# SPEED: Precise and Efficient Static Estimation of Program Computational Complexity

## (Full Version)

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Krishna K. Mehra
Microsoft Research India
kmehra@microsoft.com

Trishul Chilimbi
Microsoft Research
trishulc@microsoft.com

## Abstract

This paper describes an inter-procedural technique for computing symbolic bounds on the number of statements a procedure executes in terms of its scalar inputs and user-defined quantitative functions of input data-structures. Such computational complexity bounds for even simple programs are usually disjunctive, non-linear, and involve numerical properties of heaps. We address the challenges of generating these bounds using two novel ideas.

We introduce a proof methodology based on multiple counter instrumentation (each counter can be initialized and incremented at potentially multiple program locations) that allows a given linear invariant generation tool to compute linear bounds individually on these counter variables. The bounds on these counters are then composed together to generate total bounds that are non-linear and disjunctive. We also give an algorithm for automating this proof methodology. Our algorithm generates complexity bounds that are usually precise not only in terms of the computational complexity, but also in terms of the constant factors.

Next, we introduce the notion of user-defined quantitative functions that can be associated with abstract data-structures, e.g., length of a list, height of a tree, etc. We show how to compute bounds in terms of these quantitative functions using a linear invariant generation tool that has support for handling uninterpreted functions. We show application of this methodology to commonly used data-structures (namely lists, list of lists, trees, bit-vectors) using examples from Microsoft product code. We observe that a few quantitative functions for each data-structure are usually sufficient to allow generation of symbolic complexity bounds of a variety of loops that iterate over these data-structures, and that it is straightforward to define these quantitative functions.

The combination of these techniques enables generation of precise computational complexity bounds for real-world examples (drawn from Microsoft product code and C++ STL library code) for some of which it is non-trivial to even prove termination. Such automatically generated bounds are very useful for early detection of egregious performance problems in large modular codebases that are constantly being changed by multiple developers who make heavy use of code written by others without a good understanding of their implementation complexity.



*POPL'09,* January 18–24, 2009, Savannah, Georgia, USA.


***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

***General Terms*** Performance, Verification

***Keywords*** Symbolic Complexity Bounds, Termination Analysis, Counter Instrumentation, Quantitative Functions

## 1. Introduction

Modern software development has embraced modular design and data abstraction. While this increases programmer productivity by enabling code reuse, it potentially creates additional performance problems. Examples include hidden algorithmic complexity where a linear-time algorithm encapsulated inside a simple API call gives rise to quadratic complexity, when embedded inside an O(n) loop. Software performance testing attempts to address these issues but faces two fundamental limitations–it is often too little or too late. First, due to resource constraints a program is typically tested on only a small subset of its inputs and the performance problem may not manifest on these inputs. Second, these performance tests are time consuming and are typically only run periodically for large software projects. Consequently, many performance problems show up very late in the software development process when it is hard to redesign/re-architect the system to correctly fix the problem, or even worse, after software ships.

In this paper, we present a static analysis that can compute symbolic complexity bounds for procedures in terms of their inputs. Even though this information does not mirror the real running time of programs (which also depends on low-level architectural details like caches and pipelines), it can be used to provide useful insights into how a module performs as a function of its inputs at an abstract level, and can produce early warnings about potential performance issues. The same analysis can also be used for bounding other kinds of resources (e.g., memory) consumed by a procedure.

The hard part in computing complexity bounds is to bound the total number of loop iterations (or recursive procedure call invocations). There are 4 challenges in computing these bounds.

1. Even for simple arithmetic programs, these bounds are **disjunctive** (i.e., they involve use of `Max` operator, as in Example `Dis1` in Figure 2 and Example `NestedMultiple` in Figure 4, for both of which the bound is $\texttt{Max}(0, n - x_0)$ + $\texttt{Max}(0, m - y_0)$) and **non-linear** (Examples `SimpleMultipleDep` and `NestedMultipleDep` in Figure 4, for both of which the bound is $n \times (1 + m)$, assuming $n, m \geq 0$). This usually happens in presence of control flow inside loops.

2. For complicated programs (Example `Equals` in Figure 1, Example `Dis2` in Figure 2, Example 4 in Figure 6, Example `Traverse` in Figure 9), it is **hard to even prove termination**, and computing bounds ought to be an even harder task.
3. It is desirable to compute **precise bounds** (in terms of both the computational complexity as well as the constant factors) as opposed to computing any bound. Consider Example `SimpleSingle2` in Figure 3 and Examples `SimpleMultiple` and `SimpleMultipleDep` in Figure 4. The termination argument for each of these examples (as provided by termination techniques based on disjunctively well-founded relations [26, 6, 1]) is the same: between any two successive (not necessarily consecutive) loop iterations either $x$ increases and is bounded above by $n$, or $y$ increases and is bounded above by $m$. This implies a total bound of $(n+1) \times (m+1)$ (along with the observation that both $x$ and $y$ start out with a value of 0 and assuming $n, m \geq 0$) for each of these examples. In contrast, our technique can produce precise bounds of $\texttt{Max}(n, m)$, $n + m$, and $n \times (1 + m)$ respectively for these examples.
4. For loops that iterate over **data-structures**, expressing bounds requires use of some numerical functions over data-structures (e.g., length of a list, height of a tree). and computing those bounds may require some sophisticated shape analysis, which is a difficult task.

We address the first two challenges by using a counter instrumentation methodology (Section 3), wherein multiple counters are introduced at different cut-points (back-edges and recursive call sites) and are incremented and initialized at multiple locations (as opposed to simply using a single counter for all cut-points). These counters are such that the individual bound on each counter is linear (and hence can be computed using a linear invariant generation tool). Furthermore, these individual bounds can be composed together to yield a total bound, which may be disjunctive or non-linear (Section 3.1). We describe an algorithm that produces such a counter instrumentation scheme (Section 4).

The third challenge is addressed by ensuring that our algorithm produces a *counter-optimal* instrumentation scheme, which uses an optimal number of counters with an optimal number of dependencies between them – a criterion that we illustrate usually leads to precise estimation of bounds (Section 3.2).

We address the final challenge for abstract data-structures [1] by introducing the notion of user-definable *quantitative functions*, each of which is a numeric function over some tuple of data-structures. (Whenever we use the term data-structure, we mean abstract data-structure.) The user specifies the semantics of these functions by means of annotating each data-structure method with how it may update some quantitative attribute of any object. Given such a specification, we show how to use a linear invariant generation tool with support for uninterpreted functions to generate linear bounds on counter variables in terms of quantitative functions of input data-structures. We show application of this methodology to commonly used data-structures (namely lists, trees, lists of lists, bit-vectors) from Microsoft product code, wherein we observe that usually a few quantitative functions are sufficient to allow computation of precise loop bounds, and that it is straight-forward to define these quantitative functions.

We have implemented these ideas inside a tool called SPEED that computes precise symbolic bounds for several real-life examples drawn from Microsoft product code as well as C++ Standard Template Library (STL) code. For some of these examples, even proving termination is non-trivial. Note that our technique for estimating computational complexity does not assume program termination. Instead, existence of an upper bound on the number of loop iterations provides a free termination argument and may even yield a simpler and more efficient alternative to termination strategies pursued in [1, 6], which rely on synthesizing ranking functions for loops [26, 5, 4].

We start with a brief description of our overall methodology along with some examples in Section 2.

[1] those that are referenced and updated through a well-defined interface

## 2. Overall Methodology

The basic idea of our methodology is to instrument monitor variables (henceforth referred to as counter variables) to count the number of loop iterations and then statically compute a bound on these counter variables in terms of program inputs using an invariant generation tool. In principle, given a powerful invariant generation oracle, it is sufficient to instrument a single counter variable (which is initialized to 0 at the beginning of the procedure and is incremented by 1 at each back-edge) and then use the invariant generation oracle to compute bounds on the counter variable. However, even for a simple program, such an approach would require the invariant generation tool to be able to compute invariants that are disjunctive, non-linear, and that can characterize sophisticated heap shapes. No such invariant generation tool exists, and even if it did, it would not be scalable.

We present a two-tiered approach to address the above challenges: (a) Introduce multiple counters, each of which may be initialized and incremented at multiple locations. This avoids the need for disjunctive and non-linear invariants. (b) Require the user to define some quantitative functions over abstract data-structures. This avoids the need for sophisticated shape analysis.

***Example 1*** Consider the `Equals` procedure (taken from Microsoft product code) for string buffer data-structure `StringBuffer` in Figure 1. A string buffer is implemented as a list of chunks. A chunk consists of a character array `str` whose total size is bounded above by a global variable `size` and its `len` field denotes the total number of valid characters in the array `str`.

It is non-trivial to prove whether the outer loop in the `Equals` function even terminates. The only way to break out of the outer loop is when $\texttt{chunk1.str}[i1] \neq \texttt{chunk2.str}[i2]$ (Line 7) or when $(i1 < 0) \vee (i2 < 0)$ (Line 25). Since there is no information provided regarding the contents of `chunk1.str` and `chunk2.str`, we can restrict our attention to tracking $(i1 < 0) \vee (i2 < 0)$. The value of both $i1$ and $i2$ decreases in each iteration of the outer loop as well as in the first inner loop; hence if this were the only update to $i1$ and $i2$, the outer loop would terminate. However, note that the value of $i1$ and $i2$ may increase in the second and third inner loops (at Lines 15 and 22 respectively). Instead, consider the following (counter-intuitive) proof argument:

- The total number of times the second inner loop at Line 11 executes (or, more precisely, the back-edge $(17, 11)$ is taken) is bounded above by the length of list $s1$, denoted by $\texttt{Length}(s1)$. (This is because everytime the back-edge $(17, 11)$ is taken, `chunk1` advances forward over the list $s1$.) This information can be obtained by computing a bound on counter variable $c_1$ that has been instrumented in the procedure to count the total number of times the back-edge $(17, 11)$ is taken.
- Similarly, the total number of times the third inner loop at Line 18 executes (i.e., the back-edge $(24, 18)$ is taken) is bounded above by the length of list $s2$, denoted by $\texttt{Length}(s2)$, and this information can be obtained by computing a bound on counter $c_2$.
- The number of times the first inner loop (at Line 6) as well as the outer loop (at Line 5) executes for each iteration of the second inner loop or third inner loop is bounded above by `size`.

```
  int size;
  Equals(StringBuffer s1, StringBuffer s2) {
1   c1 := 0; c2 := 0; c3 := 0;
2   chunk1 := s1.GetHead(); chunk2 := s2.GetHead();
3   Assume(0 ≤ chunk1.len, chunk2.len < size);
4   i1 := chunk1.len − 1; i2 := chunk2.len − 1;
5   for(; ;) {
6     while (i1 ≥ 0 ∧ i2 ≥ 0) {
7       if (chunk1.str[i1] ≠ chunk2.str[i2]) return 0;
8       i1 − −; i2 − −;
9       c3 := c3 + 1;
10    }
11    while (i1 < 0) {
12      chunk1 := s1.GetNext(chunk1);
13      if (chunk1 == null) break;
14      Assume(0 ≤ chunk1.len < size);
15      i1 := i1 + chunk1.len;
16      c1 := c1 + 1; c3 := 0;
17    }
18    while (i2 < 0) {
19      chunk2 := s2.GetNext(chunk2);
20      if (chunk2 == null) break;
21      Assume(0 ≤ chunk2.len < size);
22      i2 := i2 + chunk2.len;
23      c2 := c2 + 1; c3 := 0;
24    }
25    if (i1 < 0) return (i2 < 0);if (i2 < 0) return 0;
26    c3 := c3 + 1;
27  }
28  return 1;
29 }
```

**Figure 1.** The Equals method of String Buffer data-structure as implemented in one of Microsoft's product code. The bold instrumentation of counter variables $c_1, c_2, c_3$ is part of our proof methodology, which implies a bound of $\texttt{Length}(s1)+\texttt{Length}(s2)+\texttt{size}\times(1+\texttt{Length}(s1)+\texttt{Length}(s2))$ on the total number of loop iterations for this procedure.

This information can be obtained by computing a bound on counter variable $c_3$ that has been instrumented appropriately in the procedure to count the total number of iterations of the first inner loop as well as the outer loop in between any two iterations of the second or third inner loops. (Note $c_3$ is initialized to 0 whenever $c_1$ or $c_2$ is incremented).

- Hence, the total number of loop iterations (both inner and outer) is bounded above by $\texttt{Length}(s1)+\texttt{Length}(s2)+\texttt{size}\times(1+\texttt{Length}(s1)+\texttt{Length}(s2))$.

***Example 2*** Consider the example `Dis1` shown in Figure 2. It is easy to see that the loop terminates because the if-branch is executed inside the loop until $y < m$, followed by execution of the then-branch until $x < n$. However, computing a bound on the number of loop iterations is a bit subtle. We cannot simply say that the loop executes for $(m-y_0)+(n-x_0)$ iterations since $y_0$ may be greater than $m$ (which would make the expression $m-y_0$ negative) and/or $x_0$ may be greater than $n$ (which would make the expression $n-x_0$ negative). Instead consider the following proof argument.

- If the then-branch is ever executed, it is executed for at most $m-y_0$ iterations. The bound $m-y_0$ can be obtained by computing a bound on counter variable $c_1$ (*inside* the if-branch), which has been instrumented to count the number of iterations of the if-branch (or, equivalently, the number of times the back-edge $(5,3)$ is taken).

```
  Dis1(int x0, y0, n, m) {
1   c1 := c2 := 0;
2   x := x0; y := y0;
3   while (x < n)
4     if (y < m)
5       y := y + 1; c1++;
6     else
7       x := x + 1; c2++;
8 }
```

```
  Dis2(int x0, z0, n){
1   c1 := c2 := 0;
2   x := x0; z := z0;
3   while (x < n)
4     if (z > x)
5       x := x + 1; c1++;
6     else
7       z := z + 1; c2++;
8 }
```

**Figure 2.** These examples have disjunctive bounds (which require use of max operator). Example (a) is quite a simple example, yet its bounds are subtle: $\texttt{Max}(0,n-x_0)+\texttt{Max}(0,m-y_0)$. Example (b) is a more sophisticated example taken from [6], where it is used to motivate the technique of well-founded disjunctive ranking functions for proving termination. Its timing bound is $\texttt{Max}(0,n-x_0)+\texttt{Max}(0,n-z_0)$. The bold instrumentation of counter variables $c_1, c_2$ is part of our proof methodology that can can compute precise bounds for both examples with equal ease.

- If the else-branch is ever executed, it is executed for at most $n-x_0$ iterations. The bound $n-x_0$ can be found by computing a bound on counter variable $c_2$ (*inside* the then-branch), which has been instrumented to count the number of iterations of the then-branch (or, equivalently, the number of times the back-edge $(7,3)$ is taken).
- Hence, the total number of loop iterations is bounded by $\texttt{Max}(0,m-y_0)+\texttt{Max}(0,n-x_0)$.

Note that in this example, it is important to use different counters to count the number of iterations of each branch (or, equivalently, each back-edge). This is because if the same counter variable would have been used on both the back-edges, then a linear invariant generation tool would not be able to compute bounds on it, for the simple reason that the bound is disjunctive (i.e., it involves use of max operator). Also, consider the example `Dis2` in Figure 2, where the if-branch and then-branch alternate several times. This is a non-trivial example even from the viewpoint of proving termination. This example has been taken from [6], where it is used to motivate the technique of well-founded disjunctive ranking functions for proving termination. However, its timing bound can be estimated to be $\texttt{Max}(0,n-x_0)+\texttt{Max}(0,n-z_0)$ in a manner very similar to that of example `Dis1`, wherein we compute bounds on the counters $c_1$ and $c_2$ at back-edges $(5,3)$ and $(7,3)$ respectively, and then add these bounds after maxing them out with 0.

Our methodology for automating such proof arguments for computing timing bounds involves the following steps in the order mentioned.

1. Defining Quantitative/Numerical Attributes for data-structures. The user declares some numerical-valued functions over data-structures. For example, length of a list, or height of a tree. The semantics of these functions is defined by annotating each data-structure method with its effect on the numerical functions associated with relevant data-structures. For example, the delete method associated with a list reduces its length by 1. For purpose of communicating the semantics of these functions to an invariant generation tool (used in the next step), we instrument each invocation of a data-structure method with its effect on the quantitative functions as defined by the user. This allows for treating the quantitative function as uninterpreted functions, which eases up the task of an invariant generation tool. Section 5 describes this methodology in more detail along with a case-study of examples drawn from Microsoft product code.

2. Generating a proof structure. This corresponds to choosing a set of counter variables and for each counter variable selecting the locations to initialize it to 0 and the locations to increment it by 1 (e.g., the bold counter instrumentation for the examples in Figures 1 and 2). The counters are chosen such that the given invariant generation tool can compute bounds on the counter variables at appropriate locations in terms of the scalar inputs and quantitative functions of input data-structures. We use a linear invariant generation tool, with support for handling uninterpreted functions, to generate these bounds. Section 3 describes the notion of a proof structure in more detail and also introduces the notion of an counter-optimal proof structure to enable generation of precise timing bounds. Section 4 describes an algorithm to generate a counter-optimal proof structure.
3. Composing the bounds on counter variables to obtain the final desired bound. Theorem 1 (in Section 3) describes how to compose bounds on counters to obtain a bound on the number of loop iterations. Theorem 4 (in Section 6) describes how to obtain a bound on the total cost of a procedure, given any cost-metric that maps atomic program statements to some cost.

## 3. Proof Structure

Obtaining a proof structure involves choosing a set of fresh counter variables $S$ and, for each counter variable, deciding the locations (from among the back-edges or procedure entry point) to initialize it to 0 and deciding the back-edges to increment it by 1 along with the following constraints.

- Each back-edge $q$ should be instrumented with an increment to some counter variable (denoted by $M(q)$). [2]
- There should be no cyclic dependencies between counter variables. A counter variable $c_1$ is said to depend on another counter variable $c_2$ if $c_1$ is initialized to 0 at a back-edge where $c_2$ is incremented by 1.
- The invariant generation tool is able to provide a symbolic bound $B(q)$ at each back-edge $q$ on the counter variable $M(q)$ in terms of the inputs to the procedure.

We now formally define a proof structure.

DEFINITION 1 (Proof Structure for a procedure $P$). *Let $S$ be a set of counter variables and let $M$ be a function that maps each back-edge in $P$ to some counter variable from set $S$. Let $G$ be any DAG structure over $S \cup \{r\}$ with $r$ as the unique root node. Let $B$ be a function that maps each back-edge in $P$ to some symbolic bound over inputs of $P$. Then, the tuple $(S, M, G, B)$ is a proof-structure (with respect to a given invariant generation tool) if for all back-edges $q$ in procedure $P$, the given invariant generation tool can be used to establish bound $B(q)$ on counter variable $M(q)$ at $q$ in the procedure $\texttt{Instrument}(P, (S, M, G))$.*

DEFINITION 2 ($\texttt{Instrument}(P, (S, M, G))$). *Let $P$ be some given procedure. We define $\texttt{Instrument}(P, (S, M, G))$ to be the procedure obtained from $P$ by instrumenting it as follows:*

- *Each back-edge $q$ in $P$ is instrumented with an increment (by 1) to counter variable $M(q)$.*
- *Each back-edge $q$ in $P$ is instrumented with an initialization (to 0) of any counter variable $c'$ that is an immediate successor of $M(q)$ in $G$, i.e., $(M(q), c') \in G$.*
- *The procedure entry point is instrumented with an initialization (to 0) of any counter variable $c'$ that is an immediate successor of the root node $r$ in $G$, i.e., $(r, c') \in G$.*

EXAMPLE 1. *The following tuple $(S, M, G, B)$ is a proof structure for procedure `Equals` shown in Figure 1. This proof structure also corresponds to the textual argument for the complexity of procedure `Equals` in Section 2.*

$$\begin{aligned} S &= \{c_1, c_2, c_3\} \\ M &= \{(27,5) \mapsto c_3, (10,6) \mapsto c_3, \\ &\quad (17,11) \mapsto c_1, (24,18) \mapsto c_2\} \\ G &= \{(r,c_1), (r,c_2), (r,c_3), (c_1,c_3), (c_2,c_3)\} \\ B &= \{(27,5) \mapsto \texttt{size}, (10,6) \mapsto \texttt{size}, \\ &\quad (17,11) \mapsto \texttt{Length}(s1), (24,18) \mapsto \texttt{Length}(s2)\} \end{aligned}$$

*Above $(27, 5)$ denotes the back-edge from program location 27 to location 5 (and so on).*

*The instrumented procedure $\texttt{Instrument}(\texttt{Equals}, (S, M, G))$ corresponds to bold instrumentation of the counter variables $c_1$, $c_2$ and $c_3$ as shown in Figure 1.*

### 3.1 Computing Bounds from a Proof Structure

Theorem 1 below describes how to compute a bound on the total number of loop iterations in a procedure, given a proof structure. Later in Section 6, we show how to extend Theorem 1 to compute symbolic bounds on the total cost of a procedure, given any cost-metric that maps atomic program statements to some cost. Note that it is this process that introduces disjunctions and non-linearity in our computation of bounds on loop iterations.

THEOREM 1 (Bound on Loop Iterations). *Let $(S, M, G, B)$ be a proof structure for procedure $P$. Then, $U$ as defined below denotes an upper bound on the total number of iterations of all loops in procedure $P$.*

$$\begin{aligned} U &= \sum_{c \in S} \texttt{TotalBound}(c) \\ \texttt{TotalBound}(r) &= 0 \\ \texttt{TotalBound}(c) &= \texttt{Max}\Big(\{0\} \bigcup \{B(q) \mid M(q) = c\}\Big) \times \\ &\quad \left(1 + \sum_{(c',c) \in G} \texttt{TotalBound}(c')\right) \end{aligned}$$

PROOF: We claim that $\texttt{TotalBound}(c)$ denotes the sum of the number of traversals of all those back-edges $q$ such that $M(q) = c$. This can be proved by induction on the topological order of the DAG $G$.

The total number of loop iterations is given by the sum of the number of traversals of each back-edge. The result now follows from the fact that each back-edge is mapped to some counter □

EXAMPLE 2. *For the proof structure shown in Example 1, application of Theorem 1 yields: $\texttt{TotalBound}(c_1) = \texttt{Length}(s_1)$, $\texttt{TotalBound}(c_2) = \texttt{Length}(s_2)$, $\texttt{TotalBound}(c_3) = \texttt{size} \times (1 + \texttt{Length}(s_1) + \texttt{Length}(s_2))$. Hence, $U = \texttt{Length}(s_1) + \texttt{Length}(s_2) + \texttt{size} \times (1 + \texttt{Length}(s_1) + \texttt{Length}(s_2))$.*

### 3.2 Counter-Optimal Proof Structure

A procedure might have multiple proof structures. Some of these proof structures may be better than the others in terms of yielding a better bound on the total number of loop iterations (as defined

[2] The effectiveness of our proof methodology can be increased by expanding shared paths in a loop body into disjoint paths, each with its own back-edge. In essence, what we really require is to associate each acyclic path between two cut-points with a counter variable.

in Theorem 1). We can define an optimal proof structure to be one that yields a bound which is not larger than the bound provided by any other proof structure. However, there are several practical issues with this definition. It is not clear how to generate such an optimal proof structure, or even check whether a given proof structure is optimal. Even the simpler problem of comparing the bounds implied by two proof structures is hard because it would involve reasoning about multiplication and max operators (Note that the bounds on loop iterations given by Theorem 1 involve both multiplication and max operators). Instead we propose the notion of counter-optimal proof structure that is usually an optimal proof structure, and allows for efficient checking whether a given proof structure is counter-optimal or not. We also give an algorithm (in Section 4) to generate such a counter-optimal proof structure.

Our definition for counter-optimal proof structure is inspired by the following two observations that hold for almost all of the examples that we have come across in practice.

***Fewer counters usually lead to a better bound.*** Use of fewer counters, if possible, usually leads to generation of a better bound. This is illustrated by a variety of examples described in Figure 3. Each of these examples have two back-edges. Use of a single counter for the two back-edges in each of the first three examples produces a bound of $n$. On the other hand, use of multiple counters leads to bounds of $2n$ or $n^2$ depending on the relative ordering of the counters in the DAG $G$ (i.e., depending on where the counters are initialized). However, note that we cannot always use a single counter. The examples in Figure 4 illustrate some situations when multiple counters can be used to compute bounds, but a single counter is not sufficient.

***Lesser dependency between counters usually leads to a better bound.*** Multiple counters, when used, might need to have some dependency between them (in the DAG $G$). This is definitely the case when the total number of loop iterations are non-linear as is illustrated by the examples `SimpleMultipleDep` and `NestedMultipleDep` in Figure 4. However, in some case, multiple counters need not have a dependency between them. This is usually the case when the bounds are max-linear (i.e., a disjunction of linear bounds) as is illustrated by the examples `SimpleMultiple` and `NestedMultiple` in Figure 4. Use of extraneous dependencies usually leads to imprecise bounds. For example, if a dependency is introduced among the two counter variables (in case of examples `SimpleMultiple` and `NestedMultiple`), then it leads to a quadratic bound as opposed to a linear bound.

The above two observations motivate the following transformations respectively over a triple $(S, M, G)$ that is part of a proof structure $(S, M, G, B)$.

DEFINITION 3 (Node Merging Operation). *Given a triple $(S, M, G)$, and any two counters $c_1, c_2 \in S$, none of which is transitively dependent on the other in $G$, the node merging operation yields a triple $(S', M', G')$, where $S' = S - \{c_2\}$, $M'$ is same as $M$ except that it maps those back-edges $q$ to $c_1$ that were mapped by $M$ to $c_2$, and $G'$ is obtained from $G$ by removing the node $c_2$ and adding the successors/predecessors of $c_2$ to those of $c_1$.*

DEFINITION 4 (Edge Deletion Operation). *Given a triple $(S, M, G)$, and an edge $(c_1, c_2) \in G$, the edge deletion operation yields a triple $(S', M', G')$ where $S' = S$, $M' = M$ and $G' = G - \{(c_1, c_2)\}$.*

We are now ready to define a counter-optimal proof structure.

DEFINITION 5 (Counter-Optimal Proof Structure). *A proof structure $(S, M, G, B)$ is* counter-optimal *if*

- *It has a minimal number of counters. More formally, any node merging operation over $(S, M, G)$ yields a triple that is not part of any proof structure.*
- *It has a minimal number of dependencies between counters. More formally, any edge deletion operation over $(S, M, G)$ yields a triple that is not part of any proof structure.*

Next, we describe how to generate a counter-optimal proof structure.

## 4. Algorithm for constructing a counter-optimal proof structure

In this section, we describe an efficient algorithm for constructing a proof structure, and in fact, one that is counter-optimal. Our algorithm runs in time that in worst-case is quadratic in the number of back-edges (modulo the time taken by the invariant generation tool). In contrast, note that the number of triples $(S, M, G)$ is exponential in the number of back-edges. Hence, a naive full state space search to find any proof structure $(S, M, G, B)$ would be too expensive.

The algorithm strikes the right balance between two opposite challenges.

- Introducing more counters and more dependencies between counters increases (not decreases) the ability of an invariant generation tool to generate bounds on counters. It is always possible to map each back-edge to a distinct counter, but the algorithm cannot simply make all counters depend on all other counters. This would lead to a cyclic dependency of counters, and $G$ would not be a DAG, and hence (proof of) Theorem 1 would break down. So the challenge is to find an appropriate set of acyclic dependencies between counters in DAG $G$. One may wonder if the dependencies between counters correspond to the nesting relationship between the corresponding back-edges. This is unfortunately not the case; the example in Figure 1 aptly illustrates that the flow-graph structure of the program can sometimes be quite misleading.
- To generate a counter-optimal proof structure, the algorithm would want to use a minimum number of counters, and a minimum number of dependencies between counters.

The algorithm for constructing a proof structure $(S, M, G, B)$ for a given procedure $P$ is described in Figure 5. Each iteration of the loop in Line 8 attempts to map a new back-edge $q$ to a counter. The algorithm first tries to use any existing counter variable (to ensure that the number of counter variables generated are optimal/minimal, which is one of the necessary requirements for a proof structure to be counter-optimal). If it fails, the algorithm tries to introduce a new counter variable $c$. The new counter $c$ can be introduced in an exponential number of ways, each corresponding to choosing some subset of other existing counters as the immediate predecessors of $c$ in DAG $G$. This exponential search is avoided by the following two observations.

- Creation of an additional dependency preserves the ability of the invariant generation tool to compute bounds.
- The counter-optimality requirement enforces minimality of dependencies between counters.

Lines 17-21 make use of the above two observations to search for minimal dependencies for the new counter by starting with all possible dependencies and then removing them one by one if possible. However, it is possible that even adding all possible dependencies may not be sufficient for the invariant generation tool to compute bounds at back-edge $q$. In that case, the algorithm

```
SimpleSingle(int n)
  x := 0;
  while (x < n)
    if (*) x := x + 1;
    else x := x + 1;
```

```
SequentialSingle(int n)
  x := 0;
  while (x < n && nondet())
    x := x + 1;
  while (x < n)
    x := x + 1;
```

```
NestedSingle(int n)
  x := 0;
  while (x < n)
    while (x < n && nondet())
      x := x + 1;
  x := x + 1;
```

```
SimpleSingle2(int n,m)
  x := 0; y := 0;
  while (*)
    if (x < n) x++; y++;
    else if (y < m) x++; y++;
    else break;
```

**Figure 3.** This set of simple diverse examples demonstrate that use of fewer counters, whenever possible, leads to better bounds. Each of these examples have two back-edges. Use of single counter for the two back-edges in each of the first three examples produces a bound of $n$. On the other hand, use of multiple counters leads to bounds of $2n$ or $n^2$ depending on the relative ordering of the counters in the DAG $G$ (i.e., depending on where the counters are initialized).

```
SimpleMultiple(int n, m)
  x := 0; y := 0;
  while (x < n)
    if (y < m) y := y + 1;
    else x := x + 1;
```

```
NestedMultiple(int x0, y0, n, m)
  x := x0; y := y0;
  while (x < n)
    while (y < m && nondet())
      y := y + 1;
    x := x + 1;
```

```
SimpleMultipleDep(int n, m)
  x := 0; y := 0;
  while (x < n)
    if (y < m) y++;
    else y := 0; x++;
```

```
NestedMultipleDep(int n, m)
  x := 0;
  while (x < n)
    y := 0; x := x + 1;
    while (y < m)
      y := y + 1;
```

**Figure 4.** (a) This set of diverse examples demonstrate the need for multiple counters. This is because if a single counter is used, then the invariants required to establish bounds are either disjuntive (`SingleMultiple` or `NestedMultiple`) or non-linear (`SingleMultipleDep` or `NestedMultipleDep`). (b) Also, the examples `SingleMultipleDep` and `NestedMultipleDep` demonstrate the need for creating a dependency edge between two counters in DAG $G$ (which corresponds to initializing one of the counters to 0 whenever the other counter is incremented). This is because if multiple counters are used, but are not related in the DAG $G$, then the invariant required to establish bounds is still non-linear.

postpones the decision of choosing a counter for the back-edge $q$ for a future iteration of the outer loop in Line 6.

There might be multiple iterations of the outer loop in Line 6 since a back-edge $q$ that could not be mapped to a counter in an earlier iteration of the loop in Line 8 may now be mapped to a counter since some new back-edges have been mapped to new counters in an earlier iteration. This allows for initializing the counter corresponding to back-edge $q$ to 0 at those back-edges, which in turn, may lead to the generation of a bound at back-edge $q$ by the invariant generation tool.

EXAMPLE 3. *We briefly illustrate the working of the algorithm on `Equals` procedure in Figure 1 to generate the proof structure in Example 1. If the loop in Line 8 starts out by mapping back-edges corresponding to the outer loop and the first inner loop to some counter variable(s), it won't succeed. However, the loop would succeed in mapping the back-edges for the second and third inner loops to new counters $c_1$ and $c_2$ respectively that need be dependent only on $r$; in this setting, invariant generation tool is able to bound $c_1$ and $c_2$ by `Length`$(s_1)$ and `Length`$(s_2)$ respectively at the respective back-edges. The algorithm then performs one more iteration of the outermost loop in line 6 and now the loop in Line 8 succeeds in mapping the back-edges corresponding to outer loop and the first inner loop to a new counter. The algorithm further realizes that they can be mapped to the same counter $c_3$, but $c_3$ should be dependent on $r, c_1, c_2$; in this setting, invariant generation tool is able to bound $c_3$ by `size` at both the back-edges.*

### 4.1 Correctness

It is not difficult to see that the algorithm generates a counter-optimal proof structure (if it generates one). However, the interesting part is that the algorithm, in spite of its greediness to be counter-optimal, does generate a proof structure, if there exists one. The following theorem establishes these properties.

THEOREM 2. *The algorithm always generates a proof structure, if there exists one. Furthermore, if the algorithm generates a proof structure, it generates a counter-optimal one.*

PROOF: It is easy to see that if the algorithm generates a proof structure, then it generates a counter-optimal one. This is because the algorithm generates a new counter for a back-edge (Line 13) only when the back-edge cannot be mapped to an existing counter (see conditional in Line 9). This implies that the new counter cannot be merged with the existing counters. Hence, the proof structure generated has a minimal number of counters. Also, the number of dependencies created for a new counter is minimal as is evident from the loop in Lines 19-21 that exhaustively tries to remove any redundant dependency.

We now show that the algorithm generates a proof structure, if there exists one. This follows from the fact in each iteration of the loop in Lines 8-21, the tuple $(S, M, G, B)$ forms a *extensible partial-proof structure* (as defined below in Definition 6), which can be extended to form a proof structure. This fact can be proved by induction on the number of loop iterations using Theorem 3 stated below, whose proof is given in Appendix A. □

DEFINITION 6 (Extensible Partial-proof Structure).
*A* partial-proof structure *is a tuple $(S, M, G, B)$ that satisfies all properties of a proof structure except we allow $M$ and $B$ to be partial functions over back-edges (in which case the definition of `Instrument`$(P, (S, M, G))$ is the same as in Definition 2 except that we only instrument those back-edges that are mapped by $M$). A partial-proof structure $(S, M, G, B)$ is* extensible *if it can be extended (by extending the maps $M$ and $B$, as well as the DAG $G$) to yield a proof-structure.*

THEOREM 3. *Let $(S, M, G, B)$ be any extensible partial-proof structure. Let $(S', M', G', B')$ be any partial-proof structure such that the triple $(S', M', G')$ is obtained from $(S, M, G)$ by a node merging operation (Definition 3) or an edge deletion operation (Definition 4). Then, $(S', M', G', B')$ is an extensible partial-proof structure.*

```
GenerateCounterOptimalProofStructure(Procedure P)
S := ∅;
foreach back-edge q
    M(q) ← undefined;  B(q) ← undefined;
G := Empty DAG;
change := true;
while change
   change := false;
   foreach back-edge q s.t. M(q) is undefined
      If ∃c ∈ S s.t.((B' := Gen(S, M[q ← c], G)) ≠ ⊥)
         M(q) ← c;  B := B';
         change := true;
      Else
         Let c be some fresh counter variable.
         S' := S ∪ {c};  M' = M[q ← c];
         G' := G ∪ {(c', c) | c' ∈ S, c' ≠ c} ∪ {(r, c)};
         if (B' := (Gen(S', M', G')) == ⊥) continue;
         (S, M, G, B) := (S', M', G', B');
         change := true;
         foreach (c', c) ∈ G:
            if ((B' := Gen(S, M, G − {(c, c')})) ≠ ⊥)
               G := G − {c', c)};  B := B';
if ∃ backedge q s.t. M(q) is undefined, then Fail
return (S, M, G, B);

Gen(S, M, G)
  P' := Instr(P, (S, M, G));
  Run invariant generation tool on P' to generate
  invariant I_q at any back-edge q in P'.
  B' := empty map;
  foreach back-edge q in P' s.t. M(q) is defined
     I'_q := Existentially eliminate all variables
             from I_q except counter M(q) and inputs.
     If I'_q implies an invariant of form M(q) ≤ u,
             B' := B'[q ← u];
     Else return ⊥;
  return B';
```

**Figure 5.** Algorithm to construct a counter-optimal proof-structure for an input procedure $P$, if there exists one. The function $\texttt{Gen}(S, M, G)$ returns a map $B$ that maps each back-edge $q$, s.t. $M(q)$ is defined, to some bound on counter variable $M(q)$ at back-edge $q$ in the procedure $\texttt{Instrument}(P, (S, M, G))$ (obtained by running the invariant generation tool and existentially eliminating the temporary variables from the invariant at $q$); if no bound could be computed for any such back-edge $q$, it returns $\bot$.

## 5. Quantitative Functions over Data-structures

In this section, we introduce the notion of user-definable *quantitative functions* over abstract data-structures (those that are referenced and updated through a well-defined interface). These functions serve two purposes.

- They allow a linear invariant generation tool with support for uninterpreted functions to discover linear bounds over counter variables (in terms of these quantitative functions of input data-structures). The invariant generation tool need not bother about sophisticated heap shapes.
- They are very readable since they have been defined by the user herself. Hence, these allow the user to get a quick sense of the complexity of a procedure (which is otherwise expressible using only a sophisticated logical formula over heap shapes).

Each quantitative function is associated with a tuple of abstract data-structures. In that regard, a quantitative function is similar to a ghost field except that a quantitative function can be associated with a tuple of data-structures, while a ghost field is associated with a single data-structure.

EXAMPLE 4. *Consider a `List` data-structure that maintains a linked list of objects whose type is `ListElement`. We can associate a quantitative function `Len` with a `List` $L$ (supposed to denote the length of list $L$) and we can associate a quantitative function `Pos` with a pair of `ListElement` $e$ and `List` $L$ (supposed to denote the position of a list-element $e$ inside list $L$, if $e$ belongs to $L$; otherwise it is don't care).*

The user annotates each method of an abstract data-structure with how it may affect the quantitative attributes of the input data-structures, and how it determines the quantitative attributes of the output data-structures. These effects are specified in an imperative style by a sequence of (possibly guarded) assignments and assume statements using the program syntax except that the user may also use quantitative functions (applied to appropriate arguments) wherever numeric variables can be used. The user can use only those variables that are in scope at the method declaration level (i.e., the inputs to the method, and the outputs of the method) with one exception. We also allow for use of fresh variables on the left side of an assignment with the interpretation being that the assignment is for all possible instantiations of that fresh variable. This is because a method may change the quantitative attributes associated with multiple objects that are not in scope when the method is invoked.

EXAMPLE 5. *Figure 6(a) describes the effects of some `List` methods on quantitative functions `Len` and `Pos` . The methods $L.\texttt{GetNext}(e2), L.\texttt{GetPrevious}(e2), L.\texttt{Remove}(e)$ all have the precondition that $e2$ belongs to $L$. The method $L.\texttt{Insert}(e)$ and $L.\texttt{Append}(e)$ have the precondition that $e$ does not already belong to $L$. $L.\texttt{Insert}(e)$ inserts $e$ at the front of list $L$, while $L.\texttt{Append}(e)$ appends $e$ at the end of list $L$. The method $L.\texttt{Splice}(L')$ moves all elements from $L'$ to the end of list $L$. The method $L_1.\texttt{MoveTo}(e, L_2)$ removes element $e$ from list $L_1$ and inserts it at beginning of list $L_2$ (and has precondition that $e$ belongs to $L_1$). The effect of method call $L_1.\texttt{MoveTo}(e, L_2)$ involves decrementing $\texttt{Len}(L_1)$ and incrementing $\texttt{Len}(L_2)$ by 1. Furthermore, inserting an element at the beginning of $L_2$ increases position $\texttt{Pos}(e', L_2)$ of all list-elements $e'$ in $L_2$ by 1. Stating this requires use of a free variable $e'$. Similarly, removal of an element from $L_1$ decreases (by 1) position $\texttt{Pos}(e', L_1)$ of all list-elements $e'$ that are after $e$ in $L$.*

***Principles*** There are two principles to keep in mind when defining these quantitative functions.

- (Precision) In general, defining more quantitative functions increases the possibility that the invariant generation tool will be able to compute a bound. However, we observed that for several commonly used data-structures in the source code of a large MS product code, the number of quantitative functions required for computing bounds is small.
- (Soundness) Whatever quantitative functions the user defines, they are always sound from the tool's viewpoint since it takes the semantics of these functions is what the user defines them to be. However, the user has an intended semantics for these functions in her mind. It is thus the user's responsibility to ensure that the user has conservatively estimated the effect of different methods over the quantitative attributes of different objects with respect to the intended semantics. (A recent work [13] can be used to check the soundness of the user specifications w.r.t. the intended semantics, if the intended semantics can be described as the size of some partition in an appropriate logic.)

We carried out an exhaustive case-study of the source code of a large Microsoft product to identify commonly used data-structures:

| List Operation | Effect on Quantitative Functions |
|---|---|
| $e := L.\texttt{Head}()$ | $\texttt{Assume}(e = \texttt{null} \Rightarrow \texttt{Len}(L) = 0); \texttt{Assume}(e \neq \texttt{null} \Rightarrow \texttt{Len}(L) > 0); \texttt{Pos}(e, L) := 0;$ |
| $e := L.\texttt{Tail}()$ | $\texttt{Assume}(e = \texttt{null} \Rightarrow \texttt{Len}(L) = 0); \texttt{Assume}(e \neq \texttt{null} \Rightarrow \texttt{Len}(L) > 0); \texttt{Pos}(e, L) := \texttt{Len}(L) - 1;$ |
| $t := L.\texttt{IsEmpty}()$ | $\texttt{Assume}(t = \texttt{true} \Rightarrow \texttt{Len}(L) = 0); \texttt{Assume}(t = \texttt{false} \Rightarrow \texttt{Len}(L) > 0)$ |
| $e1 := L.\texttt{GetNext}(e2)$ | $\texttt{Pos}(e1, L) := \texttt{Pos}(e2, L) + 1; \texttt{Assume}(0 \leq \texttt{Pos}(e2, L) < \texttt{Len}(L));$ |
| $e1 := L.\texttt{GetPrevious}(e2)$ | $\texttt{Pos}(e1, L) := \texttt{Pos}(e2, L) - 1; \texttt{Assume}(0 \leq \texttt{Pos}(e2, L) < \texttt{Len}(L));$ |
| $L.\texttt{RemoveHead}()$ | if $(\texttt{Len}(L) > 0)$ $\{$ $\texttt{Len}(L) := \texttt{Len}(L) - 1; \texttt{Pos}(e', L) := \texttt{Pos}(e', L) - 1$ $\}$; |
| $L.\texttt{Remove}(e)$ | $\texttt{Len}(L) := \texttt{Len}(L) - 1;$ if $(\texttt{Pos}(e, L) < \texttt{Pos}(e', L))$ $\texttt{Pos}(e', L) := \texttt{Pos}(e', L) - 1;$ |
| $L.\texttt{Insert}(e)$ | $\texttt{Len}(L) := \texttt{Len}(L) + 1; \texttt{Pos}(e', L) := \texttt{Pos}(e', L) + 1; \texttt{Pos}(e, L) := 0;$ |
| $L.\texttt{Append}(e)$ | $\texttt{Len}(L) := \texttt{Len}(L) + 1; \texttt{Pos}(e, L) := \texttt{Len}(L) - 1;$ |
| $L.\texttt{Splice}(L')$ | $\texttt{Len}(L) := \texttt{Len}(L) + \texttt{Len}(L'); \texttt{Len}(L') := 0; \texttt{Pos}(e, L)$ |
| $L_1.\texttt{MoveTo}(e, L_2)$ | $\texttt{Len}(L_1) := \texttt{Len}(L_1) - 1; \texttt{Len}(L_2) := \texttt{Len}(L_2) + 1; \texttt{Pos}(e, L_2) := 0; \texttt{Pos}(e', L_2) := \texttt{Pos}(e', L_2) + 1;$ if $(\texttt{Pos}(e, L_1) < \texttt{Pos}(e', L_1))$ $\texttt{Pos}(e', L_1) := \texttt{Pos}(e', L_1) - 1;$ |

**(a) Semantics** of Quantitative Functions Len and Pos

| | Some looping patterns over lists (from Microsoft product code) | Loop Invariant | Complexity |
|---|---|---|---|
| 1. | for $(e := f; e \neq \texttt{null}; e := L.\texttt{GetNext}(e))$; | $c = \texttt{Pos}(e, L)\text{-}\texttt{Pos}(f, L)$ $\wedge$ $\texttt{Pos}(e, L) \leq \texttt{Len}(L)$ | $\texttt{Len}(L) -$ $\texttt{Pos}(f, L)$ |
| 2. | for $(; !L.\texttt{IsEmpty}(); L.\texttt{RemoveHead}())$; | $c = \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) \ \wedge \ \texttt{Len}(L) \geq 0$ | $\texttt{Old}(\texttt{Len}(L))$ |
| 3. | for $(e := L.\texttt{Head}(); e \neq \texttt{null};)$ $tmp := e; e := L.\texttt{GetNext}(e);$ if (*) $L.\texttt{Remove}(tmp)$; | $c = \texttt{Pos}(e, L) + \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) \wedge$ $\texttt{Pos}(e, L) \leq \texttt{Len}(L)$ | $\texttt{Old}(\texttt{Len}(L))$ |
| 4. | $ToDo.\texttt{Init}(); Done.\texttt{Init}(); L.\texttt{MoveTo}(L.\texttt{Head}(), ToDo);$ while $(!ToDo.\texttt{IsEmpty}())$ $e := ToDo.\texttt{Head}(); ToDo.\texttt{RemoveHead}();$ $Done.\texttt{Insert}(e);$ foreach successor $s$ in $e.\texttt{Successors}()$ if $(L.\texttt{Contains}(s))$ $L.\texttt{MoveTo}(s, ToDo)$; | $c_1 \leq \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) - \texttt{Len}(ToDo)$ $\wedge \ c_1 = \texttt{Len}(Done) \ \wedge \ \texttt{Len}(L) \geq 0 \wedge$ $\texttt{Len}(ToDo) \geq 0$ | $\texttt{Old}(\texttt{Len}(L))$ |
| | for $(e := Done.\texttt{Head}(); e \neq \texttt{null}; e := Done.\texttt{GetNext}(e))$; | $c_2 \leq \texttt{Pos}(e, Done)$ $\wedge \ \texttt{Pos}(e, Done) \leq \texttt{Len}(Done))$ $\wedge \texttt{Len}(Done) \leq \texttt{Old}(\texttt{Len}(L)$ | $\texttt{Old}(\texttt{Len}(L))$ |

**(b) Examples**

**Figure 6.** In Table (b), Column 1 contains examples of looping patterns over lists from MS product code. Column 2 describes (interesting part of) the inductive loop invariant (computed by our invariant generation tool) that relates an instrumented loop counter $c$ with appropriate quantitative attributes. Column 3 shows an upper bound on loop iterations as obtained from the invariant in Column 2.

lists, list of lists, bit-vectors, trees. We found that the above mentioned principles are easy to satisfy for these data-structures. In particular, we found that a few quantitative functions are effective enough to express the timing complexity (as well as the invariants required to compute the timing complexity) of a variety of loops that iterate over the corresponding data-structure. We also observe that it is easy and natural to write down the update to these quantitative functions for the methods supported by the corresponding data-structure.

### 5.1 Invariant Generation over Quantitative Functions

In order to allow for invariant generation over quantitative functions, we need to communicate the meaning of the quantitative functions to the invariant generation tool. We do this by instrumenting each data-structure method call-site with the effect that it has on the quantitative functions of inputs and outputs of the method call. This is done by substituting the formals input and return parameters in the user specification by the actuals at the call site. The only issue is with respect to the assignments that involve free variables in the specification. These can be handled by instantiating these assignments with all expressions of appropriate type that are live at that program point. However, this (potentially expensive) eager approach can be avoided by instantiating these assignments (during the invariant generation process) with only those expressions that are present in the invariants computed by the (flow-sensitive) invariant generation tool immediately before the method call site.

The above instrumentation allows for treating the quantitative functions as uninterpreted functions because their semantics has explicitly been encoded in the program. Now, we can simply use a linear invariant generation tool that has been extended with support for uninterpreted functions and aliasing [3] to compute linear invariants over quantitative functions.

We know of two techniques that extend a linear invariant generation tool with support for uninterpreted functions.

- Abstract Interpretation based technique. Gulwani and Tiwari have described a general mechanism for combining the transfer functions of two given abstract interpreters to generate an abstract interpreter than can discover invariants over combination of domains [15]. We can use this methodology to combine an abstract interpreter for linear arithmetic (such as the one based on polyhedron domain [7]) with an abstract interpreter for uninterpreted functions [14]. We have implemented our invariant generation tool using this methodology.
- Constraint-based invariant generation technique. Beyer et al. have described how to extend constraint-based techniques for generating numerical invariants to synthesis of invariants expressible in the combined theory of linear arithmetic and uninterpreted function symbols [2]. Unlike fixed-point computation based techniques like abstract interpretation, constraint-based

[3] Aliasing is required to reason whether an update to an uninterpreted function such as $\texttt{Len}(L_1)$ can affect $\texttt{Len}(L_2)$: this is done by checking whether or not $L_1$ and $L_2$ are aliased.

techniques are goal-directed and do not suffer from precision losses due to widening. However, these advantages come at the cost of programmer specified invariant templates. Our application is a good fit for such a scenario because we are indeed looking for a specific kind of invariant, one that relates the loop counter with quantitative functions.

***Examples*** Figure 6(b) shows some examples of looping patterns over lists from Microsoft product code. Column 2 of the table lists the interesting part of the inductive loop invariant generated by our invariant generation tool after these loops have been instrumented with a single counter $c$. Column 3 lists the bounds computed from these invariants (by existential elimination of all temporary variables or variables that get modified in the program). Some of these invariants use the term $\texttt{Old}(t)$ that refers to the value of $t$ at the beginning of the procedure (It is useful to create a copy of the quantitative functions of input data-structures at the beginning since these might get destructively updated in the program).

Example 1 iterates over a list starting from a list-element $f$ in list $L$ and following the next links. Example 2 iterates over a list by deleting its head in each iteration. Example 3 is more interesting and combines list enumeration with destructive update of the list. Note that the inductive loop invariant (which is automatically discovered) is a bit tricky in this case.

Example 4 is the most challenging example because the while loop iterates over a $ToDo$ list whose length may decrease as well as increase in each loop iteration. Overall, elements are moved from the input list $L$ to the $ToDo$ list, which are then moved to $Done$ list. However, the order in which the vertices are moved to $Done$ list is the depth-first traversal order of the list-elements $e$, which are also graph nodes whose successors are given by the `Successors` method. Bounding the loop iterations of the outer while loop requires computing the non-trivial invariant $c_1 \leq 1 + \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) - \texttt{Len}(ToDo)$, which is something that is easily computed by our invariant generation tool. Also, note that it is easy to see that an upper bound on the number of loop iterations of the for-loop (after the while-loop) is the length of the $Done$ list. However, computing this upper bound in terms of the inputs requires relating the length of the $Done$ list in terms of the length of the input list $L$; this relationship is: $\texttt{Len}(Done) \leq \texttt{Old}(\texttt{Len}(L))$. Discovering this relationship requires computing the loop invariant $\texttt{Len}(Done) \leq 1 + \texttt{Old}(\texttt{Len}(L)) - \texttt{Len}(L) - \texttt{Len}(ToDo)$ in the first loop, which is again easily computed by our invariant generation tool. This illustrates another advantage of the quantitative functions in the overall process. The quantitative functions are not only useful for expressing loop bounds, but also allow the invariant generation tool to relate numerical properties of different data-structures, which is important to express the loop bounds in terms of inputs.

### 5.2 Composite Data-structures

Composite data-structures like list of lists, array of lists (hastables) or n-ary trees, may have more interesting quantitative attributes that can be associated with constituent data-structures. This happens when the quantitative attribute of a top-level data-structure may be a function of the quantitative attributes of the nested data-structures.

A challenge that arises in such situations is that update of a nested data-structure may not only affect the quantitative functions of the nested data-structure, but may also affect the quantitative functions of the top-level data-structure. To address this, we propose defining another function at the level of a nested data-structure that maps it to the top-level data-structure of which it is a part of. A disadvantage of this approach is that it is not modular. However, we feel that this will not be a problem in practice since the annotations are only provided at the data-structure level.

We illustrate this methodology for some useful quantitative functions that can be associated with a list of lists, besides the functions `Len` and `Pos` defined earlier. Let $L$ be any top-level list of elements $e$, where each element $e$ is a list of nodes $f$.

- $\texttt{TotalNodes}(L)$: Sum of length of all lists $e'$, where $e'$ is an element of $L$.

  $$\texttt{TotalNodes}(L) = \texttt{Sum}\{\texttt{Len}(e') \mid L.BelongsTo(e')\}$$

- $\texttt{MaxNodes}(L)$: Maximum length of any list $e'$, where $e'$ is an element of $L$.

  $$\texttt{MaxNodes}(L) = \texttt{Max}\{\texttt{Len}(e') \mid L.BelongsTo(e')\}$$

- $\texttt{TotalPos}(e, L)$: Sum of lengths of all lists $e'$, where $e'$ lies before $e$ in $L$ (i.e., if $e$ belongs to $L$, otherwise it is don't care).

  $$\texttt{TotalPos}(e, L) = \texttt{Sum}\{\texttt{Len}(e') \mid L.BelongsTo(e') \wedge \texttt{Pos}(e', L) < \texttt{Pos}(e, L)\}$$

Note that the quantitative attribute $\texttt{TotalNodes}(L)$ of the top-level list $L$ gets affected whenever any change is made to list $e$. In order to appropriately update $\texttt{TotalNodes}(L)$ in such cases, we propose introducing a function `Owner` that maps $e$ to its top-level list $L$. This idea is borrowed from the literature on ownership fields [21].

- $\texttt{Owner}(e)$: Top-level list $L$ to which the nested list $e$ belongs. (If $e$ is not a nested list, then $\texttt{Owner}(()e)$ is don't care.)

Table (a) in Figure 7 describes the updates to these functions by some list operations. Rows 1-5 show how these functions are affected by some list operations when invoked over a list of lists. Row 6 shows how these functions are affected by $Remove$ operation when invoked over a nested list.

Table (b) in Figure 7 shows some examples of looping patterns over list of lists from Microsoft product code. Example 1 iterates over the top-level list, and hence its complexity is simply the length of the top-level list. Example 2 iterates over the top-level list, but also processes all nodes in the nested lists, and hence its complexity is the sum of the top-level list and the lengths of all the nested lists, which can be expressed using the quantitative functions `Len` and `TotalNodes`. Example 3 finds a specific node in the nested list and does this by walking over the top-level list and then walking over an appropriate nested list. Its complexity can be expressed using the quantitative function `MaxNodes`.

Example 4 is most interesting since it walks over the top-level list $L$ as well as all the nested lists deleting each element one by one. Its complexity is expressible using the quantitative functions `Len` and `TotalNodes`. However, the interesting point to note here is that the top-level list as well as the nested lists are being destructively updated while they are being traversed. The destructive update to a nested list $e$ also requires an update to the quantitative attributes of the top-level list $L$, which is accessed using $\texttt{Owner}(e)$.

### 5.3 Applicability of Quantitative Functions

The methodology of quantitative functions need not be restricted to recursive data-structures, but can also be applied to data-structures like bit-vectors (which are otherwise hard to reason about). Bit-vectors have quite a few interesting quantitative functions associated with them. E.g., total number of bits: $\texttt{Bits}(a)$, total number of 1 bits: $\texttt{Ones}(a)$, position of the least significant 1 bit: $\texttt{One}(a)$, etc. Appendix B describes these quantitative functions in more detail and the effect of bit-wise operators on these functions, and also gives examples of several looping patterns from Microsoft product code-base that can be analyzed using these functions. Figure 8 describes one of such examples `Iterate`, which masks out the least significant consecutive chunk of 1s from $b$ in each loop iteration. Our tool is able to compute the inductive loop invariant

| | List Operation | Effect on Quantitative Functions |
|---|---|---|
| 1. | $e := L.\texttt{Head}()$ | $\texttt{Assume}(e = \texttt{null} \Rightarrow \texttt{Len}(L) = 0)$; $\texttt{Assume}(e \neq \texttt{null} \Rightarrow \texttt{Len}(L) > 0)$; $\texttt{Pos}(e, L) := 0$; $\texttt{TotalPos}(e, L) := 0$; $\texttt{Assume}(\texttt{Len}(e) \leq \texttt{MaxNodes}(L))$; $\texttt{Owner}(e) := L$; |
| 2. | $t := L.\texttt{IsEmpty}()$ | $\texttt{Assume}(t = \texttt{true} \Rightarrow \texttt{TotalNodes}(L) = 0 \wedge \texttt{MaxNodes}(L) = 0)$; $\texttt{Assume}(t = \texttt{false} \Rightarrow \texttt{TotalNodes}(L) > 0)$; |
| 3. | $e1 := L.\texttt{GetNext}(e2)$ | $\texttt{TotalPos}(e1, L) := \texttt{TotalPos}(e2, L) + \texttt{Len}(e2)$; $\texttt{Assume}(\texttt{TotalPos}(e1, L) \leq \texttt{TotalNodes}(L))$; $\texttt{Assume}(\texttt{Len}(e1) \leq \texttt{MaxNodes}(L))$; $\texttt{Owner}(e1) := L$; |
| 4. | $L.\texttt{Remove}(e)$ | $\texttt{TotalNodes}(L) := \texttt{TotalNodes}(L) - \texttt{Len}(e)$; $\{\text{int } tmp := \texttt{MaxNodes}(L)$; $\texttt{MaxNodes}(L) :=?$; $\texttt{Assume}(\texttt{MaxNodes}(L) \leq tmp);\}$ if $(\texttt{Pos}(e, L) < \texttt{Pos}(e', L))$ $\texttt{TotalPos}(e', L) := \texttt{TotalPos}(e', L) - \texttt{Len}(e)$; |
| 5. | $L.\texttt{Insert}(e)$ | $\texttt{TotalNodes}(L) := \texttt{TotalNodes}(L) + \texttt{Len}(e)$; $\texttt{MaxNodes}(L) := \texttt{Max}(\texttt{MaxNodes}(L), \texttt{Len}(e))$; $\texttt{TotalPos}(e', L) := \texttt{TotalPos}(e', L) + \texttt{Len}(e)$; $\texttt{TotalPos}(e, L) := 0$; $\texttt{Owner}(e) := L$; |
| 6. | $e.\texttt{Remove}(f)$ | $\texttt{TotalNodes}(\texttt{Owner}(e)) := \texttt{TotalNodes}(\texttt{Owner}(e))$-1; if (*) $\texttt{MaxNodes}(\texttt{Owner}(e)) := \texttt{MaxNodes}(\texttt{Owner}(e))$-1; if $(\texttt{Pos}(e, \texttt{Owner}(e)) < \texttt{Pos}(e', \texttt{Owner}(e)))$ $\texttt{TotalPos}(e', \texttt{Owner}(e)) := \texttt{TotalPos}(e', \texttt{Owner}(e)) - 1$; |

**(a) Semantics** of Quantitative Functions TotalNodes,MaxNodes,TotalPos and the function Owner.

| | Looping patterns over list-of-lists (from MS product code) | Loop Invariant | Complexity |
|---|---|---|---|
| 1. | for $(e := L.\texttt{Head}(); e \neq \texttt{null}; e := L.\texttt{GetNext}(e))$; | $c = \texttt{Pos}(e, L) \ \wedge \ \texttt{Pos}(e, L) \leq \texttt{Len}(L)$ | $\texttt{Len}(L)$ |
| 2. | for $(e := L.\texttt{Head}(); e \neq \texttt{null}; e := L.\texttt{GetNext}(e))$ for $(f := e.\texttt{Head}(); f \neq \texttt{null}; f := e.\texttt{GetNext}(f))$; | $c = \texttt{Pos}(e, L) + \texttt{TotalPos}(e, L) \wedge$ $\texttt{Pos}(e, L) \leq \texttt{Len}(L) \wedge$ $\texttt{TotalPos}(e, L) \leq \texttt{TotalNodes}(L)$ | $\texttt{Len}(L)+$ $\texttt{TotalNodes}(L)$ |
| 3. | for $(e := L.\texttt{Head}(); e \neq \texttt{null}; e := L.\texttt{GetNext}(e))$ if $(\texttt{rand}())$ break; for $(f := e.\texttt{Head}(); f \neq \texttt{null}; f := e.\texttt{GetNext}(f))$; | $c = \texttt{Pos}(e, L) + \texttt{Pos}(f, e) \wedge$ $\texttt{Pos}(f, e) \leq \texttt{Len}(e) \wedge$ $\texttt{Len}(e) \leq \texttt{MaxNodes}(L)$ | $\texttt{Len}(L)+$ $\texttt{MaxNodes}(L)$ |
| 4. | for $(e := L.\texttt{Head}(); e \neq \texttt{null}; e := L.\texttt{RemoveHead}())$ for $(f := e.\texttt{Head}(); f \neq \texttt{null}; f := e.\texttt{RemoveHead}())$; | $c = \texttt{Old}(\texttt{Len}(L)) - \texttt{TotalNodes}(L)$ $+\texttt{Old}(\texttt{TotalNodes}(L)) - \texttt{Len}(L) \wedge$ $\texttt{Owner}(e) = L$ | $\texttt{Old}(\texttt{Len}(L))+$ $\texttt{Old}(\texttt{TotalNodes}(L))$ |

**(b) Examples**

**Figure 7.** In Table (b), Column 1 contains some examples of looping patterns over lists of lists from MS product code. Column 2 describes (interesting part of) the inductive loop invariant, as computed by our tool, after the back-edges in each example were all instrumented with the same counter $c$. Invariants shown are for outer loop (Examples 1,2,4) and for the second loop (Example 3). Column 3 shows an upper bound on loop iterations as obtained from the invariant in Column 2.

```
  Iterate(unsigned int a) {
1   b := a;
2   while (_BitScanForward(&id1, b))
                //set all bits before id1 in b
3     b := b | ((1 << id1) − 1);
4     if (_BitScanForward(&id2, ∼ b)) break;
                //reset all bits before id2 in b
5     b := b & (∼((1 << id2) − 1));
6 }
```

**Figure 8.** One of many loops that iterate over bit-vectors taken from MS product code. The $\_BitScanForward(\&id, b)$ function returns a non-zero value iff the bit-vector $b$ contains a 1 bit, in which case $id$ is set to the position of the least significant 1 bit.

$2c \leq 1 + \texttt{One}(b) - \texttt{One}(a) \ \wedge \ c \leq 1 + \texttt{Ones}(a) - \texttt{Ones}(b)$ when the loop is instrumented with the counter variable $c$. This implies bounds of both $\texttt{Ones}(a)$ as well as $(\texttt{Bits}(a) - \texttt{One}(a))/2$ on the total number of loop iterations.

Appendix C describes interesting quantitative functions for trees, namely number of nodes in a tree, or the height of a tree, and also describes several looping patterns from STL library codebase that can be analyzed using these quantitative functions. One such example is also provided in Figure 9.

## 6. Inter-procedural Computational Complexity

Theorem 1 described in Section 4 describes a strategy to compute symbolic bound on the total number of loop iterations given a proof structure. We can use a similar strategy to compute symbolic bounds on the total cost of a procedure given any cost metric that maps atomic program statements (i.e., non procedure-call statements) to some cost. For example, if we associate a unit cost with each statement, then we obtain a symbolic bound on the total number of statements executed by the procedure. If instead we associate each statement with the amount of resources it consumes [4] (e.g., memory it allocates), we obtain a bound on the total resource consumption of the procedure.

The cost of procedures in a program is computed in a bottom-up ordering of the call graph. Since the call graph can have cycles because of recursive procedures, we decompose the call graph into a DAG of maximal strongly connected components (SCCs) and process this DAG in reverse topological order.

***Non-recursive Procedures*** For a non-recursive procedure (not a part of any non-trivial SCC), we use the approach mentioned below.

We define the cost $\|q\|$ of a back-edge $q$ between locations $\ell$ and $\ell'$ to be the maximum of the costs of any (acyclic) path that starts at procedure entry or any counter instrument location, and ends at $\ell$ without visiting any other counter instrument location. We define the cost of any acyclic path to be the sum of the cost of all statements on the path and the cost of executing any procedure call. The cost $\|\text{call } Q(\vec{v})\|$ of executing procedure call $Q(\vec{v})$ is obtained from the cost $\|Q\|$ of procedure $Q$, which has already

[4] Sometimes resources can also be released by a statement (as in case of memory deallocation), in which case, one needs to obtain a *lower* bound $L$ on the amount of resource released by the statement, and then associate that statement with a cost of $-L$.

been computed in terms of the formal parameters $\vec{y}$ of procedure $Q$. We first replace the formal parameters $\vec{y}$ in $Q$ by actuals $\vec{v}$. Since $\vec{v}$ might not be the inputs of the procedure whose summary is being computed, we need a way to express $\|Q\|[\vec{v}/\vec{y}]$ as a function of procedure inputs $\vec{x}$. We do this by making use of the invariant (generated by the invariant generation tool) at the call site $I_{\texttt{call}}$ (which relates the actuals $\vec{v}$ with the procedure inputs $\vec{x}$) as follows.

$$\|\text{call } Q(\vec{v})\| \quad := \quad \texttt{Project}_{upper}(\|Q\|[\vec{v}/\vec{y}], I_{\texttt{call}}, \vec{x})$$

where the function $\texttt{Project}_{upper}(e, \phi, V)$ denotes an upper bound on variable $t$ that is implied by the formula $\exists V' : t \leq e \wedge \phi$, where $V'$ is the set of all variables that occur in $e$ and $\phi$ except $V$, and $t$ is some fresh variable.[5] See an example below.

EXAMPLE 6. *Suppose we are computing a summary for procedure $P(x_1, x_2)$ that calls a procedure $Q(y_1, y_2)$ whose summary is already computed as $y_1 - 2y_2$. Suppose $P$ calls $Q$ with arguments $v_1, v_2$, where $v_1, v_2$ are such that $(v_1 - v_2 \leq x_1) \wedge (v_2 \geq x_2)$. Then, the above $\texttt{Project}_{upper}$ operation helps estimate an upper bound on the cost of calling $Q$ inside $P$ as $x_1 - x_2$ by existentially quantifying out $v_1, v_2$ from the formula $\exists v_1, v_2[t \leq v_1 - 2v_2 \wedge (v_1 - v_2 \leq x_1) \wedge (v_2 \geq x_2)]$ to obtain $t \leq x_1 - x_2$.*

Having defined the cost of a back-edge $q$ (with respect to any given cost metric over statements), we now describe in the theorem below how to compute the total cost of a non-recursive procedure. Proof of Theorem 4 follows easily from the proof of Theorem 1.

THEOREM 4. *The total cost $\|P\|$ of a procedure $P$ given a cost metric for atomic statements, and a proof structure $(S, M, G, B)$ can be computed as*

$$\|P\| \quad = \quad \sum_{c \in S} (1 + \texttt{TotalBound}(c)) \times \texttt{Max}\{\|q\| \mid M(q) = c\}$$

*where $\texttt{TotalBound}(c)$ is as defined in Theorem 1.*

***Recursive Procedures*** Let $P_1(\vec{x}), \ldots, P_n(\vec{x})$ be some set of mutually recursive procedures. For each procedure $P_i(\vec{x})$, we create a new procedure $P_i'(\vec{x})$ that simply calls $P_i(\vec{x})$ after copying $\vec{x}$ into global $\vec{x'}$. Now, to compute the complexity of any procedure $P_i$, we simply put the procedures $P_i'$ along with the procedures $P_1, \ldots, P_n$ into a module, and marking procedure $P_i'$ as an entry point. We now run the algorithm in Figure 5 for generating an optimal proof structure over this module with the following differences.

The proof structure now has an additional requirement. The map $M$ is required to map back-edges in all the procedures as well as the locations immediately *before* any recursive procedure call site to some *global* counter variable. A global counter variable is incremented and initialized in $\texttt{Instrument}(P, (S, M, G))$ in exactly the same manner as local counter variables with one exception. If $r$ is an immediate predecessor of $c$ in the DAG $G$, then $c$ is initialized to 0 at the beginning of the entry point procedure $P_i'$ (as opposed to any of the recursive procedures). The correctness of the algorithm now follows from the observation that a global counter introduced before a recursive call-site is counting how many times is that call-site invoked (just as a counter variable at a back-edge counts the number of iterations of the back-edge).

[5] The function $\texttt{Project}_{upper}$ is used to compute bound on complexity of a procedure call $Q(\vec{v})$ in terms of the procedure inputs $\vec{x}$. (Note $\|Q\|$ has been computed in terms of formals $\vec{y}$. We replace that by actuals $\vec{v}$, but we still need to relate that to $\vec{x}$). The existential formula $\exists V' : t \leq e \wedge \phi$ formalizes the meaning of $\texttt{Project}_{upper}$. Essentially, we want to eliminate variables from a bounds expression in presence of some invariants, but the notion of existential elimination is only defined for formulas. Use of variable $t$ allows us to model the problem of existential elimination from an expression to the problem of existential elimination from a formula.

```
Traverse(Tree T, TreeElt e)
1  y := e;
2  while (y ≠ null)
3      c := c + 1;
4      r := T.GetRight(y);
5      Traverse(T, r);
6      y := T.GetLeft(y);
7      c := c + 1;
```

```
Tree T';
TreeElt e';
int c;
Traverse'(Tree T, TreeElt e)
  T' := T; e' := e; c := 0;
  Traverse(T, e);
```

**Figure 9.** An example of a recursive procedure `Traverse` that calls itself recursively inside a loop in presence of recursive data-structures. The bold instrumentation of global counter variable $c$ inside procedure `Traverse` and the construction of procedure $\texttt{Traverse}'$ is part of our proof methodology for bounding number of recursive procedure call invocations as well as loop iterations in the procedure `Traverse`.

Also, another difference is that the invariant generation tool is supposed to compute bounds on the global counter variables in terms of the globals $\vec{x'}$ (as opposed to the procedure inputs $\vec{x}$) in an *inter-procedural setting*. (After computation of these bounds, we simply replace $\vec{x'}$ by $\vec{x}$). An intra-procedural invariant generation tool can be extended to an (context-insensitive) inter-procedural setting using the standard two phased approach. The first phase computes procedure summaries that relate the inputs of a procedure to the outputs. The actual results are computed in the second phase, wherein inputs are propagated down the call graph from the entry points. We implemented this algorithm and we give below an example of the invariants and bounds that were generated.

EXAMPLE 7. *Consider the recursive procedure `Traverse` in Figure 9 taken from C++ STL library code. The procedure calls itself recursively inside a loop to traverse a tree. The procedure has been instrumented with a global variable c to keep track of the number of recursive procedure call invocations and loop iterations. The global counter c is initialized inside the entry procedure $\texttt{Traverse}'$, which simply invokes `Traverse` after copying its arguments $T$ and $e$ into globals $T'$ and $e'$. Let $\texttt{Nodes}(e, T)$ refer to a quantitative function that denotes the number of nodes below node $e$ in tree $T$. The inter-procedural invariant generation scheme described above computes the inter-procedural invariant $c \leq 2(1 + \texttt{Nodes}(e', T') - \texttt{Nodes}(e, T))$ [6] (after computing the procedure summary $c^{out} - c^{in} = 2 + 2(\texttt{Nodes}(e, T))$, which in turn requires the inductive loop invariant $c - c^{in} = 2(\texttt{Nodes}(e', T') - \texttt{Nodes}(y, T))$ at program point 3). Existential elimination of input parameter $e$ yields the relation $c \leq 2(1 + \texttt{Nodes}(e', T'))$, thereby implying a bound of $2(1 + \texttt{Nodes}(e, T))$ (after we rename $e'$ back by $e$ in the relation) on the recursive call invocations and loop iterations.*

Having obtained an appropriate proof structure (which has counter increments at recursive call-sites), we now use the same process that we used for non-recursive procedures (described in Theorem 4) to compute the total cost. The cost of a recursive call-site location $s$ is defined in a similar way that we define the cost of a back-edge. Note that now an acyclic path will not refer to any of the recursive procedures $P_1, \ldots, P_j$, since we have instrumented

[6] In other words, the value of global counter $c$ is proportional to the number of nodes that have been traversed, which is equal to difference of total nodes $\texttt{Nodes}(e', T')$ and nodes yet-to-be traversed $\texttt{Nodes}(e, T)$. Observe that this inductive invariant requires reference to original inputs with which `Traverse` was first invoked. This reference is provided by copying of the inputs $T$ and $e$ to fresh global variables $T'$ and $e'$ in procedure $\texttt{Traverse}'$. A similar technique is used in program verification to allow for relating pre and post states.

a counter increment at those locations (in the extended notion of proof structure for recursive procedures).

## 7. Preliminary Evaluation

We have built a prototype tool called SPEED that implements the ideas described in this paper and automatically computes symbolic complexity bounds of procedures (written in C/C++) in terms of the scalar inputs of the procedure as well as any user-defined quantitative functions. Our implementation uses the Microsoft Phoenix compiler infrastructure [25] for the front-end. We have implemented the abstract interpretation based invariant generation tool in C#, as a Phoenix plug-in.

Our tool operates by slicing a procedure with respect to the statements that affect the number of loop iterations (by tracking statements that affect the conditionals in loop exit nodes). The sliced version is usually much smaller, and is thus a useful optimization for our invariant generation tool. We avoid interprocedural invariant generation for non-recursive procedures by simply inlining part of the called procedures that affect the number of loop iterations. This happens for a relatively very few cases.

Our tool is able to automatically compute bounds for all the examples presented in this paper and takes on an average 0.25 seconds for each of the examples. Even though the size of the examples as described in the paper is small, these examples are sliced versions of large procedures (ranging up to few hundreds of lines of code) from real code.

We have also run our tool on several other examples from Microsoft product code and C++ STL Library code. Our tool is able to compute precise bounds for more than half of the loops that we have seen in practice. Among the examples that our tool is able to handle, there are several examples with loops whose nesting depth goes up to 5, while their complexity is still linear or quadratic since they iterate over the same variables in multiple loops. These are loops for which an unsound analysis that simply estimates the cost of a procedure by measuring the nesting depth would provide too imprecise results. The cases that we have found our tool currently cannot handle fall into one of two categories:

- Loop bounds depend on some environment assumption, discovering which requires a global analysis (while our invariant generation tool performs only local analysis). These environment assumptions range from class invariants (e.g., $m > 0$, required to prove bounds in case of `while`$(i \leq n)\{i := i + m\}$, or some precondition on the input (e.g., $x < n$, required to prove bounds in case of `for`$(i := x; i \neq n; i$`++`$)$ to some contract between concurrently executing threads, or some liveness property guaranteed by the operating system (e.g., a loop calling a low-level system API until it returns true).
- Generating linear bounds on counter variables requires path-sensitive invariant generation (while our invariant generation tool is path-insensitive). For example,
  - `while` $(0 \leq x \leq n)$ $\{$`if` $(b)$ $x := x+t$; `else` $x$`:=`$x-t$;$\}$
  - $t :=$ $b$`?` $1 : -1$;
    `while` $(x \leq n)$ $\{$`if` $(b)$ $x := x+t$; `else` $x := x-t$;$\}$

We also identified an instance of a recursive procedure `V` where our tool failed to compute a bound, and the reason was traced to the bound being exponential in size of the input tree. This complexity surprised the developer who wrote that code. The looping pattern of the recursive function `V(Tree` $T$, `TreeElement` $e$`)` consisted of `if (`$e \neq$`null) { V(`$T$`,`$T$`.GetLeft(`$e$`)); V(`$T$`,`$T$`.GetRight(`$e$`));`
`if (`$b$`){V(`$T$`,`$T$`.GetLeft(`$e$`)); V(`$T$`,`$T$`.GetRight(`$e$`));} }`

We are currently working with some MS product groups to incorporate SPEED into their code check-in process. They are specifically interested in creating a database of the symbolic complexities of all the procedures in their huge code-base and then pointing out any significant differences in symbolic complexity bounds whenever any developer checks in performance-critical code, e.g., code that is executed within atomic boundaries (probably higher up in the call graph), or code that touches hot data-structures.

## 8. Related Work

***Type System Based Approaches*** Danielsson presented a type system targeted towards reasoning about complexity of programs in lazy functional languages [9]. Crary and Weirich presented a type system for reasoning about resource consumption, including time [8]. Hughes and Pareto proposed a type and effect system on space usage estimation based on the notion of sized types for a variant of ML such that well typed programs are proven to execute within the given memory bounds [20]. In these approaches, no effort is made to *infer* any bounds; instead they provide a mechanism for certifying the bounds once they are provided by the programmer. In contrast, our technique infers bounds. Hofmann and Jost statically infer linear bounds on heap space usage of first-order functional programs running under a special memory mechanism [19]. They use linear typing and an inference method through linear programming to derive these bounds. Their linear programming technique requires no fix-point analysis but it restricts the memory effects to a linear form without disjunction. In contrast, our focus is on computing time bounds (which are usually disjunctive and non-linear) for imperative programs.

***Worst-case Execution Time Analysis*** There is a large body of work on estimating worst case execution time (WCET) in the embedded and real-time systems community [28, 29]. The WCET research is more orthogonally focused on distinguishing between the complexity of different code-paths and low-level modeling of architectural features such as caches, branch prediction, instruction pipelines. For establishing loop bounds, WCET techniques either require user annotation, or use simple techniques based on pattern matching [18] or some simple numerical analysis. [16] describes an interval analysis based approach for automatic computation of loop bounds. However, it analyzes single-path executions of programs (i.e., using input data corresponding to one execution). Hence, their bounds are in real seconds, while our bounds are symbolic functions of inputs. [17] determines loop bounds in synchronous programs and linear hybrid systems by using a relational linear analysis to compute linear bounds on the delay or timer variables of the system. In contrast, our focus is on automatic computation of loop bounds (in general purpose programs), which are usually disjunctive, non-linear, and involve numerical properties of heap. None of the WCET techniques can automatically detect such bounds.

***Termination Techniques*** Recently, there have been some new advances in the area of proving termination of loops based on discovering disjunctively well-founded ranking functions [26] or lexicographic polyranking functions [4]. [6, 1] have successfully applied the fundamental result of [26] on disjunctively well-founded relations to prove termination of loops in real systems code. It may perhaps be possible to obtain bounds from certain kind of ranking functions given the initial state at the start of the loop. However, the ranking function abstraction is too weak to compute precise bounds. As illustrated in Introduction, programs with significantly varying computational complexity are proved terminating by [6, 1] because of the same ranking function, and hence there is no way to distinguish between the complexities of these procedures given the same termination proof. In contrast, our technique can generate more precise timing bounds.

***Symbolic Bound Generation*** Gulavani and Gulwani have described the design of a rich numerical domain to generate non-

linear disjunctive invariants [12], and they have applied it to generating bounds for timing analysis. However, their system requires the user to describe the set of *important expressions* (over which the linear relationships are to be tracked) as well as the set of variables that should occur within max operator for each loop. Furthermore, their technique only applies to arithmetic programs. However, their work can be better used as an instance of invariant generation tool within our framework (after extension with uninterpreted functions).

ACE analyzes a functional language FP to derive a time-complexity function [23] by repeatedly applying a large library of rewrite rules to transform the step-counting version of the original recursive program into a non-recursive bound function. However, bound generation is very sensitive to the order in which the rules are applied and the rules required to produce a bound are specific to a programming practice. Rosendahl describes a system to compute complexity bounds of programs written in a first-order subset of LISP using abstract interpretation [27]. His system outputs a time-bound program that is not guaranteed to be in closed form. In addition, the technique only works for programs where recursion is controlled by structural constraints, such as length of a list due to limitations of the analysis.

[11] computes symbolic bounds by curve-fitting timing data obtained from profiling. Their technique has the advantage of measuring real amortized complexity; however the results are not sound for all inputs.

[24] presents a formalism for user-defined resources on which bounds can be computed in terms of input data sizes. In contrast, we focus on computational-complexity bounds and present a formalism for user-defined input data-structure size measures for expressing computational-complexity bounds.

***Reducing pointers to integer programs*** Our notion of quantitative functions is related to recent work on reducing (primarily singly linked list-based) pointers to integer programs [10, 3, 22] after conducting alias analysis in a prepass. In comparison, our approach is limited in two ways. It applies only to abstract data-types, and aliasing has to be taken care of by the uninterpreted functions domain. However, our approach has two key advantages: It allows for arbitrary user annotations for any data-structure at run-time (as opposed to encoding the semantics of the integer variables for a specific data-structure inside the analysis). Furthermore, these annotations are not necessarily limited to pointer based data-structures, but also apply to say bit-vectors, which are otherwise quite complicated to reason about. Another technical advantage that our approach offers is that use of uninterpreted functions avoids the need for eagerly creating (several) integer variables (since congruence based data-structures can represent uninterpreted function terms with common sub-expressions succinctly).

## 9. Conclusion

This paper describes some fundamental principles for instrumenting monitor variables such that computing bounds on these monitor variables and appropriately composing them yields a bound on the total number of loop iterations. The importance of instrumenting multiple counter variables, each of which may be initialized and incremented at multiple locations, lies in being able to use linear invariant generation tools to compute precise bounds that are disjunctive as well as non-linear (which is usually the case in presence of control-flow inside loops). In particular, (a) We avoid the problem of generating disjunctive invariants by introducing multiple counters. (b) We avoid the problem of generating non-linear invariants by introducing dependencies between counters, i.e. initializing a counter at multiple locations corresponding to increment of other counters. (c) We address the issue of precision by introducing minimal number of counters with minimal number of dependencies.

The paper also introduces the notion of quantitative functions over abstract data-structures that allow a linear invariant generation tool with support for uninterpreted functions to compute bounds for loops that would otherwise require sophisticated shape analysis.

Based on these ideas, our tool is able to automatically generate (with minimal user annotation for providing quantitative functions) precise timing bounds for sophisticated real-life examples for which it is non-trivial to even prove termination.

## Acknowledgements

We thank Eric Koskinen for reading an earlier draft of this paper and providing useful comments. We thank Lakshmisubrahmanyam Velaga for implementing the abstract interpreter for uninterpreted functions, and a general combination framework for abstract interpreters, which was used as part of the SPEED tool.

## References

[1] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[2] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI'07*, pages 378–394.

[3] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.

[4] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, pages 1349–1361, 2005.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, pages 488–502, 2005.

[6] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[7] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL*, 1978.

[8] K. Crary and S. Weirich. Resource bound certification. In *POPL*, pages 184–198, 2000.

[9] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144, 2008.

[10] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.

[11] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, 2007.

[12] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384, 2008.

[13] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, 2009.

[14] S. Gulwani and G. C. Necula. A Polynomial-Time Algorithm for Global Value Numbering. In *SAS*, pages 212–227, 2004.

[15] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *PLDI*, pages 376–386, 2006.

[16] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, pages 57–66, 2006.

[17] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 11(2), '97.

[18] C. A. Healy, M. Sjodin, V. Rustagi, D. B. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[19] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[20] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in bounded space: Towards Embedded ML Programming. In *ICFP*, '99.

[21] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516, 2004.

[22] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.

[23] D. L. Métayer. Ace: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[24] J. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, pages 348–363, 2007.

[25] Microsoft Phoenix Compiler Infrastructure, http://research.microsoft.com/phoenix/.

[26] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, July 2004.

[27] M. Rosendahl. Automatic Complexity Analysis. In *FPCA*, pages 144–156, New York, NY, USA, 1989. ACM Press.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrm. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[29] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In *CAV*, pages 22–36, 2008.

## A. Proof of Theorem 3

PROOF: Let $(S, M, G, B)$ be any extensible partial-proof structure that can be extended to a proof structure $(S', M', G', B')$. Let $(S_1, M_1, G_1, B_1)$ be a partial-proof structure such that the triple $(S_1, M_1, G_1)$ has been obtained from $(S, M, G)$ by applying either a node merging operation or an edge deletion operation. We show below that in either case $(S_1, M_1, G_1, B_1)$ is an extensible partial-proof structure.
Let $P' = \texttt{Instrument}(P, (S', M', G'))$
and $P_1 = \texttt{Instrument}(P, (S_1, M_1, G_1))$.

- Consider the case when the triple $(S_1, M_1, G_1)$ is obtained from $(S, M, G)$ by merging two counter variables $c_1, c_2$ into $c_1$. Consider the following choice for $(S'_1, M'_1, G'_1, B'_1)$.

$$\begin{aligned} S'_1 &= S' - \{c_2\} \cup \{c'_1, c'_2\} \\ M'_1(q) &= M_1(q) \text{ if } M_1(q) \text{ is defined} \\ &= \sigma_1(M'(q)) \text{ otherwise} \\ G'_1 &= G_1 \cup \{(\sigma_1(c), \sigma_1(c')) \mid (c, c') \in G' - G\} \cup \\ &\qquad \{(\sigma_2(c), \sigma_2(c')) \mid (c, c') \in G' - G\} \\ B'_1(q) &= B_1(q) \text{ if } B_1(q) \text{ is defined} \\ &= B'(q) \text{ otherwise} \end{aligned}$$

  where $\sigma_1(c_1) = c'_1$, $\sigma_1(c_2) = c'_2$, and $\sigma_1(c) = c$ for any $c$ that is different from $c_1$ and $c_2$, $\sigma_2(c_2) = c_1$ and $\sigma_2(c) = c$ for any $c$ that is different from $c_2$.

  It is easy to see that $(S'_1, M'_1, G'_1, B'_1)$ is an extension of $(S_1, M_1, G_1, B_1)$. To show that $(S'_1, M'_1, G'_1, B'_1)$ is a proof structure, we need to show that for any back-edge $q$ in $P$, the invariant generation tool can produce a bound of $B'_1(q)$ on counter variable $M'_1(q)$ at back-edge $q$ in
  $P'_1 = \texttt{Instrument}(P, (S'_1, M'_1, G'_1))$. Consider any back-edge $q$. One of the following two cases arise.

  - $B_1(q)$ is defined. In this case, $M'_1(q) = M_1(q)$. Note that $P'_1$ initializes and increments the counter variable $M_1(q)$ at exactly the same locations (back-edges) where $P_1$ initializes and increments them respectively. Hence, $B'_1(q) = B_1(q)$.
  - $B_1(q)$ is not defined. In this case $M'_1(q) = \sigma_1(M'(q))$. Note that $P'_1$ initializes the counter variable $M'_1(q)$ at a super-set of the locations where $P'$ initializes $M'(q)$, and $P'_1$ increments the counter variable $M'_1(q)$ at a sub-set of the locations (back-edges) where $P'$ increments $M'(q)$. Hence, the invariant generation tool will be able to compute a bound that is at least as good as $B'(q)$ on counter variable $M'_1(q)$ at back-edge $q$.

- Consider the case when the triple $(S_1, M_1, G_1)$ is obtained from $(S, M, G)$ by deleting an edge $(c_1, c_2)$. Consider the following choice for $(S'_1, M'_1, G'_1, B'_1)$.

$$\begin{aligned} S'_1 &= S' \cup \{c'_2\} \\ M'_1(q) &= M_1(q) \text{ if } M_1(q) \text{ is defined} \\ &= \sigma_1(M'(q)) \text{ otherwise} \\ G'_1 &= G_1 \cup \{(\sigma_1(c), \sigma_1(c')) \mid (c, c') \in G' - G\} \cup \\ &\qquad \{(c, c') \mid (c, c') \in G' - G\} \\ B'_1(q) &= B_1(q) \text{ if } B_1(q) \text{ is defined} \\ &= B'(q) \text{ otherwise} \end{aligned}$$

  where $\sigma_1(c_2) = c'_2$ and $\sigma_1(c) = c$ for any $c$ that is different from $c_2$.

  It is easy to see that $(S'_1, M'_1, G'_1, B'_1)$ is an extension of $(S_1, M_1, G_1, B_1)$. To show that $(S'_1, M'_1, G'_1, B'_1)$ is a proof-structure, we need to show that for any back-edge $q$ in $P$, the invariant generation tool can produce a bound of $B'_1(q)$ on counter variable $M'_1(q)$ at back-edge $q$ in
  $P'_1 = \texttt{Instrument}(P, (S'_1, M'_1, G'_1))$. The proof is now exactly same as the above case.

□

## B. Bit-Vectors

Bit-vectors can be associated with a few quantitative functions that are useful to express and compute the complexity of loops that iterate over bit-vectors. These quantitative functions are:

- `Bits`(b): Number of bits in b.
- `Ones`(b): Number of 1 bits in b.
- `One`(b): Position of the first 1 bit in $b$ starting from right and counting from zero, (i.e., if there is any 1 bit in $b$. Otherwise, `One`$(b)$ is don't care.)
- `LastOne`$(b)$: Position of the first 1 bit in $b$ starting from left and counting from zero, if there is any 1 bit in $b$ (i.e., if there is any 1 bit in $b$. Otherwise, `LastOne`$(b)$ is don't care.)

Similarly, we can also define the quantitative functions `Zeros`$(b)$, `Zero`$(b)$, and `LastZero`$(b)$.

For simplicity, we assume that all bit-vectors have the same number of bits. Table (a) in Figure 10 describes how somes of these quantitative functions are affected by standard bit-vector operations. The function $\_BitScanForward(\&id, b)$ returns `true` iff the bit-vector $b$ contains a 1 bit, in which case it sets $id$ to the position of the first 1 bit from the right side, and counting from zero. The integer $index$ by which a bit-vector is left-shifted in $a := b << index$ should be non-negative. The other bit-vector operations are self-explanatory.

Table (b) in Figure 10 shows some examples of looping patterns over bit-vectors from Microsoft product code. The loops in examples 1 and 2 iterate over a bit-vector $b$ by setting the least significant

| Bit-vector Operation | Effect on Quantitative Functions |
|---|---|
| $a := \sim b$ | $\texttt{Ones}(a) := \texttt{Zeros}(b)$; $\texttt{One}(a) := \texttt{Zero}(b)$; $\texttt{Zero}(a) := \texttt{One}(b)$; |
| $res :=$ _BitScanForward$(\&index, b)$ | $index :=?$; $\texttt{Assume}(res = \texttt{true} \Rightarrow (\texttt{Ones}(b) = 0 \ \wedge \ \texttt{Zero}(b) = 0))$; <br> $\texttt{Assume}(res = \texttt{false} \Rightarrow (index = \texttt{One}(b) \ \wedge \ \texttt{Ones}(b) > 0 \ \wedge \ 0 \leq \texttt{One}(b) < \texttt{Bits}(b)))$; |
| $a := b << index$ | $\texttt{One}(a) := index + \texttt{One}(b)$; $\texttt{Ones}(a) :=?$; $\texttt{Assume}(\texttt{Ones}(a) \leq \texttt{Ones}(b))$; <br> if $(index > 0)$ $\texttt{Zero}(a) := 0$; |
| $a := b - 1$ | $\texttt{One}(a) :=?$; $\texttt{Ones}(a) :=?$; $\texttt{Zero}(a) := \texttt{One}(b)$; |
| $t := (b \neq 0)$ | $\texttt{Assume}(t = \texttt{true} \Rightarrow (0 \leq \texttt{One}(b) < \texttt{Bits}(b) \ \wedge \ \texttt{Ones}(b) \geq 1))$; <br> $\texttt{Assume}(t = \texttt{false} \Rightarrow (\texttt{Ones}(b) = 0 \ \wedge \ \texttt{Zero}(b) = 0))$; |
| $a := b \ \& \ c$ | $\texttt{Ones}(a) :=?$; $\texttt{One}(a) :=?$; $\texttt{Zero}(a) :=?$; <br> $\texttt{Assume}(\texttt{Zero}(c) = \texttt{One}(b) \Rightarrow \texttt{Ones}(a) \leq \texttt{Max}(0, \texttt{Ones}(b) - 1) \wedge \texttt{One}(a) > \texttt{One}(b))$; <br> $\texttt{Assume}(\texttt{One}(c) = \texttt{Zero}(b) \Rightarrow \texttt{Ones}(a) \leq \texttt{Max}(0, \texttt{Ones}(c) - 1) \wedge \texttt{One}(a) > \texttt{One}(c))$; <br> $\texttt{Assume}(\texttt{Ones}(a) \leq \texttt{Ones}(b) \ \wedge \ \texttt{Ones}(a) \leq \texttt{Ones}(c)) \ \wedge \ \texttt{One}(a) \geq \texttt{One}(b) \ \wedge \ \texttt{One}(a) \geq \texttt{One}(c))$; <br> $\texttt{Assume}(\texttt{Zero}(a) \leq \texttt{Zero}(b) \ \wedge \ \texttt{Zero}(a) \leq \texttt{Zero}(c))$; |
| $a := b \mid c$ | $\texttt{Ones}(a) :=?$; $\texttt{One}(a) :=?$; $\texttt{Zero}(a) :=?$; <br> $\texttt{Assume}(\texttt{One}(c) = \texttt{Zero}(b) \Rightarrow \texttt{Zero}(a) > \texttt{Zero}(b))$; <br> $\texttt{Assume}(\texttt{Zero}(c) = \texttt{One}(b) \Rightarrow \texttt{Zero}(a) > \texttt{Zero}(c))$; <br> $\texttt{Assume}(\texttt{Ones}(a) \geq \texttt{Ones}(b) \ \wedge \ \texttt{Ones}(a) \geq \texttt{Ones}(c)) \ \wedge \ \texttt{One}(a) \leq \texttt{One}(b) \ \wedge \ \texttt{One}(a) \leq \texttt{One}(c))$; <br> $\texttt{Assume}(\texttt{Zero}(a) \geq \texttt{Zero}(b) \ \wedge \ \texttt{Zero}(a) \geq \texttt{Zero}(c))$; |

**(a) Semantics** of Quantitative Functions One, Ones, and Zero.

| | Some looping patterns over bit-vectors (from Microsoft product code) | Loop Invariant | Complexity |
|---|---|---|---|
| 1. | for $(b := a;$ _BitScanForward$(\&id, b); b := b\&(\sim (1 << id)))$; | $c \leq 1 + \texttt{Ones}(a) - \texttt{Ones}(b)$ | $\texttt{Ones}(a)$ |
| 2. | for $(b := a; b \neq 0; b := b\&(b - 1))$; | $c \leq 1 + \texttt{Ones}(a) - \texttt{Ones}(b)$ | $\texttt{Ones}(a)$ |
| 3. | for $(b := a; b \neq 0; b := b << 1)$; | $c \leq 1 + \texttt{One}(b) - \texttt{One}(a)$ | $\texttt{Bits}(a) - \texttt{One}(a)$ |
| 4. | for $(b := a;$ _BitScanForward$(\&id1, b);)$ { <br> $b := b \mid ((1 << id1) - 1)$; // set all bits before id1 <br> if _BitScanForward$(\&id2, \sim b)$ break; <br> $b := b \ \& \ (\sim ((1 << id2) - 1))$; // reset all bits before id2 <br> }; | $2c \leq 1 + \texttt{One}(b) - \texttt{One}(a) \wedge$ <br> $c \leq 1 + \texttt{Ones}(a) - \texttt{Ones}(b)$ | $\texttt{Min}\{(\texttt{Bits}(a) - \texttt{One}(a))/2,$ <br> $\texttt{Ones}(a)\}$ |

**(b) Examples**

**Figure 10.** Column 1 contains some examples of bit-vector looping patterns from Microsoft product code. Column 2 describes the (interesting part of the) loop invariant, as computed by our tool, that relates an instrumented loop counter variable $c$ with some quantitative attributes of the bit-vectors modified in the loop. Column 3 shows an upper bound on the number of loop iterations in terms of the input bit-vector $a$.

1 bit to 0. Example 3 iterates over a bit vector $b$ by left shifting the bit-sequence by one bit. Example 4 iterates over a bit-vector $b$ by setting the first chunk of all consecutive 1 bits (from the right side) in $b$ to 0. Observe that an upper bound on the number of iterations of such a loop can be described in two orthogonal ways using the quantitative attributes that we have introduced. Clearly $\texttt{Ones}(a)$ is an upper bound, but so is $(\texttt{Bits}(a) - \texttt{One}(a))/2$ because note that the least significant one bit moves by at least 2 steps in each loop iteration.

The quantitative functions $\texttt{LastOne}(b)$ and $\texttt{LastZero}(b)$ are useful, for example, when the looping pattern involves right shifting the bits (as opposed to left shifting).

## C. Trees

The following quantitative functions are useful to express and compute bounds on loops that iterate over trees.

- $\texttt{Height}(T)$: Height of tree $T$.
- $\texttt{Nodes}(T)$: Total number of nodes in tree $T$.
- $\texttt{Height}(e, T)$: Height of node (or tree-element) $e$ in tree $T$ (i.e., if $e$ belongs to $T$; otherwise it is don't care).
- $\texttt{Nodes}(e, T)$: Total number of nodes below $e$ in tree $T$. (i.e., if $e$ belongs to $T$; otherwise it is don't care).

The table in Figure 11(a) describes the effect of standard tree methods on some of these quantitative functions. The method $T.Insert(e)$ has the precondition that $e$ does not already belong to $T$. The method $T.Remove(e)$ has the precondition that $e$ belongs to $T$. The Table (b) in Figure 11(b) shows some examples of looping patterns over trees as taken from C++ STL library code.

| | Tree Operation | Effect on Quantitative Functions |
|---|---|---|
| 1. | $e := T.\texttt{Root}()$ | $\texttt{Height}(e) := \texttt{Height}(T)$; $\texttt{Nodes}(e) := \texttt{Nodes}(T)$; |
| 2. | $e_1 := T.\texttt{GetRight}(e_2)$ | $\texttt{Height}(e_1) :=?$; $\texttt{Assume}(\texttt{Height}(e_1) \leq \texttt{Height}(e_2) - 1)$;<br>$\texttt{Nodes}(e_1) := \texttt{Nodes}(e_2) - \texttt{Nodes}(T.\texttt{GetLeft}(e_2)) - 1$; |
| 3. | $e_1 := T.\texttt{GetLeft}(e_2)$ | $\texttt{Height}(e_1) :=?$; $\texttt{Assume}(\texttt{Height}(e_1) \leq \texttt{Height}(e_2) - 1)$;<br>$\texttt{Nodes}(e_1) := \texttt{Nodes}(e_2) - \texttt{Nodes}(T.\texttt{GetRight}(e_2)) - 1$; |
| 4. | $T.\texttt{Insert}(e)$ | $\texttt{Nodes}(T) := \texttt{Nodes}(T) + 1$; if (*) $\texttt{Height}(T) := \texttt{Height}(T) + 1$;<br>if (*) $\texttt{Height}(e', T) := \texttt{Height}(e', T) + 1$; if (*) $\texttt{Nodes}(e', T) := \texttt{Nodes}(e', T) + 1$;<br>$\texttt{Height}(e, T) :=?$; $\texttt{Assume}(1 \leq \texttt{Height}(e, T) \leq \texttt{Height}(T))$;<br>$\texttt{Nodes}(e, T) :=?$; $\texttt{Assume}(1 \leq \texttt{Nodes}(e, T) \leq \texttt{Nodes}(T))$; |
| 5. | $T.\texttt{Remove}(e)$ | $\texttt{Nodes}(T) := \texttt{Nodes}(T) - 1$; if (*) $\texttt{Height}(T) := \texttt{Height}(T) - 1$;<br>if (*) $\texttt{Height}(e', T) := \texttt{Height}(e', T) - 1$; if (*) $\texttt{Nodes}(e', T) := \texttt{Nodes}(e', T) - 1$;<br>$\texttt{Height}(e, T) :=?$; $\texttt{Assume}(1 \leq \texttt{Height}(e, T) \leq \texttt{Height}(T))$;<br>$\texttt{Nodes}(e, T) :=?$; $\texttt{Assume}(1 \leq \texttt{Nodes}(e, T) \leq \texttt{Nodes}(T))$; |

**(a) Semantics** of Quantitative Functions `Nodes` and `Height` for both trees and tree-elements.

| | Some recursive calling patterns over trees (from C++ STL library code) | Procedure Summary | Complexity |
|---|---|---|---|
| 1. | $\texttt{Traverse}(\texttt{Tree}\ T, \texttt{TreeElt}\ e)$ {<br>if $(e \neq \texttt{null})$<br>$\texttt{Traverse}(T.\texttt{GetLeft}(e))$;$\texttt{Traverse}(T.\texttt{GetRight}(e))$;<br>} | $c^{out} - c^{in} \leq 1 + 2(\texttt{Nodes}(e, T))$ | $1 + 2(\texttt{Nodes}(e, T))$ |
| 2. | $\texttt{Search}(\texttt{Tree}\ T, \texttt{TreeElt}\ e)$ {<br>if $(e == \texttt{null})$ return `false`;<br>if $(e.\texttt{data} > elt)$ return $\texttt{Search}(T.\texttt{GetRight}(e))$<br>else if $(e.\texttt{data} < elt)$ return $\texttt{Search}(T.\texttt{GetLeft}(e))$<br>else return `true`<br>} | $c^{out} - c^{in} \leq 1 + \texttt{Height}(e, T)$ | $1 + \texttt{Height}(e, T)$ |
| 3. | $\texttt{Traverse2}(\texttt{Tree}\ T, \texttt{TreeElt}\ e)$ {<br>for $(y := e; y \neq \texttt{null}; y := T.\texttt{GetLeft}(e))$<br>$\texttt{Traverse2}(T.\texttt{GetRight}(e))$<br>} | $c^{out} - c^{in} \leq 2(1 + \texttt{Nodes}(e, T))$ | $2(1 + \texttt{Nodes}(e, T))$ |

**(b) Examples**

**Figure 11.** In Table(b), Column 1 contains some examples of looping patterns over trees from C++ STL library code base. Column 2 describes the procedure summary computed by our invariant generation tool that relates the values $c^{in}$ and $c^{out}$ of the instrumented global counter variable $c$ (instrumented as per the strategy described in Section 6) at procedure entry and exit respectively with appropriate quantitative attributes. Column 3 shows an upper bound on the number of recursive procedure call invocations in terms of quantitative functions of the inputs.