Compositional Certified Resource Bounds

Quentin Carbonneaux   Jan Hoffmann   Zhong Shao

Yale University

Abstract

The goal of quantitative resource analysis is to provide developers with quantitative information about the runtime behavior of software at development time. Recent years have seen tremendous progress in automatically deriving worst-case resource bounds, yet many challenges in specifying, interactively deriving, formally certifying, and composing resource bounds remain unsolved.

This paper describes a novel quantitative resource analysis framework that tackles these challenges for C programs. The analysis framework consists of three parts: First, a parametric cost semantics formalizes the resource consumption of CompCert Clight programs. Second, a quantitative Hoare logic enables users to interactively develop resource bounds in the Coq Proof Assistant. The logic is proved sound with respect to the cost semantics, and shallow embedding enables a derived bound to be any function that is definable in Coq. Third, an automatic amortized resource analysis for Clight computes derivations in the quantitative Hoare logic. It is the first automatic amortized analysis that can derive bounds that depend on negative integers or differences between numbers, which is crucial to handle typical systems code. Both, the quantitative logic and the automatic amortized analysis are naturally compositional and can be combined to semi-automatically derive global resource bounds. An experimental evaluation demonstrates the practicality of the analysis framework. The expressivity of the logic is shown by manually deriving customized bounds that are tailored to specific algorithms. A comparison of the automatic amortized analysis with other automatic tools on 30 challenging loop and recursion patterns from the literature and open-source software shows that the bounds derived by the automatic amortized analysis are often more precise.

1. Introduction

In software engineering and software verification, we often would like to have static information about the quantitative behavior of programs. For example, stack and heap-space bounds are important to ensure the reliability of safety-critical systems [15, 37]. Static energy usage information is critical for autonomous systems and has applications in cloud computing [17, 16]. Worst-case time bounds can help to create constant-time implementations that prevent side-channel attacks [31, 9]. Loop and recursion-depth bounds are used to ensure the accuracy of programs that are executed on unreliable hardware [14] and complexity bounds are needed to verify cryptographic protocols [8]. In general, quantitative performance information at design time can provide useful feedback for developers.

Static analysis of quantitative properties of imperative programs is an active research area and recent years have seen many innovations. Notable tools that have been developed include SPEED [22], KoAT [13], PUBS [1], Rank [3], and LOOPUS [38]. While the these tools can derive impressive results for realistic software, there still exist shortcomings that hamper the application of existing techniques in practice:

- Analysis tools are black boxes that either deliver a result or fail without enabling *user interaction* to manually or semi-automatically derive bounds for challenging parts of the program.
- The computed bounds are often *non-compositional* local bounds (for a single (nested) loop) that are difficult to combine to global whole program bounds.
- Existing techniques often rely on complex external tools such as abstract interpretation–based invariant generation [22] or translation of the program into a term-rewriting system [13, 38] without providing *verifiable certificates* for the correctness of the derived bound.

While there has been much progress in static quantitative analysis, Knuth correctly points out in a recent interview [33] that the state-of-the-art in formal quantitative methods still falls short in comparison with semantic techniques.

> $[\cdots]$ Thanks to the work of Floyd, Hoare, and others, we have formal definitions of semantics, and tools by which we can verify that sorting is indeed always achieved. My job is to go beyond correctness, to an analysis of such things as the program's running time $[\cdots]$. I'm 100% sure that my recurrence correctly describes the program's performance, and all of my colleagues agree with me that the recurrence is "obviously" valid. Yet I have no formal tools by which I can prove that my recurrence is right. I don't really understand my reasoning processes at all!
> – Donald E. Knuth, 2014

In this work, we develop a resource analysis framework for C programs that is based on a solid semantic foundation. The choice of C is primarily motivated by our ongoing work on the formally verified hypervisor kernel CertiKOS [19]. CertiKOS is mainly developed in C and is supposed to provide verified guaranties on timing and memory usage. Moreover, C is a natural choice because it is still the most widely used language for system development; in particular in safety-critical embedded and real-time systems where resource bound analyses are often required by regulatory authorities [34]. Our *contributions* are as follows:

1. We define a operational cost semantics for CompCert Clight that defines the resource consumption of terminating and diverging executions. The cost is parametric in a user-definable cost metric and can be negative if resources are released during an execution.
2. We develop a quantitative Hoare logic for interactively deriving resource bounds for Clight programs. The logic is implemented and proved sound in the Coq Proof Assistant with respect to the cost semantics of Clight.
3. We describe an automatic static analysis that computes bounds together with derivations in the quantitative Hoare logic. The analysis is inspired by type-based amortized resource analysis for functional programs [28, 25]. It computes derivations

by generating simple linear constraints that can be solved by an off-the-shelf LP solver, and does not require any fixpoint computations to obtain loop invariants.

4. We show with a publicly available prototype implementation, and an experimental evaluation, that our novel amortized resource analysis works precisely and efficiently for challenging loop and recursion patterns, and that the derived constant factors in the bounds are close to or identical with the optimal ones.

To the best of our knowledge, this article presents the first resource analysis framework for C that makes it possible to combine nontrivial automatically derived bounds with interactively derived bounds in a proof system that produces verifiable certificates for the bounds. Our approach complements existing work since it provides a semantic foundation for the computation of bounds while still providing support for automation that sometimes goes beyond the capabilities of existing techniques. Furthermore, the automatic amortized analysis and the quantitative logic are naturally compositional and describe the resource behavior of code fragments without referring to the source code. As a result, we directly derive global bounds that are functions of the input parameters of the program. Another unique feature of our system is, that it can handle resources like memory that may become available during execution.

Our starting point is a recent work [15] in which we have formally verified bounds on the stack usage of C programs. A key part of this stack-bound verification is based on a quantitative Hoare logic for deriving abstract *stack bounds* for CompCert Clight programs that depend on the depth of (recursive) function calls. Our *first contribution* is a generalization of the quantitative Hoare logic for CompCert Clight that enables us to derive resource bounds that are parametric in the resource of interest. In the implementation of the logic, we use a shallow embedding in Coq which makes the logic very flexible. There is practically no limitation on the format of the derived resource bounds since they can be any function that is definable in Coq. The derived bounds can also be parametric in a set of cost metrics or specifically apply to one fixed cost metric.

The precision and expressivity of the quantitative Hoare logic provide an ample foundation for reasoning about resource consumption of Clight programs. However, reasoning about quantitative properties can often be more tedious than reasoning about intentional properties. Consequently, automation is inevitable to derive resource bounds for large code bases like CertiKOS. Such an automation was easily achievable in the case of constant stack bounds for a code base that does not contain recursive functions [15]. In general however, resource bounds have to be parametric in the arguments of a function and depend on the number of loop iterations and (recursive) function calls that are performed by the function.

For functional programs, there exist resource analysis systems—based on type-based amortized resource analysis [28, 26]—that can automatically derive complex polynomial bounds with tight constant factors. This type-based amortized resource analysis has been an inspiration in the design of the quantitative Hoare logic and is therefore a natural candidate to automate the reasoning. Type-based amortized resource analysis works well for functional programs with pattern matching but it has been a long-time open problem to apply it to C-like programs with control flow that depends on integer arithmetic, negative numbers, and non-linear control flow.

Our *main contribution* is an automatic amortized resource analysis for CompCert Clight that computes derivations in the quantitative Hoare logic. It is the first automatic amortized resource analysis that can derive bounds that depend on negative integers. It also handles programs with mutually recursive functions as well as `break` and `return` statements. The potential-based approach of amortized analysis provides a single mechanism for analyzing programs that need special treatment in other techniques; including

- interaction between *sequential loops or function calls* through size changes of variables,
- *nested loops* that influence each others number of iterations,
- and *tricky iteration patterns* as found for instance in the Knuth-Morris-Pratt algorithm for string search.

The main innovation that makes the analysis practical for C is the use of interval sizes in potential functions instead of the sizes of variables. This leads to precise bounds of programs whose resource consumption can be described as a function of sizes $|[x, y]| = \max(0, y-x)$ of intervals of integer variables. Such bounds arise for instance from standard *for loops* $\mathsf{for}(\mathsf{i} = \mathsf{x}; \mathsf{i} + \mathsf{K} \leqslant \mathsf{y}; \mathsf{i} = \mathsf{i} + \mathsf{K})$ where the step-size $K > 0$ is a constant.

Despite the apparent simplicity of the new analysis system, it is able to reproduce results from the literature (e.g, SPEED [22]) that have been obtained using sophisticated abstract interpretation–based methods such as disjunctive invariant generation [22] and symbolic backward execution [20]. In contrast with abstract interpretation–based methods, our technique does not require any fixpoint computations to obtain loop invariants. The mechanism we designed is able to leverage local assertions such as $x < y$ that we collect along the branching points of the program to obtain global resource invariants. We achieve this by generating a linear constraint system that reflects the resource cost and size changes of integer variables in the program; a solution of the linear program immediately yields a resource usage bound for the C program.

Following the development steps of automatic amortized analysis for functional programs [28, 23], we deliberately restrict ourselves to linear bounds in the automatic analysis (there are no restrictions for the manually-derived bounds in the logic). More specifically, bounds have the form $\sum_{a,b} q_{(a,b)}|[a, b]|$ where a and b are integer variables or constants. The reason for our focus on linear bounds is mainly clarity of presentation. We already experimented with an extension to multivariate resource polynomials [25, 24] but this would make the inference rules considerably more involved and should better be described separately. However, we developed the linear inference system so that the extension to polynomial bounds shall work smoothly.

We have evaluated the automatic analysis with system code and examples from the literature. Appendix A contains more than 30 challenging loop and recursion patterns that we collected from open source software and the literature. Our analysis can find asymptotically tight bounds for all but 1 of these patterns, and in most cases the derived constant factors are tight. To compare our automatic analyzer with existing techniques, we tested our examples with tools such as KoAT [13], Rank [3], and LOOPUS [38]. Our experiments show that the bounds that we derive are often more precise than those derived by existing tools. Only LOOPUS [38], which also uses amortization techniques, is able to achieve a similar precision. Several micro benchmarks demonstrate the practicality and expressiveness of the quantitative Hoare logic. For example, we derive a logarithmic bound for a binary search function, a bound that depends on the contents of an array to describe the exact cost of a function that finds a maximal element in an array, and a linear bound that amortizes the cost of k successive increments to a binary counter.

2. Overview and Examples

In this section, we informally introduce the quantitative program logic and the automatic amortized analysis for Clight programs.

2.1 Quantitative Hoare Logic

The idea that drives the design of our framework is amortized analysis [39]. Assume that a program S executes on a starting state

```
$\{\#_1(a) + 2k\}$
while (k > 0) {
  x=0;
  $\{\#_1(a) + 2k\}$
  while (x < N && a[x] == 1) {
    $\{(a[x] = 1) + \#_1(a) + 2k\}$
    a[x]=0;
    $\{\#_1(a) + 2k + 1\}$
    tick(1);
    $\{\#_1(a) + 2k\}$
    x++; }
  $\{(x \geqslant N \vee a[x] = 0) + \#_1(a) + 2k\}$
  if (x < N) {
    $\{(a[x] = 0) + \#_1(a) + 1 + 2(k-1) + 1\}$
    a[x]=1;
    $\{\#_1(a) + 2(k-1) + 1\}$
    tick(1); }
  $\{\#_1(a) + 2(k-1)\}$
  k--;
  $\{\#_1(a) + 2k\}$
} $\{\#_1(a)\}$
```

Figure 1. Example derivation using amortized reasoning. We write $\#_1(a)$ for $\#\{i \mid 0 \leqslant i < N \wedge a[i] = 1\}$ and use the tick metric that assigns cost n to the statement $\mathsf{tick(n)}$ and 0 to all other operations.

σ and consumes n resource units of some user-defined quantity. We denote that by writing $(S, \sigma) \Downarrow_n \sigma'$ where σ' is the program state after the execution. The basic idea of amortized analysis is to define a *potential function* Φ that maps program states to non-negative numbers and to show that $\Phi(\sigma) \geqslant n$ if σ is a program state such that $(S, \sigma) \Downarrow_n \sigma'$ to prove a bound on the resource usage.

To obtain a compositional reasnoning we also have to take into account the state resulting from a program's execution. We thus use two potential functions, one that applies before the execution, and one that applies after. The two functions must respect the relation $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ for all states σ and σ' such that $(S, \sigma) \Downarrow_n \sigma'$. Intuitively, $\Phi(\sigma)$ must provide enough *potential* for both, paying for the resource cost of the computation and paying for the potential $\Phi'(\sigma')$ on the resulting state σ'. That way, if $(\sigma, S_1) \Downarrow_n \sigma'$ and $(\sigma', S_2) \Downarrow_m \sigma''$, we get $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ and $\Phi'(\sigma') \geqslant m + \Phi''(\sigma'')$. This can be composed as $\Phi(\sigma) \geqslant (n+m) + \Phi''(\sigma'')$. Note that the initial potential function Φ provides an upper bound on the resource consumption of the whole program. What we have observed is that, if we define $\{\Phi\}\, S\, \{\Phi'\}$ to mean

$$\forall \sigma\, n\, \sigma'.\, (\sigma, S) \Downarrow_n \sigma' \implies \Phi(\sigma) \geqslant n + \Phi'(\sigma') \;,$$

then we get the following familiar looking rule.

$$\frac{\{\Phi\}\, S_1\, \{\Phi'\} \qquad \{\Phi'\}\, S_2\, \{\Phi''\}}{\{\Phi\}\, S_1; S_2\, \{\Phi''\}}$$

Similarly, other language constructs lead to rules for the potential functions that look very similar to Hoare logic or effect system rules. These rules enable interactive reasoning about resource usage in a flexible and compositional way, which, as a side effect, produces a certificate for the derived resource bound.

It is also possible to incorporate boolean conditions into a bound to express that the bound is only valid for a certain class of inputs. To this end, we allow the potential function Φ in the pre- and postconditions to take non-negative numbers or ∞ (infinity) as values. Infinity, plays the same role as *false* in Hoare logic. Boolean assertions can be embedded into potential functions by mapping *false* to ∞ and *true* to 0. We can then use the logic to prove a concrete bound for a given resource metric, or a bound that is parametric in a set of resource metrics. For example, we can derive

```
$\cdot;\ 0 + \frac{T}{K}\cdot|[x, y]| + 0\cdot|[y, x]| \vdash$
while (x+K<=y) {
  $x + K \leqslant y;\ 0 + \frac{T}{K}\cdot|[x, y]| + 0\cdot|[y, x]| \vdash$
  x=x+K;
  $\dashv x \leqslant y;\ T + \frac{T}{K}\cdot|[x, y]| + 0\cdot|[y, x]| \vdash$
  tick(T);
  $\dashv x \leqslant y;\ 0 + \frac{T}{K}\cdot|[x, y]| + 0\cdot|[y, x]|$
}
$\dashv x \geqslant y; 0 + \frac{T}{K}\cdot|[x, y]| + 0\cdot|[y, x]|$
```

Figure 2. Automatic derivation of a tight bound on the number of ticks for a standard *for loop*. The parameters $K > 0$ and $T > 0$ are not program variables but denote concrete constants in the program. The reasoning works uniformly for all such K and T.

the following quantitative triple for a binary-search function $\mathsf{bsearch}$ that holds for all stack metrics M.

$$\{(Z = \log_2(h - \ell)) + Z{\cdot}M_{\mathsf{bsearch}}\}\ \mathsf{bsearch(x, l, h)}\ \{Z{\cdot}M_{\mathsf{bsearch}}\}$$

A stack metric assigns cost 0 to all evaluation steps except function calls. Before a call $f(x)$, we consume $M_f > 0$ resources and after the call we return M_f resources that can be used in subsequent function calls. So M_f corresponds to the stack-frame size of the function f. In the pre- and postcondition, the logical variable Z is used to relate the size of the input with the potential that is left after the function call. The logical part $Z = \log_2(h - \ell)$ of the precondition is 0 if $Z = \log_2(\sigma_0(h) - \sigma_0(\ell))$ in the initial state σ_0 and ∞ otherwise.

For simplicity, assume for the remainder of this section that we are interested in bounding the number of *ticks* in a program. That is, we use the *tick metric* that assigns cost n to the function call $\mathsf{tick(n)}$ and cost 0 to all other operations. The argument n is an integer and a negative number means that $-n$ resources are returned.

We say that the analysis is amortized because the use of a potential function Φ enables us to amortize the cost of operations like in typical textbook examples that use the potential method of Tarjan's amortized analysis. As an illustrative example, Figure 1 shows a program that repeatedly increments a binary counter implemented with a boolean-valued (0 or 1) array. The derived bound in the precondition states that the number of ticks executed by the program is bounded by $\#_1(a) + 2k$. Here, k is the number of increments to the counter that are performed by the program and $\#_1(a) = \#\{i \mid 0 \leqslant i < N \wedge a[i] = 1\}$ is the number of 1's in the array a. We have inserted the statement $\mathsf{tick(1)}$ into the program whenever we update the array a. A naive analysis of the algorithm would lead to a quadratic bound since there can be a linear number of updates in the inner loop.

To derive the bound for the program in Figure 1 in the quantitative Hoare logic, we follow the textbook reasoning [18]. When we assign $\mathsf{a[x]} = 0$ in the inner loop, we know that $a[x] = 1$ before the assignment. As a result, we have $\#_1(a) = \#_1(a') + 1$ if a' is the array a after the assignment. So we obtain one resource unit in addition to the invariant (i.e., $\Phi = \#_1(a) + 2k + 1$) that we can use to pay for the cost of the statement $\mathsf{tick(1)}$. To pay for the tick statement in the conditional, we use the fact that $k > 0$ to write $2k$ as $1 + 2(k-1) + 1$. Dual to the first assignment, we have $\#_1(a) + 1 = \#_1(a')$ if a is the array before and a' is the array after the assignment $\mathsf{a[x]} = 1$. After paying one potential unit for the statement $\mathsf{tick(1)}$, we are left with the potential $\#_1(a) + 2(k-1)$. But since we increment k at the end of the outer loop, we can establish again our loop invariant $\#_1(a) + 2k$.

See Section 4 for more a formal definition of the logic and additional example derivations.

```
while (n>x) {
  n>x; |[x,n]|+|[y,m]| ⊢
  if (m>y)
    m>y; |[x,n]|+|[y,m]| ⊢
    y=y+1;
    ⊣ ·; 1+|[x,n]|+|[y,m]|
  else
    n>x; |[x,n]|+|[y,m]| ⊢
    x=x+1;
    ⊣ ·; 1+|[x,n]|+|[y,m]|
  ⊣ ·; 1+|[x,n]|+|[y,m]| ⊢
  tick(1);
  ⊣ ·; |[x,n]|+|[y,m]|
}
```

$|[x,n]| + |[y,m]|$

speed_1

```
while (x<n) {
  x<n; |[x,n]|+|[z,n]| ⊢
  if (z>x)
    x<n; |[x,n]|+|[z,n]| ⊢
    x=x+1;
    ⊣ ·; 1+|[x,n]|+|[z,n]|
  else
    z⩽x, x<n; |[x,n]|+|[z,n]| ⊢
    z=z+1;
    ⊣ ·; 1+|[x,n]|+|[z,n]|
  ⊣ ·; 1+|[x,n]|+|[z,n]| ⊢
  tick(1);
  ⊣ ·; |[x,n]|+|[z,n]|
}
```

$|[x,n]| + |[z,n]|$

speed_2

```
while (z-y>0) {
  y<z; 3.1|[y,z]|+0.1|[0,y]| ⊢
  y=y+1;
  ⊣ ·; 3+3.1|[y,z]|+0.1|[0,y]| ⊢
  tick(3);
  ⊣ ·; 3.1|[y,z]|+0.1|[0,y]|
}
⊣ ·; 3.1|[y,z]|+0.1|[0,y]| ⊢
while (y>9) {
  y>9; 3.1|[y,z]|+0.1|[0,y]| ⊢
  y=y-10;
  ⊣ ·; 1+3.1|[y,z]|+0.1|[0,y]| ⊢
  tick(1);
  ⊣ ·; 3.1|[y,z]|+0.1|[0,y]|
}
```

$3.1|[y,z]| + 0.1|[0,y]|$

t08a

```
while (n<0) {
  n<0; P(n,y) ⊢
  n=n+1;
  ⊣ ·; 59+P(n,y) ⊢
  y=y+1000;
  ⊣ ·; 9+P(n,y) ⊢
  while (y>=100 && *){
    y>99; 9+P(n,y) ⊢
    y=y-100;
    ⊣ ·; 14+P(n,y) ⊢
    tick(5);
    ⊣ ·; 9+P(n,y)
  }
  ⊣ ·; 9+P(n,y) ⊢
  tick(9);
  ⊣ ·; P(n,y)
}
```

$59|[n,0]|+0.05|[0,y]|$

t27

Figure 3. Derivations of bounds on the number of ticks for challenging examples. Examples *speed_1* and *speed_2* (from [22]) use *tricky iteration patterns*, *t08a* contains *sequential loops* so that the iterations of the second loop depend on the first, and *t27* contains interacting *nested loops*. In the potential functions, we only mention the non-zero terms and in the logical context Γ we only mention assertions that we actually use in the reasoning. In Example *t27*, we use the abbreviation $P(n,y) := 59|[n,0]|+0.05|[0,y]|$.

2.2 Automatic Amortized Analysis

A program logic provides a principled foundation for statically analyzing programs. However, program logics need to be supported by automatic methods to be useful in practice.

To enable automation, we fix the shape of the potential functions Φ so that it becomes possible to use *linear programming* (LP) to compute a derivation in the logic. If we assume for now that a program state σ simply maps variables to integers then we require

$$\Phi(\sigma) = q_0 + \sum_{x,y \in \mathrm{dom}(\sigma)} q_{(x,y)} \cdot |[\sigma(x), \sigma(y)]|$$

where $q_{(x,y)} \in \mathbb{Q}_0^+$ and $|[a,b]| = \max(b-a, 0)$. Note that q_0 is the constant potential and that we treat constants as program variables. For instance, we always have $q_{(0,x)}|[0,x]|$ as a component of $\Phi(\sigma)$ if $x \in \mathrm{dom}(\sigma)$. We then develop an inference rule for each syntactic construct that derives a sound triple $\{\Phi\}\, P\, \{\Phi'\}$ in the quantitative logic but enables inference using an LP solver.

This idea is best explained by example. If we again use the tick metric that assigns cost n to the function call $\mathsf{tick(n)}$ and cost 0 to all other operations then the cost of the following example can be bounded by $|[x,y]|$.[1]

```
while (x<y) { x=x+1; tick(1); }                    (Example 1)
```

To derive this bound in our automatic amortized analysis, we start with the initial potential $\Phi_0 = |[x,y]|$ (short for $\Phi_0(\sigma) = |[\sigma(x), \sigma(y)]|$) which we also use as the loop invariant. For the loop body we then (like in Hoare logic) have to derive a triple like $\{\Phi_0\}\, \mathsf{x} = \mathsf{x} + 1; \mathsf{tick}(1)\, \{\Phi_0\}$. We can only do so if we utilize the fact that $x < y$ at the beginning of the loop body. If we denote the updated version of x after the assignment by x' then the relation $|[x,y]| = |[x',y]| + 1$ between the potential before and after the assignment $\mathsf{x} = \mathsf{x} + 1$ holds. This means that we have the potential $|[x,y]| + 1$ before the statement $\mathsf{tick}(1)$. Since $\mathsf{tick}(1)$ consumes one resource unit, we end up with potential $|[x,y]|$ after the loop body and have established the loop invariant again.

[1] Note that there are no restrictions on the signs of the input variables in the examples.

Note that our notion of a potential function is a generalization of the concept of a ranking function. A potential function can be used like a ranking function if we use the tick metric and add the statement $\mathsf{tick}(1)$ to every back edge of the program (loops and function calls). However, a potential function is more flexible. For example, we can use a potential function to prove that Example 2 does not consume any resources in the tick metric.

```
while (x<y) { tick(-1); x=x+1; tick(1); }  (Example 2)
```

Similarly we can prove that Example 3 can be bounded by $10|[x,y]|$. In both cases, we reason exactly like in the first version of the while loop to prove the bound. Of course, such loops with different tick annotations can be seamlessly combined in a larger program.

```
while (x<y) { x=x+1; tick(10); }             (Example 3)
```

More formally, we develop (Section 5) a judgement

$$\Gamma; Q \vdash S \dashv Q'; \Gamma'$$

such that Γ contains logical assertions like $x < y$ that we collect along the conditional branches of the program and Q is a family of coefficents $Q = (q_{(a,b)})_{a,b \in \mathrm{scope}}$ that is indexed by the variables currently in scope. The logical context Γ is simply a conjunction of inequalities between linear combinations of variables. It is used, for example, to determine at variable assignments if we can extract constant potential to pay for future cost. It is important to note that the reasoning about assertions in Γ is very basic. We do not perform any fixpoint computations and only derive trivial loop invariants about variables that are unchanged in the loop body.

Figure 2 shows a derivation of the bound $\frac{T}{K} \cdot |[x,y]|$ on the number of ticks for a generalized version of Example 1 in which we increment x by a constant $K > 0$ and consume $T > 0$ resources in each iteration. The reasoning is similar to the one of Example 1 except that we obtain the potential $K \cdot \frac{T}{K}$ after the assignment. Note that the logical assertions in Γ are only used in the rule for the assignment $\mathsf{x} = \mathsf{x} + \mathsf{K}$. To the best of our knowledge, no other implemented tool for C is currently capable of deriving a tight bound on the cost of such a loop. For $T = 1$ (many systems focus on the number of loop iterations without a cost model) and $K = 10$, KoAT [13] computes the bound $|x|+|y|+10$, Rank [3] computes the bound $y - x - 7$, and LOOPUS [38] computes the bound $y - x - 9$. Only PUBS [1] computes the tight bound $0.1(y - x)$ if we translate the program into a term-rewriting system by hand.

To automate the reasoning, we first introduce an unknown rational variable for each factor in the potential functions. We then use our inference rules (see Section 5) to emit linear constraints on these variables that enforce that variable assignments that respect the constraints correspond to sound potential annotations. For instance if $K = 1$ in Figure 2 then we would have the annotation $q_0 + q_{(x,y)} \cdot |[x, y]| + q_{(y,x)} \cdot |[y, x]|$ before the assignment and $p_0 + p_{(x,y)} \cdot |[x, y]| + p_{(y,x)} \cdot |[y, x]|$ after the assignment, where q_i and p_i are unknown and must satisfy constraints like $p_{(x,y)} = q_{(x,y)}$, $p_{(y,x)} = q_{(y,x)}$, and $p_0 = q_0 + q_{(x,y)} - q_{(y,x)}$.

It might be surprising that we only track simple linear relations in the logical context Γ. While it would of course be possible to keep track of more sophisticated assertions, this simple form is sufficient for the examples we considered and we can efficiently decide queries such as $\Gamma \implies x < y$? that we make in the assignment rule. Nevertheless, this logical part of our analysis system is orthogonal to the quantitative part and can be easily extended if necessary.

As mentioned earlier, the automatic analysis can handle challenging example programs without special tricks or techniques. Examples *speed_1* and *speed_2*, that are taken from previous work [22], demonstrate that our method can handle *tricky iteration patterns*. The SPEED tool [22] derives the same bounds as our analysis but requires heuristics for its counter instrumentation. These loops can also be handled with inference of *disjunctive invariants*, but in the abstract interpretation community, these invariants are known to be notoriously difficult to generate. In example *speed_1* we have one loop that first increments variable y up to m and then increments variable x up to n. We derive the tight bound $|[x, n]| + |[y, m]|$. Example *speed_2* is even trickier, and we found it hard to find a bound manually. However, using potential transfer reasoning as in amortized analysis, it is easy to prove the tight bound $|[x, n]| + |[z, n]|$.

Example *t08a* shows the ability of the analysis to discover interaction between *sequenced loops* through size change of variables. We accurately track the size change of y in the first loop by transferring the potential 0.1 from $|[y, z]|$ to $|[0, y]|$. Furthermore, *t08a* shows again that we do not handle the constants 1 or 0 in any special way. In all examples we could replace 0 and 1 with other constants like in the second loop and still derive a tight bound. The only information, that the analyzer needs is $y \geqslant c$ before assigning $y = y - c$. Example *t27* shows how amortization can be used to handle *interacting nested loops*. In the outer loop we increment the variable n until $n = 0$. In each of the $|[n, 0]|$ iterations, we increment the variable y by 1000. Then we non-deterministically (expressed by $*$) execute an inner loop that decrements y by 100 until $y < 100$. The analysis discovers that only the first execution of the inner loop depends on the initial value of y. We again derive tight constant factors.

As mentioned, the analysis also handles advanced control flow like break and return statements, and mutual recursion. Figure 4 contains two mutually-recursive functions with their automatically derived tick bounds. The function count_down decrements its first argument x until it reaches the second argument y. It then recursively calls the function count_up, which is dual to count_down. Here, we count up y by 2 and recursively call count_down. Our analysis is the only available system that computes a tight bound on this example. The analysis amounts to computing the meeting point of two trains that approach each other with different speeds.

We only show a small selections of the programs that we can handle automatically here. In Appendix A is a list of more than 30 classes of challenging programs that we can automatically analyze. Section 6 contains a more detailed comparison with other toolsfor automatic bound derivation.

```
void count_down (int x,int y) {
  if (x>y) { tick(1); count_up(x-1,y); } }

void count_up (int x, int y) {
  if (y+1<x) { tick(1); count_down(x,y+2); } }
```

$$0.33 + 0.67|[y, x]| \quad (\mathsf{count\_down}(x, y))$$
$$0.67|[y, x]| \quad (\mathsf{count\_up}(x, y))$$

t39

Figure 4. Two mutually-recursive functions with the computed tick bounds. The derived constant factors are tight.

3. Syntax and Semantics

We implemented our cost semantics and the quantitative Hoare logic in Coq for *CompCert Clight*. Clight is the most abstract intermediate language used by CompCert. Mainly, it is a subset of C in which loops can only be exited with a break statement and expressions are free of side effects.

Syntax. In this article, we describe our system for a subset of Clight that is sufficient to discuss the general ideas. This subset is given by the following grammar.

$$S := \mathsf{assert}\ E \mid \mathsf{skip} \mid \mathsf{break} \mid \mathsf{return}\ x \mid x \leftarrow E \mid x \leftarrow f(x^*)$$
$$\mid \mathsf{loop}\ S \mid \mathsf{if}(E)\ S\ \mathsf{else}\ S \mid S; S \mid \mathsf{tick}(n)$$

Expressions E are left abstract in our presentation. For our analysis framework, it is only important that they are side-effect free. The most notable difference to full Clight is that we can only assign to variables and thus do not consider operations that update the heap. Moreover, function arguments and return values are assumed to be variables. This is only for simplifying the presentation; in the implementation we can deal with heap updates and general function calls and returns. However, we have not implemented our framework for function pointers, goto statements, continue statements, and switch statements.

We include the built-in primitive assert e that terminates the program if the argument e evaluates to false and has no effect otherwise. This is useful to express assumptions on the inputs of a program for the automatic analysis. We also add the built-in function tick(n) that can be called with a constant integer n as a flexible way to model resource consumption or release (if n is negative).

Semantics. CompCert Clight's operational semantics is based on small-step transitions and continuations. Expressions—which do not have side effects—are evaluated in a big-step fashion. Here, we describe a simplified version of Clight's semantics for the subset we consider.

A program state $\sigma = (\theta, \gamma)$ is composed of two maps from variable names to integers. The first map, $\theta : \text{Locals} \to \mathbb{Z}$, assigns integers to local variables of a function, and the second map, $\gamma : \text{Globals} \to \mathbb{Z}$, gives values to global variables of the program. In this article, we assume that all values are integers but in the implementation we support all data types of Clight. The evaluation function $[\![\cdot]\!]$ maps an expression $e \in E$ to a value $[\![e]\!]_\sigma \in \mathbb{Z}$ in the program state σ.

For simplicity, we assume that local variables and global variables are always different. We just write $\sigma(x)$ to obtain the value of x in program state σ. Such a lookup is defined as

$$(\theta, \gamma)(x) = \begin{cases} \theta(x) & \text{if } x \in \text{dom}(\theta) \\ \gamma(x) & \text{if } x \in \text{dom}(\gamma) \end{cases} .$$

For a program state σ, we we write $\sigma[x \mapsto v]$ for the program state that maps x to v and behaves as σ for all other variables, regardless whether $x \in \text{dom}(\theta)$ or $x \in \text{dom}(\gamma)$.

$$\frac{\mathsf{istrue}\ [\![e]\!]_\sigma}{(\sigma, \mathsf{assert}\ e, K, c) \rightarrow_M (\sigma, \mathsf{skip}, K, c{-}M_a)}\ (\text{S:ASSERT})$$

$$(\sigma, \mathsf{break}, \mathsf{Kseq}\, S\, K, c) \rightarrow_M (\sigma, \mathsf{break}, K, c)\ (\text{S:BRKSEQ})$$

$$(\sigma, \mathsf{break}, \mathsf{Kloop}\, S\, K, c) \rightarrow_M (\sigma, \mathsf{skip}, K, c{-}M_b)\ (\text{S:BRKLOOP})$$

$$(\sigma, \mathsf{return}\ x, \mathsf{Kseq}\, S\, K, c) \rightarrow_M (\sigma, \mathsf{return}\ x, K, c)\ (\text{S:RETSEQ})$$

$$(\sigma, \mathsf{return}\ x, \mathsf{Kloop}\, S\, K, c) \rightarrow_M (\sigma, \mathsf{return}\ x, K, c)\ (\text{S:RETLOOP})$$

$$\frac{\sigma = (\_, \gamma) \qquad \sigma' = (\theta, \gamma)[r \mapsto \sigma(x)]}{(\sigma, \mathsf{return}\ x, \mathsf{Kcall}\, r\, \theta\, K, c) \rightarrow_M (\sigma', \mathsf{skip}, K, c{-}M_r)}\ (\text{S:RETCALL})$$

$$\frac{\sigma' = \sigma[x \mapsto [\![e]\!]_\sigma]}{(\sigma, x \leftarrow e, K, c) \rightarrow_M (\sigma', \mathsf{skip}, K, c{-}M_u{-}M_e(e))}\ (\text{S:UPDATE})$$

$$\frac{\begin{array}{c}\Sigma f = (\vec{x}, S_f) \\ \sigma = (\theta, \gamma) \qquad \sigma' = (\vec{x} \mapsto \sigma(\vec{y}), \gamma)\end{array}}{(\sigma, r \leftarrow f(\vec{y}), K, c) \rightarrow_M (\sigma', S_f, \mathsf{Kcall}\, r\, \theta\, K, c{-}M_f)}\ (\text{S:CALL})$$

$$(\sigma, \mathsf{loop}\ S, K, c) \rightarrow_M (\sigma, S, \mathsf{Kloop}\, S\, K, c)\ (\text{S:LOOP})$$

$$(\sigma, \mathsf{skip}, \mathsf{Kloop}\, S\, K, c) \rightarrow_M (\sigma, \mathsf{loop}\ S, K, c{-}M_l)\ (\text{S:SKIPLOOP})$$

$$\frac{\mathsf{istrue}\ [\![e]\!]_\sigma \qquad c' = c{-}M_c^1{-}M_e(e)}{(\sigma, \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2, K, c) \rightarrow_M (\sigma, S_1, K, c')}\ (\text{S:IFTRUE})$$

$$\frac{\mathsf{isfalse}\ [\![e]\!]_\sigma \qquad c' = c{-}M_c^2{-}M_e(e)}{(\sigma, \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2, K, c) \rightarrow_M (\sigma, S_2, K, c')}\ (\text{S:IFFALSE})$$

$$\frac{}{(\sigma, S_1; S_2, K, c) \rightarrow_M (\sigma, S_1, \mathsf{Kseq}\, S_2\, K, c)}\ (\text{S:SEQ})$$

$$(\sigma, \mathsf{skip}, \mathsf{Kseq}\, S\, K, c) \rightarrow_M (\sigma, S, K, c{-}M_s)\ (\text{S:SKIPSEQ})$$

$$\frac{}{(\sigma, \mathsf{tick}(n), K, c) \rightarrow_M (\sigma, \mathsf{skip}, K, c{-}M_t(n))}\ (\text{S:TICK})$$

Figure 5. Rules of the operational semantics of statements.

The small-step semantics is standard, except that it tracks the resource consumption of a program. The semantics is parametric in the resource of interest for the user of our system. We achieve this independence by parameterizing evaluations with a resource *metric* M; a tuple of nine rational numbers , one map $M_e : \mathrm{Expr} \rightarrow \mathbb{Q}$ from expressions to rational numbers, and one map $M_t : \mathbb{Z} \rightarrow \mathbb{Q}$ from integers to rational numbers.

$$M = (M_a, M_b, M_r, M_u, M_f, M_l, M_c^1, M_c^2, M_s, M_e(E), M_t(n))$$

Each of these rational numbers indicates the amount of resource consumed by a corresponding operation. If the assigned resource cost is negative then it means that resources are released. The metric that we use in the implementation can also depend on other information that is statically available such as the name of the called function or the number of arguments.

Figure 5 contains the reduction rules of the semantics. The rules define a rewrite system for program configurations of the form (σ, S, K, c), where σ is the program state, S is the statement being executed, K is a continuation that describes what remains to be done after the execution of S, and $c \in \mathbb{Q}$ is the non-negative number of resources available for further execution. All rules that can decrease c have the implicit side condition that the resource quantity available *before* the step is non-negative. This means that we allow (σ, S, K, c) with $c < 0$ on the right-hand side transition relation $\rightarrow_M^*$ to indicate that the execution ran out of resources. However, every execution that reaches such a state is stuck.

A continuation K represents the context of the execution of a given statement. A continuation can be the empty continuation Kstop (used to start a program), a sequenced computation $\mathsf{Kseq}\, S\, K$, a loop continuation $\mathsf{Kloop}\, S\, K$, or the continuation $\mathsf{Kcall}\, r\, \theta\, K$ of a function call.

$$K := \mathsf{Kstop} \mid \mathsf{Kseq}\, S\, K \mid \mathsf{Kloop}\, S\, K \mid \mathsf{Kcall}\, r\, \theta\, K$$

During the execution we assume a fixed function context Σ that maps function names to a list of variables and the statement that defines the function body. It is used in the rule S:CALL.

The intuitive meaning of an evaluation judgement $(\sigma, S, K, c) \rightarrow_M^* (\sigma', S', K', c')$ is the following. If the statement S is executed in program state σ, with continuation K, and with c resources available then—after a finite number of steps—the evaluation will reach the new machine state (σ', S', K', c') and there are c' resources available. If $c' \geqslant 0$ then the execution did not run out of resources and the resource consumption up to this point is $c - c'$. If this difference is negative then resource became available during the execution. If however $c' < 0$ then the execution ran out of resources and is stuck. The cost of the execution is then $c \geqslant 0$.

Lemma 1 summarizes the main properties of the resource counter of the semantics. It can be proved by induction on the number of steps.

Lemma 1. *1. If $(\sigma, S, K, c) \rightarrow_M^k (\sigma', S', K', c')$ and $n \geqslant 0$ then $(\sigma, S, K, c{+}n) \rightarrow_M^k (\sigma', S', K', c'{+}n)$.*

2. If $(\sigma, S, K, c) \rightarrow_M^k (\sigma', S', K', c')$ and $(\sigma, S, K, d) \rightarrow_M^k (\sigma', S', K', d')$ then $c - c' = d - d'$.

If we assume that we have a *positive resource metric*, that is, a metric that assigns a positive cost to each evaluation step then we can link bounds on the resource consumption under the metric to termination. This is formalized by Lemma 2.

Lemma 2. *Let M be a positive metric and assume that there exists $b \in \mathbb{Q}_0^+$ such that $(\sigma, S, K, c) \rightarrow_M^* (\sigma', S', K', c') \implies c{-}c' \leqslant b$. Then there exists $B \in \mathbb{N}$ such that $(\sigma, S, K, d) \rightarrow_M^n (\sigma'', S', K', d')$ for some n, d, d' and σ'' implies $n \leqslant B$.*

4. Quantitative Hoare Logic

In this section we describe a simplified version of the quantitative Hoare logic that we use in Coq to interactively prove resource bounds. We generalize classic Hoare logic to express not only classical boolean-valued assertions but also assertions that talk about the future resource usage. Instead of the usual assertions $P : \mathit{State} \rightarrow \mathit{bool}$ of Hoare logic we use assertions

$$P : \mathit{State} \rightarrow \mathbb{Q}_0^+ \cup \{\infty\}\ .$$

This can be understood as a refinement of boolean assertions where *false* is ∞ and *true* is refined by $\mathbb{Q}_0^+$. We write Assn for $\mathit{State} \rightarrow \mathbb{Q}_0^+ \cup \{\infty\}$ and $\bot$ for $\lambda\sigma.\infty$. We sometimes call assertions *potential functions*. To use Coq's support for propositional reasoning, assertions have the type $\mathit{State} \rightarrow \mathbb{Q}_0^+ \rightarrow \mathrm{Prop}$ in the implementation. For a given $\sigma \in \mathit{State}$, such an assertion can be seen as a set $B \subseteq \mathbb{Q}$ of valid bounds. However, we find the presentation in this article easier to read.

Due to break and return statements of Clight, there are different possible ways to exit a block of code. We also have to keep track of the resource specifications of functions. To account for this in the

$$\frac{}{\Delta; B; R \vdash \{Q\}\, \mathsf{skip}\, \{Q\}}\ (\text{L:Skip}) \qquad \frac{}{\Delta; B; R \vdash \{M_b + B\}\, \mathsf{break}\, \{Q\}}\ (\text{L:Break})$$

$$\frac{}{\Delta; B; R \vdash \{R\,(\sigma(x))\}\, \mathsf{return}\ x\, \{Q\}}\ (\text{L:Return}) \qquad \frac{}{\Delta; B; R \vdash \{\lambda\sigma.\, M_u + M_e(e) + Q\,\sigma[x \mapsto [\![e]\!]_\sigma]\}\ x \leftarrow e\ \{Q\}}\ (\text{L:Update})$$

$$\frac{\Delta; B; R \vdash \{P\}\, S_1\, \{Q' + M_s\} \qquad \Delta; B; R \vdash \{Q'\}\, S_2\, \{Q\}}{\Delta; B; R \vdash \{P\}\, S_1; S_2\, \{Q\}}\ (\text{L:Seq})$$

$$\frac{\Delta(f) = \forall z\, \vec{v}\, v.(P_f\, z\, \vec{v}, Q_f\, z\, v) \qquad P \models P_f\, y\, (\sigma(\vec{x})) \wedge A \qquad \forall v.\, (Q_f\, y\, v \wedge A \models \lambda\sigma.\, Q\, \sigma[r \mapsto v])}{\Delta; B; R \vdash \{M_f + P\}\, r \leftarrow f(\vec{x})\, \{Q - M_r\}}\ (\text{L:Call})$$

$$\frac{}{\Delta; B; R \vdash \{\mathsf{istrue}\ [\![e]\!]_\sigma \implies Q + M_a\}\ \mathsf{assert}\ e\, \{Q\}}\ (\text{L:Assert}) \qquad \frac{}{\Delta; B; R \vdash \{Q + M_t(n)\}\ \mathsf{tick}(n)\, \{Q\}}\ (\text{L:Tick})$$

$$\frac{\Delta; Q; R \vdash \{I\}\, S\, \{I + M_l\}}{\Delta; B; R \vdash \{I\}\ \mathsf{loop}\ S\, \{Q\}}\ (\text{L:Loop})$$

$$\frac{\Delta; B; R \vdash \{\mathsf{istrue}\ [\![e]\!]_\sigma + P - M_c^1\}\, S_1\, \{Q\} \qquad \Delta; B; R \vdash \{\mathsf{isfalse}\ [\![e]\!]_\sigma + P - M_c^2\}\, S_2\, \{Q\}}{\Delta; B; R \vdash \{P + M_e(e)\}\ \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2\, \{Q\}}\ (\text{L:If})$$

$$\frac{P \models P' \qquad \Delta; B'; R' \vdash \{P'\}\, S\, \{Q'\} \qquad Q' \models Q \qquad B' \models B \qquad \forall v.\, (R'\, v \models R\, v)}{\Delta; B; R \vdash \{P\}\, S\, \{Q\}}\ (\text{L:Weaken})$$

$$\frac{\Delta; B; R \vdash \{P\}\, S\, \{Q\} \qquad x \in \mathbb{Q}_0^+}{\Delta; B + x; R + x \vdash \{P + x\}\, S\, \{Q + x\}}\ (\text{L:Frame})$$

$$\frac{\begin{array}{c}\Delta \cup \Delta'; B; R \vdash \{P\}\, S\, \{Q\} \\ \forall f\, P_f\, Q_f.\, \Delta'(f) = \forall z\, \vec{v}\, v.(P_f\, z\, \vec{v}, Q_f\, z\, v) \rightarrow \forall y\, \vec{v}.\, (\Delta \cup \Delta'; \bot; Q_f\, y \vdash \{P_f\, y\, \vec{v}\}\, S_f\, \{\bot\})\end{array}}{\Delta; B; R \vdash \{P\}\, S\, \{Q\}}\ (\text{L:Extend})$$

Figure 6. Rules of the Quantitative Hoare Logic

logic, our quantitative Hoare triples have the form

$$\Delta; B; R \vdash \{Q\}\, S\, \{Q'\}\ .$$

The triple $\{Q\}\, S\, \{Q'\}$ consists of a statement S and two assertions $Q, Q' : Assn$. It corresponds to triples in classic Hoare logic and the intuitive meaning is as follows. If S is executed with starting state σ, the empty continuation Kstop, and at least $P(\sigma)$ resources available then the evaluation does not run out of resources and there are at least $Q(\sigma')$ resources left if the evaluation terminates in state σ'. The assertion $B : Assn$ provides the postcondition for the case in which the code block S is exited by a break statement. So if the execution is terminated in state σ' with a break then $B(\sigma')$ resources are available. Similarly, $R : \mathbb{Z} \rightarrow Assn$ is the postcondition for the case in which the code block S is exited by a return x statement. The integer argument of R is the return value. Finally, the function context of judgements that we write Δ is a mapping from function names to specifications of the form

$$\forall z\, \vec{v}\, v.(P_f\, z\, \vec{v}, Q_f\, z\, v).$$

The assertion $P_f\, z\, \vec{v}$ is the precondition of the function f and the assertion $Q_f\, z\, v$ is its postcondition. They are both parameterized by an arbitrary logical variable z (which can be a tuple) that relates the function arguments with the return value. The precondition also depends on $\vec{v}$, the values of the arguments at the function invocation. Similarly, the postcondition depends on the return value v of the function. The use of logical variables to express relations between different states of an execution is a standard technique of Hoare logic. To ensure soundness, we require that P_f and Q_f do not depend on the local variables on the stack, that is, $\forall z\, \vec{v}\, \theta\, \theta'\, \gamma.\, P_f\, z\, \vec{v}\, (\theta, \gamma) = P_f\, z\, \vec{v}\, (\theta', \gamma)$.

For two assertions $P, Q : Assn$, we write $P \models Q$ to if for all program state σ $P(\sigma) \geqslant Q(\sigma)$.

Rules of the Quantitative Logic. Figure 6 shows the inference rules of the quantitative logic. The rules are slightly simplified in comparison to the implemented rules in Coq. The main difference is that the presented version does not formalize the heap operations.

In the rule L:Skip, we do not have to account for any resource consumption. As a result, the precondition Q can be any (potential) function and we only have to make sure that we do not end up with more potential. Since the execution of skip leaves the program state unchanged, we can simply use the precondition as postcondition. The potential functions B for the break and R for the return part of the postcondition are not reachable and can therefore be arbitrary.

In the rule L:Assert, we use the notation istrue $[\![e]\!]_\sigma \implies Q + M_a$ to express that we require potential $Q + M_a$ in the precondition if e evaluates to $true$ in the current program state. If e evaluates to $false$ then the potential in the precondition can be arbitrary since the program will be terminated.

In the rules L:Break and L:Return, the postcondition can be arbitrary since it is unreachable. Instead, we have to justify the potential functions B and R that hold after a return and a break, respectively. In L:Break, we require to have potential $M_b + B$ in the precondition: M_b to pay for the execution cost of break and B to pay for the potential after the break. In L:Return we only require to have potential R in the precondition to pay for the potential after the return. The reason is that we found it to be more convenient to account for the execution cost M_r of the return in rule L:CALL for function calls.

The rule L:UPDATE is the standard assignment rule of Hoare logic. With the substitution $\sigma[x \mapsto [\![e]\!]_\sigma]$ in the precondition we ensure that Q evaluates to the same number as in the postcondition. We also require that have we the constant potential $M_u + M_e(e)$ available in the precondition to pay for the cost of the evaluation of e and the update.

The rule L:SEQ rule is crucial to understand how the quantitative Hoare logic works. To account for early exits of statements, we must ensure in the break part B of S_1's judgement that the break part B of of $S_1; S_2$ holds. The same is true for the return part R of the judgements for S_1 and S_2. The interaction between the actual pre- and postconditions is analogous to standard Hoare logic. In the postcondition of S_1 we account for the cost M_s of the execution of the sequence.

In the rule L:LOOP, the break part of the loop body S becomes the postcondition of the loop statement. We use an arbitrary B as the break part of the judgement for loop S since its operational semantics ensures that it can only terminate with a skip or a return. The precondition I of the loop is the loop invariant. That is why we require to have potential I available in the precondition of the loop body S. In the postcondition of S, the potential must be sufficient to pay for the invariant I and the cost M_l of the loop iteration.

The L:IF is similar to the rule for the conditional in classic Hoare logic. In the preconditions of the judgments for the two branches S_1 and S_2 we lift the boolean assertions isfalse $[\![e]\!]_\sigma$ and isfalse $[\![e]\!]_\sigma$ to quantitative assertions. In the precondition of the rule L:TICK we account for the execution cost $M_t(n)$ of the tick statement that depends on the integer n.

The rule L:CALL accounts for the execution cost of both M_f for function calls and M_r for return statements. This justifies that the L:RETURN rule does not account for resource consumption. The pre- and postcondition P_f and Q_f are taken from the function context Δ. The assertions in the context are parametric with respect to both the values of the function arguments and the return value. This allows us to specify a bound for a function whose resource consumption depends on its arguments. The arguments are instantiated by the call rule using the result of the evaluation of the argument variables in the current state. Recall that we require that P_f and Q_f do not depend on the local variables on the call stack, that is, $\forall z\, \vec{v}\, \theta\, \theta'\, \gamma.\, P_f\, z\, \vec{v}\, (\theta, \gamma) = P_f\, z\, \vec{v}\, (\theta', \gamma)$. To transfer potential that depends on local variables of the callee from the precondition P to the postcondition Q, we use an assertion $A : Assn$ that is independent of global variables, that is, $\forall \theta\, \gamma\, \gamma'.\, A(\theta, \gamma) = A(\theta, \gamma')$. It is still possible to express relations between global and local variables using logical variables (see the following paragraph for details).

Finally, we describe the rules which are not syntax directed. There are two weakening rules available in the quantitative Hoare logic. The framing rule L:FRAME is designed to weaken a statement by stating that if S needs P resources to run and leaves Q resources available after its execution, then it can very well run with $P + c$ resources and return $Q + c$ resources. The consequence L:CONSEQ rule is directly imported from classical Hoare logic except that instead of using the logical implication $\Rightarrow$ we use the quantitative $\models$ that point-wise applies $\geqslant$. This rule indeed weakens the statement since it requires more resource to run the statement and yields less than what has been proved to be available after its termination.

Logical Variables and Shallow Embedding. When specifying a program, it is often necessary to relate certain invariants in the pre- and postcondition. A standard solution of Hoare logic is to use *logical variables* [32]. These additional variables (also called auxiliary state in the literature) are constant across the derivation. For example, if Z is such a logical variable, we can specify the function double() which doubles the global variable x as

$$\{z = Z\}\, \mathsf{double}()\, \{x = 2 \cdot Z\}.$$

When formalizing Hoare logics in a proof assistant one can either fully specify the syntax and semantics of assertions and hence get a deep embedding, or use the assertions of the host theory to get a shallow embedding. Because of its flexibility, we used the latter approach in our development. This choice makes it possible to have logical variables almost for free: we can simply use the variable and binding mechanisms of Coq, our host theory. When an logical variable is needed we introduce it using a universal quantifier of Coq before using the logic to derive triples. For example, the Coq theorem for the above example would look as follows.

```
Theorem dbl_triple: forall Z,
  qtriple (λσ.σ(x) = Z) (double()) (λσ.σ(x) = 2*Z).
```

However, this trick alone is often not sufficient when working with a recursive function f. In that case we apply the L:EXTEND rule of the logic. First we add a specification $\forall z\, \vec{v}\, v.(P_f\, z\, \vec{v}, Q_f\, z\, v)$ of f to the function context Δ. Then we proceed to prove the function body S_f with this induction hypothesis. In this process it can be the case that we have to use the induction hypothesis with a different value of a logical variable (e.g., because the values of the arguments in the recursive call differ from the values of the arguments of the callee). To cope with this problem, assumptions in the function context Δ are universally quantified over logical variables. The L:EXTEND rule uses the host proof assistant to require that the triple on f's body is proved for every possible logical variable y.

Using the Quantitative Logic. In the following we demonstrate the use of the logic with two example derivations.

In the example in Figure 7 we derive a precise runtime bound on a program that searches a maximal element in an array. The cost metric that we use simply counts the assignments performed by the program. Hence, the resource cost is closely related to the number of times the test a[i] > m is true during the execution. If we define

$$A(i) = \#\{k \mid i \leqslant k < N \wedge \forall\, 0 \leqslant j < k.\, a[j] < a[k]\}\,.$$

where we write $\#S$ for the cardinal of the set S then $A(1) + 1$ is the number of "maximum candidates" in the array a seen by the algorithm. $A(1)$ is bounded by N, the size of the array. So any automated tool would at best derive the linear bound $2 \cdot N$ for that program. But with the expressivity of our logic it is possible to use the previous set cardinal directly and precisely tie the bound to the initial contents of the array. The non-trivial part of this derivation is finding the loop invariant $(m = \max_{k \in [0,i-1]} a[k]) + A(i) + (N - i)$ for the while loop. When the condition a[i] > m is true, we know that we encountered a "maximum so far" because m is a maximal element of a[0 . . . i], thus $A(i) = 1 + A(i + 1)$ and we get one potential unit to pay for the assignment. In the other case, no maximum so far is encountered so $A(i) = A(i + 1)$.

The example in Figure 8 shows a use case for logical variables as well as a metric for stack consumption. In a stack metric, we account a constant cost for a function call ($M_f > 0$) that is returned after the call ($M_r = -M_f < 0$). All other resource cost are 0. We are interested in showing that a binary search function bsearch has logarithmic stack consumption. We use a logical variable Z in the function specification $\{(Z = \log_2(h - l)) + Z \cdot M_{\mathsf{bsearch}}\}\ \{Z \cdot M_{\mathsf{bsearch}}\}$ to express that the stack required by the function is returned after the call. The critical step in the proof is the application of the L:CALL rule to the recursive call. At this point the context Δ contains the specification $\forall y\, (x, l, h)\, \_.((\lambda y\, (\_, l, h).\, (y = \log_2(h{-}l)){+}y{\cdot}M_{\mathsf{bsearch}}), (\lambda y\, \_.\, y{\cdot}M_{\mathsf{bsearch}}))$. Using the rule L:CALL it is possible to instantiate y with $Z{-}1$, and because $M_f = M_{\mathsf{bsearch}}$ and $M_r = -M_{\mathsf{bsearch}}$ in the stack metric. The rest of the proof does

```
{$A(0) + N$}
i=1; m=a[0];
{$(i{=}1 \wedge m = a[0]) + A(1) + (N-1)$}
while (i < N) {
  {$(m = \max_{k \in [0,i-1]} a[k]) + A(i) + (N-i)$}
  if (a[i] > m)
    m=a[i];
  {$(m = \max_{k \in [0,i]} a[k]) + A(i+1) + (N-i)$}
  i=i+1;
  {$(m = \max_{k \in [0,i-1]} a[k]) + A(i) + (N-i)$}
} {$0$}
```

Figure 7. Example derivation where we wrote $A(i)$ for $\#\{k \mid i \leqslant k \leqslant N \wedge \forall 0 \leqslant j < k.\, a[j] < a[k]\}$, the metric used here assigns a cost of 1 to every assignment and 0 to all other operations.

```
{$(Z = \log_2(h-l)) + Z \cdot M_{\mathsf{bsearch}}$}
bsearch(x,l,h) {
  if (h-l > 1) {
    {$(Z \geqslant 1 \wedge Z = \log_2(h-l)) + Z \cdot M_{\mathsf{bsearch}}$}
    m = h + (h-l)/2;
    {$(m = \frac{h+l}{2} \wedge Z \geqslant 1 \wedge Z = \log_2(h-l)) + Z \cdot M_{\mathsf{bsearch}}$}
    if (a[m]>x) h=m; else l=m;
    {$(Z-1 = \log_2(h-l)) + (Z-1) \cdot M_{\mathsf{bsearch}} + M_{\mathsf{bsearch}}$}
    l = bsearch(x,l,h); }
  {$(Z-1) \cdot M_{\mathsf{bsearch}} - (-M_{\mathsf{bsearch}})$}
  return l;
} {$Z \cdot M_{\mathsf{bsearch}}$}
```

Figure 8. Example derivation of a stack usage bound for a binary search program. The used resource metric defines the cost M_{bsearch} before the function call and $-M_{\mathsf{bsearch}}$ after the call. M_{bsearch} is the stack frame size of the function bsearch. Z is a logical variable.

not involve any resource manipulation and is just bookkeeping of logical assertions.

Soundness of Quantitative Hoare Triples. We already gave an intuition of the meaning of judgements derived in the logic. To make it formal, we define the *resource safety* $\mathit{safe}(n, P, S, K)$ of an assertion and a program configuration as $\forall \sigma\, c\, m\, c'$.

$$(m{\leqslant}n \wedge P(\sigma){\leqslant}c \wedge (\sigma, S, K, c) \to_M^m (\_,\_,\_,c')) \implies c'{\geqslant}0.$$

This predicate is step indexed by an integer n that is used for induction in the soundness proof for the function-call and loop cases. The constraint $c' \geqslant 0$ imposed by the definition ensures that the program does not stop because of a resource error (recall that a negative resource counter on the right-hand side is a resource failure). However, it does not rule out memory safety errors or assertion failures. This is because our logic does not prove any safety or correctness theorems but only focuses on resource usage. An interesting detail in the definition is the natural number m. We simply use it to ensure that $\mathit{safe}(n+1, P, S, K) \implies \mathit{safe}(n, P, S, K)$. This would not be the case if we replaced all occurrences of m by n in the definition of validity.

The resource safety of a continuation K is defined using three assertions, one for each of the possible outcomes of a program statement. We define it as follows.

$$\begin{aligned} \mathit{safeK}(n, B, R, Q, K) := \quad & \mathit{safe}(n, B, \mathsf{break}, K) \\ & \wedge \mathit{safe}(n, \lambda\sigma.\, R\,(\sigma(x)), \mathsf{return}\ x, K) \\ & \wedge \mathit{safe}(n, Q, \mathsf{skip}, K) \end{aligned}$$

We can now define the *semantic validity* of a judgement $B; R \vdash \{P\}\, S\, \{Q\}$ of the quantitative logic without function context as

$$\begin{aligned} & \mathit{valid}(n, B, R, P, S, Q) := \\ & \forall m\, K\, x \geqslant 0.\, (m \leqslant n \wedge \mathit{safeK}(m, B{+}M_b{+}x, R{+}x, Q{+}x, K)) \\ & \quad \implies \mathit{safe}(m, P + x, S, K)\ . \end{aligned}$$

Note how the validity of a triple embeds the frame rule of our logic. This refinement is necessary to have a stronger induction hypothesis available during the proof. We again need to add the auxiliary m to ensure that $\mathit{valid}(n{+}1, B, R, P, S, Q)$ implies $\mathit{valid}(n, B, R, P, S, Q)$.

Using the semantic validity of triples we define the validity of a function context Δ, written $\mathit{validC}(n, \Delta)$, as

$$\begin{aligned} \forall f.\, \Delta(f) = \forall z\, \vec{v}\, v.(P_f\, z\, \vec{v}, Q_f\, z\, v) \implies \\ \forall z\, \vec{v}.\ \mathit{valid}(n, \bot, Q_f\, z, P_f\, z\, \vec{v}, S_f, \bot), \end{aligned}$$

where S_f is the body of the function f. A full judgement that mentions a non-empty function context Δ is in fact a guarded statement: it makes assumptions on some functions' behavior. The predicate $\mathit{validC}(n, \Delta)$ gives the precise meaning of the assumptions made. It is also step-indexed to prove the soundness of the L:EXTEND rule by induction. We are now able to state the soundness of the quantitative logic.

Theorem 1 (Soundness of the logic)**.** *If* $\Delta; B; R \vdash \{P\}\, S\, \{Q\}$ *is derivable then* $\forall n.\ \mathit{validC}(n, \Delta) \implies \mathit{valid}(n{+}1, B, R, P, S, Q)$.

The difference $\delta = 1$ between the index in the triple validity and the one in context validity arises from the soundness proofs of L:CALL and L:EXTEND. For L:CALL, the language semantics makes one step and proceeds with the function body, so we must have $\delta \leqslant 1$ to use the assumptions in Δ. For L:EXTEND, we have to show that $\Delta \cup \Delta'$ is a valid context for n steps. The induction hypothesis in that case says that if $\Delta \cup \Delta'$ is valid for m steps, Δ' is valid for $m + \delta$ steps. So if we want to solve this goal by induction, it is necessary that $\delta \geqslant 1$. These two constraints force δ to be exactly one in the theorem statement.

Assume that S is a complete program and Δ is empty. By expanding the definitions we see that Δ is valid for every n and that Kstop is safe for every n. So we derive

$$\Delta; B; R \vdash \{P\}\, S\, \{Q\} \implies \forall n.\ \mathit{safe}(n, P, S, \mathsf{Kstop}).$$

This means that from any starting state σ, $P(\sigma)$ provides enough resources for any run of the program S. This setting is actually the main use case of the previous theorem which is stated as a stronger result to allow a proof by induction.

5. Automatic Amortized Analysis

In this section we present the automatic amortized analysis that we use in our implementation to derive resource bounds for C programs.

Linear Potential Functions. To find derivations in the quantitative Hoare logic automatically, we have to focus on bounds that have a certain shape. The general form of *potential functions* (or assertions) that we consider is

$$\Phi(\sigma) = q_0 + \sum_{x,y \in \mathrm{dom}(\sigma) \wedge x \neq y} q_{(x,y)} \cdot |[\sigma(x), \sigma(y)]|\ .$$

Here $\sigma : (\text{Locals} \to \mathbb{Z}) \times (\text{Globals} \to \mathbb{Z})$ is again a simplified program state as introduced in Section 3, $|[a, b]| = \mathit{max}(0, b-a)$, and $q_i \in \mathbb{Q}_0^+$. To simplify the references to the linear coefficients q_i, we introduce an *index set* I. This set is defined to be $\{0\} \cup \{(x, y) \mid x, y \in \mathit{Var} \wedge x \neq y\}$. Each index i corresponds to a *base function* f_i in the potential function: 0 corresponds to the constant function $\sigma \mapsto 1$, and (x, y) corresponds to $\sigma \mapsto |[\sigma(x), \sigma(y)]|$. Using these notations we can rewrite the above equality as

$$\Phi = \sum_{i \in I} q_i f_i.$$

We often write xy to denote the index (x, y). The family $(f_i)_{i \in I}$ is actually a basis (in the linear-algebra sense) for potential functions. This allows us to uniquely represent any linear potential function Φ as a *quantitative annotation* $Q = (q_i)_{i \in I}$, that is, a family of rational numbers where only a finite number of elements are not zero. For soundness, we require that potential functions always give non-negative results. This is acheived by restricting coefficients in a quantitative annotation to be always non-negative.

In the potential functions, we treat constants as global variables that cannot be assigned to. For example, if the program contains the constant 8128 then we have a variable c_{8128} and $\sigma(c_{8128}) = 8128$. We assume that every program state includes the constant c_0.

Logical State. In addition to the quantitative annotations our automatic amortized analysis needs to maintain a minimal logical state to justify certain operations on quantitative annotations. For example when analyzing the code $x \leftarrow x + y$, it is helpful to know the sign of y to determine which intervals will increase or decrease. The knowledge needed by our rules can be inferred by local reasoning (i.e., in basic blocks without recursion and loops) within usual theories (e.g. Presburger arithmetic or bit vectors).

In contrast to the quantitative annotations, logical contexts Γ in the presented inference system are left abstract. This allows for simpler rules and leaves room for future improvements. Our implementation uses conjunctions of linear inequalities as logical state. We never compute fixpoints and take $\top$ as pre- and postconditions for functions. It has proved to be sufficient for the variety of examples we are interested in or found in the literature.

Judgements of the Automatic Analysis. The inference system for the automatic amortized analysis is defined in Figure 9. The inference rules derive judgements of the form

$$(\Gamma_B, Q_B); (\Gamma_R, Q_R); (\Gamma, Q) \vdash S \dashv (\Gamma', Q').$$

These judgements correspond to the logic judgements $\Delta; B; R \vdash \{P\}\, S\, \{Q\}$ where each assertion splits in two parts, a logical part Γ and a quantitative part Q. That means that (Γ_B, Q_B) is the postcondition of break statements, (Γ_R, Q_R) is the postcondition for return statements, and $(\Gamma, Q) \vdash S \dashv (\Gamma', Q')$ can be understood as a simple Hoare triple. We leave out the function context Δ in the presentation to avoid cluttering notations. Instead we assume a fixed function context Δ that is implicit on all judgements. Function contexts are treated exactly as in the logic. The correspondence between judgements of the automation and the ones of the quantitative logic is made formal by our soundness proof of the automatic amortized analysis at the end of this section.

As a convention, if Q and Q' are quantitative annotations then we assume that $Q = (q_i)_{i \in I}$ is a family with elements q_i, $Q' = (q'_i)_{i \in I}$, etc. The notation $Q \pm n$ used in many rules defines a new context Q' such that $q'_0 = q_0 \pm n$ and $\forall i \neq 0.\, q'_i = q_i$. We have the implicit side condition that all rational coefficients are non-negative. Finally, if a rule mentions Q and Q' and leaves the latter undefined at some index i we will implicitly assume that $q'_i = q_i$.

We describe the automatic amortized analysis for a subset of expressions of Clight. Assignments must have the form $x \leftarrow y$ or $x \leftarrow x \pm y$. In the implementation, a Clight program is converted into this form prior to analysis without changing the resource cost. This achieved by using a series of *cost-free assignments* that do not result in additional cost in the analysis. Non-linear operations such as $x \leftarrow z * y$ are simply handled by assigning zero potential to coefficients like q_{xa} and q_{ax} that contain x after the assignment.

Inference Rules. Most of the rules correspond exactly to the respective rules of the quantitative Hoare logic. For example, the rule Q:SKIP simply reuses its postcondition as precondition. This is sound since evaluating a skip statement never costs any resources and does not change the program state. Similarly, the rule Q:TICK enforces that the quantitative annotation of the precondition is at least $M_t(n)$ to pay for the resource consumption. Since the program state is also unchanged in this case, the logical part Γ of the precondition can soundly be reused in the postcondition. In the rule Q:BREAK, where the linear control flow is broken, the postcondition (Γ', Q') can be arbitrary. The precondition is taken from the break part of the judgement similarly as it is done in the quantitative logic. The rule Q:RETURN exhibits a slight difference to the quantitative logic. While we have a function $\mathbb{Z} \to Assn$ to represent return values in the logic, we assume that a special variable ret is part of the index set of Q_R. The substitution $Q[ret/x]$ denotes an potential annotation Q' with $q'_{xy} = q_{rety}$ and $q'_{yx} = q_{yret}$ for all y and $q'_i = q_i$ otherwise.

Most interesting are the rules Q:INCP, Q:DECP, and Q:INC for increments and decrements, which describe how the potential is distributed after a size change of a variable. The rule Q:INCP is for increments $x \leftarrow x + y$ and Q:DECP is for decrements $x \leftarrow x - y$, they both apply only when we can deduce from the logical context Γ that $y \geqslant 0$. Of course, we have symmetrical rules Q:INCN and Q:DECN that can be applied if y is not positive. The rules are equivalent in the case where $y = 0$. Finally, the rule Q:INC can be applied if we cannot deduce any information about y. In the implementation, the rules for increments and decrements are combined into one syntax directed rule where the specialized rules take priority over Q:INC.

To explain how rules for increment and decrement work, it is sufficient to understand the rule Q:INCP. The others follow the same idea and are symmetrical. In Q:INCP, the program updates a variable x with $x + y$ where $y \geqslant 0$. Since x is changed, the quantitative annotation must be updated to reflect the change of the program state. We write x' for the value of x after the assignment. Since x is the only variable changed, only intervals of the form $[u, x]$ and $[x, u]$ will be resized. Note that for any u, $[x, u]$ will get smaller with the update, and if $x' \in [x, u]$ we have $|[x, u]| = |[x, x']| + |[x', u]|$. But $|[x, x']| = |[0, y]|$ which means that the potential q'_{0y} in the postcondition can be increased by q_{xu} under the guard that $x' \in [x, u]$. Dually, the interval $[v, x]$ can get bigger with the update. We know that $|[v, x']| \leqslant y + |[v, x]|$. So we decrease the potential of $[0, y]$ by q_{vx} to pay for this change. The rule ensures this only for $v \notin \mathcal{U}$ because we know that $x \leqslant v$ otherwise, and thus $|[v, x]| = 0$.

Another interesting rule is Q:CALL. It needs to account for the changes to the stack caused by the function call, the arguments/return value passing, and the preservation of local variables. We can sum up the main ideas of the rule as follows.

- The potential in the pre- and postcondition of the function specification is equalized to its matching potential in the callee's pre- and postcondition.
- The potential of intervals $|[x, y]|$ is preserved if x and y are local variables,
- The unknown potentials after the call (e.g. $|[x, g]|$, with x local and g global) are set to zero in the postcondition.

If x and y are local variables and $f(x, y)$ is called, Q:CALL splits the potential of $|[x, y]|$ in two parts, one part to perform the computation in the function f and one part to keep for later use after the function call. This splitting is realized by the equations $Q = P + S$ and $Q' = P' + S'$. Arguments in the function precondition (Γ_f, Q_f) are named using a fixed vector $\vec{args}$ of names different from all program variables. This prevents name conflicts from happening and ensures that the substitution $[\vec{args}/\vec{x}]$ is meaningful. Symmetrically, we use the unique name ret to represent the return value in the function's postcondition (Γ'_f, Q'_f).

The rule Q:WEAK is not syntax directed. In the implementation we apply Q:WEAK before loops and between the two statements

$$\frac{}{B; R; (\Gamma, Q) \vdash \textsf{skip} \dashv (\Gamma, Q)!}\ (\text{Q:Skip}) \qquad \frac{}{(\Gamma, Q_B); R; (\Gamma, Q_B{+}M_b) \vdash \textsf{break} \dashv (\Gamma', Q')}\ (\text{Q:Break})$$

$$\frac{}{B; R; (\Gamma, Q{+}M_t(n)) \vdash \textsf{tick}(n) \dashv (\Gamma, Q)}\ (\text{Q:Tick}) \qquad \frac{P = Q_R[ret/x] \qquad \Gamma = \Gamma_R[ret/x] \qquad \forall i \in \textsf{dom}(P).\, p_i = q_i}{B; (\Gamma_R, Q_R); (\Gamma, Q) \vdash \textsf{return}\ x \dashv (\Gamma', Q')}\ (\text{Q:Return})$$

$$\frac{q'_{xy}, q'_{yx} \in \mathbb{Q}_0^+ \qquad \forall u.(q_{yu} = q'_{xu} + q'_{yu} \wedge q_{uy} = q'_{ux} + q'_{uy})}{B; R; (\Gamma[x/y], Q{+}M_u{+}M_e(y)) \vdash x \leftarrow y \dashv (\Gamma, Q')}\ (\text{Q:Update})$$

$$\frac{(\Gamma', Q'); R; (\Gamma, Q) \vdash S \dashv (\Gamma, Q{+}M_l)}{B; R; (\Gamma, Q) \vdash \textsf{loop}\ S \dashv (\Gamma', Q')}\ (\text{Q:Loop})$$

$$\frac{\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x + y \in [x, u]\} \qquad q'_{0y} = q_{0y} + \sum_{u \in \mathcal{U}} q_{xu} - \sum_{v \notin \mathcal{U}} q_{vx}}{B; R; (\Gamma[x/x{+}y], Q{+}M_u{+}M_e(x{+}y)) \vdash x \leftarrow x + y \dashv (\Gamma, Q')}\ (\text{Q:IncP})$$

$$\frac{\begin{array}{c}\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x - y \in [u, x]\} \\ q'_{y0} = q_{y0} + \sum_{u \in \mathcal{U}} q_{ux} - \sum_{v \notin \mathcal{U}} q_{xv}\end{array}}{B; R; (\Gamma[x/x{-}y], Q{+}M_u{+}M_e(x{-}y)) \vdash x \leftarrow x - y \dashv (\Gamma, Q')}\ (\text{Q:DecP})$$

$$\frac{\begin{array}{c}M = M_u + M_e(x{\pm}y) \\ q'_{0y} = q_{0y} - \sum_v q_{vx} \qquad q'_{y0} = q_{y0} - \sum_v q_{xv}\end{array}}{B; R; (\Gamma[x/x{\pm}y], Q{+}M) \vdash x \leftarrow x \pm y \dashv (\Gamma, Q')}\ (\text{Q:Inc}) \qquad \frac{\begin{array}{c}B; R; (\Gamma \wedge e, Q{-}M_c^1) \vdash S_1 \dashv (\Gamma', Q') \\ B; R; (\Gamma \wedge \neg e, Q{-}M_c^2) \vdash S_2 \dashv (\Gamma', Q')\end{array}}{B; R; (\Gamma, Q{+}M_e(e)) \vdash \textsf{if}(e)\ S_1\ \textsf{else}\ S_2 \dashv (\Gamma', Q')}\ (\text{Q:If})$$

$$\frac{B; R; (\Gamma, Q) \vdash S_1 \dashv (\Gamma', Q'{+}M_s) \qquad B; R; (\Gamma', Q') \vdash S_2 \dashv (\Gamma'', Q'')}{B; R; (\Gamma, Q) \vdash S_1; S_2 \dashv (\Gamma'', Q'')}\ (\text{Q:Seq})$$

$$\frac{\begin{array}{c}(\Gamma_f, Q_f, \Gamma'_f, Q'_f) \in \Delta(f) \\ \text{Loc} = \text{Locals}(Q) \qquad \forall i \neq j.\, x_i \neq x_j \qquad c \in \mathbb{Q}_0^+ \qquad Q = P + S \qquad Q' = P' + S \qquad U = Q_f[\vec{args}/\vec{x}] \\ U' = Q'_f[ret/r] \qquad \forall i \in \text{dom}(U).\, p_i = u_i \qquad \forall i \in \text{dom}(U').\, p'_i = u'_i \qquad \forall i \notin \text{dom}(U').\, p'_i = 0 \qquad \forall i \notin \text{Loc}.\, s_i = 0\end{array}}{B; R; (\Gamma_f[\vec{args}/\vec{x}] \wedge \Gamma_{\text{Loc}}, Q{+}c{+}M_f) \vdash r \leftarrow f(\vec{x}) \dashv (\Gamma'_f[ret/r] \wedge \Gamma_{\text{Loc}}, Q'{+}c{-}M_r)}\ (\text{Q:Call})$$

$$\frac{}{B; R; (\Gamma, Q{+}M_a) \vdash \textsf{assert}\ e \dashv (\Gamma \wedge e, Q)}\ (\text{Q:Assert}) \qquad \frac{\begin{array}{c}\Sigma f = (\vec{y}, S_f) \\ B; (\Gamma'_f, Q'_f); (\Gamma_f[\vec{args}/\vec{y}], Q_f[\vec{args}/\vec{y}]) \vdash S_f \dashv (\Gamma', Q')\end{array}}{(\Gamma_f, Q_f, \Gamma'_f, Q'_f) \in \Delta(f)}\ (\text{Q:Extend})$$

$$\frac{\begin{array}{c}B; R; (\Gamma_2, Q_2) \vdash S \dashv (\Gamma'_2, Q'_2) \qquad \Gamma_1 \models \Gamma_2 \\ Q_1 \succeq_{\Gamma_1} Q_2 \qquad \Gamma'_2 \models \Gamma'_1 \qquad Q'_2 \succeq_{\Gamma'_2} Q'_1\end{array}}{B; R; (\Gamma_1, Q_1) \vdash S \dashv (\Gamma'_1, Q'_1)}\ (\text{Q:Weak})$$

$$\frac{\begin{array}{c}\mathcal{L} = \{xy \mid \exists l_{xy}{\in}\mathbb{N}\,.\,\Gamma \models l_{xy} \leqslant |[x, y]|\} \qquad \mathcal{U} = \{xy \mid \exists u_{xy}{\in}\mathbb{N}\,.\,\Gamma \models |[x, y]| \leqslant u_{xy}\} \\ \forall i \in \mathcal{U}.\, q'_i \geqslant q_i - r_i \qquad \forall i \in \mathcal{L}.\, q'_i \geqslant q_i + p_i \qquad \forall i \notin \mathcal{U}{\cup}\mathcal{L}{\cup}\{0\}.\, q'_i \geqslant q_i \qquad q'_0 \geqslant q_0 + \sum_{i \in \mathcal{U}} u_i r_i - \sum_{i \in \mathcal{L}} l_i p_i\end{array}}{Q' \succeq_\Gamma Q}\ (\text{Relax})$$

Figure 9. Inference rules of the quantitative analysis.

of a sequential composition. We could integrate weakening into every syntax directed rule but this simple heuristic helps to make the analysis efficient. The high-level idea of Q:WEAK is the same as in the rule L:WEAK of the quantitative logic: If we have a sound judgement, then it is sound to add more potential to the precondition and remove potential from the postcondition. The concept of *more potential* is formalized by the relation $Q' \succeq_\Gamma Q$ that is defined in the rule RELAX. Here, Q and Q' are potential annotations and Γ is a logical context. The rule RELAX deals also with the important task of transferring constant potential (represented by q_0) to interval sizes and vice versa. If we can deduce from the logical context that the interval size $|[x, y]| \geqslant l$ is larger then a constant l then we can transfer potential $q_{xy}{\cdot}|[x, y]|$ form the interval to constant potential $l{\cdot}q_{xy}$ and guarantee that we do not gain potential. Conversely, if $|[x, y]| \leqslant u$ for a constant u then we can transfer constant potential $u{\cdot}q_{xy}$ to the interval potential $q_{xy}{\cdot}|[x, y]|$ without gaining potential.

Automatic Inference via LP Solving. We separate the search of a derivation into two steps. As a first step we go through the whole program and apply inductively the inference rules of the automatic amortized analysis. During this process our tool uses symbolic names for the rational coefficients (q_i) in the rules. Every time a linear constraint must be satisfied by these coefficients, it is recorded in a global list using the symbolic names. We then feed the collected constraints to an off the shelf LP solver [2]. If the solver successfully

[2] We currently use Coin-Or's CLP.

finds a solution, we know that a derivation exists for the considered program. We can then extract the initial Q from the solver and get a resource bound for the program. To get a full derivation we simply extract the complete solution from the solver, and apply it to the symbolic names (q_i) of the coefficients in the derivation. If the LP solver fails to find a solution, an error is reported to the user.

Soundness Proof. The soundness of the analysis builds on the quantitative logic. We translate judgements of the automatic amortized analysis as defined in Figure 9 into quantitative Hoare triples. We define a translation function $\mathcal{T}$, such that if a judgement J in the automatic analysis is derivable, $\mathcal{T}(J)$ is derivable in the quantitative logic. By using $\mathcal{T}$ to translate derivations of the automatic analysis to derivations in the quantitative logic we can automatically obtain a certified resource bound for the analyzed program.

The translation of an assertion (Γ, Q) in the automatic analysis is defined by

$$\mathcal{T}(\Gamma, Q) := \lambda\sigma.\, (\Gamma(\sigma)) + \Phi_Q(\sigma),$$

where we write Φ_Q for the unique linear potential function defined by the quantitative annotation Q. We also need to translate the assumptions in function contexts of the automatic analysis. We define $\mathcal{T}(\Gamma_f, Q_f, \Gamma'_f, Q'_f) := \forall z\, \vec{v}\, v.$

$$(\lambda\_\,\vec{v}\,\sigma.\, \mathcal{T}(\Gamma_f, Q_f)(\sigma[\vec{args}\mapsto\vec{v}]), \lambda\_\, v\,\sigma.\, \mathcal{T}(\Gamma'_f, Q'_f)(\sigma[ret\mapsto v]))$$

These definitions let us translate the judgement $J = B; R; P \vdash S \dashv P'$ in the context Δ by

$$\mathcal{T}(J) := (\lambda f.\, \mathcal{T}(\Delta(f))); \mathcal{T}(B); \mathcal{T}(R) \vdash \{\mathcal{T}(P)\}\, S\, \{\mathcal{T}(P')\}.$$

The soundness of the automatic analysis can now be stated formally with the following theorem.

Theorem 2 (Soundness of the automatic analysis). *If J is a judgement derived with the rules of Figure 9, then $\mathcal{T}(J)$ is a quantitative Hoare triple derivable with the rules of Figure 6.*

The proof of this theorem is constructive and basically maps each rule of the automatic analysis directly to its counterpart in the quantitative logic. The most tricky parts are the translations of the rules for increments and decrements and the rule Q:WEAK for weakening because we have to show that the preconditions of the rules L:WEAK or L:UPDATE, respectively, are met. For instance we prove the following lemma to show the soundness of the translation of Q:WEAK.

Lemma 3 (Relax). *If $Q' \succeq_\Gamma Q$ and σ is a program state such that $\sigma \models \Gamma$ then we have $\Phi_{Q'}(\sigma) \geqslant \Phi_Q(\sigma)$.*

Proof. We observe the following inequations.

$$\begin{aligned}\Phi_{Q'}(\sigma) &= \sum_i q'_i f_i(\sigma)\\ &\geqslant q_0 + \sum_{i\in\mathcal{U}} u_i p_i - \sum_{i\in\mathcal{L}} l_i p_i + \sum_{i\in\mathcal{U}}(q_i - p_i) f_i(\sigma)\\ &\quad + \sum_{i\in\mathcal{L}}(q_i + p_i) f_i(\sigma) + \sum_{i\notin\mathcal{U}\cup\mathcal{L}} q_i\\ &\geqslant \Phi_Q(\sigma) + \sum_{i\in\mathcal{U}}(u_i - f_i(\sigma))p_i + \sum_{i\in\mathcal{L}}(f_i(\sigma) - l_i)p_i\\ &\geqslant \Phi_Q(\sigma).\end{aligned}$$

Since $\sigma \models \Gamma$, using the transitivity of $\models$ and the definition of $\mathcal{U}$ and $\mathcal{L}$ we get $\forall i \in \mathcal{L}.\, l_i \leqslant f_i(\sigma)$ and $\forall i \in \mathcal{U}.\, f_i(\sigma) \leqslant u_i$. These two inequalities justify the last step. □

We prove similar lemmas for the rules that deal with assignment. With the computational content of the proof, we have an effective algorithm to construct certificates for upper bounds derived automatically.

```
int srch(
  int t[], int n, /* haystack */
  int p[], int m, /* needle */
  int b[]
) { int i=0, j=0, k=-1;
    while (i < n) {
      while (j >= 0 && t[i]!=p[j]) {
        k = b[j];
        assert(k > 0 && k <= j + 1);
        j -= k; tick(1)
      }
      i++, j++;
      if (j == m) break;
      tick(1);
    }
    return i;
  }
```

AAA	$1+2\lvert[0,n]\rvert$
Rank	$O(n^2)$
LOOPUS	$2+3\max(n,0)$

Figure 11. The Knuth-Morris-Pratt algorithm for string search.

6. Experimental Evaluation

We have experimentally evaluated the practicality of our automatic amortized analysis with more than 30 challenging loop and recursion patterns from open-source code and the literature [22, 21, 20]. A full list of examples together with the derived bounds is given in Appendix A.

Figure 10 shows five representative loop patterns from the evaluation. Example *t08* is a slightly modified version of *t08a* which is described in the introduction. Example *t19* demonstrates the compositionaliy of the analysis. The program consists of two loops that decrement a variable i. In the first loop, i is decremented down to 100 and in the second loop i is decremented further down to -1. However, between the loops we assign $\mathsf{i} = \mathsf{i} + \mathsf{k} + 50$. So in total the program performs $52 + |[-1, i]| + |[0, k]|$ increments. Our analysis finds this tight bound because our amortized analysis naturally takes into account the relation between the two loops.

At first sight, example *t30* appears to be a simple loop that decrements the variable x down to zero. However, a closer look reveals that the loop actually decrements both input variables x and y down to zero before terminating. In the loop body, first x is decremented by one. Then the values of the variables x and y are switched using the local variable t as a buffer. Our analysis infers the tight bound $|[0, x]| + |[0, y]|$. Sometimes we need some assumptions on the inputs in order to derive a bound. Example *t15* is such a case. We assume here that the input variable y is non-negative and write $\mathsf{assert}(\mathsf{y} >= 0)$. If we enter the loop then we know that $x > 0$ and we can obtain constant potential from the assignment $\mathsf{x} = \mathsf{x} - 1$. After the assignment we know that $x \geqslant y$ and $y \geqslant 0$. As a consequence, we can share the potential $|[0, x]|$ before the assignment $\mathsf{x} = \mathsf{x} - \mathsf{y}$ between $|[0, x]|$ and $|[0, y]|$ after the assignment. In this way, we derive a tight linear bound.

Example *t13* shows how amortization can be used to find linear bounds for nested loops. The outer loop is iterated $|[0, x]|$ times. In the conditional, we either (the branching condition is arbitrary) increment the variable y or we execute an inner loop in which y is counted back to 0. The analysis computes a tight bound. Again, the constants 0 and 1 in the inner loop can as well be replace by something more interesting, say 9 and 10 like in example *t08*. Then we still obtain a tight linear bound.

Finally, Figure 11 contains the search function of the Knuth-Morris-Pratt algorithm for string search. Our automatic amortized analysis finds the tight linear bound $1 + 2|[0, n]|$. We need to assert that the elements $b[j]$ of the failure table b are in the interval $[1, j + 1]$. This is guaranteed by construction of the table in the initialization procedure of the algorithm, which we can also analyze

	t08 (modified t08a)	t19	t30	t15	t13
	`while (y > x) { x++; tick(1); } while (x > 2) { x -= 3; tick(1); }`	`while (i > 100) { i--; tick(1) } i += k+50; while (i >= 0) { i--; tick(1); }`	`while (x > 0) { x--; t=x, x=y, y=t; tick(1); }`	`assert(y >= 0); while (x > y) { x -= y+1; for (z=y; z > 0; z--) tick(1); tick(1); }`	`while (x > 0) { x--; if (*) y++; else while (y > 0) { y--; tick(1); } tick(1); }`
AAA	$1.33\|[x,y]\|+0.33\|[0,x]\|$	$50+\|[-1,i]\|+\|[0,k]\|$	$\|[0,x]\|+\|[0,y]\|$	$\|[0,x]\|$	$2\|[0,x]\|+\|[0,y]\|$
Rank	$2+y-x(?)$	$54+k+i$	—	$2+2x-y$	$0.5\cdot y^2+yx+0.5\cdot x^2\dots$
LOOPUS	$\max(0,x-2)$ $+2\max(0,y-x)$	$\max(0,i-100)$ $+\max(0,k+i+51)$	—	—	$2\max(x,0)+\max(y,0)$

Figure 10. Comparison of resource bounds derived by different tools on several examples with linear bounds. AAA stands for our automatic amortized analysis for Clight. The output of Rank has been manually simplified to fit the table.

	KoAT	Rank	LOOPUS	SPEED	AAA
#bounds	9	24	20	14	32
#lin. bounds	9	21	20	14	32
#best bounds	0	0	11	14	29
#tested	14	33	33	14	33

Table 1. Comparison of our automatic amortized analysis with other automatic tools. We have not been able to obtain a version of SPEED [22] and just use the bounds that have been reported by the authors. Similarly, KoAT [13] does currently not work on C programs and we use the bounds that have been reported in the author's experimental evaluation.

automatically. We need the assertion since we do not infer any logical assertion on the contents of the heap. Rank derives a complex quadratic bound and LOOPUS derives a linear bound.

To compare our tool with existing work, we focused on loop bounds and use a simple metric that counts the number of back edges (i.e., number of loop iterations) that are followed in the execution of the program because most other tools only bound this specific cost. In Figure 10, we show the bounds we derived (AAA) together with the bounds derived by LOOPUS [38] and Rank [3]. We also contacted the authors of SPEED but have not been able to obtain this tool. KoAT [13] and PUBS [1] currently cannot operate on C code and the examples would need to be manually translated into a term-rewriting system to be analyzed by these tools. For Rank it is not totally clear how the computed bound relates to the C program since the computed bound is for transitions in an automaton that is derived from the C code. For instance, the bound $2 + y - x$ that is derived for *t08* only applies to the first loop in the program.

Table 1 summarizes the results of our experiments. It shows for each tool the number of derived bounds (#bounds), the number of asymptotically tight bounds (#lin. bounds), the number of bounds with the best constant factors in comparison with the other tools (#best bounds), and the number of examples that we were able to test with the tool (#tested). Since we were not able to run the experiments for KoAT and SPEED, we simply used the bounds the have been reported by the authors of the respective tools. The results show that our automatic amortized analysis outperforms the existing tools on our example programs. However, this experimental evaluation has to be taken with a grain of salt. Our goal in this work is not to set a new standard for automatic bound analysis but only to show that our approach has advantages on examples with linear bounds. Overall the existing tools are more powerful since they can derive polynomial bounds and support more features of C. We were particularly impressed by LOOPUS which is very robust, works on large C files, and derives very precise bounds. We did not include the running times of the tools in the table since all tested tools work very efficiently and need less then a second on every tested example.

Since we have a formal cost semantics, we can run our examples with this semantics for different inputs and measure the cost to compare it to our derived bound. Figure 12 shows such a comparison for example *t08*. One can see that the derived constant factors are the best possible if the input variable x is non-negative. If x is negative then the bound is only slightly off.

7. Related Work

Most closely related to this article is our previous work on end-to-end stack-bound verification of Clight programs [15]. In that work, we have implemented and verified a quantitative logic to reason about stack-space usage. In this article, we present a more general quantitative Hoare logic that is parametric in the resource of interest. The main innovation is the novel automatic amortized analysis for Clight programs that computes logical derivations for non-trivial bounds for programs with loops and recursion.

Our work has been inspired by type-based amortized resource analysis for functional programs [28, 23, 26]. There are three major improvements over previous work in the current paper, which presents the first automatic amortized resource analysis for C. First, we solved the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on (possibly negative) integers. Second, we extended the analysis system to deal with non-linear control flow that is introduced by `break` and `return` statements. Third, for the first time, we have combined an automatic amortized analysis with a system for interactively deriving bounds.

In the development of our quantitative Hoare logic we have drawn inspiration from mechanically verified Hoare logics. Nipkow's [36] description of his implementations of Hoare logics in Isabelle/HOL has been helpful to understand the interaction of auxiliary variables with the consequence rule. Appel's separation logic for CompCert Clight [5] has been a blueprint for the general structure of the quantitative logic. Since we do not deal with memory safety, our logic is much simpler and it would be possible to integrate it with Appel's logic. The continuation passing style that we use in the quantitative logic is not only used by Appel [5] but also in Hoare logics for low-level code [35, 30].

There exist quantitative logics that are integrated into separation logic [7, 27] and they are closely related to our quantitative logic. However, the purpose of these logics is slightly different since they focus on the verification of bounds that depend on the shape of heap data structures and they are not implemented for C. Also closely related to our logic is a VDM-style logic for reasoning about resource usage of an abstract fragment of JVM byte code by Aspinall et al. [6]. Their logic is not Hoare-style, does not target C code, and is not designed for interactive bound development but to produce certificates for bounds derived for high-level functional programs.

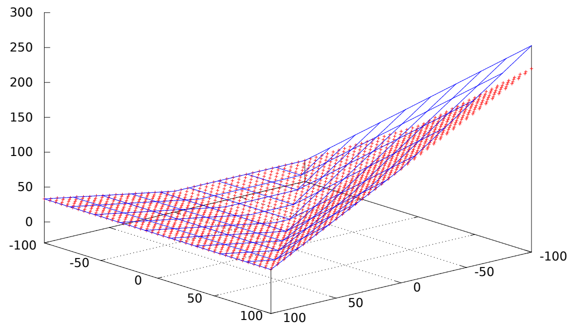

Figure 12. The automatically derived bound $1.33|[x, y]| + 0.33|[0, x]|$ (blue lines) and the measured runtime cost (red crosses) for example *t08*. For $x \geqslant 0$ the derived bound is tight.

There exist many tools that can automatically derive loops and recursion bounds for imperative programs such as SPEED [22, 20], KoAT [13], PUBS [1], Rank [3], ABC [10] and LOOPUS [40, 38]. These tools are based on abstract interpretation–based invariant generation and/or term rewriting techniques, and they derive impressive results on realistic software. The importance of amortization to derive tight bounds is well known in the resource analysis community [4, 29, 38]. Currently, the only other available tools that can be directly applied to C code are Rank and LOOPUS. Our analysis framework does not aim to set a new standard for automatic bound analysis. Our contribution is rather a principled approach that produces certificates that are proved sound with respect to a formal cost semantics. Moreover, we have a system that enables both automatic and interactive bound derivation, a formal soundness proof in Coq, and a method that can handle resources that can be released (e.g., memory). However, as we have shown in Section 6, our automatic amortized analysis matches the state of the art in automatic bound analysis for linear bounds and sometimes even derives better constant factors than the other tools. Since Rank and Loopus do not handle recursion, our automatic amortized analysis is also the only available tool that can derive bounds for recursive C programs.

There are techniques [12] that can compute the memory requirements of object oriented programs with region based garbage collection. These systems infer invariants and use external tools that count the number of integer points in the corresponding polytopes to obtain bonds. The described technique can handle loops but not recursive or composed functions.

We are only aware of two verified quantitative analysis systems. Albert et al. [2] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for Java Card programs. However, they do not have a formal cost semantics and do not prove the bounds correct with respect to a cost model. Blazy et al. [11] have verified a loop bound analysis for CompCert's RTL intermediate language. However, there is no way to interactively derive bounds or to deal with resources like memory usage. Furthermore, Blazy et al.'s automatic bound analysis does not compute symbolic bounds.

8. Conclusion

We have introduced an novel quantitative analysis system for CompCert Clight programs. To the best of our knowledge, this article presents the first resource analysis framework for C that makes it possible to combine non-trivial automatically derived bounds with interactively derived bounds in a proof system that produces verifiable certificates for the bounds. The main technical innovations are a quantitative Hoare logic for reasoning about user-defined resource cost and an automatic amortized analysis that derives bounds for programs whose resource consumption depends on (possibly negative) integers and non-sequential control flow as introduced by `break` and `return` statements.

We will continue to improve our framework to reason more precisely about resource usage of system software such as the hypervisor kernel CertiKOS [19], which we are currently developing and verifying. For one thing, we will improve the automation to derive *polynomial bounds* and to handle non-linear size changes, as already developed for functional programs [24]. For another thing, we will build on previous work [27] to generalize the quantitative Hoare logic to handle *concurrent programs* with locks and lock-free data structures.

A. Catalog of Automatically Analyzed Programs

In this appendix we provide a non-exhaustive catalog of classes of programs that can be automatically analyzed by our system. For simplicity, and to compare our analysis with existing tools, we use a cost metric that counts the number of loop iterations and function calls. This is the only cost metric supported by tools like KoAT [13] and LOOPUS [38]. Sometimes we also use the *tick metric* if we want to discuss features such as resource restitution. The tick metric assigns cost n to the statement `tick(n)` and cost 0 to all other statements. Of course, the examples can also be analyzed with any other cost metric in our system.

We assume that free variables in the code snippets are the inputs of the program. Some of the examples contain constants on which the computed bound depends. These constants are randomly chosen to present an example but the analysis works for other constants as well. Note however that it is sometimes crucial that constants are positive (or negative) or that other relations hold.

Table 2 contains the details of our comparative evaluation.

Amortization and Compositionality Figures 13, 14, and 15 show code snippets that need amortization and compositionality to obtain a whole program bound.

Example *t07* demonstrates two different features of the analysis. For one thing it shows that we can precisely track size changes inside loops. In the first loop, we increment y by 2 in each of the $|[0, x]|$ iterations. An in the second loop, we decrement y. For another thing it shows that we automatically recognize dead code if we find conflicting assertions on a branching path: After the second loop we know $y \leqslant 0$ and as a result can assign arbitrary potential inside the third loop where we know that $y > 0$. As a result, we obtain a tight bound.

Example *t08* shows the ability of the analysis to handle negative and non-negative numbers. Note that there are no restrictions on the signs of y and x. We also see again that we accurately track the size change of x in the first loop. Furthermore, *t08* shows that we do not handle the constants 1 or 0 in any special way. In all examples you could replace 0 and 1 with other constants like we did in the second loop and still derive a tight bound. The only information, that the analyzer needs is $x \geqslant c$ before assigning $x = x - c$.

In example *t10* we also do not restrict the inputs x and y. They can be negative, positive, or zero. The star `*` in the conditional, stands for an arbitrary assertion. In each branch of the conditional we can obtain the constant potential 1 since the interval size $|[y, x]|$ is decreasing.

Example *t13* shows how amortization can be used to handle tricky nested loops. The outer loop is iterated $|[0, x]|$ times. In the conditional, we either (the branching condition is again arbitrary) increment the variable y or we execute an inner loop in which y is counted back to 0. The analysis computes a tight linear bound for this program. Again, the constants 0 and 1 in the inner loop can as

```
while (y-x>0) {
  x = x+1;
}
while (x>2) {
  x=x-3;
}
```

$1.33|[x,y]| + 0.33|[0,x]|$

t08

```
while (x-y>0) {
  if (*)
    y=y+1;
  else
    x=x-1;
}
```

$|[y,x]|$

t10

```
while (x>0) {
  x=x-1;
  if (*)
    y=y+1;
  else {
    while (y>0)
      y=y-1;
  }
}
```

$2|[0,x]| + |[0,y]|$

t13

```
while (n<0) {
  n=n+1;
  y=y+1000;
  while (y>=100 && *){
    y=y-100;
  }
}
```

$11|[n,0]| + 0.01|[0,y]|$

t27

Figure 13. Amortization and Compositionality (a).

```
while (x>0) {
  x=x-1;
  y=y+2;
}
while (y>0) {
  y=y-1;
}
while (y>0) {
  y=y+1;
}
```

$1 + 3|[0,x]| + |[0,y]|$

t07

```
while (x>y) {
  x=x-1;
  x=x+1000;
  y=y+1000;
}
while (y>0) {
  y=y-1;
}
while (x<0) {
  x=x+1;
}
```

$1002|[y,x]| + |[x,0]| + |[0,y]|$

t28

```
assert (y>=0);
while (x-y>0) {
  x=x-1;
  x=x-y;
  z=y;
  while (z>0) {
    z=z-1;
  }
}
```

$|[0,x]|$

t15

```
assert (y>=0);
while (x-y>0) {
  x=x-1;
  x=x-y;
  z=y;
  z=z+y;
  z=z+100;
  while (z>0) {
    z=z-1;
  }
}
```

$101|[y,x]|$

t16

Figure 14. Amortization and Compositionality (b).

```
while (i>100) {
  i--;
}
i=i+k+50;
while (i>=0) {
  i--;
}
```

$50 + |[-1,i]| + |[0,k]|$

t19

```
while (x<y) {
  x=x+1;
}
while (y<x) {
  y=y+1;
}
```

$|[x,y]| + |[y,x]|$

t20

```
while (x>0) {
  x=x-1;
  t=x;
  x=y;
  y=t;
}
```

$|[0,x]| + |[0,y]|$

t30

```
flag=1;
while (flag>0) {
  if (n>0 && *) {
    n=n-1;
    flag=1;
  } else
    flag=0;
}
```

$1 + |[0,n]|$

t47

Figure 15. Amortization and Compositionality (c).

well be replace by something more interesting, say 9 and 10 like in example *t08*. Then we still obtain a tight linear bound.

Example *t27* is similar to example *t13*. Instead of decrementing the variable x in the outer loop we this time increment the variable n till $n = 0$. In each of the $|[n,0]|$ iterations, we increment the variable y by 1000. We then execute an inner loop that increments y by 100 until $y = 0$. The analysis can derive that only the first execution of the inner loop depends on the initial value of y. We again derive a tight bound.

Example *t28* is particularly interesting. In the first loop we decrement the size $|[y,x]|$. However, we also shift the interval $[y,x]$ to the interval $[y+1000, x+1000]$. The analysis can derive that this does not change the size of the interval and computes the tight loop bound $|[y,x]|$. The additional two loops are in the program to show that the size tracking in the first loop works accurately. The second loop is executed $|[0,y]| + 1000|[y,x]|$ times in the worst case. The third loop is executed $|[x,0]| + |[y,x]|$ in the worst case (if x and y are negative).

Sometimes we need some assumptions on the inputs in order to derive a bound. Example *t15* is such a case. We assume here that the input variable y is non-negative and write $\mathsf{assert}(y >= 0)$. The semantic of assert is that it has no effect if the assertion is true and that the program is terminated without further cost otherwise. If we enter the loop then we know that $x > 0$ and we can obtain constant potential from the assignment $\mathsf{x = x - 1}$. After the assignment we know that $x \geqslant y$ and $y \geqslant 0$. As a consequence, we can share the potential $|[0,x]|$ before the assignment $\mathsf{x = x - y}$ between $|[0,x]|$ and $|[0,y]|$ after the assignment. In this way, we derive a tight linear bound.

Example *t16* is an extension of example *t15*. We again assume that y is non-negative and use the same mechanism to iterate the outer loop as in *t15*. In the inner loop, we also count the variable z down to zero and perform $|[0,z]|$ iterations. However, instead of assigning $\mathsf{z = y}$, we assign $\mathsf{z = 2y + 100}$. The analysis computes the linear bound $101|[0,x]|$. The assignment of potential to the size interval $|[0, x]|$ in instead of $|[y, x]|$ is a random choice of the LP solver.

Example *t19* consists of two loops that decrement a variable i. In the first loop, i is decremented down to 100 and in the second loop i is increment further down to -1. However, between the loops we assign $i = i + k + 50$. So in total the program performs $50 + |[-1, i]| + |[0, k]|$ iterations. Our analysis finds this tight bound because our amortized analysis naturally takes into account the relation between the two loops. Techniques that do not use amortization derive a more conservative bound such as $50 + |[-1, i]| + |[0, k]| + |[100, i]|$.

Example *t20* shows how we can handle programs in which bounds contain absolute values like $|x - y|$. The first loop body is only executed if $x < y$ and the second loop body is only executed if $y < x$. The analyzer finds a tight bound.

At first sight, example *t30* appears to be a simple loop that decrements the variable x down to zero. However, a closer look reveals that the loop actually decrements both input variables x and y down to zero before terminating. In the loop body, first x is decremented by one. Then the values of the variables x and y are switched using the local variable t as a buffer. Our analysis infers the tight bound $|[0, x]| + |[0, y]|$.

Example *t47* demonstrates how we can use integers as Booleans to amortize the cost of loops that depend on boolean flags. The outer loop is executed as long as the variable flag is "true", that is, $\mathsf{flag} > 0$. Inside the loop, there is a conditional that either (if $n > 0$) decrements n and assigns $\mathsf{flag} = 1$, or (if $n \geqslant 0$) leaves n unchanged and assigns $\mathsf{flag} = 0$. The analyzer computes the tight bound $1 + |[0, n]|$. The potential in the loop invariant is $|[0, \mathit{flag}]| + |[0, n]|$. In the *then* branch of the conditional, we use the potential $|[0, n]|$ and the fact that $n > 0$. In the *else* branch, we use the potential $|[0, \mathit{flag}]|$ and the fact that $\mathit{flag} = 1$.

From the Literature Our analyzer can derive almost all linear bounds for programs that have been described as challenges in the literature on bound generation. We found only one program with a linear bound for which our analyzer could not find a tight bound: Example (*fig4_5*) from [21] requires *path-sensitive reasoning* to derive a bound.

Examples *fig2_1* and *fig2_2* are taken from Gulwani et al [22]. They are both handled by the SPEED tool but require inference of a *disjunctive invariant*. In the abstract interpretation community, these invariants are known to be notoriously difficult to handle. In example *fig2_1* we have one loop that first increments variable y up to m and then increments variable x up to n. We derive the tight bound $|[x, n]| + |[y, m]|$. Example *fig2_2* is more tricky and trying to understand how it works may be challenging. However, with the amortized analysis in mind, using the potential transfer reasoning, it is almost trivial to prove a bound. While the SPEED tool has to find a fairly involved invariant for the loop, our tool is simply reasoning locally and works without any clever tricks. We obtain the tight bound $|[x, n]| + |[z, n]|$.

Example *nested_multiple* is similar to example *fig2_1*. Instead of incrementing variable y in the outer loop, y is here potentially incremented multiple times in each iteration of the outer loop. The idea of example *nested_single* is similar. However, instead of incrementing variable y in the inner loop, we increment x, the counter variable of the outer loop. Our analyzer derives a tight bound for both programs. Note that a star * in a branching condition denotes an arbitrary boolean condition that might of course change while iterating (non-deterministic choice).

Example *sequential_single* is like example *nested_single*. The only difference is that the inner loop of *nested_single* is now evaluated after the outer loop. Example *simple_multiple* is a variant of example *fig2_1* and *simple_single* is a simple variant of *nested_single*. We derive tight bounds for all aforementioned programs.

Example *simple_single_2* uses conditionals and a break statement to control loop iterations. If $x < N$ then variables x and y are incremented. Otherwise, if $y < M$ then the same increment is executed. If $y \geqslant M$ and $x \geqslant N$ then the loop is terminated with a break. Our tool computes the bound $|[0, M]| + |[0, N]|$. This bound is tight in the sense that there are inputs (such as $M = -100$ and $N = 100$) for which the bound precisely describes the execution cost. However, SPEED can compute the more precise bound $\max(N, M)$. We currently cannot express this bound in our system.

Example *fig4_2* from [21] is quite involved. Amortized reasoning helps to understand how we derive the bound $1 + 2|[0, n]|$. We start with potential $1 + 2|[0, n]|$ and use the fact that $vb = 0$ to establish the potential $1 + 2|[0, n]| + |[0, vb]|$ that serves as a loop invariant. In the *if branch* of the conditional, we use the constant potential 1 of the invariant to pay for the potential of $|[0, vb]|$. Since we also know that $|[0, n]| > 0$ we obtain constant potential 2 that we use to pay for the loop iteration (1 unit) and to establish the loop invariant again (1 unit). In the *else* branch, we use the potential $|[0, vb]|$ and the fact $vb > 0$ to obtain 1 potential units to pay for the loop iteration.

In example *fig4_4* it is essential that m is positive. That ensures that we can obtain constant potential for the interval size $|[0, i]|$ in the *else* branch of the conditional since $|[0, i]|$ decreases. Example *fig4_5* is an examples that we can not handle automatically. The execution is bounded because the boolean value of the test $\mathsf{dir} == \mathsf{fwd}$ does not change during the iteration of the loop. As a result, the variable i is either counted down to 0 or up to n. Our tool cannot handle example *fig4_5* because we don't do path sensitive reasoning. Note however that it would be more efficient to move the test $\mathsf{dir} == \mathsf{fwd}$ outside of the loop (this would be also done by an optimizing compiler). The resulting program can then be analyzed by our tool.

Example *ex1* from [20] specifically focuses on the code in the *if* statement. So we use the *tick metric* and insert tick(1) inside the if statement to derive a bound on the number of times the code in the if statement is executed. Note that we cannot derive a bound for the whole program since the outer loop is executed a quadratic number of times. Nevertheless it is straightforward to derive a bound on the number of ticks using the amortized approach: In the if statement we know that $n > 0$ and assign $\mathsf{n} = \mathsf{n} - 1$. So we can use the potential of the interval size $|[0, n]|$ to pay for the tick.

Similar to example *ex1*, example *ex2* form [20] focuses on the number of iterations of the *outer loop*. While the whole program is not terminating, the number of iterations of the outer loop is bounded by $|[0, n]|$. Finally, example *ex2* is similar to example *nested_single*, and *ex3* is a variant of example *t47*.

Recursive Functions Our approach can naturally deal with mutually-recursive functions. The recursion patterns can be exactly the same that are used in iterations of loops. In the following, we present three simple examples that illustrate the analysis of functions.

Example *t37* illustrates that the analyzer is able to perform interprocedural size tracking. The function copy adds the argument x to the argument y if x is positive. However, this addition is done in steps of 1 in each recursive call. The function count_down recursively decrements its argument down to 0. The derived bound $3 + 2|[0, x]| + 2|[0, y]|$ is for the function main in which we first add x to y using the function copy and then count down the variable y using the function count_down. The derived bound is tight.

Example *t39* uses mutual recursion. The function count_down is similar to the function with the same name in example *t37*. However, we do not count down to 0 but to a variable y that is passed as an argument and we call the function count_up afterwards. The function count_up is dual to count_down. Here, we count up y by 2 and recursively call count_down. For the function main, which

```
while (n>x) {
  if (m>y)
    y = y+1;
  else
    x = x+1;
}
```

$|[x,n]| + |[y,m]|$

fig2_1

```
while (x<n) {
  if (z>x)
    x=x+1;
  else
    z=z+1;
}
```

$|[x,n]| + |[z,n]|$

fig2_2

```
while (x<n) {
  while (y<m) {
    if (*) break;
    y=y+1;
  }
  x=x+1;
}
```

$|[x,n]| + |[y,m]|$

nested_multiple

```
x=0;
while (x<n) {
  x=x+1;
  while (x<n) {
    if (*) break;
    x=x+1;
  }
}
```

$|[0,n]|$

nested_single

Figure 16. Examples from Gulwani et al's SPEED [22] (a).

```
x=0;
while (x<n) {
  if (*) break;
  x=x+1;
}
while (x<n)
  x=x+1;
```

$|[0,n]|$

sequential_single

```
x=0; y=0;
while (x<n) {
  if (y<m)
    y=y+1;
  else
    x=x+1;
}
```

$|[0,m]| + |[0,n]|$

simple_multiple

```
x=0;
while (x<n) {
  if (*)
    x=x+1;
  else
    x=x+1;
}
```

$|[0,n]|$

simple_single

```
x=0; y=0;
while (*) {
  if (x<N) {
    x=x+1; y=y+1;
  } else if (y<M ) {
    x=x+1; y=y+1;
  } else
    break;
}
```

$|[0,M]| + |[0,N]|$

simple_single_2

Figure 17. Examples from Gulwani et al's SPEED [22] (b).

```
assert n>0;
assert m>0;
va = n; vb = 0;
while (va>0 && *) {
  if (vb<m) {
    vb=vb+1;
    va=va-1;
  } else {
    vb=vb-1;
    vb=0;
  }
}
```

$1 + 2|[0,n]|$

fig4_2

```
assert (0<m);
i = n;
while (i>0 && *) {
  if (i<m)
    i=i-1;
  else
    i=i-m;
}
```

$|[0,n]|$

fig4_4

```
assert (0 < m < n);
i=m;
while (0<i<n) {
  if (dir==fwd) i++;
  else i--;
}
```

$---$

fig4_5

Figure 18. Examples from [21].

```
i=0;
while (i<n) {
  j=i+1;
  while (j<n) {
    if (*) {
      tick(1);
      j=j-1; n=n-1;
    }
    j=j+1;
  }
  i=i+1;
}
```

$|[0,n]|$ ticks

ex1

```
while (n>0 && m>0) {
  n--; m--;
  while (nondet()) {
    n--; m++;
  };
  tick(1);
}
```

$|[0,n]|$ ticks

ex2

```
while (n>0) {
  t = x;
  n=n-1;
  while (n>0) {
    if (*) break;
    n=n-1;
  }
}
```

$|[0,n]|$

ex3

```
flag=1;
while (flag>0) {
  flag=0;
  while (n>0 && *) {
    n=n-1;
    flag=1;
  }
}
```

$1 + 2|[0,n]|$

ex4

Figure 19. Examples from [20].

```
void count_down (int x) {
  int a = x;
  if (a>0) {
    a = a-1;
    count_down(a);
  }
}
int copy (int x, int y) {
  if (x>0) {
    x = x-1;
    y = y+1;
    y=copy(x,y);
  };
  return y;
}
void main (int x,int y) {
  y = copy (x,y);
  count_down(y);
}
```

$3 + 2|[0, x]| + |[0, y]|$

t37

```
void count_down (int x,int y)
{ int a = x;
  if (a>y) {
    a = a-1;
    count_up(a,y);
  }
}

void count_up (int x, int y)
{ int a = y;
  if (a+1<x) {
    a = a+2;
    count_down(x,a);
  }
}

void main (int y, int z) {
  count_down(y,z);
}
```

$1.33 + 0.67|[z, y]|$

t39

```
void produce () {
  while (x>0) {
    tick(-1); x=x-1; y=y+1;
  }
}
void consume () {
  while (y>0) {
    y=y-1; x=x+1; tick(1);
  }
}
void main (int y, int z) {
  consume(); produce(); consume();
}
```

$|[0, y]|$ ticks

t46

Figure 20. Programs with (recursive) functions

```
int srch(
  int t[], int n, /* haystack */
  int p[], int m, /* needle */
  int b[]
) {
  int i=0, j=0, k=-1;

  while (i < n) {
    while (j >= 0 && t[i]!=p[j]) {
      k = b[j];
      assert(k > 0);
      assert(k <= j + 1);
      j -= k;
    }
    i++, j++;
    if (j == m)
      break;
  }
  return i;
}
```

$1 + 2|[0, n]|$

Knuth-Morris-Pratt

```
int gcd(int x, int y) {
  if (x <= 0) return y;
  if (y <= 0) return x;

  for (;;) {
    if (x>y) x -= y;
    else if (y>x) y -= x;
    else return x;
  }
}
```

$|[0, x]| + |[0, y]|$

Greatest Common Divisor

```
void qsort(int a[], int lo, int hi) {
  int m1, m2, n;

  if (hi - lo < 1) return;

  n = nondet(); /* partition the array */
  assert( n > 0 );
  assert( lo + n <= hi );

  m1 = n + lo;
  m2 = m1 - 1;

  qsort(a, m1, hi);
  qsort(a, lo, m2);
}

void main(int a[], int len) {
  qsort(a, 0, len);
}
```

$1 + 2|[0, \mathsf{len}]|$

Quick Sort

Figure 21. Well-Known Algorithms

calls count_down(y, z), the analyzer computes the tight bound $1.33 + 0.67|[z, y]|$.

Example *t46* shows a program that uses and returns resources. Again, we use the *tick metric* and the function tick to describe the resource usage. The function produce produces $|[0, x]|$ resources, that is, in each of the $|[0, x]|$ iterations, it receives one resource unit. Similarly, the function consume consumes $|[0, y]|$ resources. The analyzer computes the tight bound $|[0, y]|$ for the function main. This is only possible since amortized analysis naturally tracks the size changes to the variables x and y, and the interaction between consume and produce.

Well-Known Algorithms Figure 21 shows well-known algorithms that can be automatically analyzed in our framework. Example *Knuth-Morris-Pratt* shows the search function of the Knuth-Morris-Pratt algorithm for string search. To derive a bound for this algorithm we have add an assertion that indicates the bounds of the values that are stored in the array b. Using this information we derive a tight linear bound.

Example *Greatest Common Divisor* is the usual implementation of the GCD algorithm. We automatically derive a linear bound. Note that the two tests at the beginning that check if the inputs are positive are essential. Finally, example *Quick Sort* shows a skeleton of the quick sort sorting algorithm. Since we can only derive linear bounds we left out the inner loop that swaps the array elements and determines the position $\mathsf{n} + \mathsf{lo}$ of the pivot. We only assert that $\mathsf{lo} < \mathsf{n} + \mathsf{lo} <= \mathsf{hi}$. We then derive a tight linear bound.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor.*

File	KoAT		Rank		LOOPUS		SPEED	AAA
`gcd.c`	?		$(((2{+}1)\ldots$	$O(n)$	—		?	$\lvert[0,x]\rvert+\lvert[0,y]\rvert$
`kmp.c`	?		$(((2{+}(n{+}\ldots$	$O(n^2)$	$\mathrm{mx}(n,0)\ldots$	$O(n)$	?	$1{+}2\lvert[0,n]\rvert$
`qsort.c`	?		—		—		?	$1{+}2\lvert[0,\mathsf{len}]\rvert$
`speed_pldi09 fig4_2.c`	—		$(((2{+}n)\ldots$	$O(n)$	—		$\frac{n}{m}+n$	$1{+}2\lvert[0,n]\rvert$
`speed_pldi09 fig4_4.c`	—		$(((2{+}(-1\ldots$	$O(n)$	—		$\frac{n}{m}+m$	$\lvert[0,n]\rvert$
`speed_pldi09 fig4_5.c`	$28d+7g+27$	$O(n)$	$(((2{+}(-1\ldots$	$O(n)$	—		$\mathrm{mx}(n,n-m)$	—
`speed_pldi10 ex1.c`	—		—		—		n	$\lvert[0,n]\rvert$
`speed_pldi10 ex3.c`	—		$(((2{+}(-1\ldots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
`speed_pldi10 ex4.c`	$110a+33$	$O(n)$	—		—		$n+1$	$1{+}2\lvert[0,n]\rvert$
`speed_popl10 fig2_1.c`	$9a+9b+\ldots$	$O(n)$	$((2{+}((-y\ldots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$\lvert[x,n]\rvert+\lvert[y,m]\rvert$
`speed_popl10 fig2_2.c`	$6a{+}9b{+}3c+5$	$O(n)$	$((2{-}x\ldots$	$O(n)$	$\mathrm{mx}(0,(x+1{-}z)\ldots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,n{-}z)$	$\lvert[x,n]\rvert+\lvert[z,n]\rvert$
`speed_popl10 nstd_multiple.c`	—		$((2{-}x{+}n\ldots$	$O(n^2)$	$\mathrm{mx}(0,m{-}y)+\mathrm{mx}(0,n{-}x)$	$O(n)$	$\mathrm{mx}(0,m{-}y){+}\mathrm{mx}(0,n{-}x)$	$\lvert[x,n]\rvert+\lvert[y,m]\rvert$
`speed_popl10 nstd_single.c`	$48b+16$	$O(n)$	$(((1{-}x{+}n\ldots$	$O(n)$	$\mathrm{mx}(0{,}n{-}1)\ldots$	$O(n)$	n	$\lvert[0,n]\rvert$
`speed_popl10 sqntl_single.c`	$21b+6$	$O(n)$	$((2-x{+}n\ldots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
`speed_popl10 smpl_multiple.c`	$9c+10d+7$	$O(n)$	$((2{-}y{+}m\ldots$	$O(n)$	$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,m]\rvert+\lvert[0,n]\rvert$
`speed_popl10 smpl_single2.c`	$20d+12c+17$	$O(n)$	—		$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,n]\rvert+\lvert[0,m]\rvert$
`speed_popl10 smpl_single.c`	$4b+6$	$O(n)$	$((2{-}x{+}n\ldots$	$O(n)$	$\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
`t07.c`	?		$2+x$	$O(n)$	$\mathrm{mx}(x,0)\ldots$	$O(n)$	?	$1{+}3\lvert[0,x]\rvert+\lvert[0,y]\rvert$
`t08.c`	?		$((2{+}z{-}y\ldots$	$O(n)$	$\mathrm{mx}(0{,}y{-}2)\ldots$	$O(n)$	?	$1.33\lvert[y,z]\rvert{+}0.33\lvert[0,y]\rvert$
`t10.c`	?		$((2{-}y{+}x\ldots$	$O(n)$	$\mathrm{mx}(0,x{-}y)$	$O(n)$	?	$\lvert[y,x]\rvert$
`t11.c`	?		$((2{-}y{+}m\ldots$	$O(n)$	$\mathrm{mx}(0,n{-}x))+\mathrm{mx}(0,m{-}y)$	$O(n)$	?	$\lvert[x,n]\rvert+\lvert[y,m]\rvert$
`t13.c`	?		$(((1{+}y^2/2\ldots$	$O(n^2)$	$2{\cdot}\mathrm{mx}(x,0)+\mathrm{mx}(y,0)$	$O(n)$	?	$2\lvert[0,x]\rvert+\lvert[0,y]\rvert$
`t15.c`	?		$((1{+}x\ldots$	$O(n)$	—		?	$\lvert[0,x]\rvert$
`t16.c`	?		$((-99{\cdot}y\ldots$	$O(n)$	—		?	$101\lvert[0,x]\rvert$
`t19.c`	?		$((153{+}k\ldots$	$O(n)$	$\mathrm{mx}(0{,}i{-}10^2){+}\mathrm{mx}(0{,}k{+}i{+}51)$	$O(n)$	?	$50{+}\lvert[-1,i]\rvert+\lvert[0,k]\rvert$
`t20.c`	?		$(2{-}y{+}x\ldots$	$O(n)$	$2{\cdot}\mathrm{mx}(0{,}y{-}x){+}\mathrm{mx}(0{,}x{-}y)$	$O(n)$	?	$\lvert[x,y]\rvert+\lvert[y,x]\rvert$
`t27.c`	?		—		$10^3\mathrm{mx}(0,-n)\ldots$	$O(n)$	?	$0.01\lvert[n,y]\rvert{+}11\lvert[n,0]\rvert$
`t28.c`	?		$((1{-}y{+}x\ldots$	$O(n)$	$10^3\,\mathrm{mx}(0,x-y)\ldots$	$O(n)$	?	$\lvert[x,0]\rvert+\lvert[0,y]\rvert+1002\lvert[y,x]\rvert$
`t30.c`	?		—		—		?	$\lvert[0,x]\rvert+\lvert[0,y]\rvert$
`t37.c`	?		—		—		?	$3{+}2\lvert[0,x]\rvert+\lvert[0,y]\rvert$
`t39.c`	?		—		—		?	$1.33{+}0.67\lvert[z,y]\rvert$
`t46.c`	?		—		—		?	$\lvert[0,y]\rvert$
`t47.c`	?		$4+n$	$O(n)$	$1+\mathrm{mx}(n,0)$	$O(n)$	?	$1{+}\lvert[0,n]\rvert$

Table 2. Experimental evaluation comparing the bounds generated KoAT, Rank, LOOPUS, SPEED, and our automatic amortized analysis on several challenging linear examples. Results for KoAT and SPEED were extracted from previous publications because KoAT cannot take C programs as input in its current version and SPEED is not available. Entries marked with ? indicate that we cannot test the respective example with the tool. Entries marked with — indicate that the tool failed to produce a result.

Comput. Sci., 413(1):142–159, 2012.

[2] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Fundamental Approaches to Software Engineering - 15th Int. Conf. (FASE'12)*, pages 130–145, 2012.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[4] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symposium (SAS'12)*, pages 405–421, 2012.

[5] A. W. Appel et al. *Program Logics for Certified Compilers*. Cambridge University Press, 2013.

[6] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[7] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[8] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 90–101, 2009.

[9] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie. System-Level Non-Interference for Constant-Time Cryptography. *IACR Cryptology ePrint Archive*, 2014:422, 2014.

[10] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

[11] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th Int. Conf. (VSTTE'13)*, 2013. To appear.

[12] V. A. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *7th Int. Symp. on Memory Management (ISMM'08)*, pages 141–150, 2008.

[13] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[14] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *28th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'13*, pages 33–52, 2013.

[15] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *Conf. on Prog. Lang. Design and Impl. (PLDI'14)*, page 30, 2014.

[16] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX Annual Technical Conference (USENIX'10)*, 2010.

[17] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy Types. In *27th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'12*, pages 831–850, 2012.

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

[19] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Asia Pacific Workshop on Systems (APSys'11)*, 2011.

[20] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

[21] S. Gulwani, S. Jain, and E. Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, pages 375–385, 2009.

[22] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[23] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[24] J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[25] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[26] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[27] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative Reasoning for Proving Lock-Freedom. In *28th ACM/IEEE Symposium on Logic in Computer Science (LICS'13)*, 2013.

[28] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[29] M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Joint 25th RTA and 12th TLCA*, 2014.

[30] J. B. Jensen, N. Benton, and A. Kennedy. High-Level Separation Logic for Low-Level Code. In *40th ACM Symp. on Principles of Prog. Langs. (POPL'13)*, pages 301–314, 2013.

[31] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Emb. Sys., 11th Int. Workshop (CHES'09)*, pages 1–17, 2009.

[32] T. Kleymann. Hoare Logic and Auxiliary Variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.

[33] D. E. Knuth. Twenty Questions for Donald Knuth. `http://www.informit.com/articles/article.aspx?p=2213858`, 2014. Accessed: 2014-06-21.

[34] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013. ISSN 0740-7459.

[35] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *33th ACM Symp. on Principles of Prog. Langs. (POPL'06)*, pages 320–333, 2006.

[36] T. Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, volume 62 of *NATO Science Series*, pages 341–367. Springer, 2002.

[37] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005.

[38] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[39] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[40] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symposium (SAS'11)*, 2011.