ reachable (from S_e) at ℓ:

(A2) if a is weaken, then $\Gamma_\ell(\sigma) \geqslant \Gamma_{\ell'}(\sigma)$;

(A3) if a is $\mathsf{guard}\ e$, then $\Gamma_\ell(\sigma) = \Gamma_{\ell'}(\sigma)$;

(A4) if a is $v \leftarrow e$, then $\Gamma_\ell(\sigma) = \Gamma_{\ell'}(\sigma[e/v])$;

(A5) if a is $\mathsf{call}\ Q$, then $\exists \Gamma' \in \Psi(Q).\, \Gamma_\ell = \Gamma'_{Q_e} \wedge \Gamma_{\ell'} = \Gamma'_{Q_x}$.

The reachability condition in Definition 1 ensures that the inequalities are required to hold only for states that can actually appear in an actual execution trace. These admissibility conditions provide us with a principled way to obtain upper bounds on the potential goal, as demonstrated by Proposition 1. Moreover, the local nature of these conditions makes the generation of constraints described in Section 6 a local and compositional process.

Proposition 1. *For every program state σ reachable at ℓ, if Ψ is an admissible IPA and (ℓ', σ') is such that $(\ell, \sigma) \to^* (\ell', \sigma')$, then for all $\Gamma \in \Psi(P)$, $\Gamma_\ell(\sigma) \geqslant \Gamma_{\ell'}(\sigma')$.*

Proposition 1 and (A1) imply the existence of a procedure annotation $\Gamma \in \Psi(S)$ such that any execution state (ℓ, σ) on a trace $(S_e, \sigma_e) \to^* (S_x, \sigma_x)$ gives an upper bound $\Gamma_\ell(\sigma)$ on the goal evaluated on the final state $g(\sigma_x)$; in particular, $\Gamma_{S_e}(\sigma_e) \geqslant g(\sigma_x)$. This property is intuitively clear as all the admissibility conditions of Definition 1 constrain the values of the potential functions to decrease as the program progresses. Suppose now that g measures, in the final state, the size of the value stored in a variable v (e.g., the length of a list or the absolute value of an integer). Then each of the potential functions in the procedure annotation Γ expresses an upper bound on the size of v in the final state in terms of the *local state* at other program points. That is, our analysis tracks the size of v backwards through the program points.

Note that the negation of any admissible IPA for a goal $-g$ will provide *lower bounds* on the goal g. This observation shows that the orientation we chose for the inequality of (A2) is irrelevant and allows computation of both upper and lower bounds.

Resource Analysis. When the potential goal g is set, finding an admissible IPA for a program gives upper bounds on g at all program points. These bounds can be readily used to account for the resource consumption of a program. For instance, if a bound on the number of iterations of a loop is desired, the loop can be instrumented with a counter z initially set to 0 and incremented at every iteration.

```
z = 0         # instrumentation counter
while ...:
    ...
    z = z + 1  # counting iterations
```

The final value of z is the number of times the loop body was executed. So if we write ℓ_1 and ℓ_2 for the program points before and after the loop, respectively, an admissible IPA Ψ for $g(\sigma) := \sigma(z)$ will provide a bound Γ_{ℓ_1} for the number of loop iterations. The actual number of iterations when the loop starts in state σ_1 is bounded by $\Gamma_{\ell_1}(\sigma_1)$. Note that the bound on the value of z in the *final* state is expressed in terms of the values of variables in the *initial* state: Annotations provide cross-program invariants.

High-Water Mark Resource Consumption. For resources that can be freed (e.g. memory, connections, etc.), we often want to know the highest amount of resource required: the "net" consumption at the end of the program is not enough. On this aspect, our admissibility departs slightly from previous works on automated amortized analysis [11, 17]. In these papers, a sound annotation bounds not only the final value of a resource counter, but also the high-water mark of this counter. So if a program allocates 3 integers, then frees them, an admissible annotation in this paper is 0, since no memory is in use at the end of the execution. However, the high-water mark consumption of this program is 3.

We will now explain how to modify our setting to accommodate this difference. In contrast to previous work that has the resource counter as a semantic instrumentation, we can use a standard program variable z to track the available resources. Allocation grows this counter variable and freeing lessens it. We now show that from an admissible IPA Ψ, by adding an additional requirement, we can obtain a *water-mark-tracking* IPA.

Proposition 2. *A water-mark-tracking IPA Ψ, is an admissible IPA such that: ($\star$) for any point ℓ, $\Gamma \in \Psi(P)$, and state σ reachable at ℓ, $\Gamma_\ell(\sigma) \geqslant \sigma(z)$.*

Given such an IPA Ψ, a trace $(\ell_0, \sigma_0) \to (\ell_1, \sigma_1) \to \ldots \to (\ell_n, \sigma_n)$, and $\Gamma \in \Psi(P)$, we have $\Gamma_{\ell_i}(\sigma_i) \geqslant \max_{j \geqslant i} \sigma_j(z)$.

Proof. We give the intuition on a trace with no procedure calls. By induction on $n - i$. If $n = i$, the result holds by ($\star$). Otherwise, we have $\Gamma_{\ell_i}(\sigma_i) \geqslant \Gamma_{\ell_{i+1}}(\sigma_{i+1})$ and, by induction, $\Gamma_{\ell_i}(\sigma) \geqslant \max_{j \geqslant i+1} \sigma_j(z)$. We conclude using ($\star$) on σ_i.

Note that, like admissibility conditions, the condition ($\star$) is *local* but implies a *global* water-mark property. In a practical implementation, the condition ($\star$) can be enforced naturally using the framework we describe in Section 5. The non-negativity requirement of the potential in previous works is exactly the condition ($\star$), since their potential functions are essentially $\Gamma - z$ in this work.

5 Rewrite Functions

Because of admissibility condition (A2), any automated analysis using the potential method needs to enforce *weakening* conditions of the form $\Gamma \geqslant \Gamma'$ on potential functions. In this section, we present a new principled approach to weakening in AARA-based systems: *rewrite functions*. To our knowledge, they subsume all the existing potential-weakening and potential-rewriting mechanisms. Moreover, they provide a language for a user to interact with an AARA-based system, which is a feature unique to this work.

Definition 2 (Rewrite Function). *We say that F_ℓ is a rewrite function at a program point ℓ when for any state σ reachable at ℓ, $F_\ell(\sigma) \geqslant 0$.*

In other words, F_ℓ is the left-hand side of a program invariant $F_\ell(\sigma) \geqslant 0$.

Example. Assume that the potential function at program point ℓ is $\Gamma := 2y + z$, and we are looking for a weakening $\Gamma' := k' + k'_x \cdot x + k'_y \cdot y + k'_z \cdot z$ such that $\Gamma \geqslant \Gamma'$ holds. We will assume that nothing is known about the sign of variables x, y, and z, and thus pointwise constraints $2 \geqslant k'_y$, $1 \geqslant k'_z$, $0 \geqslant k'_x$, and $0 \geqslant k'$ would not ensure that $\Gamma \geqslant \Gamma'$ holds.

Now assume that $y \geqslant z + x$ and $z \geqslant x$ are invariants that hold at ℓ, which means that we have two rewrite functions $F_1 := -x + y - z$ and $F_2 := -x + z$. Write Γ as follows:

$$\Gamma = 2y + z - (0 \cdot F_1 + 0 \cdot F_2). \qquad (1)$$

Because F_1 and F_2 mean that $-x + y - z \geqslant 0$ and $-x + z \geqslant 0$, respectively, we obtain a value that is less than or equal to Γ by choosing any positive coefficients to replace either/both of the 0s in Equation (1). For instance, by choosing both coefficients to be 2, we have

$$\Gamma = 2y + z - (0 \cdot F_1 + 0 \cdot F_2) \geqslant 2y + z - (2 \cdot F_1 + 2 \cdot F_2) = 4x + z,$$

and thus we can choose Γ' to be $4x + z$. (Rather than making specific choices for the coefficients $\{u_i\}$, however, we will leave it to the LP solver to choose values that allow it to solve the overall constraint system generated for the program.)

In short, we can systematize potential weakening as a two-step process.

1. If the original potential at ℓ is V, write the weakened potential as $V - \sum u_i \cdot F_i$, where $\{F_i\}$ is the set of rewrite functions available at ℓ. (In the example, $V = 2y + z$.)
2. Choose values for the coefficients $\{u_i\}$ such that $u_i \geqslant 0$, for all i. (In the example, $u_1 = 2 \geqslant 0$, and $u_2 = 2 \geqslant 0$.)

We can express this process using standard linear-algebra notation. Let the coefficients of Γ (i.e., $\boldsymbol{k}$), Γ' ($\boldsymbol{k'}$), and $\boldsymbol{u}$ be column vectors, and let the matrix F represent the set $\{F_i\}$ of rewrite functions, with each F_i a column of F. The constraints generated to express the allowable rewrites of Γ into Γ' as per items 1 and 2 are

$$(\boldsymbol{k'} = \boldsymbol{k} - F\boldsymbol{u}) \wedge (\boldsymbol{u} \geqslant 0). \tag{2}$$

In the example above, we have

$$\begin{pmatrix} k' \\ k'_x \\ k'_y \\ k'_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ -1 & -1 \\ 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \wedge \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \geqslant \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

There are many solutions to this system. The LP solver is free to pick any one that allows the overall system of constraints to be satisfied.

Rewrite Idempotence. An interesting consequence of the algebraic formulation of weakenings introduced above is that, for a fixed set of rewrite functions, multiple compositions of the linear weakening system are never necessary. In practice, this property means that our automated system never needs to apply a weakening operation twice consecutively: once is always sufficient.

We call the set of coefficients for the initial and final potential in a satisfying assignment of a linear weakening system a *solution.* Then the following property holds:

Proposition 3 (Rewrite Idempotence). *The set of solutions for one weakening application and the set of solutions for two or more composed weakening applications are identical.*

Rewrite Hints. In practice, a system needs a source of rewrite functions to use at the different points in the program. Because rewrite functions are obtained from program invariants, abstract interpretation [15]—using various abstract domains—can be used to obtain rewrite functions for the different points in the program: given such a source of invariants, a system would employ heuristics to choose which invariants should be used as rewrite functions in the constraint system passed to the LP solver.

Occasionally, however, a program requires a complex transfer of potential. With previous implementations of amortized analysis, one was faced with two choices: either rewrite the program, or modify the analysis. Both alternatives have drawbacks.

– Rewriting the program provides only an indirect means for obtaining the desired effect, and it can be hard to understand whether a given program rewriting will allow the analyzer to establish the desired bound.

– Modifying the analysis to enable more complex transfers of potential requires the whole soundness proof to be redone as well as a good knowledge of the implementation of the analyzer.

Rewrite functions offer a third option. A programmer can manually specify rewrite functions as hints to be used to analyze the complex parts of a program. These hints, in contrast with typical assertions, both have no runtime effect and do not compromise soundness. In particular, before using a rewrite function, the analyzer would ask an oracle (e.g., an SMT solver or an abstract interpreter) if the value of the user-supplied rewrite function is provably non-negative at that program point. This approach provides a good fallback mechanism in case the heuristics of the analyzer are not sophisticated enough to identify an appropriate set of rewrite functions.

6 Automatic Potential Inference

In this section, we describe a general framework to automate the inference of potential. To emphasize the modularity of the method presented, we leave expressions and base potential functions $(b_i)_{1\leqslant i\leqslant N}$ unspecified. However, we make two formal requirements necessary to automate the inference process.

Requirement 1: Basis Stability. Ideally, we would like the basis to be linearly stable under all expression substitutions. However, this requirement is too strong in practice. As an escape hatch, for each assignment "$x \leftarrow e$", we assume an *exclusion set* $X_{v\leftarrow e}$ of all the base functions that are not expressible as linear combinations of the basis after the substitution $[e/v]$. An important consequence of this definition is that if $b \notin X_{v\leftarrow e}$, there is a family of coefficients (k_i) such that $b[e/v] = \sum_i k_i \cdot b_i$.

As an example, we show that the requirement is met if the expressions are increments by a constant $u + c$ where $c \in \mathbb{Z}$, the base functions $(b_i)_i$ are all the monomials over program variables of total degree d or less, and $X_{v\leftarrow e} = \varnothing$ for all assignments. If the maximum power of v in b is v^n (with $n \leqslant d$), then

$$b[u + c/v] = \sum_{0\leqslant j\leqslant n} \binom{n}{j} c^{n-j} \frac{u^j b}{v^n}.$$

In this sum, the products $\binom{n}{j}c^{n-j}$ are the constant coefficients (k_i) above and the multiplicands $u^j b/v^n$ are base functions. Indeed, they are monomials over program variables of degree $n \leqslant d$. Note that when v does not appear in b, n is 0 and the above sum correctly degenerates to b.

The exclusion set enables the implementation of practical tools. For example, during pre-processing it is often desirable to abstract a statement into one or more non-deterministic assignments "$v \leftarrow \star$". These assignments cause any base function that depends on the assigned variable to not be linearly stable. This situation leads us to define $X_{v\leftarrow\star} = \{b_i \mid v \in b_i\}$, where we write $v \in b_i$ to express that b_i depends on v.

Requirement 2: Rewrite Functions. We also assume that we are provided with a set of rewrite functions for every program point. Recall that, by Definition 2, for any program point ℓ and any program state σ reachable at that point, such a rewrite function F_ℓ satisfies $F_\ell(\sigma) \geqslant 0$.

Generating a Linear System. In the following, we explain how to generate a procedure annotation for a procedure S, given a potential goal g. We assume that procedure annotations of all the procedures called by S are available. In an actual implementation, the method described here would be implemented by a function calling itself recursively to generate all the procedure annotations needed transitively by the input program. The annotation generated is parameterized by LP variables that are constrained by a linear system. We also sketch the proof that a satisfying assignment of the linear system describes an admissible potential annotation, as specified in Definition 1.

The potential function associated with each program point of S is a "template" potential function $\Gamma_\ell(\sigma) = \sum_i k_i \cdot b_i(\sigma)$, where each k_i belongs to LP, a set of LP variables, and $(b_i)_i$ is the family of base functions. For every program edge (ℓ, a, ℓ'), we constrain the coefficients $(k_i)_i$ and $(k'_i)_i$ of Γ_ℓ and $\Gamma_{\ell'}$ by case analysis on the kind of the action a. To be able to use matrix notation, we define the column vectors $\boldsymbol{k} := (k_1, \ldots, k_N)^\intercal$ and $\boldsymbol{k'} := (k'_1, \ldots, k'_N)^\intercal$.

• **Case $v \leftarrow e$.** Because of Requirement 1, substituting an expression e for v in a base function $b_i \notin X_{v\leftarrow e}$ is a linear operation. Without loss of generality, assume that the exclusion set $X_{v\leftarrow e}$ is $\{b_i \mid N_X < i \leqslant N\}$. The constraints generated are

$$(\boldsymbol{k} = Q\boldsymbol{k'}) \wedge \bigwedge_{N_X < i \leqslant N} k'_i = 0.$$

The $N \times N$ matrix Q contains zeroes everywhere but in the first N_X columns. The j^{th} column ($j \leqslant N_X$) is the result of the substitution $b_j[e/v]$; that is: $b_j[e/v] = \sum_i q_{i,j} \cdot b_i$. The expansion exists because $b_j \notin X_{v\leftarrow e}$ when $j \leqslant N_X$. These coefficients are constants known before the generation of the linear program, and only depend on the choice of the basis.

The transformation $\Gamma_{\ell'}[e/v]$ on $\Gamma_{\ell'}$ is constrained to be linear by setting the coefficients of base functions in the exclusion set to zero. This linear transformation is then encoded as a matrix multiplication. Thus, for all states σ, $\Gamma_\ell(\sigma) = \Gamma_{\ell'}[e/v](\sigma) = \Gamma_{\ell'}(\sigma[e/v])$ and the admissibility condition (A4) is satisfied.

• **Case weaken.** To encode a weakening via a set of linear constraints, we make use of the rewrite functions provided in Requirement 2. As explained in Section 5, we relate the potential functions Γ_ℓ and $\Gamma_{\ell'}$ using an ℓ-specific set of unknowns, $\boldsymbol{u}_\ell$, as coefficients in a linear combination of the rewrite functions available at ℓ. Following Equation (2), let F_ℓ denote the matrix of rewrite functions available at ℓ (in which the i^{th} column of F is the i^{th} rewrite function F_i). In matrix notation, the constraints generated are

$$(\boldsymbol{k'} = \boldsymbol{k} - F_\ell \boldsymbol{u}_\ell) \wedge (\boldsymbol{u} \geqslant 0).$$

• **Case call P.** When calling a procedure, we need to know what base functions depend only on global variables and what depend only on local variables of the caller. We call these sets GF and LF, respectively. Note that these two sets do not form a partition: some potential functions in the complement of $GF \cup LF$ depend on both local and global variables. We write $(k^e_i)_i$ and $(k^x_i)_i$ for the entry and exit annotations of P. With this notation, the analysis generates the following system of constraints.

- The potential associated with global variables only is passed from the caller to the callee, and fetched back after the return: $\bigwedge_{i \in GF} k_i = k^e_i \wedge k^x_i = k'_i$.
- Local variables of the callee start and end without making any contribution to the potential: $\bigwedge_{i \notin GF} k^e_i = 0 \wedge k^x_i = 0$.
- The potential associated with local variables before the call can be recovered after: $\bigwedge_{i \in LF} k'_i = k_i$.
- Finally, we consider the coefficients of base functions that depend on both local and global variables. Such coefficients are constrained to be zero at the call and return sites in the caller because we do not know how their base function will evolve through the call: $\bigwedge_{i \notin GF \cup LF} k_i = 0 \wedge k'_i = 0$.

The admissibility argument in this case is more subtle than the ones given before. It follows from inductive reasoning and leverages a *framing* lemma to pass the potential of local variables through the call. Our implementation also makes use of *resource polymorphism*, as explained in greater detail below.

• **Case** guard e**.** We require that $\boldsymbol{k'} = \boldsymbol{k}$, and the admissibility condition (A3) is trivially satisfied.

• **Potential Goal.** Finally, we constrain the annotation of the exit point Γ_{S_x} to be the potential goal g pointwise, as in the guard case above. This fulfills (A1).

Constraint Solving. The generated constraints can be solved by an off-the-shelf LP solver. To derive the best possible bound, we minimize the potential $\sum_i k^e_i \cdot b_i(\sigma)$ that is associated with the entry point of the procedure. When all the base functions $b_i(\sigma)$ are non-negative, we use a weighted sum $\sum_i w_i k^e_i$ as objective function in the linear program. The weights w_i can be set to assign a higher priority to the coefficients of higher-degree base functions b_i.

A more robust method for non-negative base functions that allows us to prioritize the minimization of the coefficients of high-degree base functions is to use the support for efficient iterative solving that is provided by modern LP solvers: We first use the objective function $\sum_i k^e_{j_i}$ to minimize the coefficients $k^e_{j_i}$ of the base functions b_{j_i} with the highest degree. If the LP solver finds a solution, then we add the constraint $\sum_i k^e_{j_i} = o_0$ where o_0 is the objective value, and re-run the solver to optimize the coefficients for the lower-degree base functions.

It is not always the case that all base functions are non-negative in the initial state. In the implementation, we use a linear system to rewrite the initial potential as $X + \sum k_i F_i$ where $(F_i)_i$ is a family of rewrite functions in the initial state. We then constrain X to be 0 and minimize the coefficients k_i as described before.

Resource Polymorphism. In practice, constraining procedures to always use the same initial and final potential is too restrictive because procedures can be executed in different contexts. In the absence of recursion, procedure bodies can be inlined, and different bounds will be derived for the various call sites. However, inlining strategies are not applicable to recursive procedures.

An example of such a situation is displayed in Figure 3(a). In this example, we look for an upper bound on n at the program exit. It is easy to show by induction that n is invariant across a call of `P`. However, if the analysis naively uses the equality constraints $\Gamma_e = \Gamma_c$ and $\Gamma_r = \Gamma_x$, no bound can be found. Indeed, Γ_x has to be set to the goal n, and then $\Gamma_r = \Gamma_x[n+1/n] = \{n+1\}$ by the admissibility condition (A4); but $n + 1 \neq n$, preventing $\Gamma_r = \Gamma_x$. One solution to this problem is to allow a framing to be performed at call sites. In the example in Figure 3(a), the frame used is 1.

With non-linear procedures, frames can be of higher degree. However, we cannot merely add a term like $5n$ to a potential function before and after a procedure call because the variable n can be modified by the procedure. A sound and practical approach to infer frames that we use in our implementation has been pioneered in [18]; it uses as a frame an annotation obtained by another run of the analysis on the callee with a smaller set of base functions.

```
def P():
    Γ_e := {n}
    if n > 0:
        n = n - 1
        Γ_c := {n+1}
        P()
        Γ_r := {n+1}
        n = n + 1
    Γ_x := {n}
```

(a)

```
def Q():
    Γ_e := {z+(n choose 2)}
    if n > 0:
        n = n - 1
        Γ_c := {z+(n choose 2)+n}
        Q()
        Γ_c := {z+n}
        z = z + n
        Γ_r := {z}
        n = n + 1
    Γ_x := {z}
```

(b)

Fig. 3. Procedures where resource-polymorphic recursion is used to infer a bound.

Consider, for example, the procedure `Q` in Figure 3(b). It is a variant of `P` in which we added the assignment "`z = z + n`" after the recursive call. Assume that our potential goal is $\Gamma_x = \{z\}$. If z_0 and n_0 are the values of z and n before the call of `Q` then we have $z = z_0 + \binom{n_0}{2}$ after the call. Consequently, Γ_e is a sound potential annotation for the entry point of `Q`. To justify this potential annotation, we attempt to use the same annotation Γ_e for the potential before the recursive call. Note, however, that we have some additional potential n available in Γ_c. To transfer this potential to the return point Γ_r we have to analyze how n changes during a call to `Q`. As with the example on Figure 3(a), n is invariant across the call, and we can perform a similar analysis on `Q` to derive $\Gamma'_e = \{n\}$ and $\Gamma'_x = \{n\}$ for the entry and exit points of `Q`. The annotations before and after the recursive call can now use the combined annotations $\Gamma_e + \Gamma'_e$ and $\Gamma_x + \Gamma'_x$, respectively.

7 Pastis: A Practical Implementation for Integer Programs

We implemented the framework of Section 6 in a tool that computes polynomial resource bounds for imperative integer programs. Programs are internally represented as described in Section 2, but the tool accepts as input both a minimal imperative language and LLVM bitcode. The expressions accepted are additions, subtractions, and multiplications of constants and program variables; a special random expression is used to represent all other operations (e.g., shifts and divisions). The base functions are picked among the monomials M defined below.

$$\text{(Monomials)} \qquad M := 1 \mid v \mid M_1 \cdot M_2 \mid \max(0, P) \qquad v \in \mathcal{V}$$
$$\text{(Polynomials)} \qquad P := k \cdot M \mid P_1 + P_2 \qquad k \in \mathbb{Q}$$

Heuristics to Select Base and Rewrite Functions. In Pastis, we make use of invariants generated by abstract interpretation to generate basis and rewrite functions. For example, if some variable v can be proved non-negative at one program point, we add the base function $\max(0, v)$ and a set of rewrite functions that will be needed to transfer potential to and from this base function. For instance, we register the rewrite function $\max(0, v) - \max(0, v - 1) - 1$. As shown in Figure 4, this rewrite function can be used before a decrement when $v \geqslant 1$. Higher-degree base functions are introduced by considering successive powers and products of linear base functions.

```
if v > 0:
  $\{\max(0, v)\} \geqslant$
  $\{\max(0, v-1)+1\}$
  v = v - 1
  $\{\max(0, v)+1\}$
```

Fig. 4.

By using as base functions the lengths of intervals that can be formed by pairs of program variables (e.g., $\max(0, b - a)$), Pastis strictly generalizes C4B [11]: any derivation in that system can be encoded in the framework of Section 6. Moreover, our work is more general because it allows higher-degree base functions, as well as base functions that do not match exactly the interval pattern.

Simple Polynomial Example. The program shown in Figure 5 has a polynomial bound on the loop counter z. In the annotations, we use a saturating subtraction operation $a \dot{-} b := \max(0, a - b)$. (In our implementation, $\max(0, \cdot)$ is used, as explained above.) As we will see, the annotations of the example already form a valid certificate. We found them by solving the system of linear constraints derived using the method of Section 6 on this program text. In this example, because there are no procedure calls, there are only two kinds of checks to do: (i) assignment checks, and (ii) potential-rewrite checks. The potential rewrites are all marked with a "$\geqslant$" sign (lines 4, 10, 15, and 16).

```
1  def nested():
2    $\{\binom{n \dot{-} 0}{2} + z\}$
3    while n > 0:
4     $\geqslant \{\binom{n \dot{-} 1}{2} + (n \dot{-} 1) + z\}$
5      n = n - 1
6      $\{\binom{n \dot{-} 0}{2} + (n \dot{-} 0) + z\}$
7      m = n
8      $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\}$
9      while m > 0:
10      $\geqslant \{\binom{n \dot{-} 0}{2} + (m \dot{-} 1) + (z+1)\}$
11       m = m - 1
12       $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + (z+1)\}$
13       z = z + 1
14       $\{\binom{n \dot{-} 0}{2} + (m \dot{-} 0) + z\}$
15    $\geqslant \{\binom{n \dot{-} 0}{2} + z\}$
16   $\geqslant \{z\}$
```

Fig. 5. Polynomial example.

The most interesting parts of the reasoning are the "potential transfers" in this program. The core idea is that the quadratic potential associated with the counter of the outer loop will, at each decrement (line 5), generate a linear potential used to pay for the increments of the counter z in the inner loop. The potential behavior of the inner loop is then very similar to the one given in the introductory example of Section 3. The only difference is that all annotations carry an extra quadratic part $\binom{n \dot{-} 0}{2}$ that remains unchanged through the loop.

The validity of the potential rewrites can be justified with rewrite functions as explained in Section 5. For example, on line 4, we use the rewrite function $F := \binom{n \dot{-} 0}{2} - \binom{n \dot{-} 1}{2} - (n \dot{-} 1)$ to show that $\binom{n \dot{-} 0}{2} + z \geqslant \binom{n \dot{-} 1}{2} + (n \dot{-} 1) + z$. Indeed, the right-hand side of the inequality can be rewritten as $\binom{n \dot{-} 0}{2} + z - (1 \cdot F)$. Note that the rewrite function F can be used on line 4 because it is under the check "`while n > 0`". It could not be used on line 2, where no information about n is known yet.

Polynomial Example with Recursion. Figure 6 contains an implementation of the core of the Quicksort algorithm. All array operations are abstracted away and we look for a bound on the variable z at the end of the procedure. Note that we pass two arguments to the procedure `qsort`. In our implementation, as in the programs of Section 2, the arguments are passed via global variables. This approach is similar to machine calling conventions that use registers to pass arguments.

The bound Γ_e on z is quadratic. We can express it precisely using the binomial basis. Indeed, we will see below that Γ_e is a tight worst-case bound on z. We left all the annotations on the inner loop unspecified because they follow exactly the same pattern as the ones in the inner loop above. The interesting parts of the derivation are the two weakening hints on lines 4 and 12, and the two recursive calls. The first weakening uses the binomial identity $\binom{X+1}{2} = \binom{X}{2} + X$. This identity is applied with $X := h \dot{-} (l+1)$ because in this branch $h \dot{-} (l+1) = h \dot{-} l - 1$. The current implementation of our system is not able to infer this rewrite function from the body of the recursive function alone; we thus use an explicit rewrite hint. Similarly on line 12, we use a hint to prove that

```
1  def qsort(l, h):
2    $\Gamma_e := \{\binom{h \dot{-} l}{2} + z\}$
3    if l < h:
4        hint
5          $\geqslant \{\binom{h \dot{-} (l+1)}{2} + (h \dot{-} (l+1)) + z\}$
6        m = l
7        $\{\binom{h \dot{-} (l+1)}{2} + (h \dot{-} (m+1)) + z\}$
8        while m < h-1 and random:
9            m = m + 1
10           z = z + 1
11       $\{\binom{h \dot{-} (l+1)}{2} + z\}$
12       hint
13         $\geqslant \{\binom{h \dot{-} (m+1)}{2} + \binom{m \dot{-} l}{2} + z\}$
14       qsort(l, m)
15       $\{\binom{h \dot{-} (m+1)}{2} + z\}$
16       qsort(m+1, h)
17   $\{z\}$
```

Fig. 6. Quicksort core.

$$\binom{X+Y}{2} \geqslant \binom{X}{2} + \binom{Y}{2} + X \cdot Y \geqslant \binom{X}{2} + \binom{Y}{2}, \qquad (3)$$

where $X := h \dot{-} (m+1)$ and $Y := m \dot{-} l$.

Note that $X + Y = h \dot{-} (l+1)$ because $l \leqslant m < h$.

Let us now discuss the recursive calls on lines 14 and 16. As in Section 6, we can describe the potential before and after the calls piecewise. On line 14, after the arguments are assigned, $\binom{m \dot{-} l}{2} + z$ will become $\binom{h \dot{-} l}{2} + z = \Gamma_e$. This quantity is the potential passed to the recursive call. The term $\binom{h \dot{-} (m+1)}{2}$ only depends on local variables and remains unchanged by the call; it is thus framed and retrieved on line 15. Finally, z, the final potential of `qsort`, is returned on line 15 and added to the frame. The call of line 16 follows a similar logic, but without any frame.

The procedure `qsort` exhibits its worst-case behavior when the internal loop goes all the way from l to $h-1$. Note that all the weakenings of the derivation are actually pure potential rewrites ($\geqslant$ is in fact $=$), except for the one on line 13. On that line, the term $X \cdot Y$ of Equation (3) is lost potential. However, in the worst-case scenario, $m = h-1$ on line 13 and thus $X \cdot Y = 0$, making the weakening a pure potential rewrite, too. Thus, in the worst case of Quicksort, no potential is ever lost and the bound $\binom{h \dot{-} l}{2} + z$ is exact: $\Gamma_e(\sigma_e) = \sigma_x(z)$, where σ_e and σ_x are, respectively, the entry and exit states of `qsort`.

8 Generation of Coq Proof Objects

In the framework of Section 6, IPAs are natural candidates for proof certificates. In this section, we explain how to leverage this observation and generate Coq proofs from the coefficients returned by a successful run of the LP solver. The Coq files generated depend on a small library described below, and can be checked completely automatically without modifications. The certified theorems state that the derived bounds are sound with respect to a Coq formalization of our interprocedural control-flow graphs and do not rely on any unproved assumptions. In particular, the certificates also include a soundness proof of the invariants that we derived with a simple abstract interpretation.

Benefits of Formal Verification. The benefits of checking bounds with a proof assistant are three-fold. First, it greatly increases the confidence in generated bounds, which is especially critical considering that we observed LP solvers silently overflow and return an unsound solution. All resource-analysis tools using LP solvers are currently vulnerable to this issue. Second, proof certificates are a license to implement aggressive heuristics and optimizations in our tool without risking unnoticed soundness issues. Third, it allows the integration of resource bounds into larger formal developments. We think that automating the inference of resource-bound theorems will enable a new class of software verification where not only the correctness is proved, but also quantitative properties are proved, such as real-time guarantees and memory usage, which are often neglected.

Coq Support Library. We implemented a support library that is used by all the Coq certificates generated. This library contains a formal definition of the control-flow graphs presented in Section 2, and a generalized version of the IPAs presented in Section 4. These generalized IPAs (Coq terms of type `IPA`) express in a single annotation the results of both the abstract interpretation and the potential annotations. A set of admissibility conditions on IPAs is defined as a predicate "`IPA_VC: IPA → Prop`", which is designed to be easily checked automatically. A theorem similar to Proposition 1 gives a semantic meaning to this verification condition.

Finally, to automate fully the checking of the certificates produced, a set of relatively small tactics is defined (130 lines of Coq); they are tightly coupled with the proof-generation module of Pastis. The checking of IPAs is split into two tasks for each edge: (i) checking the validity of the abstract-state transformation, and (ii) checking the corresponding admissibility condition in Definition 1. For (i), we use the Coq decision procedure `lia` for linear-integer arithmetic. This logic is sufficient because the abstract interpretation merely derives linear constraints between program variables. Similarly for (ii), the inequalities to check on potential annotations are linear in program variables with coefficients in $\mathbb{Q}$. Because the coefficients are constants, the checking of inequalities is reduced to $\mathbb{Z}$, and also automated using `lia`. Equalities are checked with the generic `ring` tactic. Finally, one ad hoc tactic handles the normalization of potential annotations using the $\max(0, \cdot)$ function, which is not handled natively by `lia`.

Importantly, according to the Coq reference manual [32], the `lia` tactic is complete. This property means that a failure when checking a proof certificate can only be explained by an invalid certificate. Invalid certificates can be the result of a bug in our tool—we found one in the abstract interpreter during the development—or of invalid coefficients in the potential annotations (e.g., because of overflows or rounding errors in the LP solver).

Generated Coq Files. A generated file starts by defining the program analyzed as a control-flow graph with procedure calls. Then, for each procedure and program point, the results of the abstract-interpretation procedure and the potential annotations are listed. From these two pieces of information, multiple procedure annotations are defined and aggregated in a single global IPA for the complete program (referred to as "`ipa`" in the Coq file). This IPA is proved to satisfy the verification condition in a theorem

```
Theorem admissible_ipa: IPA_VC ipa.
Proof. prove_ipa_vc. Qed.
```

The proof of admissibility is always a single call to the tactic `prove_ipa_vc` imported from the library described above. This tactic call is where the bulk of the checking happens. Finally, a user-readable theorem expresses the soundness of the bound that was generated. For example

```
Theorem bound_valid: forall s1 s2,
  steps P_start (proc_start P_start) s1 (proc_end P_start) s2 →
  ( s2 G_z <= (3#2) * s1 G_g + (1#2) * s1 G_g^2 )%Q.
Proof. prove_bound ipa admissible_ipa P_start. Qed.
```

In this theorem, `s1` and `s2` are the initial and final program states, respectively, of the execution of the `start` procedure (which appears in the hypothesis as "`steps P_start ...`"). In this example, the goal was set to $\{z\}$, i.e., the value of the global variable z at the end of the execution (shown as "`s2 G_z`"). It is bounded in terms of the initial value of another global variable g. A rational number a/b is represented in Coq using the notation `a#b`. The proof of this theorem leverages both the main theorem about admissible IPAs—proved once and for all in our library—and one annotation of the `start` procedure that has the goal as final potential.

Rewrite Functions. Rewrite functions are also checked for validity in the Coq implementation. One challenge in this step is that rewrite functions often contain non-linear expressions (e.g., the binomial functions used in Section 7). This issue prevents us from using the built-in automation of Coq directly. To solve this issue, we used a small domain-specific language (DSL). Elements of this DSL are interpreted (in Coq) as a pair of an actual rewrite function (a map from states to rationals) and a conjunction of linear conditions. The conjunction is, by design of the DSL, a sufficient condition proving the non-negativity of the rewrite function. This trick lets us once more reuse Coq's linear-integer automation to lighten the proof burden. When checking weakening steps in the tactic `prove_ipa_vc`, all the rewrite functions that were used by the LP solver are put in the proof context as hints for `lia` to use.

9 Experimental Evaluation

Tool	Bounded	$O(1)$	$O(n)$	$O(n^2)$	$> O(n^2)$	Timeout	Proof ✓
Loopus'15	806	205	489	97	15	6	N/A
Pastis	459	187	229	43	0	127	424
Loopus'14	431	200	188	43	0	20	N/A
KoAT	430	253	138	35	4	161	N/A
CoFloCo	386	200	148	38	0	217	N/A

Table 1. Experimental evaluation of Pastis on 1659 functions from the cBench benchmark. Only Pastis can generate proof certificates, whence the four occurrences of "N/A" in the last column. The tools were run with a timeout of 60 seconds.

Benchmark Set. To evaluate the performance of Pastis, we used the benchmark suite of the paper that presented Loopus'15 [30]. That paper compares Loopus'15 to Loopus'14 [29], KoAT [10], and CoFloCo [16]. The comprehensive program set used is based on the compiler optimization Collective Benchmark (cBench) which contains 211,892 lines of C code. These C files are pre-processed to extract all functions into independent files; those files are then translated to the various input formats of the tools compared. Pastis accepts directly LLVM bitcode and processes it to extract a control-flow graph as described in Section 2. The benchmark was run in intraprocedural mode because only KoAT and Pastis support procedure calls.

Machine. The experimental evaluation was run on a machine equipped with an Intel Xeon CPU clocked at 3.10GHz and 32GB of memory.

Results. Table 1 contains a digest of the experiments. The results for Loopus'14, KoAT, and CoFloCo were taken from the evaluation done by Sinn et al. [30]. By the number of examples only, Pastis is in the same ballpark as the majority of other tools. We also found that Pastis infers bounds quickly in most cases: 98% are found in less than 3 seconds.

Loopus'15 deserves special mention as it performs remarkably well; we were impressed by its versatility on this benchmark set. In addition to its sophisticated intraprocedural loop-analysis algorithm, Loopus'15 implements many practical ad hoc features. Among others, the C types are retrieved from the debugging information in the LLVM bitcode; some heuristics to identify loops on null-terminated C strings and files are built in; and finally, at the expense of compositionality, large sections of code can be represented symbolically with a Scheme-like syntax in the case where top-level loops are preceded with complex straight-line code. Similar features are not yet implemented in Pastis.

From the "Proof ✓" column, we can see that most but not all the bounds that Pastis generated were successfully checked by Coq. Coq usually processes the generated files quickly: 311 files take less than 10 seconds to check, and another 83 files take less than 20 seconds. The checking failures are caused by imprecision in the LP certificates and by our conversion function from floating-point numbers to rational numbers. Especially on higher-degree problems, it is common to see a base function in a loop invariant assigned a small, non-zero, bogus coefficient. Past a certain threshold, our extraction mechanism will output a small rational number when zero is actually needed. On examples with large constants, we also observed LP-solver overflows leading to obviously unsound bounds. As a practical counter-measure, our LLVM-bitcode reader replaces all "large" constants with non-deterministic expressions.

10 Related Work

Resource-Bound Analysis. The analysis method presented is based on automatic amortized resource analysis (AARA). This analysis technique has been introduced for functional programs [17,19,20,33] and has also been applied to derive bounds for object-oriented programs [21] and heap-manipulating imperative programs [6]. In contrast to our work, none of the previous work focuses on proof certificates or deriving bounds for imperative code that depend on integers. Most related to our work is the recent application of AARA to derive linear bounds for imperative integer programs [11]. The main benefits of our work are the derivation of polynomial bounds, the flexibility introduced by rewrite functions, and the automatic verification of resource certificates in Coq.

Amortized analysis has been formalized in the proof assistants Isabelle/HOL and Coq to verify manually the complexity of algorithms and data-structures [14, 26]. Our focus is on integer programs rather than sophisticated data-structures and algorithms, and our technique verifies programs automatically.

Resource-bound analyses that focus on integer programs include CAMPY [31], KoAT [10], Rank [5], CoFloCo [16], COSTA [4] and Loopus [30]. The advantages of our method include LP-based bound inference, natural compositionality, bound inference for recursive procedures, and automatically-checked Coq certificates. Recent work on an interprocedural analysis that finds procedure summaries in non-linear arithmetic [22] has shown how such information can be used to find resource bounds.

We are only aware of three other papers that describe tools that can generate bounds with machine-verified certificates. Carbonneaux et al. [12] use Coq and the verified CompCert C compiler to derive stack bounds for assembly code that are verified by Coq. In contrast to the present paper, their technique does not use linear constraint solving and automatic verification is limited to constant stack bounds. Blazy et al. [9] describe a loop-bound estimation for WCET analysis that is formally verified in Coq. An advantage of our method is that we generate complex symbolic bounds and can naturally handle recursive functions. Albert et al. [3] have used the KeY program verifier to automatically verify bounds generated by the COSTA bound analyzer. While the overall methodology is similar, we show that we can handle a large set of benchmarks, perform bounds inference via linear programming, and focus on integer bounds rather than bounds that depend on the sizes of data structures.

Template-Based Methods. Several program-analysis methods use the idea of (i) choosing a template that characterizes the kind of invariants of a program that are sought, (ii) extracting an appropriate set of (linear) constraints from the program, and (iii) solving the constraints. For example, Template Constraint Matrices (TCMs) are a parameterized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. [27] gave meet, join, and a set of abstract transformers for all TCM domains. Monniaux [24] gave an algorithm that finds the best transformer in a TCM domain across a straight-line block and good transformers across more complicated control flow.

Müller-Olm and Seidl [25] showed how to obtain invariants that are polynomial equalities of bounded degree. Their method uses a finite-height domain that is a vector space whose basis elements are (transformers on) the set of monomials in which polynomial invariants can be expressed.

Bagnara et al. [8] presented a technique to generate invariants that are polynomial inequalities of bounded degree. Their technique introduces additional variables (dimensions) to represent nonlinear terms, and uses convex polyhedra to represent polynomial cones in the extended set of variables.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Automatic inference of resource consumption bounds. In: LPAR (2012)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: ESOP (2007)
3. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Román-Díez, G.: Verified resource guarantees for heap manipulating programs. In: FASE (2012)
4. Albert, E., Fernández, J.C., Román-Díez, G.: Non-cumulative resource analysis. In: TACAS (2015)
5. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: SAS (2010)
6. Atkey, R.: Amortised resource analysis with separation logic. In: ESOP (2010)
7. Avanzini, M., Lago, U.D., Moser, G.: Analysing the complexity of functional programs: Higher-order meets first-order. In: ICFP (2012)
8. Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of basic semi-algebraic invariants using convex polyhedra. In: SAS (2005)
9. Blazy, S., Maroneze, A., Pichardie, D.: Formal verification of loop bound estimation for WCET analysis. In: VSTTE (2013)
10. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating runtime and size complexity analysis of integer programs. In: TACAS (2014)
11. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: PLDI (2015)
12. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for C programs. In: PLDI (2014)
13. Cerný, P., Henzinger, T.A., Kovács, L., Radhakrishna, A., Zwirchmayr, J.: Segment abstraction for worst-case execution time analysis. In: ESOP (2015)
14. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: ITP (2015)

15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
16. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: APLAS (2014)
17. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: POPL (2017)
18. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In: APLAS (2010)
19. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL (2011)
20. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL (2003)
21. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: ESOP (2006)
22. Kincaid, Z., Breck, J., Forouhi Boroujeni, A., Reps, T.: Compositional recurrence analysis revisited. In: PLDI (2017)
23. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: POPL (2017)
24. Monniaux, D.: Automatic modular abstractions for template numerical constraints. LMCS 6(3) (2010)
25. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
26. Nipkow, T.: Amortized Complexity Verified. In: ITP (2015)
27. Sankaranarayanan, S., Sipma, H., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI (2005)
28. Serrano, A., López-García, P., Hermenegildo, M.V.: Resource usage analysis of logic programs via abstract interpretation using sized types. TPLP 14(4-5) (2014)
29. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable approach to bound analysis and amortized complexity analysis. In: CAV (2014)
30. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In: FMCAD (2015)
31. Srikanth, A., Sahin, B., Harris, W.R.: Complexity verification using guided theorem enumeration. In: POPL (2017)
32. The Coq Development Team: Reference Manual (Version 8.6). `https://coq.inria.fr/distrib/current/refman/index.html`
33. Vasconcelos, P.B., Jost, S., Florido, M., Hammond, K.: Type-based allocation analysis for co-recursion in lazy functional languages. In: ESOP (2015)