**Extended version**

# Compositional Certified Resource Bounds

Quentin Carbonneaux Jan Hoffmann Zhong Shao

Yale University

## Abstract

This paper presents a new approach for automatically deriving worst-case resource bounds for C programs. The described technique combines ideas from amortized analysis and abstract interpretation in a unified framework to address four challenges for state-of-the-art techniques: compositionality, user interaction, generation of proof certificates, and scalability. *Compositionality* is achieved by incorporating the potential method of amortized analysis. It enables the derivation of global whole-program bounds with local derivation rules by naturally tracking size changes of variables in sequenced loops and function calls. The resource consumption of functions is described abstractly and a function call can be analyzed without access to the function body. *User interaction* is supported with a new mechanism that clearly separates qualitative and quantitative verification. A user can guide the analysis to derive complex non-linear bounds by using auxiliary variables and assertions. The assertions are separately proved using established qualitative techniques such as abstract interpretation or Hoare logic. *Proof certificates* are automatically generated from the local derivation rules. A soundness proof of the derivation system with respect to a formal cost semantics guarantees the validity of the certificates. *Scalability* is attained by an efficient reduction of bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. The analysis framework is implemented in the publicly-available tool $C^4B$. An experimental evaluation demonstrates the advantages of the new technique with a comparison of $C^4B$ with existing tools on challenging micro benchmarks and the analysis of more than 2900 lines of C code from the cBench benchmark suite.

## 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 [Regehr et al. 2005]. Static energy usage information is critical for autonomous systems and has applications in cloud computing [Cohen et al. 2012; Carroll and Heiser 2010]. Worst-case time bounds can help create constant-time implementations that prevent side-channel attacks [Käsper and Schwabe 2009; Barthe et al. 2014]. Loop and recursion-depth bounds are used to ensure the accuracy of programs that are executed on unreliable hardware [Carbin et al. 2013] and complexity bounds are needed to verify cryptographic protocols [Barthe et al. 2009]. In general, quantitative resource information can provide useful feedback for developers.

Available techniques for *automatically* deriving worst-case resource bounds fall into two categories. Techniques in the first category derive impressive bounds for numerical imperative programs, but are not compositional. This is problematic if one needs to derive global whole-program bounds. Techniques in the second category derive tight whole-program bounds for programs with regular loop or recursion patterns that decrease the size of an individual variable or data structure. They are highly compositional, scale for large programs, and work directly on the syntax. However, they do not support multivariate interval-based resource bounds (e.g., $x - y$) which are common in C programs. Indeed, it has been a long-time open problem to develop compositional resource analysis techniques that can work for typical imperative code with non-regular iteration patterns, signed integers, mutation, and non-linear control flow.

Tools in the first category include SPEED [Gulwani et al. 2009b], KoAT [Brockschmidt et al. 2014], PUBS [Albert et al. 2012a], Rank [Alias et al. 2010], and LOOPUS [Sinn et al. 2014]. They lack compositionality in at least two ways. First, they all base their analysis on some form of *ranking function* or *counter instrumentation* that is linked to a local analysis. As a result, loop bounds are arithmetic expressions that depend on the values of variables just before the loop. This makes it hard to give a resource bound on a sequence of loops and function calls in terms of the input parameters of a function. Second, while all popular imperative programming languages provide a function or procedure abstraction, available tools are not able to abstract resource behavior; instead, they have to inline the procedure body to perform their analysis.

Tools in the second category originate form the *potential method* of amortized analysis and type systems for functional programs [Hofmann and Jost 2003; Hoffmann et al. 2012]. It has been shown that class definitions of object-oriented programs [Hofmann and Jost 2006] and data-structure predicates of separation logic [Atkey 2010] can play the role of the type system in imperative programs. However, a major weakness of existing potential-based techniques is that they can only associate *potential* with individual program variables or data structures. For C programs, this fails for loops as simple as `for(i=x;i<y;i++)` where $y - i$ decreases, but not $|i|$.

A general problem with existing tools (in both categories) is user interaction. When a tool fails to find a resource bound for a program, there is no possibility for sound user interaction to guide the tool during bound derivation. For example, there is no concept of manual proofs of resource bounds; and no framework can support composition of manually derived bounds with automatically inferred bounds.

This paper presents a new compositional framework for automatically deriving resource bounds on C programs. This new approach is an attempt to unify the two aforementioned categories: It solves the compositionality issues of techniques for numerical imperative code by adapting amortized-analysis–based techniques from the functional world. Our automated analysis is able to infer resource bounds on C programs with mutually-recursive functions and integer loops. The resource behavior of functions can be summarized in a functional specification that can be used at every call site without accessing the function body. To our knowledge this is the first technique based on amortized analysis that is able to derive bounds that depend on negative numbers and differences of variables. It is also the first resource analysis technique for C that deals naturally with recursive functions and sequenced loops, and can handle resources that may become available during execution (e.g., when freeing memory). Compared to more classical approaches based on ranking functions, our tool inherits the benefits of amortized reasoning. Using only one simple mechanism, it handles:

- interactions between *sequential loops or function calls* through size changes of variables,
- *nested loops* that influence each other with the same set of modified variables,
- and *amortized bounds* as found, for example, in the Knuth-Morris-Pratt algorithm for string search.

The main innovations that make amortized analysis work on imperative languages are to base the analysis on a Hoare-like logic and to track multivariate quantities instead of program variables. This leads to precise bounds expressed as functions of sizes $|[x, y]| = \max(0, y - x)$ of intervals. A distinctive feature of our analysis system is that it reduces linear bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. This enables the efficient inference of global bounds for larger programs. Moreover, our local inference rules automatically generate proof certificates that can be easily checked in linear time.

The use of the potential method of amortized analysis makes user interaction possible in different ways. For one thing, we can directly combine the new automatic analysis with manually derived bounds in a previously-developed quantitative Hoare logic [Carbonneaux et al. 2014] (see Section 7). For another thing, we describe a new mechanism that allows the separation of quantitative and qualitative verification (see Section 6). Using this mechanism, the user can guide the analysis by using auxiliary variables and logical assertions that can be verified by existing qualitative tools such as Hoare logic or abstract interpretation. In this way, we can benefit from existing automation techniques and provide a middle-ground between fully automatic and fully manual verification for bound derivation. This enables the semi-automatic inference of non-linear bounds, such as polynomial, logarithmic, and exponential bounds.

We have implemented the analysis system in the tool $C^4B$ and experimentally evaluated its effectiveness by analyzing system code and examples from the literature. $C^4B$ has automatically derived global resource bounds for more than 2900 lines of C code from the cBench benchmark suite. Appendix C 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 one of these patterns, and in most cases the derived constant factors are tight. To compare $C^4B$ with existing techniques, we tested our examples with tools such as KoAT [Brockschmidt et al. 2014], Rank [Alias et al. 2010], and LOOPUS [Sinn et al. 2014]. Our experiments show that the bounds that we derive are often more precise than those derived by existing tools. Only LOOPUS [Sinn et al. 2014], which also uses amortization techniques, is able to achieve a similar precision.

Examples from cBench and micro benchmarks demonstrate the practicality and expressiveness of the user guided bound inference. 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 bound that amortizes the cost of $k$ successive increments to a binary counter (see Section 6).

In summary, we make the following *contributions*.

- We develop the first automatic amortized analysis for C programs. It is naturally compositional, tracks size changes of variables to derive global bounds, can handle mutually-recursive functions, generates resource abstractions for functions, derives proof certificates, and handles resources that may become available during execution.
- We show how to automatically reduce the inference of *linear* resource bounds to efficient LP solving.
- We describe a new method of harnessing existing qualitative verification techniques to guide the automatic amortized analysis to derive *non-linear* resource bounds with LP solving.
- We prove the soundness of the analysis with respect to a parametric cost semantics for C programs. The cost model can be further customized with function calls ($\mathsf{tick}(\mathsf{n})$) that indicate resource usage.
- We implemented our resource bound analysis in the publicly-available tool $C^4B$.

- We present experiments with $C^4B$ on more than 2900 lines of C code. A detailed comparison shows that our prototype is the only tool that can derive global bounds for larger C programs while being as powerful as existing tools when deriving linear local bounds for tricky loop and recursion patterns.

## 2. The Potential Method

The idea that underlies the design of our framework is amortized analysis [Tarjan 1985]. Assume that a program $S$ executes on a starting state $\sigma$ and consumes $n$ resource units of some user-defined quantity. We denote that by writing $(S, \sigma) \Downarrow_n \sigma'$ where $\sigma'$ is the program state after the execution. The basic idea of amortized analysis is to define a *potential function* $\Phi$ that maps program states to non-negative numbers and to show that $\Phi(\sigma) \geqslant n$ if $\sigma$ is a program state such that $(S, \sigma) \Downarrow_n \sigma'$. Then $\Phi(\sigma)$ is a valid resource bound.

To obtain a compositional reasoning 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 $\sigma$ and $\sigma'$ 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 $\sigma'$. 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 $\Phi$ 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

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

This rule already shows a departure from classical techniques that are based on ranking functions. Reasoning with two potential functions promotes compositional reasoning by focusing on the sequencing of programs. In the previous rule, $\Phi$ gives a bound for $S_1; S_2$ through the intermediate potential $\Phi'$, even though it was derived on $S_1$ only. 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 reasoning about resource usage in a flexible and compositional way, which, as a side effect, produces a certificate for the derived resource bound.

The derivation of a resource bound using potential functions is best explained by example. If we use a cost model that assigns cost $n$ to the function call $\mathsf{tick}(\mathsf{n})$ and cost 0 to all other operations then the cost of the following example can be bounded by $|[x, y]| = \max(y - x, 0)$.

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

**Figure 1:** 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.

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

To derive this bound, we start with the initial potential $\Phi_0 = |[x, y]|$, which we also use as the loop invariant. For the loop body we have (like in Hoare logic) to derive a triple $\{\Phi_0\}\, \mathsf{x} = \mathsf{x} + 1; \mathsf{tick}(1)\, \{\Phi_0\}$. We can only do so if we utilize the fact that $x < y$ at the beginning of the loop body. The reasoning then works as follows. We start with the potential $|[x, y]|$ and the fact that $|[x, y]| > 0$ before the assignment. 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.

Figure 1 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. In the figure, we separate logical assertions from potential functions with semicolons. Note that the logical assertions 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 computes the bound $|x| + |y| + 10$, Rank computes the bound $y - x - 7$, and LOOPUS computes the bound $y - x - 9$. Only PUBS computes the tight bound $0.1(y - x)$ if we translate the program into a term-rewriting system by hand. We will show in the following sections that the potential method makes automatic bound derivation straightforward.

The concept 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 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.

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

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

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.

In general, the potential based approach addresses several challenges in resource bound analysis. This includes amortized bounds in which individual loop iterations have different resource consumption, sequential loops or function calls that require the tracking of size changes of variables, and nested loops that interact by modifying the same set of variables.

## 3. Compositional Resource-Bound Analysis

In this section we describe the high-level design of the automatic amortized analysis that we implemented in $C^4B$. Examples explain and motivate our design decisions.

***Linear Potential Functions.*** To find resource bounds automatically, we first need to restrict our search space. In this work, we focus on the following form of potential functions, which can express tight bounds for many typical programs and allows for inference with *linear programming*.

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

Here $\sigma : (\mathrm{Locals} \to \mathbb{Z}) \times (\mathrm{Globals} \to \mathbb{Z})$ is a simplified program state that maps variable names to integers, $|[a,b]| = \max(0, b-a)$, and $q_i \in \mathbb{Q}$. 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

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

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 $\Phi$ 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. As a notable difference from previous works based on amortized-analysis, we do not require all the coefficients of our potential function to be positive. In practice, we will see using a semantic argument that despite this additional freedom, potential functions always remain positive.

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 $c_0$.

***Abstract Program State.*** In addition to the quantitative annotations, our automatic amortized analysis needs to maintain a minimal abstract 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).

The abstract program state is represented as *logical contexts* in the derivation system used by our automated tool. Our implementation finds these logical contexts using abstract interpretation with the domain of linear inequalities. We observed that the rules of the analysis often require only minimal local knowledge. This means that it is not necessary for us to compute precise loop invariants and only a rough fixpoint (e.g. keeping only inequalities on variables unchanged by the loop) is sufficient to obtain good bounds.

***Challenging Loops.*** One might think that our set of potential functions is too simplistic to be able to express and prove bounds for realistic programs. Nevertheless, we can handle challenging example programs without special tricks or techniques. Examples *speed_1* and *speed_2* in Figure 2, which are taken from previous work [Gulwani et al. 2009b], demonstrate that our method can handle *tricky iteration patterns*. The SPEED tool [Gulwani et al. 2009b] 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 $|[x,n]| + |[z,n]|$.

***Nested and Sequenced Loops.*** Example *t08a* in Figure 2 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* in Figure 2 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-

```
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)}
  tick(9);
} {·; P(n,y)}
```

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

**t27**

**Figure 2:** Derivations of bounds on the number of ticks for challenging examples. Examples *speed_1* and *speed_2* (from [Gulwani et al. 2009b]) 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 $\Gamma$ we only mention assertions that we use. In Example *t27*, we use the abbreviation $P(n,y) := 59|[n,0]|+0.05|[0,y]|$.

```
void c_down (int x,int y) {
  if (x>y) {tick(1); c_up(x-1,y);}
}
void c_up (int x, int y) {
  if (y+1<x) {tick(1); c_down(x,y+2);}
}
```

$0.33 + 0.67|[y,x]|$ (c_down(x,y))
$0.67|[y,x]|$ (c_up(x,y))

**t39**

```
for (; l>=8; l-=8)
  /* process one block */
  tick(N);
for (; l>0; l--)
  /* save leftovers */
  tick(1);
```

$\frac{N}{8}|[0,l]|$ if $N \geqslant 8$
$7\frac{8-N}{8} + \frac{N}{8}|[0,l]|$ if $N < 8$

**t61**

```
for (;;) {
  do { l++; tick(1); }
    while (l<h && *);
  do { h--; tick(1); }
    while (h>l && *);
  if (h<=l) break;
  tick(1); /* swap elems. */ }
```

$2 + 3|[l,h]|$

**t62**

**Figure 3:** Example *t39* shows two mutually-recursive functions with the computed tick bounds. Example *t61* and *t62* demonstrate the unique compositionality of our system. In *t61*, $N \geqslant 0$ is a fixed but arbitrary constant.

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.

***Mutually Recursive Functions.*** As mentioned, the analysis also handles advanced control flow like break and return statements, and mutual recursion. Example *t39* in Figure 3 contains two mutually-recursive functions with their automatically derived tick bounds. The function c_down decrements its first argument $x$ until it reaches the second argument $y$. It then recursively calls the function c_up, which is dual to c_down. Here, we count up $y$ by 2 and call c_down. $C^4B$ is the only available system that computes a tight bound. The analysis amounts to computing the meeting point of two trains that approach each other with different speeds.

***Compositionality.*** With two concrete examples from open-source projects we demonstrate that the compositionality of our method is indeed crucial in practice.

Example *t61* in Figure 3 is typical for implementations of block-based cryptographic primitives: Data of arbitrary length is consumed in blocks and the leftover is stored in a buffer for future use when more data is available. It is present in all the block encryption routines of PGP and also used in performance critical code to unroll a loop. For example we found it in a bit manipulating function of the libtiff library and a CRC computation routine of MAD, an MPEG decoder. This looping pattern is handled particularly well by our method. If $N \geqslant 8$, $C^4B$ infers the bound $\frac{N}{8}|[0,l]|$, but if $N < 8$, it infers $7\frac{8-N}{8} + \frac{N}{8}|[0,l]|$. The selection of the block size (8) and the cost in the second loop (tick(1)) are random choices and $C^4B$ would also derive tight bound for other values.

To understand the resource bound for the case $N < 8$, first note that the cost of the second loop is $|[0,l]|$. After the first loop, we still have $\frac{N}{8}|[0,l]|$ potential available from the invariant. So we have to raise the potential of $|[0,l]|$ from $\frac{N}{8}$ to 1, that is, we must pay $\frac{8-N}{8}|[0,l]|$. But since we got out of the first loop, we know that $l < 8$, so it is sound to only

pay $7\frac{8-N}{8}$ potential units instead. This level of precision and compositionality is only achieved by our novel analysis, no other available tool derives the aforementioned tight bounds.

Example *t62* (Figure 3) is the inner loop of a quick sort implementation in cBench. More precisely, it is the partitioning part of the algorithm. This partition loop has linear complexity, and feeding it to our analysis gives the worst-case bound $2 + 3|[l, h]|$. This bound is not optimal but it can be refined by rewriting the program. To understand the bound, we can reason as follows. If $h \geqslant l$ initially, the cost of the loop is 2. Otherwise, the cost of each round (at most 3) can be payed using the potential of $[l, h]$ by the first increment to $l$ because we know that $l < h$. The two inner loops can also use $[l, h]$ to pay for their inner costs. KoAT fails to find a bound and LOOPUS derives the quadratic bound $(h - l - 1)^2$. Following the classical technique, these tools try to find one ranking function for each loop and combine them multiplicatively or additively.

We only show a small selections of the programs that we can handle automatically here. In Appendix C is a list of more than 30 classes of challenging programs that we can automatically analyze. Section 8 contains a more detailed comparison with other tools for automatic bound derivation.

## 4. Derivation System

In the following we describe the local and compositional derivation rules of the automatic amortized analysis.

***Cost Aware Clight.*** We present the rules for a subset of Clight. Clight is the first intermediate language of the CompCert compiler [Leroy 2009]. It is a subset of C with a unified looping construct and side-effect free expressions. We reuse most of CompCert's syntax but instrument the semantics with a resource counter (an arbitrary rational number) that can be modified by tick() function calls. More details on this cost-aware semantics are provided in Section 7.

In the rules, assignments are restricted to 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. Non-linear operations such as $x \leftarrow z * y$ or $x \leftarrow a[y]$ are handled by assigning 0 to coefficients like $q_{xa}$ and $q_{ax}$ that contain $x$ after the assignment. This sound treatment ensures that no further loop bounds depend on the result of the non-linear operation.

***Judgements.*** The derivation system for the automatic amortized analysis is defined in Figure 4. The derivation rules yield judgements of the form

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

The part $\{\Gamma; Q\}\, S\, \{\Gamma'; Q'\}$ of the judgement can be seen as a quantitative Hoare triple. All assertions are split into two parts, the logical part and the quantitative part. The quantitative part $Q$ represents a potential function as a collection of rational numbers $q_i$ indexed by the index set $I$. The logical part $\Gamma$ is left abstract but is enforced by our derivation system to respect classic Hoare logic constraints. The meaning of this basic judgment is as follows: If $S$ is executed with starting state $\sigma$, the assertions in $\Gamma$ hold, $Q' \geqslant 0$, and at least $Q(\sigma)$ resources are available then the evaluation does not run out of resources and, if the execution terminates in state $\sigma'$, there are at least $Q'(\sigma')$ resources left and $\Gamma'$ holds for $\sigma'$.

The judgement is a bit more involved since we have to take into account the early exit statements break and return. This is similar to classical Hoare triples in the presence of non-linear control flow. In the judgement, $(\Gamma_B; Q_B)$ is the postcondition that holds when breaking out of a loop using break. Similarly, $(\Gamma_R; Q_R)$ is the postcondition that holds when returning from a function call with return. (This becomes clear in the rules.)

As a convention, if $Q$ and $Q'$ are quantitative annotations we assume that $Q = (q_i)_{i \in I}$ and $Q' = (q'_i)_{i \in I}$. The notation $Q \pm n$ used in many rules defines a new context $Q'$ such that $q'_0 = q_0 \pm n$ and $\forall i \neq 0.\, q'_i = q_i$. Finally, if a rule mentions $Q$ and $Q'$ and leaves the latter undefined at some index $i$ we assume that $q'_i = q_i$.

***Function Specifications.*** During the analysis, function specifications are quadruples $(\Gamma_f; Q_f, \Gamma'_f; Q'_f)$ where $\Gamma_f; Q_f$ depend on $a\vec{r}gs$, and $\Gamma'_f; Q'_f$ depend on $ret$. These parameters are instantiated by appropriate variables on call sites. A distinctive feature of our analysis is that it respects the function abstraction: when deriving a function specification it generates a set of constraints and the above quadruple; once done, the constraint set can readily be reused for every call site and the function need not be analyzed multiple times. Therefore, the derivation rules are parametric in a function context $\Delta$ that we leave implicit in the rules presented here.

***Derivation Rules.*** The rules of our derivation system must serve two purposes. They must *attach* potential to certain program variable intervals and use this potential, when it is allowed, to *pay* for resource consuming operations. These two purposes are illustrated on the Q:SKIP rule. This rule reuses its precondition as postcondition, it is explained by two observations: First, no resource is consumed by the skip operation, thus no potential has to be used to pay for the evaluation. Second, the program state is not changed by the execution of a skip statement. Thus all potential available before the execution of the skip statement is still available after. These two observations would also justify a rule with the side-condition $Q \geqslant Q'$, because some potential is always safely lost. For the sake of orthogonality, it is not how we present the rule and this form of weakening is in fact expressed as the stand-alone rule Q:WEAK described later.

The rules Q:INC and Q:DEC describe how the potential is distributed after a size change of a variable. The rule Q:INC is for incements $x \leftarrow x + y$ and the symmetric rule Q:DEC is for decrements $x \leftarrow x - y$. The two rules are equivalent in the case where $y = 0$.

$$\frac{}{B, R \vdash \{\Gamma; Q\}\,\mathsf{skip}\,\{\Gamma; Q\}}\ (\text{Q:SKIP}) \qquad \frac{}{(\Gamma; Q_B), R \vdash \{\Gamma; Q_B\}\,\mathsf{break}\,\{\Gamma'; Q'\}}\ (\text{Q:BREAK})$$

$$\frac{n < 0 \implies Q \geqslant 0}{B, R \vdash \{\Gamma; Q\}\,\mathsf{tick}(n)\,\{\Gamma; Q{-}n\}}\ (\text{Q:TICK}) \qquad \frac{P = Q_R[ret/x] \qquad \Gamma = \Gamma_R[ret/x] \qquad \forall i \in \mathrm{dom}(P).\, p_i = q_i}{B, (\Gamma_R; Q_R) \vdash \{\Gamma; Q\}\,\mathsf{return}\ x\,\{\Gamma'; Q'\}}\ (\text{Q:RETURN})$$

$$\frac{\forall u.(q_{yu} = q'_{xu} + q'_{yu} \wedge q_{uy} = q'_{ux} + q'_{uy})}{B, R \vdash \{\Gamma[x/y]; Q\}\,x \leftarrow y\,\{\Gamma; Q'\}}\ (\text{Q:UPDATE}) \qquad \frac{Q \geqslant Q' \qquad (\Gamma'; Q'), R \vdash \{\Gamma; Q\}\,S\,\{\Gamma; Q\}}{B, R \vdash \{\Gamma; Q\}\,\mathsf{loop}\ S\,\{\Gamma'; Q'\}}\ (\text{Q:LOOP})$$

$$\frac{\begin{array}{c} q'_{0y} = q_{0y} - \sum_u \max(q_{ux}, -q_{xu}) \\ q'_{y0} = q_{y0} - \sum_u \max(q_{xu}, -q_{ux}) \end{array}}{B, R \vdash \{\Gamma[x/x{+}y]; Q\}\,x \leftarrow x + y\,\{\Gamma; Q'\}}\ (\text{Q:INC}) \qquad \frac{\begin{array}{c} q'_{0y} = q_{0y} - \sum_u \max(q_{xu}, -q_{ux}) \\ q'_{y0} = q_{y0} - \sum_u \max(q_{ux}, -q_{xu}) \end{array}}{B, R \vdash \{\Gamma[x/x{-}y]; Q\}\,x \leftarrow x - y\,\{\Gamma; Q'\}}\ (\text{Q:DEC})$$

$$\frac{\begin{array}{c} B, R \vdash \{\Gamma \wedge e; Q\}\,S_1\,\{\Gamma'; Q'\} \\ B, R \vdash \{\Gamma \wedge \neg e; Q\}\,S_2\,\{\Gamma'; Q'\} \end{array}}{B, R \vdash \{\Gamma; Q\}\,\mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2\,\{\Gamma'; Q'\}}\ (\text{Q:IF}) \qquad \frac{\begin{array}{c} B, R \vdash \{\Gamma; Q\}\,S_1\,\{\Gamma'; Q'\} \\ B, R \vdash \{\Gamma'; Q'\}\,S_2\,\{\Gamma''; Q''\} \end{array}}{B, R \vdash \{\Gamma; Q\}\,S_1; S_2\,\{\Gamma''; Q''\}}\ (\text{Q:SEQ})$$

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

$$\frac{}{B, R \vdash \{\Gamma; Q\}\,\mathsf{assert}\ e\,\{\Gamma \wedge e; Q\}}\ (\text{Q:ASSERT}) \qquad \frac{\begin{array}{c} \Sigma f = (\vec{y}, S_f) \qquad Q_f \geqslant 0 \qquad Q'_f \geqslant 0 \\ B, (\Gamma'_f; Q'_f) \vdash \{\Gamma_f[a\vec{r}gs/\vec{y}]; Q_f[a\vec{r}gs/\vec{y}]\}\,S_f\,\{\Gamma'; Q'\} \end{array}}{(\Gamma_f; Q_f, \Gamma'_f; Q'_f) \in \Delta(f)}\ (\text{Q:EXTEND})$$

$$\frac{\Gamma_1 \models \Gamma_2 \qquad Q_1 \geq_{\Gamma_1} Q_2 \qquad B, R \vdash \{\Gamma_2; Q_2\}\,S\,\{\Gamma'_2; Q'_2\} \qquad \Gamma'_2 \models \Gamma'_1 \qquad Q'_2 \geq_{\Gamma'_2} Q'_1}{B, R \vdash \{\Gamma_1; Q_1\}\,S\,\{\Gamma'_1; Q'_1\}}\ (\text{Q:WEAK})$$

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

**Figure 4:** Inference rules of the quantitative analysis.

The two previous rules are essentially the same, so we only focus on the explanation of Q:INC. In this rule, the program updates a variable $x$ with $x + y$. 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. Consider $S(x) = q_{xu}|[x, u]| + q_{ux}|[u, x]|$, during the increment, $S$ will increase by at most $y \cdot \max(q_{ux}, -q_{xu})$. This means that we must decrease the potential $q_{0y}$ of $|[0, y]|$ by $\max(q_{ux}, -q_{xu})$ to pay for the change on $|[x, u]|$ and $|[u, x]|$. A symmetric argument explains the change on $q_{y0}$.

The rule Q:LOOP is a cornerstone of our analysis. To apply it on a loop body, one needs to find an invariant potential $Q$ that will pay for the iterations. At each iteration, some resources are spent by the loop body. Since the loop can only be exited with a break statement, the postcondition $\{\Gamma'; Q'\}$ for the statement $\mathsf{loop}\ S$ is used as break postcondition in the derivation for $S$.

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 across a function call 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 $(\Gamma_f; Q_f)$ are named using a fixed vector $\vec{args}$ of names different from all program variables. This prevents name conflicts 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 $(\Gamma'_f; Q'_f)$.

The rule Q:WEAK is not syntax directed. In the implementation we apply Q:WEAK before loops and between the two statements 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 following: 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 $Q' \geq_\Gamma Q$ that is defined in the rule RELAX. This rule also deals 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 $|[x,y]| \geqslant \ell$ is larger than a constant $\ell$ then we can turn the potential $q_{xy}{\cdot}|[x,y]|$ form the interval into the constant potential $\ell{\cdot}q_{xy}$ 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.

## 5. Automatic Inference via LP Solving

We separate the search of a derivation in two steps. As a first step we go through the functions of the program and apply inductively the derivation rules of the automatic amortized analysis. This is done in a bottom-up way for each strongly connected component (SCC) of the call graph. During this process our tool uses symbolic names for the rational coefficients $q_i$ in the rules. Each time a linear constraint must be satisfied by these coefficients, it is recorded in a global list for the SCC using the symbolic names. We reuse the constraint list for every call from outside the SCC.

We then feed the collected constraints to an off-the-shelf LP solver (currently CLP [COIN-OR Project 2014]). If the solver successfully finds a solution, we know that a derivation exists and extract the values for the initial $Q$ from the solver to get a resource bound for the program. To get a full derivation, we 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.

Figure 5 contains an example derivation as produced by $C^4B$. The upper case letters (with optional superscript) such as $Q^{\text{de}}$ are families of variables that are later part of the constraint system that is passed to the LP solver. For example $Q^{\text{de}}$ stands for the potential function $q_0^{\text{de}} + q_{x,0}^{\text{de}}|[x,0]| + q_{0,x}^{\text{de}}|[0,x]| + q_{x,10}^{\text{de}}|[x,10]| + q_{10,x}^{\text{de}}|[10,x]| + q_{0,10}^{\text{de}}|[0,10]|$, where the variables such as $q_{x,10}^{\text{de}}$ are yet unknown and later instantiated by the LP solver.

In general, the weakening rule can be applied after every syntax directed rule. However, it can be left out in practice at some places to increase the efficiency of the tool. The weakening operation $\geq_\Gamma$ is defined by the rule RELAX. It is parameterized by a logical context that is used to gather information on interval sizes. For example,

$$\begin{aligned} P^{\text{de}} \geq_{(\cdot)} P^{\text{we}} \equiv\ & p_{0,10}^{\text{we}} \leqslant p_{0,10}^{\text{de}} + u_{0,10} - v_{0.10} \\ & \wedge\ p_0^{\text{we}} \leqslant p_0^{\text{de}} - 10{\cdot}u_{0,10} + 10{\cdot}v_{0.10} \\ & \wedge\ \forall(\alpha,\beta) \neq (0,10).\, p_{\alpha,\beta}^{\text{we}} \leqslant p_{\alpha,\beta}^{\text{de}}\ . \end{aligned}$$

The other rules are syntax directed and applied inductively. For example, the outermost expression is a loop, so we use the rule Q:Loop at the root of the derivation tree. At this point, we do not know yet whether a loop invariant exists. But we produce the constraints $Q^{\text{lo}} = P^{\text{lo}}$ which is short for the following constraint set.

$$\begin{array}{lll} q_0^{\text{lo}} = p_0^{\text{lo}} & q_{x,0}^{\text{lo}} = p_{x,0}^{\text{lo}} & q_{0,x}^{\text{lo}} = p_{0,x}^{\text{lo}} \\ q_{x,10}^{\text{lo}} = p_{x,10}^{\text{lo}} & q_{10,x}^{\text{lo}} = p_{10,x}^{\text{lo}} & q_{0,10}^{\text{lo}} = p_{0,10}^{\text{lo}} \end{array}$$

These constraints express the fact that the potential functions before and after the loop body are equal and thus constitute an invariant.

After the constraint generation, the LP solver is provided with an objective function to be minimized. We wish to minimize the initial potential, which is a resource bound on the whole program. Here it is given by $Q$. Moreover, we would like to express that minimization of linear potential such as $q_{10,x}|[10,x]|$ takes priority over minimization of constant potential such as $q_{0,10}|[0,10]|$.

To get a tight bound, we use modern LP solvers that allow constraint solving and minimization at the same time: First we consider our initial constraint set as given in Figure 5 and ask the solver to find a solution that satisfies the constraints and minimizes the linear expression

$$1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x}\ .$$

The penalties given to certain factors are used to prioritize certain intervals. For example, a bound with $[10,x]$ will be preferred to another with $[0,x]$ because $|[10,x]| \leqslant |[0,x]|$. The LP solver now returns a solution of the constraint set and an objective value, that is, a mapping from variables to floating-point numbers and the value of the objective function with this instantiation. The solver also memorizes the optimization path that led to the optimal solution. In this case, the objective value would be 5000 since the LP solver assigns $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. We now add the constraint

$$1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x} \leqslant 5000$$

to our constraint set and ask the solver to optimize the objective function

$$q_0 + 11{\cdot}q_{0,10}\ .$$

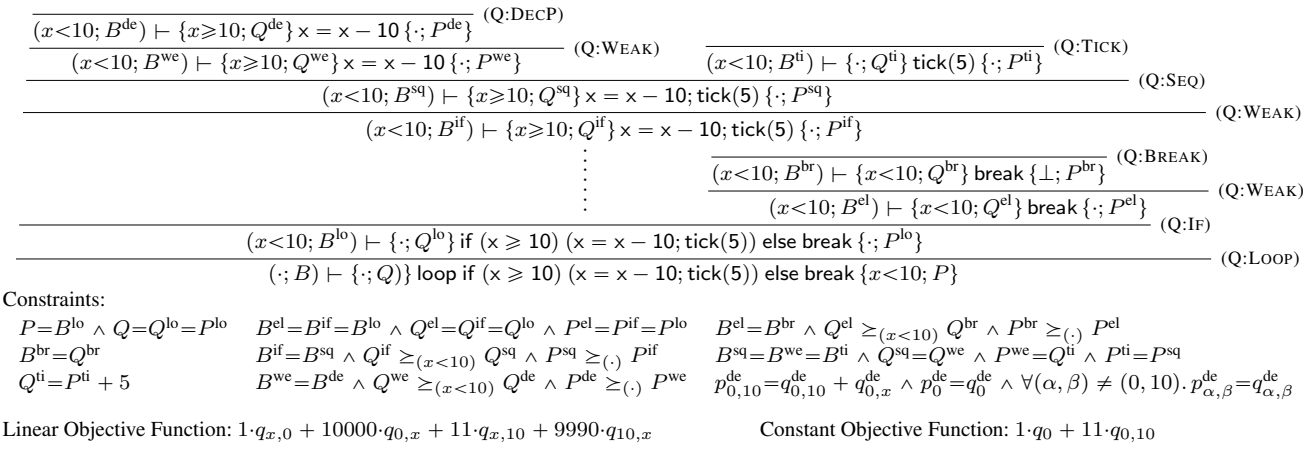


**Figure 5:** An example derivation as produced $C^4B$. The constraints are resolved by an off-the-shelf LP solver.

```
1  logical state invariant {na = #1(a)}
2  while (k > 0) {
3    x=0;
4    while (x < N && a[x] == 1) {
5      assert(na > 0);
6      a[x]=0; na--;
7      tick(1); x++; }
8    if (x < N) { a[x]=1; na++; tick(1); }
9    k--;
10 }
```

**Figure 6:** Assisted bound derivation using logical state. We write $\#_1(a)$ for $\#\{i \mid 0{\leqslant}i{<}N{\wedge}a[i]{=}1\}$. The derived bound is $2|[0,k]| + |[0,na]|$.

```
1  logical state invariant {lg > log2(h − l)}
2  bsearch(x,l,h,lg) {
3    if (h-l > 1) {
4      assert(lg > 0);
5      m = l + (h-l)/2;
6      lg--; if (a[m]>x) h=m; else l=m;
7      tick(M_bsearch);
8      l = bsearch(x,l,h,lg);
9      tick(−M_bsearch);
10   } return l;
11 }
```

**Figure 7:** Assisted bound derivation using logical state. We write $\log_2(x)$ for the integer part of logarithm of $x$ in base 2. The semi-automatically derived bound is $|[0,lg]|$.

This happens in almost no time in practice. The final solution is $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. Thus the derived bound is $0.5|[0,x]|$.

A notable advantage of the LP-based approach compared to SMT-solver–based techniques is that a satisfying assignment is a proof certificate instead of a counter example. To provide high-assurance bounds, this certificate can be checked in linear time by a simple validator.

The constraints that we generate have a particularly simple form that is known as *network problem*. Such problems can be solved in linear time in practice.

## 6. Logical State and User Interaction

While complete automation is desirable, it is not always possible since the problem of bound derivation is undecidable. In this section we present a new technique to derive complex resource bounds semi-automatically by leveraging our automation. Our goal is to develop an interface between bound derivation and established qualitative verification techniques.

When the resource bound of a program depends on the contents of the heap, or is non-linear (e.g. logarithmic, exponential), we introduce a *logical state* using *auxiliary variables*. Auxiliary variables guide $C^4B$ during bound derivation but they do not change the behavior of the program.

```
logical state invariant {nm = A(0)}
i=1; m=a[0];
while (i < N) {
  if (a[i] > m) {
    assert(nm > 0); nm--;
    m = a[i], tick(1);
  } i++, tick(1);
}
```

**Figure 8:** Example of assisted bound derivation using logical state. We write $A(i)$ for $\#\{k \mid i \leqslant k \leqslant N \wedge \forall\, 0 \leqslant j < k.\, a[j] < a[k]\}$. The bound derived is $|[0,N]| + |[0,nm]|$.

More precisely, the technique consists of the following steps. First, a program $P$ that fails to be analyzed automatically is enriched by auxiliary variables $\vec{x}$ to form a program $P_l(\vec{x})$. Second, an initial value $\vec{X}(\sigma)$ for the logical variables is selected to satisfy the proposition:

$$\forall n\, \sigma\, \sigma'.\, (\sigma, P_l(\vec{X}(\sigma))) \Downarrow_n \sigma' \Longrightarrow \exists n'{\leqslant}n.\, (\sigma, P) \Downarrow_{n'} \sigma'. \quad (*)$$

Since the annotated program and the original one are usually syntactically close, the proof of this result goes by simple induction on the resource-aware evaluation judgement. Third, using existing automation tools, a bound $B(\vec{x})$ for $P_l(\vec{x})$ is

derived. Finally this bound, instantiated with $\vec{X}$, gives the final resource bound for the program $P$.

This idea is illustrated by the program in Figure 6. The parts of the code in blue are annotations that were added to the original program text. The top-level loop increments a binary counter $k$ times. A naive analysis of the algorithm yields the quadratic bound $k \cdot N$. However, the algorithm is in fact linear and its cost is bounded by $2k + \#_1(a)$ where $\#_1(a)$ denotes the number of one entries in the array $a$. Since this number depends on the heap contents, no tool available for C is able to derive the linear bound. However, it can be inferred by our automated tool if a logical variable $na$ is introduced. This logical variable is a reification of the number $\#_1(a)$ in the program. For example, on line 6 of the example we are setting `a[x]` to 0 and because of the condition we know that this array entry was 1. To reflect this change on $\#_1(a)$, the logical variable $na$ is decremented. Similarly, on line 8, an array entry which was 0 becomes 1, so $na$ is incremented. To complete the step 2 of the systematic procedure described above, we must show that the extra assertion `na > 0` on line 5 is always true. It is done by proving inductively that $na = \#_1(a)$ and remarking that since `a[x] == 1` is true, we must have $\#_1(a) > 0$, thus the assertion `na > 0` never fails.

Another simple example is given in Figure 7 where a logarithmic bound on the stack consumption of a binary search program is proved using logical variable annotations. Once again, annotations are in blue in the program text. In this example, to ease the proof of equivalence between the annotated program and the original one, we use the inequality $lg > \log_2(h - l)$ as an invariant. This technique allows a simpler proof because, when working with integer arithmetic, it is not always the case that $\log_2(x - x/2) = \log_2(x) - 1$.

Generally, we observed that because the instrumented program is structurally same as the original one, it is enough to prove that the added assertions never fail in order to show the two programs satisfy the proposition (*). This can usually be piggybacked on standard static-analysis tools.

## 7. Soundness Proof

The soundness of the analysis builds on a new cost semantics for Clight and an extended quantitative logic. Using these two tools, the soundness of the automatic analysis described in Section 3 is proved by a translation morphism to the logic.

***Cost Semantics for Clight.*** To base the soundness proof on a formal ground, we start by defining a new cost-aware operational semantics for Clight. 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.

A program state $\sigma = (\theta, \gamma)$ is composed of two maps from variable names to integers. The first map, $\theta$ : Locals $\to \mathbb{Z}$, assigns integers to local variables of a function, and the second map, $\gamma$ : 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 $\sigma$. We write $\sigma(x)$ to obtain the value of $x$ in program state $\sigma$. Similarly, we write $\sigma[x \mapsto v]$ for the state based on $\sigma$ where the value of $x$ is updated to $v$.

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 using a tick instruction that allows the programmer to specify how much and when resource is consumed. If the resource usage is bound to the programming language (e.g. stack usage, number loop iterations and function calls), the tick statements can be inserted automatically by a pre-processing phase. Resources can be released by using a negative tick. Two sample rules of the semantics for update and tick follow.

$$\frac{\sigma' = \sigma[x \mapsto [\![e]\!]_\sigma]}{(\sigma, x \leftarrow e, K, c) \to (\sigma', \mathsf{skip}, K, c)}\ (\mathrm{U}) \qquad \frac{}{(\sigma, \mathsf{tick}(n), K, c) \to (\sigma, \mathsf{skip}, K, c{-}n)}\ (\mathrm{T})$$

The rules have as implicit side condition that $c$ is non-negative. This makes it possible to detect a resource crash as a stuck configuration where $c < 0$.

***Quantitative Hoare Logic.*** To prove the soundness of $C^4B$ we found it useful to go through an intermediate step using a quantitative Hoare logic. This logic is at the same time a convenient semantic tool and a clean way to interface manual proofs with our automation. We base it on a logic for stack usage [Carbonneaux et al. 2014], add support for arbitrary resources, and simplify the handling of auxiliary state.

We define quantitative Hoare triples as $\Delta; B; R \vdash_L \{Q\}\, S\, \{Q'\}$ where $B$, $R$, $Q$, and $Q'$ are maps from program states to an element of $\mathbb{Q} \cup \{\infty\}$ that represents an amount of resources available. The assertions $B$ and $R$ are postconditions for the case in which the block $S$ exits by a break or return statement. Additionally, $R$ depends on the return value of the current function. The meaning of the triple $\{Q\}\, S\, \{Q'\}$ is as follows: If $S$ is executed with starting state $\sigma$, the empty continuation, $Q'(\sigma) \geqslant 0$ for all state $\sigma$, and at least $Q(\sigma)$ resource units 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 $\sigma'$. The logic rules are similar to the ones in previous work.

Finally, we define a strong compositional continuation-based soundness for triples and prove the validity of all the rules in Coq. Details about the logic and its soundness proof are provided in the Appendix B.

***The Soundness Theorem.*** We use the quantitative logic as the target of a translation function for the automatic derivation system. This reveals two orthogonal aspects of the proof: on one side, it relies on amortized reasoning (the quantitative

logic rules), and on the other side, it uses combinatorial properties of our linear potential functions (the automatic analysis rules).

Technically, 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 directly obtain a certified resource bound for the analyzed program.

The translation of an assertion $(\Gamma; Q)$ in the automatic analysis is defined by

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

where we write $\Phi_Q$ for the unique linear potential function defined by the quantitative annotation $Q$. The logical context $\Gamma$ is implicitly lifted to a quantitative assertion by mapping a state $\sigma$ to 0 if $\Gamma(\sigma)$ holds and to $\infty$ otherwise. 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 $J := B, R \vdash \{P\}\, S\, \{P'\}$ in the context $\Delta$ by $\mathcal{T}(J) :=$

$$(\lambda f.\, \mathcal{T}(\Delta(f)));\mathcal{T}(B);\mathcal{T}(R) \vdash_L \{\mathcal{T}(P)\}\, S\, \{\mathcal{T}(P')\}.$$

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

**Theorem 1** (Soundness of the automatic analysis)**.** *If $J$ is a judgement derived by the automatic analysis, then $\mathcal{T}(J)$ is a quantitative Hoare triple derivable in the quantitative logic.*

The proof of this theorem is constructive and 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 they make essential use of the algebraic properties of our linear potential functions. For instance we prove the following lemma to show the soundness of the translation of Q:WEAK.

**Lemma 1** (Relax)**.** *If $Q' \geq_\Gamma Q$ and $\sigma$ 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.

## 8. 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 [Gulwani et al. 2009b,a; Gulwani and Zuleger 2010]. A full list of examples together with the derived bounds is given in Appendix C.

Figure 9 shows five representative loop patterns from the evaluation. Example *t09* is a loop that performs an expensive operation every 4 steps. $C^4B$ is the only tool able to amortize this cost over the input parameter $x$. Example *t19* demonstrates the compositionality 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 `i=i+k+50`. So in total the program performs $52 + |[-1, i]| + |[0, k]|$ ticks. Our analysis finds this tight bound because our amortized analysis naturally takes into account the relation between the two loops. Example *t30* decrements both input variables $x$ and $y$ down to zero in an unconventional way. 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 `assert(y>=0)`. The assignment `x -= y+1` in the loop is split in `x--` and `x -= y`. If we enter the loop then we know that $x > 0$, so we can obtain constant potential from `x--`. Then we know that $x \geqslant y \geqslant 0$, as a consequence we can share the potential of $|[0, x]|$ between $|[0, x]|$ and $|[0, y]|$ after `x -= y`.

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. $C^4B$ 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 10 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

| | t09 | t19 | t30 | t15 | t13 |
|---|---|---|---|---|---|
| | `i=1; j=0;`<br>`while (j<x) {`<br>`  j++;`<br>`  if (i>=4)`<br>`  i=1, tick(40);`<br>`  else i++;`<br>`  tick(1); }` | `while (i>100) {`<br>`  i--; tick(1);`<br>`} i += k+50;`<br>`while (i>=0) {`<br>`  i--; tick(1);`<br>`}` | `while (x>0) {`<br>`  x--;`<br>`  t=x, x=y, y=t;`<br>`  tick(1);`<br>`}` | `assert(y>=0);`<br>`while (x > y) {`<br>`  x -= y+1;`<br>`  for (z=y; z>0; z--)`<br>`  tick(1);`<br>`  tick(1);`<br>`}` | `while (x>0) {`<br>`  x--;`<br>`  if (*) y++;`<br>`  else`<br>`  while (y>0)`<br>`  y--, tick(1);`<br>`  tick(1); }` |
| $C^4B$ | $11\lvert[0,x]\rvert$ | $50+\lvert[-1,i]\rvert+\lvert[0,k]\rvert$ | $\lvert[0,x]\rvert+\lvert[0,y]\rvert$ | $\lvert[0,x]\rvert$ | $2\lvert[0,x]\rvert+\lvert[0,y]\rvert$ |
| Rank | $23{\cdot}x-14$ | $54+k+i$ | — | $2+2x-y$ | $0.5{\cdot}y^2{+}yx\ldots$ |
| LOOPUS | $41\max(x,0)$ | $\max(i{-}100,0)$<br>$+\max(k{+}i{+}51,0)$ | — | — | $2\max(x,0)$<br>$+\max(y,0)$ |

**Figure 9:** Comparison of resource bounds derived by different tools on several examples with linear bounds. The output of Rank has been manually simplified to fit the table.

```
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;
  }
```

| | |
|---|---|
| $C^4B$ | $1{+}2\lvert[0,n]\rvert$ |
| Rank | $O(n^2)$ |
| LOOPUS | $2{+}3\max(n,0)$ |

**Figure 10:** The Knuth-Morris-Pratt algorithm for string search.

| | KoAT | Rank | LOOPUS | SPEED | $C^4B$ |
|---|---|---|---|---|---|
| #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 $C^4B$ with other automatic tools.

| Function | LoC | Bound | Time (s) |
|---|---|---|---|
| adpcm_coder | 145 | $1+\lvert[0,\mathsf{N}]\rvert$ | 0.6 |
| adpcm_decod | 130 | $1+\lvert[0,\mathsf{N}]\rvert$ | 0.2 |
| BF_cfb64_enc | 151 | $1+2\lvert[-1,\mathsf{N}]\rvert$ | 0.7 |
| BF_cbc_enc | 180 | $2+0.25\lvert[-8,\mathsf{N}]\rvert$ | 1.0 |
| mad_bit_crc | 145 | $61.19{+}0.19\lvert[-1,\mathsf{N}]\rvert$ | 0.4 |
| mad_bit_read | 65 | $1+0.12\lvert[0,\mathsf{N}]\rvert$ | 0.05 |
| MD5Update | 200 | $133.95{+}1.05\lvert[0,\mathsf{N}]\rvert$ | 1.0 |
| MD5Final | 195 | 141 | 0.22 |
| sha_update | 98 | $2+3.55\lvert[0,\mathsf{N}]\rvert$ | 1.2 |
| PackBitsDecode | 61 | $1+65\lvert[-129,\mathsf{cc}]\rvert$ | 0.6 |
| KMPSearch | 20 | $1+2\lvert[0,n]\rvert$ | 0.1 |
| ycc_rgb_conv | 66 | $\mathsf{nr}\cdot\mathsf{nc}$ | 0.1 |
| uv_decode | 31 | $\log_2(\mathsf{UV\_NVS})+1$ | 0.1 |

**Table 2:** Derived bounds for functions from cBench.

construction of the table in the initialization procedure of the algorithm, which we can also analyze automatically. We need the assertion since we do not infer any logical assertions 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 tick statements to 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 9, we show the bounds we derived ($C^4B$) together with the bounds derived by LOOPUS [Sinn et al. 2014] and Rank [Alias et al. 2010]. We also contacted the authors of SPEED but have not been able to obtain this tool. KoAT [Brockschmidt et al. 2014] and PUBS [Albert et al. 2012a] 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 completely 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 presented in Table 3. 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 that 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. Existing tools complement $C^4B$ 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

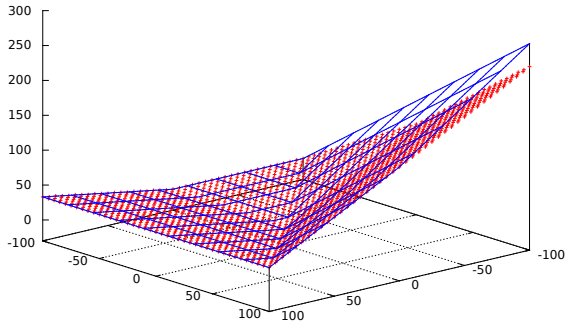


**Figure 11:** The automatically derived bound $1.33|[x,y]| + 0.33|[0,x]|$ (blue lines) and the measured runtime cost (red crosses) for Example *t08*. For $x \geqslant 0$ the bound is tight.

the table since all tested tools work very efficiently and need less then a second on every tested example.

Table 2 contains a compilation of the results of our experiments with the cBench benchmark suite. It shows a representative list of automatically derived function bounds. In total we analyzed more than 2900 lines of code. In the LoC column we not only count the lines of the analyzed function but also the ones of all the function it calls. The line count is computed on pre-processed C with all unnecessary declarations removed. We analyzed the functions by giving a cost 1 to all the back-edges in the control flow (loops, and function calls). The bounds for the functions ycc_rgb_conv and uv_decode have been inferred with user interaction as described in Section 6. The most challenging functions for $C^4B$ have unrolled loops where many variables are assigned. This stresses our analysis because the number of LP variables has a quadratic growth in program variables. Even on these stressful examples, the analysis could finish in less than 2 seconds. These times can be reduced by adding manual annotations to indicate $C^4B$ that certain variables do not have to be tracked. We could also envision to implement heuristics. For example, the sha_update function is composed of one loop calling two helper functions that in turn have 6 and 1 inner loops. In the analysis of the SHA algorithm, the compositionality of our analysis is essential to get a tight bound since loops on the same index are sequenced 4 and 2 times without resetting it. All other tools derive much larger constant factors.

With our formal cost semantics, we can run our examples for different inputs and measure the cost to compare it to our derived bound. Figure 11 shows such a comparison for Example *t08*, a variant of *t08a* from Section 3. 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.

## 9. Limitations

Our implementation does not currently support all of Clight. Programs with function pointers, goto statements, continue statements, and pointers to stack-allocated variables cannot be analyzed automatically. While these limitations concern the current implementation, our technique is in principle capable to handle them.

For the sake of simplicity, the automated system described here is restricted to finding only linear bounds. However, the amortized analysis technique was shown to work with polynomial bounds [Hoffmann et al. 2011]; we leave this extension of our system as future work.

Even certain linear programs cannot be analyzed automatically by $C^4B$, it is usually the case for programs that rely on heap invariants (like nul-terminated C strings), for programs in which resource usage depends on the result of non-linear operations (like $\%$ or $*$) in a non-trivial way, or for programs whose termination can only be proved by complex path-sensitive reasoning.

## 10. Related Work

Our work has been inspired by type-based amortized resource analysis for functional programs [Hofmann and Jost 2003; Hoffmann and Hofmann 2010; Hoffmann et al. 2012]. Here, we present the first automatic amortized resource analysis for C. None of the existing techniques can handle the example programs we describe in this work. The automatic analysis of realistic C programs is enabled by two major improvements over previous work. First, we extended the analysis system to associate potential with not just individual program variables but also multivariate intervals and, more generally, auxiliary variables. In this way, we solved the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on (possibly negative) integers without decreasing one individual number in each iteration. Second, for the first time, we have combined an automatic amortized analysis with a system for interactively deriving bounds. In particular, recent systems [Hoffmann and Shao 2014] that deal with integers and arrays cannot derive bounds that depend on values in mutable locations, possibly negative integers, or on differences between integers.

A recent project [Carbonneaux et al. 2014] has implemented and verified a quantitative logic to reason about stack-space usage, and modified the verified CompCert C compiler to translate C level bound to x86 stack bounds. This quantitative logic is also based on the potential method but has very rudimentary support for automation. It is not based on efficient LP solving and cannot automatically derive symbolic bounds. In contrast, our main contribution is an automatic amortized analysis for C that can derive parametric bounds for loops and recursive functions fully automatically. We use a more general quantitative Hoare logic that is parametric over the resource of interest.

In the development of our quantitative Hoare logic we have drawn inspiration from mechanically verified Hoare logics. Nipkow's [Nipkow 2002] 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 [Appel et al. 2013] 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 [Appel et al. 2013] but also in Hoare logics for low-level code [Ni and Shao 2006; Jensen et al. 2013].

There exist quantitative logics that are integrated into separation logic [Atkey 2010; Hoffmann et al. 2013] 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. [Aspinall et al. 2007]. 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.

There exist many tools that can automatically derive loop and recursion bounds for imperative programs such as SPEED [Gulwani et al. 2009b; Gulwani and Zuleger 2010], KoAT [Brockschmidt et al. 2014], PUBS [Albert et al. 2012a], Rank [Alias et al. 2010], ABC [Blanc et al. 2010] and LOOPUS [Zuleger et al. 2011; Sinn et al. 2014]. 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 [Alonso-Blas and Genaim 2012; Hofmann and Moser 2014; Sinn et al. 2014]. Currently, the only other available tools that can be directly applied to C code are Rank and LOOPUS. As demonstrated, $C^4B$ is more compositional than the aforementioned tools. Our technique, is the only one that can generate resource specifications for functions, deal with resources like memory that might become available, generate proof certificates for the bounds, and support user guidance that separates qualitative and quantitative reasoning.

There are techniques [Braberman et al. 2008] 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. [Albert et al. 2012b] 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. [Blazy et al. 2013] have verified a loop bound analysis for CompCert's RTL intermediate language. However, this automatic bound analysis does not compute symbolic bounds. Furthermore, there is no way to interactively derive bounds or to deal with resources like memory usage.

## 11. Conclusion

We have developed a novel analysis framework for compositional and certified worst-case resource bound analysis for C programs. The framework combines ideas from existing abstract interpretation–based techniques with the potential method of amortized analysis. It is implemented in the publicly available tool $C^4B$. To the best of our knowledge, $C^4B$ is the first tool for C programs that automatically reduces the derivation of symbolic bounds to LP solving.

We have demonstrated that our approach improves the state-of-the-art in resource bound analysis for C programs in three ways. First, our technique is naturally compositional, tracks size changes of variables, and can abstractly specify the resource cost of functions (Section 3). Second, it is easily combined with established qualitative verification to guide semi-automatic bound derivation (Section 6). Third, we have shown that the local inference rules of the derivation system automatically produce easily checkable certificates for the derived bounds (Section 7). Our system is the first amortized resource analysis for C programs. It addresses the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on signed integers and to deal with non-linear control flow that is introduced by break and return statements.

This work is the starting point for several projects that we plan to investigate in the future, such as the extension to concurrency, better integration of low-level features like memory caches, and the extension of the automatic analysis to multivariate resource polynomials [Hoffmann et al. 2011].

## A. 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.*** 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 or the stack through a pointer. 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 stack variables that have their address taken.

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 rational number $n$ as a flexible way to model resource consumption or release (if $n$ is negative).

***Semantics.*** Figure 12 contains the reduction rules of the semantics. The rules define a rewrite system for program configurations of the form $(\sigma, S, K, c)$, where $\sigma$ 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 amount of resources available for further execution. All the rules have the implicit side condition that the resource quantity available *before* the step is non-negative. This means that we allow $(\sigma, S, K, c)$ with $c < 0$ on the right-hand side transition relation $\to^*$ 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 $\Sigma$ 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 $(\sigma, S, K, c) \to^* (\sigma', S', K', c')$ is the following. If the statement $S$ is executed in program state $\sigma$, with continuation $K$, and with $c$ resources available then—after a finite number of steps—the evaluation will reach the new machine state $(\sigma', S', K', c')$ 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 2 summarizes the main properties of the resource counter of the semantics. It can be proved by induction on the number of steps.

**Lemma 2.** *1. If $(\sigma, S, K, c) \to^k (\sigma', S', K', c')$ and $n \geqslant 0$ then $(\sigma, S, K, c{+}n) \to^k (\sigma', S', K', c'{+}n)$.*
*2. If $(\sigma, S, K, c) \to^k (\sigma', S', K', c')$ and $(\sigma, S, K, d) \to^k (\sigma', S', K', d')$ then $c - c' = d - d'$.*

## B. 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 $P : \mathit{State} \to \mathit{bool}$ of Hoare logic we use assertions

$$P : \mathit{State} \to \mathbb{Q} \cup \{\infty\}\ .$$

This can be understood as a refinement of boolean assertions where *false* is $\infty$ and *true* is refined by $\mathbb{Q}_0^+$. We write $\mathit{Assn}$ for $\mathit{State} \to \mathbb{Q} \cup \{\infty\}$ and $\bot$ for $\lambda\sigma.\,\infty$. We sometimes call assertions *potential functions*. To use Coq's support for propositional reasoning, assertions have the type $\mathit{State} \to \mathbb{Q} \to \mathrm{Prop}$ in the implementation. For a given $\sigma \in \mathit{State}$, such an assertion can be seen as a set $B \subseteq \mathbb{Q}$. 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 logic, our quantitative Hoare triples have the form

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

The triple $\{Q\}\, S\, \{Q'\}$ consists of a statement $S$ and two assertions $Q, Q' : \mathit{Assn}$. It corresponds to triples in classic Hoare logic and the intuitive meaning is as follows. If $S$ is executed with starting state $\sigma$, with the empty continuation Kstop, $Q' \geqslant 0$, and at least $Q(\sigma)$ 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 $\sigma'$.

$$\frac{\mathsf{istrue}\ [\![e]\!]_\sigma}{(\sigma, \mathsf{assert}\ e, K, c) \to (\sigma, \mathsf{skip}, K, c)}\ (\text{S:ASSERT}) \qquad \frac{}{(\sigma, \mathsf{break}, \mathsf{Kseq}\ S\ K, c) \to (\sigma, \mathsf{break}, K, c)}\ (\text{S:BRKSEQ})$$

$$\frac{}{(\sigma, \mathsf{break}, \mathsf{Kloop}\ S\ K, c) \to (\sigma, \mathsf{skip}, K, c)}\ (\text{S:BRKLOOP}) \qquad \frac{}{(\sigma, \mathsf{return}\ x, \mathsf{Kseq}\ S\ K, c) \to (\sigma, \mathsf{return}\ x, K, c)}\ (\text{S:RETSEQ})$$

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

$$\frac{\sigma = (\_, \gamma) \qquad \sigma' = (\theta, \gamma)[r \mapsto \sigma(x)]}{(\sigma, \mathsf{return}\ x, \mathsf{Kcall}\ r\ \theta\ K, c) \to (\sigma', \mathsf{skip}, K, c)}\ (\text{S:RETCALL}) \qquad \frac{\sigma' = \sigma[x \mapsto [\![e]\!]_\sigma]}{(\sigma, x \leftarrow e, K, c) \to (\sigma', \mathsf{skip}, K, c))}\ (\text{S:UPDATE})$$

$$\frac{\Sigma f = (\vec{x}, S_f) \qquad \sigma = (\theta, \gamma) \qquad \sigma' = (\vec{x} \mapsto \sigma(\vec{y}), \gamma)}{(\sigma, r \leftarrow f(\vec{y}), K, c) \to (\sigma', S_f, \mathsf{Kcall}\ r\ \theta\ K, c)}\ (\text{S:CALL}) \qquad \frac{}{(\sigma, \mathsf{loop}\ S, K, c) \to (\sigma, S, \mathsf{Kloop}\ S\ K, c)}\ (\text{S:LOOP})$$

$$\frac{}{(\sigma, \mathsf{skip}, \mathsf{Kloop}\ S\ K, c) \to (\sigma, \mathsf{loop}\ S, K, c)}\ (\text{S:SKIPLOOP}) \qquad \frac{\mathsf{istrue}\ [\![e]\!]_\sigma}{(\sigma, \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2, K, c) \to (\sigma, S_1, K, c)}\ (\text{S:IFTRUE})$$

$$\frac{\mathsf{isfalse}\ [\![e]\!]_\sigma}{(\sigma, \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2, K, c) \to (\sigma, S_2, K, c)}\ (\text{S:IFFALSE}) \qquad \frac{}{(\sigma, S_1; S_2, K, c) \to (\sigma, S_1, \mathsf{Kseq}\ S_2\ K, c)}\ (\text{S:SEQ})$$

$$\frac{}{(\sigma, \mathsf{skip}, \mathsf{Kseq}\ S\ K, c) \to (\sigma, S, K, c)}\ (\text{S:SKIPSEQ}) \qquad \frac{}{(\sigma, \mathsf{tick}(n), K, c) \to (\sigma, \mathsf{skip}, K, c - n)}\ (\text{S:TICK})$$

**Figure 12:** Rules of the operational semantics of statements.

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 $\sigma'$ with a break then $B(\sigma')$ resources are available. Similarly, $R : \mathbb{Z} \to Assn$ 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 $\Delta$ 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 $P \geqslant Q$ to if for all program state $\sigma$, $P(\sigma) \geqslant Q(\sigma)$.

***Rules of the Quantitative Logic.*** Figure 13 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.

The L:TICK rule accounts for the cost $n$ of the tick statement. This rule is the only one to account directly for a cost on in the assertions because calling the tick function is the only way to change the amount of resource available. The rule also has a side condition that ensures that the potential remains positive during all the program execution. It is an essential semantic property of this system.

In the rule L:ASSERT, we use the notation $\mathsf{istrue}\ [\![e]\!]_\sigma \implies Q$ to express that we require potential $Q$ 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 any non-negative number 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 $B$ in the precondition to pay for the potential after

$$\frac{}{\Delta; B; R \vdash_L \{Q\}\,\mathsf{skip}\,\{Q\}}\,(\text{L:SKIP}) \qquad \frac{}{\Delta; B; R \vdash_L \{B\}\,\mathsf{break}\,\{Q\}}\,(\text{L:BREAK})$$

$$\frac{n < 0 \implies Q \geqslant 0}{\Delta; B; R \vdash_L \{Q\}\,\mathsf{tick}(n)\,\{Q - n\}}\,(\text{L:TICK}) \qquad \frac{}{\Delta; B; R \vdash_L \{R\,(\sigma(x))\}\,\mathsf{return}\;x\,\{Q\}}\,(\text{L:RETURN})$$

$$\frac{}{\Delta; B; R \vdash_L \{\mathsf{istrue}\;[\![e]\!]_\sigma \implies Q\}\,\mathsf{assert}\;e\,\{Q\}}\,(\text{L:ASSERT}) \qquad \frac{}{\Delta; B; R \vdash_L \{\lambda\sigma.\,Q\,\sigma[x \mapsto [\![e]\!]_\sigma]\}\,x \leftarrow e\,\{Q\}}\,(\text{L:UPDATE})$$

$$\frac{I \geqslant Q \qquad \Delta; Q; R \vdash_L \{I\}\,S\,\{I\}}{\Delta; B; R \vdash_L \{I\}\,\mathsf{loop}\;S\,\{Q\}}\,(\text{L:LOOP}) \qquad \frac{\Delta; B; R \vdash_L \{P\}\,S_1\,\{Q'\} \quad \Delta; B; R \vdash_L \{Q'\}\,S_2\,\{Q\}}{\Delta; B; R \vdash_L \{P\}\,S_1; S_2\,\{Q\}}\,(\text{L:SEQ})$$

$$\frac{\Delta; B; R \vdash_L \{\mathsf{istrue}\;[\![e]\!]_\sigma + P\}\,S_1\,\{Q\} \qquad \Delta; B; R \vdash_L \{\mathsf{isfalse}\;[\![e]\!]_\sigma + P\}\,S_2\,\{Q\}}{\Delta; B; R \vdash_L \{P\}\,\mathsf{if}(e)\;S_1\;\mathsf{else}\;S_2\,\{Q\}}\,(\text{L:IF})$$

$$\frac{\Delta(f) = \forall z\,\vec{v}\,v.(P_f\;z\,\vec{v}, Q_f\;z\,v) \qquad P \geqslant P_f\;y\,(\sigma(\vec{x})) + A \qquad \forall v.\,(Q_f\;y\,v + A \geqslant \lambda\sigma.\,Q\,\sigma[r \mapsto v])}{\Delta; B; R \vdash_L \{P\}\,r \leftarrow f(\vec{x})\,\{Q\}}\,(\text{L:CALL})$$

$$\frac{\Delta \cup \Delta'; B; R \vdash_L \{P\}\,S\,\{Q\} \qquad P_f \geqslant 0 \qquad \forall f\;P_f\;Q_f.\,\Delta'(f) = \forall z\,\vec{v}\,v.(P_f\;z\,\vec{v}, Q_f\;z\,v) \to \forall y\,\vec{v}.\,(\Delta \cup \Delta'; \bot; Q_f\;y \vdash_L \{P_f\;y\,\vec{v}\}\,S_f\,\{\bot\})}{\Delta; B; R \vdash_L \{P\}\,S\,\{Q\}}\,(\text{L:EXTEND})$$

$$\frac{P \geqslant P' \quad \Delta; B'; R' \vdash_L \{P'\}\,S\,\{Q'\} \quad Q' \geqslant Q \quad B' \geqslant B \quad \forall v.\,(R'\,v \geqslant R\,v)}{\Delta; B; R \vdash_L \{P\}\,S\,\{Q\}}\,(\text{L:WEAKEN}) \qquad \frac{\Delta; B; R \vdash_L \{P\}\,S\,\{Q\} \qquad x \in \mathbb{Q}_0^+}{\Delta; B + x; R + x \vdash_L \{P + x\}\,S\,\{Q + x\}}\,(\text{L:FRAME})$$

**Figure 13:** Rules of the Quantitative Hoare Logic

the break. In L:RETURN we only require to have potential $R$ in the precondition to pay for the potential after the return.

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.

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 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 $S$.

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.

The pre- and postcondition $P_f$ and $Q_f$ used in the L:CALL rule are taken from the function context $\Delta$. 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 $\geqslant$. 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* [Kleymann 1999]. 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 $\mathsf{double}()$ 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 a 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 $\Delta$. 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 $\Delta$ 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 14 we derive a precise runtime bound on a program that searches a maximal element in an array. Because of the location of the tick statements, the resource cost is closely related to the number of times the test $\mathsf{a[i]} > \mathsf{m}$ 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 $\mathsf{a}$ seen

```
{A(0) + N}
i=1; tick(1); m=a[0]; tick(1);
while (i < N) {
  {(m = max_{k∈[0,i−1]} a[k]) + A(i) + (N − i)}
  if (a[i] > m)
    m=a[i], tick(1);
  {(m = max_{k∈[0,i]} a[k]) + A(i + 1) + (N − i)}
  i=i+1;
  {(m = max_{k∈[0,i−1]} a[k]) + A(i) + (N − i) + 1}
  tick(1);
  {(m = max_{k∈[0,i−1]} a[k]) + A(i) + (N − i)}
} {0}
```

**Figure 14:** Example derivation where we wrote $A(i)$ for $\#\{k \mid i \leqslant k \leqslant N \wedge \forall\, 0 \leqslant j < k.\, a[j] < a[k]\}$, we use ticks in this example to count the number of assignments performed by the program.

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 $\mathsf{a[i]} > \mathsf{m}$ is true, we know that we encountered a "maximum so far" because $m$ is a maximal element of $\mathsf{a[0 \ldots i]}$, thus $A(i) = 1 + A(i+1)$ and we get one potential unit to pay for the tick in the conditional. In the other case, no maximum so far is encountered so $A(i) = A(i+1)$. As a side remark, if $K \geqslant 0$, it is possible to have $\{A(0) + N + K\}$ and $\{K\}$ as pre- and postcondition of the same program. It can be done either by adapting the proof or by applying the L:FRAME rule one top of the triple already derived. More generally, without function calls, the L:FRAME rule is admissible in our system.

The example in Figure 15 shows a use case for logical variables. On this example, we model the stack consumption: we account a constant cost for a function call ($M_f > 0$) that is returned after the call ($M_r = -M_f < 0$). We are interested in showing that a binary search function $\mathsf{bsearch}$ 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 $\Delta$ 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 $Z-1$. The rest of the proof is here to make sure that the first tick can be paid for using regular algebra on the $\log_2$ function.

***Soundness of Quantitative Hoare Triples.*** We already gave an intuition of the meaning of judgements derived in the logic. We start by defining an indexed predicate $\cdot \downarrow_n$

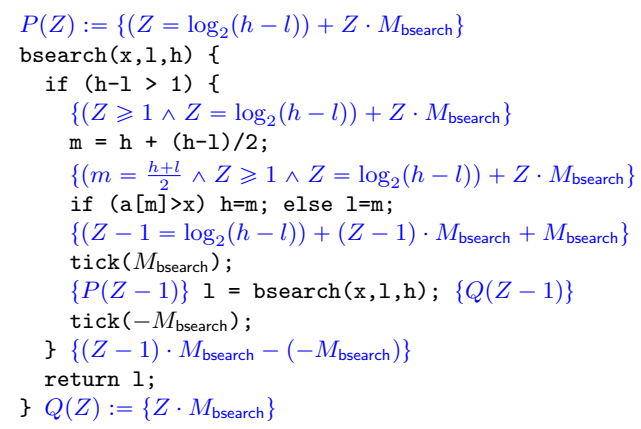


**Figure 15:** Example derivation of a stack usage bound for a binary search program. The stack usage is modeled by adding a cost $M_{\text{bsearch}}$ before the function call and $-M_{\text{bsearch}}$ after the call. $M_{\text{bsearch}}$ is the stack frame size of the function bsearch. $Z$ is a logical variable.

that defines a *resource-safe* program configuration:

$$(\sigma, S, K, c)\downarrow_n := \forall m \leqslant n.\, \forall c'.\, (\sigma, S, K, c) \to^m (\_, \_, \_, c') \implies 0 \leqslant c'.$$

This predicate means that a program configuration can execute for a certain number of steps without running out of resources. Note that nothing is said about the program safety, i.e. a stuck configuration $C$ will satisfy $\forall n.\, C\downarrow_n$. This is because our logic does not prove any safety or correctness theorems but only focuses on resource usage. We can now be used to define the resource safety of a configuration with respect to a quantitative assertion.

$$\mathit{safe}(n, P, S, K) := \forall \sigma\, c.\, P(\sigma) \leqslant c \implies 0 \leqslant c \wedge (\sigma, S, K, c)\downarrow_n .$$

Once again, we use an index in the definition of the predicate, it is used to perform proofs by induction for the function-call and loop rules of the logic. An interesting and essential property of the previous predicate is that $\mathit{safe}(n+1, P, S, K) \implies \mathit{safe}(n, P, S, K)$. To prove this weakening property, we use the quantification over all numbers smaller than $n$ in the definition of $\cdot\downarrow_n$.

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.

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

We can now define the *semantic validity* of a judgement $B; R \vdash_L \{P\}\, S\, \{Q\}$ of the quantitative logic without function context as $\mathit{valid}(n, B, R, P, S, Q) :=$

$$\forall m \leqslant n.\, \forall x \geqslant 0.\, \forall K.\ \mathit{safeK}(m, B+x, R+x, Q+x, K) \implies \mathit{safe}(m, P+x, S, K).$$

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 $\Delta$, written $\mathit{validC}(n, \Delta)$, as

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

where $S_f$ is the body of the function $f$. A full judgement that mentions a non-empty function context $\Delta$ 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 2** (Soundness)**.** *If* $\Delta; B; R \vdash_L \{P\}\, S\, \{Q\}$ *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 $\Delta$. 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, $\Delta'$ 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 $\delta$ to be exactly one in the theorem statement.

Assume that $S$ is a complete program and $\Delta$ is empty. By expanding the definitions we see that $\Delta$ is valid for every $n$ and that Kstop is safe for every $n$ if both $Q$, $R$, and $B$ are non-negative on all possible program states. So the Theorem 2 becomes

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

This means that from any starting state $\sigma$, $P(\sigma)$ 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.

## C. 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, the examples are automatically instrumented with tick statements to count the number of loop iterations and function calls. This is the only resource that can be measured by tools like KoAT [Brockschmidt et al. 2014] and LOOPUS [Sinn et al. 2014]. Sometimes we use tick statements explicitely to discuss features such as resource restitution.

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 3 contains the details of our comparative evaluation.

***Amortization and Compositionality*** Figures 16, 17, and 18 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 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 $\mathsf{assert}(\mathsf{y} >= 0)$. The semantic of $\mathsf{assert}$ 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 $\mathsf{x} = \mathsf{x} - 1$. 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 *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 $\mathsf{z} = \mathsf{y}$, we assign $\mathsf{z} = 2\mathsf{y} + 100$. 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 *t09* shows how amortized reasoning handles periodically expensive operations. The top-level loop is iterated $x$ times in a regular fashion, but every 4 iterations of this loop an expensive operation modeled by $\mathsf{tick}(40)$ is performed. Taking the worst case of the loop body would yield the pessimistic bound $41|[0,x]|$, however our tool is able to amortize the expensive operation and derives instead the tight bound $11|[0,x]|$.

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 $50 + |[-1,i]| + |[0,k]| + |[100,i]|$.

```
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 16:** Amortization and Compositionality (a).

```
while (x>0) {
  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**

```
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**

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

$101|[y,x]|$

**t16**

```
i=1;
j=0;
while (j<x) {
  j++;
  if (i>=4) {
    i=1;
    tick(40);
  } else
    i++;
  tick(1);
}
```

$11|[0,x]|$

**t09**

**Figure 17:** 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 18:** Amortization and Compositionality (c).

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 [Gulwani et al. 2009a] requires *path-sensitive reasoning* to derive a bound.

Examples *fig2_1* and *fig2_2* are taken from Gulwani et al [Gulwani et al. 2009b]. 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 [Gulwani et al. 2009a] 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 [Gulwani and Zuleger 2010] specifically focuses on the code in the *if* statement. So we insert $\mathsf{tick}(1)$ 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 [Gulwani and Zuleger 2010] focuses on the number of iterations of the *outer loop*. While the whole program is not terminating, the

```
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 19:** Examples from Gulwani et al's SPEED [Gulwani et al. 2009b] (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 20:** Examples from Gulwani et al's SPEED [Gulwani et al. 2009b] (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 21:** Examples from [Gulwani et al. 2009a].

```
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 22:** Examples from [Gulwani and Zuleger 2010].

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 inter-procedural 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 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 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 24 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 n + lo of the pivot. We only assert that lo < n + lo <= hi. We then derive a tight linear bound.

```
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 start (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 start (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 start (int y, int z) {
  consume(); produce(); consume();
}
```

$|[0, y]|$ ticks

**t46**

**Figure 23:** 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 */
  assert( n > 0 );
  assert( lo + n <= hi );

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

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

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

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

**Quick Sort**

**Figure 24:** Well-Known Algorithms

| File | KoAT | | Rank | | LOOPUS | | SPEED | $C^4B$ |
|---|---|---|---|---|---|---|---|---|
| gcd.c | ? | | $(((2+1)\dots$ | $O(n)$ | — | | ? | $\lvert[0,x]\rvert+\lvert[0,y]\rvert$ |
| kmp.c | ? | | $(((2+(n+\dots$ | $O(n^2)$ | $\mathrm{mx}(n,0)\dots$ | $O(n)$ | ? | $1+2\lvert[0,n]\rvert$ |
| qsort.c | ? | | — | | — | | ? | $1+2\lvert[0,\mathsf{len}]\rvert$ |
| speed_pldi09 fig4_2.c | — | | $(((2+n)\dots$ | $O(n)$ | — | | $\frac{n}{m}+n$ | $1+2\lvert[0,n]\rvert$ |
| speed_pldi09 fig4_4.c | — | | $(((2+(-1\dots$ | $O(n)$ | — | | $\frac{n}{m}+m$ | $\lvert[0,n]\rvert$ |
| speed_pldi09 fig4_5.c | $28d+7g+27$ | $O(n)$ | $(((2+(-1\dots$ | $O(n)$ | — | | $\mathrm{mx}(n,n-m)$ | — |
| speed_pldi10 ex1.c | — | | — | | — | | $n$ | $\lvert[0,n]\rvert$ |
| speed_pldi10 ex3.c | — | | $(((2+(-1\dots$ | $O(n)$ | $2{\cdot}\mathrm{mx}(n,0)$ | $O(n)$ | $n$ | $\lvert[0,n]\rvert$ |
| speed_pldi10 ex4.c | $110a+33$ | $O(n)$ | — | | — | | $n+1$ | $1+2\lvert[0,n]\rvert$ |
| speed_popl10 fig2_1.c | $9a+9b+\dots$ | $O(n)$ | $((2+((-y\dots$ | $O(n)$ | $\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$ | $O(n)$ | $\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$ | $\lvert[x,n]\rvert+\lvert[y,m]\rvert$ |
| speed_popl10 fig2_2.c | $6a{+}9b{+}3c+5$ | $O(n)$ | $((2{-}x\dots$ | $O(n)$ | $\mathrm{mx}(0,(x+1{-}z)\dots$ | $O(n)$ | $\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,n{-}z)$ | $\lvert[x,n]\rvert+\lvert[z,n]\rvert$ |
| speed_popl10 nstd_multiple.c | — | | $((2{-}x{+}n\dots$ | $O(n^2)$ | $\mathrm{mx}(0,m{-}y)+\mathrm{mx}(0,n{-}x)$ | $O(n)$ | $\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$ | $\lvert[x,n]\rvert+\lvert[y,m]\rvert$ |
| speed_popl10 nstd_single.c | $48b+16$ | $O(n)$ | $(((1{-}x{+}n\dots$ | $O(n)$ | $\mathrm{mx}(0,n{-}1)\dots$ | $O(n)$ | $n$ | $\lvert[0,n]\rvert$ |
| speed_popl10 sqntl_single.c | $21b+6$ | $O(n)$ | $((2-x{+}n\dots$ | $O(n)$ | $2{\cdot}\mathrm{mx}(n,0)$ | $O(n)$ | $n$ | $\lvert[0,n]\rvert$ |
| speed_popl10 smpl_multiple.c | $9c+10d+7$ | $O(n)$ | $((2{-}y{+}m\dots$ | $O(n)$ | $\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$ | $O(n)$ | $n+m$ | $\lvert[0,m]\rvert+\lvert[0,n]\rvert$ |
| 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 | $4b+6$ | $O(n)$ | $((2{-}x{+}n\dots$ | $O(n)$ | $\mathrm{mx}(n,0)$ | $O(n)$ | $n$ | $\lvert[0,n]\rvert$ |
| t07.c | ? | | $2+x$ | $O(n)$ | $\mathrm{mx}(x,0)\dots$ | $O(n)$ | ? | $1+3\lvert[0,x]\rvert+\lvert[0,y]\rvert$ |
| t08.c | ? | | $((2{+}z{-}y\dots$ | $O(n)$ | $\mathrm{mx}(0,y{-}2)\dots$ | $O(n)$ | ? | $1.33\lvert[y,z]\rvert+0.33\lvert[0,y]\rvert$ |
| t10.c | ? | | $((2{-}y{+}x\dots$ | $O(n)$ | $\mathrm{mx}(0,x{-}y)$ | $O(n)$ | ? | $\lvert[y,x]\rvert$ |
| t11.c | ? | | $((2{-}y{+}m\dots$ | $O(n)$ | $\mathrm{mx}(0,n{-}x))+\mathrm{mx}(0,m{-}y)$ | $O(n)$ | ? | $\lvert[x,n]\rvert+\lvert[y,m]\rvert$ |
| t13.c | ? | | $(((1{+}y^2/2\dots$ | $O(n^2)$ | $2{\cdot}\mathrm{mx}(x,0)+\mathrm{mx}(y,0)$ | $O(n)$ | ? | $2\lvert[0,x]\rvert+\lvert[0,y]\rvert$ |
| t15.c | ? | | $((1{+}x\dots$ | $O(n)$ | — | | ? | $\lvert[0,x]\rvert$ |
| t16.c | ? | | $((-99{\cdot}y\dots$ | $O(n)$ | — | | ? | $101\lvert[0,x]\rvert$ |
| t19.c | ? | | $((153{+}k\dots$ | $O(n)$ | $\mathrm{mx}(0,i{-}10^2)+\mathrm{mx}(0,k{+}i{+}51)$ | $O(n)$ | ? | $50{+}\lvert[-1,i]\rvert+\lvert[0,k]\rvert$ |
| t20.c | ? | | $(2{-}y{+}x\dots$ | $O(n)$ | $2{\cdot}\mathrm{mx}(0,y{-}x)+\mathrm{mx}(0,x{-}y)$ | $O(n)$ | ? | $\lvert[x,y]\rvert+\lvert[y,x]\rvert$ |
| t27.c | ? | | — | | $10^3\mathrm{mx}(0,-n)\dots$ | $O(n)$ | ? | $0.01\lvert[n,y]\rvert+11\lvert[n,0]\rvert$ |
| t28.c | ? | | $((1{-}y{+}x\dots$ | $O(n)$ | $10^3\,\mathrm{mx}(0,x-y)\dots$ | $O(n)$ | ? | $\lvert[x,0]\rvert+\lvert[0,y]\rvert+1002\lvert[y,x]\rvert$ |
| 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 3:** Comparison of the bounds generated by KoAT, Rank, LOOPUS, SPEED, and our tool $C^4B$ 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. We write $\mathrm{mx}(a,b)$ for the maximum of $a$ and $b$.

## References


E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012a.

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, 2012b.

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.

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.

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

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.

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

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.

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.

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.

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.

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.

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.

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.

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.

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

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.

COIN-OR Project. CLP (Coin-or Linear Programming). `https://projects.coin-or.org/Clp`, 2014. Accessed: 2014-11-12.

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

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, 2009a.

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, 2009b.

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

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.

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

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

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.

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.

M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

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

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.

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.

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

X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

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.

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

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

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

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

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.