Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints

Moritz Sinn¹ · Florian Zuleger¹ · Helmut Veith¹

Received: 28 November 2016 / Accepted: 29 November 2016 / Published online: 11 January 2017
© The Author(s) 2017. This article is published with open access at Springerlink.com

Abstract  Difference constraints have been used for termination analysis in the literature, where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of $x$ in the current state is at most the value of $y$ in the previous state plus some constant $c \in \mathbb{Z}$. We believe that difference constraints are also a good choice for complexity and resource bound analysis because the complexity of imperative programs typically arises from counter increments and resets, which can be modeled naturally by difference constraints. In this article we propose a bound analysis based on difference constraints. We make the following contributions: (1) our analysis handles bound analysis problems of high practical relevance which current approaches cannot handle: we extend the range of bound analysis to a class of challenging but natural loop iteration patterns which typically appear in parsing and string-matching routines. (2) We advocate the idea of using bound analysis to infer invariants: our soundness proven algorithm obtains invariants through bound analysis, the inferred invariants are in turn used for obtaining bounds. Our bound analysis therefore does not rely on external techniques for invariant generation. (3) We demonstrate that difference constraints are a suitable abstract program model for automatic complexity and resource bound analysis: we provide efficient abstraction techniques for obtaining difference constraint programs from imperative code. (4) We report on a thorough experimental comparison of state-of-the-art bound analysis tools: we set up a tool comparison on (a) a large benchmark of real-world

Electronic supplementary material  The online version of this article (doi:10.1007/s10817-016-9402-4) contains supplementary material, which is available to authorized users.

Moritz Sinn
sinn@forsyte.at

Florian Zuleger
zuleger@forsyte.at

¹ Institut für Informationssysteme, TU Wien, Favoritenstr. 9-11, Wien, Austria
C code, (b) a benchmark built of examples taken from the bound analysis literature and (c) a benchmark of challenging iteration patterns which we found in real source code. (5) Our analysis is more scalable than existing approaches: we discuss how we achieve scalability.

**Keywords** Bound analysis · Complexity analysis · Amortized analysis · Difference constraints · Static analysis · Resource bound analysis · Automatic complexity analysis · Cost analysis

### 1 Introduction

Automated program analysis for inferring program complexity and resource bounds is a very active area of research. Amongst others, approaches have been developed for analyzing functional programs [16], C# [15], C [2,7,29,35], Java [1] and Integer Transition Systems [6, 10]. Below we sketch applications in the areas of verification and program understanding. For additional motivation we refer the reader to the cited papers.

**Verification** In many applications such as embedded systems there is a hard constraint on the availability of resources such as CPU time, memory, bandwidth, etc. It is an important part of functional correctness that programs stay within their given resource limits. As a concrete example we mention that considerable effort has been invested to analyze the worst case execution time (WCET) of hard real-time systems [33]. Another application domain is security, where the goal is to derive a bound on how much secret information is leaked in order to decide whether this leakage is acceptable [31].

**Static Profiling and Program Understanding** Standard profilers report numbers such as how often certain program locations are visited and how much time is spent inside certain functions; however, no information is provided how these numbers are related to the program input. Recently, new profiling approaches have been proposed that apply curve fitting techniques for deriving a cost function, which relates size measures on the program input to the measured program performance [9,34]. We believe that automated complexity and resource bound analysis lends itself naturally as static profiling technique, because it provides the user with a symbolic expression that relates the program performance to the program input. In the same way, complexity and resource bound analysis can be used to explore unfamiliar code or to annotate library functions by their performance characteristics; we note that a substantial number of performance bugs can be attributed to a “wrong understanding of API performance features” [22].

As a final remark we discuss the relationship to termination analysis, which has been intensively studied in the last decade in the computer-aided verification community: complexity and resource bound analysis can be understood as a quantitative variant of termination analysis, where not only a qualitative “yes” answer is provided, but also a symbolic upper bound on the run-time of the program.

**Difference constraints (DCs)** have been introduced by Ben-Amram for termination analysis in [4], where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of $x$ in the current state is at most the value of $y$ in the previous state plus some constant $c \in \mathbb{Z}$. We call a program whose transitions are given by a set of difference constraints a difference constraint program (DCP).

We advocate the use of DCs for program complexity and resource bound analysis. Our key insight is that DCs provide a natural abstraction of the standard manipulations of counters in imperative programs: counter increments and decrements, i.e., $x := x + c$ resp. resets,
i.e., $x := y$, can be modeled by the $DCsx' \leq x + c$ resp. $x' \leq y$ (see Sect. 6 on program abstraction). The approach we discuss in this article exploits the expressive strength of DCs and distinguishes between counter resets and counter increments in the reasoning. In contrast, previous approaches [1,2,6,10,15,29,35] to bound analysis are not able to track increments and resets on the same level of precision and therefore often fail to infer tight bounds for a class of nested loop constructs which we identified during our experiments on real-world code (demonstrated by our experimental evaluation in Sect. 8.3). In this article we make the following contributions:

1. Our analysis handles bound analysis problems of high practical relevance which current approaches cannot handle: we extend the range of bound analysis to a class of challenging but natural loop iteration patterns which typically appear in parsing and string-matching routines as we discuss in Sect. 2. At the same time our analysis is general and can handle most of the bound analysis problems which are discussed in the literature. Both claims are supported by our experiments.

2. We advocate the idea of using bound analysis to infer invariants: we state a clear and concise formulation of invariant analysis by bound analysis on base of our abstract program model: our soundness proven algorithm (Sect. 3) obtains invariants through bound analysis, the inferred invariants are in turn used for obtaining bounds. Our bound analysis therefore does not rely on external techniques for invariant generation.

3. We demonstrate that difference constraints are a suitable abstract program model for automatic complexity and resource bound analysis: we develop appropriate techniques for abstracting imperative programs to DCPs in Sect. 6.

4. We report on a thorough experimental comparison of state-of-the-art bound analysis tools (Sect. 8): we set up a tool comparison on (a) a large benchmark of real-world C-code (Sect. 8.1), (b) a benchmark built of examples taken from the bound analysis literature (Sect. 8.2) and (c) a benchmark of challenging iteration patterns which we found in real source code (Sect. 8.3).

5. We have designed our analysis with the goal of scalability: our experiments demonstrate that our implementation outperforms the state-of-the-art with respect to scalability. We give a detailed discussion on how we achieve scalability in Sect. 10.

This article is an extension of the conference version presented at FMCAD 2015 [27]. Besides making the material more accessible through additional explanations and discussions, it adds the following contributions: (1) a discussion on the instrumentation of our analysis for resource bound analysis (Sect. 2.2). (2) A more detailed discussion and presentation of our context-sensitive bound algorithm (Sect. 3.3). (3) A more detailed discussion on how we determine local bounds and an extension to sets of local bounds (Sect. 4). (4) A complete example (Sect. 7). (5) A discussion on the relation to amortized complexity analysis (Sect. 9). (6) Additional experimental results (Sects. 8.2 and 8.3). (7) In Electronic Supplementary Material we state the soundness proofs omitted in the conference version.

2 Motivation and Related Work

Example xnusSimple stated in Fig. 1 is representative for a class of loops that we found in parsing and string matching routines during our experiments. In these loops the inner loop iterates over disjoint partitions of an array or string, where the partition sizes are determined by the program logic of the outer loop. For an illustration of this iteration scheme see Example xnua in Fig. 9 (Sect. 7), which contains a snippet of the source code after which we have
modeled Example xnuSimple. Example xnuSimple has the linear complexity \( 2n \) (we define complexity here as the total number of loop iterations, for alternative definitions see the discussion in Sect. 2.2), because the inner loop as well as the outer loop can be iterated at most \( n \) times (as argued in the next paragraph). In the following, we give an overview how our approach infers the linear complexity for Example xnuSimple:

1. **Program Abstraction** We abstract the program to a DCP over \( \mathbb{N} \) as shown in Fig. 1. The abstract variable \([x]\) represents the program expression \(\max(x, 0)\). We discuss our algorithm for abstracting imperative programs to DCPs based on symbolic execution in Sect. 6.

2. **Finding Local Bounds** We identify \([p]\) as a variable that limits the number of executions of transition \(\tau_3\): we have that \([p]\) decreases on each execution of \(\tau_3\) (\([p]\) takes values over \(\mathbb{N}\)). We call \([p]\) a local bound for \(\tau_3\). Accordingly we identify \([x]\) as a local bound for the transitions \(\tau_1, \tau_2, \tau_4, \tau_5, \tau_6\).

3. **Bound Analysis** Our algorithm (stated in Sect. 3) computes transition bounds, i.e., (symbolic) upper bounds on the number of times program transitions can be executed, and variable bounds, i.e., (symbolic) upper bounds on variable values. For both types of bounds, the main idea of our algorithm is to reason how much and how often the value of the local bound resp. the variable value may increase during program run. Our algorithm is based on a mutual recursion between variable bound analysis (“how much”, function \(VB(\nu)\)) and transition bound analysis (“how often”, function \(TB(\tau)\)). Next, we give an intuition how our algorithm computes transition bounds: for \(\tau \in \{\tau_1, \tau_2, \tau_4, \tau_5, \tau_6\}\) our algorithm computes \(TB(\tau) = [n] = n\) (note that \([n] = n\) because \(n\) has type unsigned) because the local bound \([x]\) is initially set to \([n]\) and never increased or reset. Our algorithm computes \(TB(\tau_3)\) (\(\tau_3\) corresponds to the loop at \(l_3\)) as follows: \(\tau_3\) has local bound \([p]\); \([p]\) is reset to \([r]\) on \(\tau_2\); our algorithm detects that before each execution of \(\tau_2\), \([r]\) is reset to \([0]\) on either \(\tau_0\) or \(\tau_4\), which we call the context under which \(\tau_2\) is executed; our algorithm establishes that between being reset and flowing into \([p]\) the value of \([r]\) can be incremented up to \(TB(\tau_1)\) times by 1; our algorithm obtains \(TB(\tau_1) = n\) by a recursive call; finally, our algorithm calculates \(TB(\tau_3) = [0] + TB(\tau_1) \times 1 = 0 + n \times 1 = n\). We give an example for the mutual recursion between \(TB\) and \(VB\) in Sect. 2.1.

![Fig. 1 Running Example xnuSimple, the symbol asterisk denotes non-determinism (arising from conditions not modeled in the analysis)](image-url)
void twoSCCs(uint n, uint m1, uint m2) {
    int y = n;
    int x;
    l_1 if(*)
        x = m1;
    else
        x = m2;
    l_2 while(y > 0) {
        y--;
        x = x + 2;
    }
    int y = x;
    l_3 while(z > 0) {
        z--;
    }
}

Complexity: \( TB(\tau_3) + TB(\tau_5) = n + \max(m_1, m_2) + 2n = \max(m_1, m_2) + 3n \)

Example twoSCCs

abstracted DCP of twoSCCs

Fig. 2 Running Example twoSCCs

2.1 Invariants and Bound Analysis

We motivate the need for invariants in bound analysis and sketch how our algorithm infers invariants by bound analysis. Consider Example twoSCCs in Fig. 2. It is easy to infer \( x \) as a bound for the possible number of iterations of the loop at \( l_3 \). However, in order to obtain a bound in the function parameters the difficulty lies in finding an invariant of form \( x \leq \exp(x(n, m_1, m_2)) \), where \( \exp(x(n, m_1, m_2) \) denotes an expression over the function parameters \( n, m_1, m_2 \). We show how our algorithm obtains the invariant \( x \leq \max(m_1, m_2) + 2n \) by means of bound analysis:

Our algorithm computes a transition bound for the loop at \( l_3 \) (with the single transition \( \tau_3 \)) by \( TB(\tau_3) = TB(\tau_4) \times VB([x]) = 1 \times VB([x]) = VB([x]) = TB(\tau_3) \times 2 + \max([m_1], [m_2]) = (TB(\tau_0) \times [n]) \times 2 + \max([m_1], [m_2]) = (1 \times [n]) \times 2 + \max([m_1], [m_2]) = 2n + \max(m_1, m_2) \) (note that \([n] = n, [m_1] = m_1 \) and \([m_2] = m_2 \) because \( n, m_1, m_2 \) have type unsigned). We point out the mutual recursion between \( TB \) and \( VB \): \( TB(\tau_5) \) has called \( VB(x) \), which in turn called \( TB(\tau_3) \). We highlight that the variable bound \( VB(x) \) (corresponding to the invariant \( x \leq \max(m_1, m_2) + 2n \)) has been established during the computation of \( TB(\tau_5) \).

We call the kind of invariants that our algorithm infers upper bound invariants (Definition 6). We compare our reasoning to classical invariant analysis in Sect. 2.3.

2.2 Resource Bound Analysis

We shortly discuss how resource bound analysis can be naturally formulated within our framework. We introduce a fresh variable \( c \) and add the initialization \( c = 0 \) to the beginning of the program under scrutiny. We add an increment/decrement \( c = c + k \) at program locations where a resource of cost \( k \) is consumed (\( k \) is positive) or freed (\( k \) is negative). Resource bound analysis is then equivalent to computing an upper bound on the value of the variable \( c \). We can run our algorithm \( VB(c) \) to compute a symbolic upper bound for \( c \).

In the same way we can encode related bound analysis problems: reachability bounds [15] (visits to a single location), visits to multiple transitions, loop bounds or complexity analysis.
For each of these bound analysis problems one can add a counter increment at the program locations of interest.

We illustrate the suggested encoding on the problem of computing loop bounds: for a given loop we add increments of the counter variable \( c \) to every back edge of the loop. Calling \( VB(c) \) then returns the sum of the transition bounds of all back edges of the loop. This example also illustrates how transition bounds are used for computing variable bounds in our approach.

### 2.3 Related Work

**Termination** In [4] it is shown that termination of DCPs is undecidable in general but decidable for the natural syntactic subclass of fan-in free DCPs (see Definition 12), which is the class of DCPs we use in this article. It is an open question for future work whether there is a complete algorithm for bound analysis of fan-in free DCPs.

**Bound Analysis** In [35] a bound analysis based on so-called size-change constraints \( x' \prec y \) is proposed, where \( \prec \in \{<, \leq\} \). Size-change constraints form a strict syntactic subclass of DCs. However, termination is decidable even for size-change programs that are not fan-in free and a complete algorithm for deciding the complexity of size-change programs has been developed [8]. For reasoning about inner loops [35] computes disjunctive loop summaries while such summaries are not computed by the approach discussed in this work.

In [29] a bound analysis based on constraints of the form \( x' \leq x + c \) is proposed, where \( c \) is either an integer or a symbolic constant. Because the constraints in [29,35] cannot model both increments and resets, the resulting bound analyses cannot infer the linear complexity of Example \( \text{nuSimple} \) and need to rely on external techniques for invariant analysis.

The COSTA project (e.g. [1]) obtains recurrence relations from so-called cost equations using invariant analysis based on the polyhedra abstract domain and approaches from the literature for synthesizing linear ranking functions. Closed-form solutions for the obtained recurrence relations are inferred by means of computer algebra.

The technique discussed in [10] is based on the COSTA approach and formulated in terms of cost equations. Further, paper [10] is inspired by the counter instrumentation-based approach [14] and applies the techniques [3,25] for inferring linear ranking functions. The technique of [10] achieves a high precision of the inferred bounds by means of control-flow refinement (see also Ref. [11]).

The technique discussed in [2] over-approximates the reachable states by abstract interpretation based on the polyhedra abstract domain. This information is used for generating a linear constraint problem from which a multi-dimensional linear ranking function is obtained. A bound on the number of values which can be taken by the ranking function is then obtained from the previously computed approximation of the reachable states. Importantly, the number of dimensions of the ranking function determines the degree of the bound polynomial. The approach of [2] therefore aims at inferring a ranking function with a minimal number of dimensions and thus depends on a minimal solution to the linear constraint problem which is obtained by linear optimization (Technique [2] instruments the LP-solver with an objective function).

The technique discussed in [6] applies approaches from the literature for synthesizing ranking functions thereby inferring bounds on the number of times the execution of isolated program parts can be repeated. These bounds, called time bounds, are then used to compute bounds on the absolute value of variables, so-called variable size bounds. Additional information is inferred through abstract interpretation based on the octagon abstract domain. An
overall complexity bound is deduced by alternating between time bound and variable size bound analysis. In each alternation bounds for larger program parts are obtained based on the previously computed information.

In Sect. 8 we compare our implementation against the techniques [2, 6, 7, 10, 29].

Amortized Complexity Analysis We note that inferring the linear complexity $2n$ for Example xnuSimple, even though the inner loop can already be iterated $n$ times within one iteration of the outer loop, is an instance of amortized complexity analysis [32]: the cost of executing the inner loop, averaged over all $n$ iterations of the outer loop is 1. Most previous approaches [1, 6, 10, 15, 29, 35] can establish only a quadratic bound for Example xnuSimple. A typical reasoning which fails to establish the linear complexity of Example xnuSimple is as follows: (1) the outer loop can be iterated at most $n$ times, (2) the inner loop can be iterated at most $n$ times within one iteration of the outer loop (because the inner loop has a local loop bound $p$ and $p \leq n$ is an invariant), (3) the loop bound $n^2$ is obtained from (1) and (2) by multiplication.

The recent paper [7] discusses an interesting alternative for amortized complexity analysis of imperative programs: a system of linear inequalities is derived using Hoare-style proof-rules. Solutions to the system represent valid linear resource bounds. Since bound analysis typically does not aim at some bound but tries to infer a tight bound, Ref. [7] uses linear optimization (an LP-solver instrumented by an objective function) in order to obtain a minimum solution to the problem. Interestingly, Ref. [7] is able to compute the linear bound for $l_3$ of Example xnuSimple but fails to deduce the bound for the original source code (discussed in Sect. 7). Moreover, Ref. [7] is restricted to linear bounds, while our approach derives bounds which are polynomial (see, e.g., the results in Table 12) and which contain the maximum operator (e.g., Example twoSCCs). We compare our implementation to the implementation of Ref. [7] in Sect. 8.

Invariants and Bound Analysis The powerful idea of expressing locally computed bounds in terms of the function parameters by alternating between bound analysis and variable upper bound analysis has previously been applied in [6, 12, 28]. Since Refs. [12, 28] do not give a general algorithm but deal with specific cases we focus our discussion on [6] and highlight some important differences. The technique discussed in [6] computes upper bound invariants only for the absolute values of variables; for many cases, this does not allow to distinguish between variable increments and decrements: consider the program foo(int x, int y) {while(y > 0) {x--; y--}; while(x > 0) x--;}.

The algorithm described in [6] infers the bound $|x| + |y|$ for the second loop, whereas our analysis infers the bound $\max(x, 0)$. The approach of [6] depends on global invariant analysis. E.g., given a decrement $x := x - 1$, the technique of [6] needs to check whether $x \geq 0$ holds. If $x \geq 0$ cannot be ensured, the decrement can actually increment the absolute value of $x$, and will thus be interpreted as $|x| = |x| + 1$. This can either lead to gross over-approximations or failure of bound computation if the increment of $|x|$ cannot be bounded. Since our approach does not track the absolute value but the value, it is not concerned with this problem. The technique discussed in [6] does not support amortized analysis: e.g., The technique [6] fails to compute the linear bounds for Example xnuSimple (Fig. 1), Example xnu (Fig. 9) and other examples we discuss in this article (see also the results in Sect. 8.3). On the other hand, Ref. [6] can infer bounds for functions with multiple recursive calls which is not supported by the analysis we present in this article.

Comparison to Invariant Analysis We contrast our previously discussed approach for computing a bound for the loop at $l_3$ of Example xnuSimple with classical invariant analysis:
assume that we have added a counter $c$ which counts the number of inner loop iterations (i.e., $c$ is initialized to 0 and incremented in the inner loop). For inferring $c \leq n$ through invariant analysis the invariant $c + x + r \leq n$ is needed for the outer loop, and the invariant $c + x + p \leq n$ for the inner loop. Both relate 3 variables and cannot be expressed as (parametrized) octagons (e.g., [26]). Further, the expressions $c + x + r$ and $c + x + p$ do not appear in the program, which is challenging for template based approaches to invariant analysis.

We now contrast our variable bound analysis (function $V\mathit{B}$) with classical invariant analysis: reconsider Example twoSCCs in Fig. 2. We have discussed how our algorithm obtains the invariant $x \leq \max(m_1, m_2) + 2n$ by means of bound analysis in the course of computing a bound for the loop at $l_3$. Note, that the invariant $x \leq \max(m_1, m_2) + 2n$ cannot be computed by standard abstract domains such as octagon or polyhedra; these domains are convex and cannot express non-convex relations such as maximum. The most precise approximation of $x$ in the polyhedra domain is $x \leq m_1 + m_2 + 2n$. Unfortunately, it is well-known that the polyhedra abstract domain does not scale to larger programs and needs to rely on heuristics for termination. Standard abstract domains such as octagon or polyhedra propagate information forward until a fixed point is reached, greedily computing all possible invariants expressible in the abstract domain at every location of the program. In contrast, our method $V\mathit{B}(x)$ infers the invariant $x \leq \max(m_1, m_2) + 2n$ by modular reasoning: local information about the program (i.e., local bounds and increments/resets of variables) is combined to a global program property. Moreover, our variable and transition bound analysis is demand-driven: our algorithm performs only those recursive calls that are indeed needed to derive the desired bound. We believe that our analysis complements existing techniques for invariant analysis and will find applications outside of bound analysis.

3 Program Model and Algorithm

In this section we present our algorithm for computing worst-case upper bounds on the number of executions of a given transition (transition bound) and on the value of a given program expression (variable bound and upper bound invariant).

Definition 1 (Program) Let $\Sigma$ be a set of states. A program over $\Sigma$ is a directed labeled graph $\mathcal{P} = (L, T, l_b, l_e)$, where $L$ is a finite set of locations, $l_b \in L$ is the entry location, $l_e \in L$ is the exit location and $T \subseteq L \times 2^{\Sigma \times \Sigma} \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{\lambda} l_2$ to denote a transition $(l_1, \lambda, l_2) \in T$. We call $\lambda \in 2^{\Sigma \times \Sigma}$ a transition relation. A path of $\mathcal{P}$ is a sequence $l_0 \xrightarrow{\lambda_0} l_1 \xrightarrow{\lambda_1} \cdots$ with $l_i \xrightarrow{\lambda_i} l_{i+1} \in E$ for all $i$. A run of $\mathcal{P}$ is a sequence $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ such that $l_b \xrightarrow{\lambda_0} l_1 \xrightarrow{\lambda_1} \cdots$ is a path of $\mathcal{P}$ and for all $0 < i$ it holds that $(\sigma_{i-1}, \sigma_i) \in \lambda_{i-1}$. A run $\rho$ is complete if it ends at $l_e$.

Note that a run of $\mathcal{P} = (L, T, l_b, l_e)$ starts at location $l_b$. Further note that we call an edge $l_1 \xrightarrow{\lambda} l_1 \in T$ of the program a transition, whereas $\lambda$ is its transition relation. In the following we will refer to transitions by $\tau$ and to transition relations by $\lambda$.

Transition bounds are at the core of our analysis: we infer bounds on the number of loop iterations, on computational complexity, on resource consumption, etc., by computing bounds on the number of times that one or several transitions can be executed. Before we formally define our notion of a transition bound we have to introduce some notation.
Definition 2 (Counter Notation I) Let $P(L, T, l_b, l_e)$ be a program over $\Sigma$. Let $\tau \in T$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $P$. By $\sharp(\tau, \rho)$ we denote the number of times that $\tau$ occurs on $\rho$.

In the following, we denote by ‘$\infty$’ a value s.t. $a < \infty$ for all $a \in \mathbb{Z}$ (infinity).

Definition 3 (Transition Bound) Let $P = (L, T, l_b, l_e)$ be program over states $\Sigma$. Let $\tau \in T$. A value $b \in \mathbb{N}_0 \cup \{\infty\}$ is a bound for $\tau$ on a run $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} (l_2, \sigma_2) \xrightarrow{\lambda_2} \cdots$ of $P$ iff $\sharp(\tau, \rho) \leq b$, i.e., iff $\tau$ appears not more than $b$ times on $\rho$. A function $b : \Sigma \rightarrow \mathbb{N}_0 \cup \{\infty\}$ is a bound for $\tau$ iff for all runs $\rho$ of $P$ it holds that $b(\sigma_0)$ is a bound for $\tau$ on $\rho$, where $\sigma_0$ denotes the initial state of $\rho$.

Given a program transition $\tau$, our bound algorithm (which we define below) computes a bound for $\tau$. If possible, the bound computed by our algorithm should be precise or tight, in particular the trivial bound $\Sigma \rightarrow \infty$ is (most often) of no value to us.

Definition 4 (Precise Transition Bound) Let $P = (L, T, l_b, l_e)$ be a program over states $\Sigma$. Let $\tau \in T$. We say that a transition bound $b : \Sigma \rightarrow \mathbb{N}_0 \cup \{\infty\}$ for $\tau$ is precise iff for each $\sigma_0 \in \Sigma$ there is a run $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ such that $\sharp(\tau, \rho) = b(\sigma_0)$.

Informally A transition bound is precise if it can be reached for all initial states $\sigma_0$. Note that there is exactly one precise transition bound.

Definition 5 (Tight Transition Bound) Let $P = (L, T, l_b, l_e)$ be a program over states $\Sigma$. Let $\tau \in T$. We say that a transition bound $b : \Sigma \rightarrow \mathbb{N}_0 \cup \{\infty\}$ is tight iff there is a $c > 0$ such that either (1) for all $\sigma \in \Sigma$ we have $b(\sigma) < c$ (b is bounded), or (2) there is a family of states $(\sigma_i)_{i \in \mathbb{N}}$ with $\lim_{i \to \infty} b(\sigma_i) = \infty$ (b is unbounded) such that for all $\sigma_i$ there is a run $\rho_i$ starting in $\sigma_i$ with $b(\sigma_i) \leq c \times \sharp(\tau, \rho_i)$.

Informally A transition bound is tight if it is in the same asymptotic class as the precise transition bound: let $\tau \in T$. For the special case $\Sigma = \mathbb{N}$ we have the following: let $f : \mathbb{N} \rightarrow \mathbb{N}$ denote the precise transition bound for $\tau$. Let $g : \mathbb{N} \rightarrow \mathbb{N}$ be some transition bound for $\tau$. Trivially $f \in O(g)$ ($f$ does not grow faster than $g$). Now, $g$ is tight if also $f \in \Theta(g)$ ($f$ does not always grow slower than $g$). With $f \in O(g)$ and $f \in \Omega(g)$ we have that $f \in \Theta(g)$. The same can be formulated for general state sets $\Sigma$ by mapping $\Sigma$ to the natural numbers.

We discussed in Sect. 2.1 that in the course of computing transition bounds, our analysis computes invariants of a special shape. We now formally define the form of the invariants that our analysis infers.

Definition 6 (Upper Bound Invariant) Let $P = (L, T, l_b, l_e)$ be a program over $\Sigma$. Let $e : \Sigma \rightarrow \mathbb{Z}$. Let $l \in L$. Let $e = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} (l_2, \sigma_2) \xrightarrow{\lambda_2} \cdots$ be a run of $P$. A value $b \in \mathbb{Z} \cup \{\infty\}$ is an upper bound invariant for $e$ at $l$ on $\rho$ iff $e(\sigma_i) \leq b$ holds for all $i$ on $\rho$ with $l_i = l$. A function $b : \Sigma \rightarrow \mathbb{Z} \cup \{\infty\}$ is an upper bound invariant for $e$ at $l$ iff for all runs $\rho$ of $P$ it holds that $b(\sigma_0)$ is an upper bound invariant for $e$ at $l$ on $\rho$, where $\sigma_0$ denotes the initial state of $\rho$.

We now formally define the notion local bound that we motivated in Sect. 2.

Definition 7 (Counter Notation II) Let $P = (L, T, l_b, l_e)$ be a program over $\Sigma$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $P$. Let $e : \Sigma \rightarrow \mathbb{R}$ be a norm. By $\downarrow(e, \rho)$ we denote the number of times that the value of $e$ decreases on $\rho$, i.e., $\downarrow(e, \rho) = |\{i \mid e(\sigma_i) > e(\sigma_{i+1})\}|$. 
Definition 8 (Norm) Let $\Sigma$ be a set of states. A norm $e: \Sigma \rightarrow \mathbb{Z}$ over $\Sigma$ is a function that maps the states to the integers.

Definition 9 (Local Bound) Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over $\Sigma$. Let $\tau \in T$. Let $e: \Sigma \rightarrow \mathbb{N}$ be a norm that takes values in the natural numbers. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ be a run of $\mathcal{P}$. $e$ is a local bound for $\tau$ on $\rho$ if it holds that $\rho(\tau, \rho) \leq \downarrow(e, \rho)$. We call $e$ a local bound for $\tau$ if $e$ is a local bound for $\tau$ on all runs of $\mathcal{P}$.

Discussion A natural number valued norm $e$ is a local bound for $\tau$ on a run $\rho$ if $\tau$ appears not more often on $\rho$ than the number of times the value of $e$ decreases. I.e., a local bound $e$ for $\tau$ limits the number of executions of $\tau$ on a run $\rho$ as long as certain program parts (those were $e$ increases) are not executed. We argue in Sect. 9 that in our analysis local bounds play the role of potential functions in classical amortized complexity analysis [32]. We discuss how we obtain local bounds in Sect. 4.

3.1 Difference Constraint Programs

As discussed introductory, we base our algorithm on the abstract program model of difference constraint programs which we now formally define in Definition 12. We discuss in Sect. 6 how we abstract a given program to a DCP.

Definition 10 (Variables, Symbolic Constants, Atoms) By $\mathcal{V}$ we denote a finite set of variables. By $\mathcal{C}$ we denote a finite set of symbolic constants. $\mathcal{A} = \mathcal{V} \cup \mathcal{C}$ is the set of atoms.

Definition 11 (Difference Constraints) A difference constraint over $\mathcal{A}$ is an inequality of form $x' \leq y + c$ with $x \in \mathcal{V}$, $y \in \mathcal{A}$ and $c \in \mathbb{Z}$. By $\mathcal{DC}(\mathcal{A})$ we denote the set of all difference constraints over $\mathcal{A}$.

Notation We often write $x' \leq y$ as a shorthand for the difference constraint $x' \leq y + c$.

Definition 12 (Difference Constraint Program, Syntax) A difference constraint program (DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta \mathcal{P} = (L, E, l_b, l_e)$, where $L$ is a finite set of vertices, $l_b \in L$ and $l_e \in L$ and $E \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L$ is a finite set of edges. We write $l_1 \xrightarrow{u} l_2$ to denote an edge $(l_1, u, l_2) \in E$ labeled by a set of difference constraints $u \in 2^{\mathcal{DC}(\mathcal{A})}$. We use the notation $l_1 \rightarrow l_2$ to denote an edge that is labeled by the empty set of difference constraints. $\Delta \mathcal{P}$ is fan-in-free, if for every edge $l_1 \xrightarrow{u} l_2 \in E$ and every $v \in \mathcal{V}$ there is at most one $a \in \mathcal{A}$ and $c \in \mathbb{Z}$ s.t. $v' \leq a + c \in u$.

Example Figure 10b shows a fan-in free DCP.

Definition 13 (Difference Constraint Program, Semantics) The set of valuations of $\mathcal{A}$ is the set $\text{Val}_\mathcal{A} = \mathcal{A} \rightarrow \mathbb{N}$ of mappings from $\mathcal{A}$ to the natural numbers. Let $u \in 2^{\mathcal{DC}(\mathcal{A})}$. We define $[u] \in 2^{\text{Val}_\mathcal{A} \times \text{Val}_\mathcal{A}}$ s.t. $(\sigma, \sigma') \in [u]$ iff for all $x' \leq y + c \in u$ it holds that (i) $\sigma'(x) \leq \sigma(y) + c$ and (ii) for all $s \in C \sigma'(s) = \sigma(s)$. A DCP $\Delta \mathcal{P} = (L, E, l_b, l_e)$ is a program over the set of states $\text{Val}_\mathcal{A}$ with locations $L$, entry location $l_b$, exit location $l_e$ and transitions $T = \{l_1 \xrightarrow{[u]} l_2 \mid l_1 \xrightarrow{u} l_2 \in E\}$.

Discussion A DCP is a program (Definition 1) whose transition relations are solely specified by conjunctions of difference constraints. Note that variables in difference constraint programs take values only over the natural numbers. Further note that we refer to the syntactic representation of the transition relation in form of a set of difference constraints by $u$, whereas by $[u]$ we refer to the transition relation itself.
Definition 14 (Well-defined DCP) Let $\Delta P = (L, E, l_b, l_e)$ be a DCP over atoms $A$. We say that a variable $x \in V$ is defined at $l$ if $x \in \text{def}(l)$, where $\text{def} : L \rightarrow 2^A$ is defined by $\text{def}(l) = \bigcap_{i \in I}^L \{x \mid \exists y \in V \exists c \in Z \text{ s.t. } x' \leq y + c \in u \} \cup C$.

We say that a variable $x$ is used at $l$ if $x \in \text{use}(l)$, where $\text{use} : L \rightarrow 2^A$ is defined by $\text{use}(l) = \bigcup_{i \in I}^L \{y \mid \exists x \in A \exists c \in Z \text{ s.t. } x' \leq y + c \in u\}$.

$\Delta P$ is well-defined iff $l_b$ has no incoming edges and for all $l \in L$ it holds that $\text{use}(l) \subseteq \text{def}(l)$.

Discussion A DCP $\Delta P$ is well-defined if $l_b$ has no incoming edges and for all $v \in V$ it holds that $v$ is defined at all locations at which $v$ is used (symbolic constants are always defined).

Note that for well-defined programs we in particular require $\text{use}(l_b) \subseteq \text{def}(l_b)$. Because $l_b$ has no incoming edges we have $\text{def}(l_b) = C$. Thus only symbolic constants can be used at $l_b$.

Throughout this work we will only consider DCPs that are fan-in free and well-defined.

Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $A$. Our bound algorithm, which we start to develop in the next section, computes a bound for a given transition $\tau \in E$ in form of an expression over $A$ which involves the operators $+, \times, /, \text{min}, \text{max}$ and the floor function $\lfloor \cdot \rfloor$. However, note that the norms, which are treated as atoms (elements of $A$) in the abstraction, can involve arbitrary operators (see Sect. 6).

Definition 15 (Expressions over $A$) By $\text{Expr}(A)$ we denote the set of expressions over $A \cup \mathbb{Z} \cup \{\infty\}$ that are formed using the arithmetical operators addition (+), multiplication ($\times$), maximum (max), minimum (min) and integer division of form $\lfloor \cdot \rfloor$ where $\text{expr} \in \text{Expr}(A)$ and $c \in \mathbb{N}$. The semantics function $\llbracket \cdot \rrbracket : \text{Expr}(A) \rightarrow (\text{Val}_A \rightarrow \mathbb{Z} \cup \{\infty\})$ evaluates an expression $\text{expr} \in \text{Expr}(A)$ over a state $\sigma \in \text{Val}_A$ using the usual operator semantics (we have $a + \infty = \infty$, $\min(a, \infty) = a$, etc.).

Our bound algorithm, which we define next, computes a special case of an upper bound invariant which we call a variable bound.

Definition 16 (Variable Bound) Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $A$. Let $a \in A$. We call $b \text{ s.t. } b$ is an upper bound invariant for $\llbracket a \rrbracket$ at all $l \in L$ with $a \in \text{def}(l)$ a variable bound for $a$.

Let variable $x$ of the abstract program represent the expression $\text{expr}_x$ of the concrete program. Note that by computing a variable bound for $x$ in the abstract program, we compute an upper bound invariant for $\text{expr}_x$ in the concrete program.

3.2 Algorithm

Our bound algorithm computes a bound for a given transition $\tau \in E$ based on a mapping $\xi : E \rightarrow \text{Expr}(A)$ (called local bound mapping) which assigns each transition $\tau \in E$ either (1) a bound for $\tau$ in form of an expression over the symbolic constants (i.e., $\xi(\tau) \in \text{Expr}(C)$) or (2) a local bound for $\tau$ in form of a variable (i.e., $\xi(\tau) \in V$). Note that $V \cap \text{Expr}(C) = \emptyset$. In Case (1) our algorithm (Definition 19) returns $TB(\tau) = \xi(\tau)$. In Case (2) a transition bound $TB(\tau) \in \text{Expr}(C)$ is computed by inferring how often and by how much the local transition bound $\xi(\tau) \in V$ of $\tau$ may increase during program run.

Definition 17 (Local Bound Mapping) Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $A$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\tau_0} (l_1, \sigma_1) \xrightarrow{\tau_1} \cdots$ be a run of $\Delta P$. We call a function $\xi : E \rightarrow \text{Expr}(A)$ a local bound mapping for $\rho$ if for all $\tau \in E$ it holds that either
1. $\xi(\tau) \in \text{Expr}(C)$ and $[\xi(\tau)](\sigma_0)$ is a bound for $\tau$ on $\rho$ or
2. $\xi(\tau) \in V$ and $[\xi(\tau)]$ is a local bound for $\tau$ on $\rho$.

We say that $\xi$ is a local bound mapping for $\Delta P$ if $\xi$ is a local bound mapping for all runs of $\Delta P$.

Further, our bound algorithm is based on a syntactic distinction between two kinds of updates that modify the value of a given variable $v \in V$: we identify transitions which increment $v$ and transitions which reset $v$.

**Definition 18 (Resets and Increments)** Let $\Delta P = (L, E, I_b, l_e)$ be a DCP over $A$. Let $v \in V$. We define the resets $R(v)$ and increments $I(v)$ of $v$ as follows:

$$
R(v) = \{(l_1 \xrightarrow{u} l_2, a, c) \in E \times A \times Z \mid v' \leq a + c \in u, a \neq v\}
$$

$$
I(v) = \{(l_1 \xrightarrow{u} l_2, c) \in E \times N \mid v' \leq v + c \in u, c > 0\}
$$

Given a path $\pi$ of $\Delta P$ we say that $v$ is reset on $\pi$ if there is a transition $\tau$ on $\pi$ such that $(\tau, a, c) \in R(v)$ for some $a \in A$ and $c \in Z$. We say that $v$ is incremented on $\pi$ if there is a transition $\tau$ on $\pi$ such that $(\tau, c) \in I(v)$ for some $c \in N$.

I.e., we have that $(\tau, a, c) \in R(v)$ if variable $v$ is reset to a value smaller or equal to $a + c$ when executing the transition $\tau$. Accordingly we have $(\tau, c) \in I(v)$ if variable $v$ is incremented by a value smaller or equal to $c$ when executing the transition $\tau$.

Our algorithm in Definition 19 is built on a mutual recursion between the two functions $VB(v)$ and $TB(\tau)$, where $VB(v)$ infers a variable bound for variable $v$ and $TB(\tau)$ infers a transition bound for the transition $\tau$.

**Definition 19 (Bound Algorithm)** Let $\Delta P = (L, E, I_b, l_e)$ be a DCP over $A$. Let $\xi : E \rightarrow \text{Expr}(A)$. We define $VB : A \mapsto \text{Expr}(A)$ and $TB : E \mapsto \text{Expr}(A)$ as:

$$
VB(a) = a, \text{ if } a \in C, \text{ else }
$$

$$
VB(v) = \text{Incr}(v) + \max_{(\tau, a, c) \in R(v)} (VB(a) + c)
$$

$$
TB(\tau) = {\xi(\tau)}, \text{ if } {\xi(\tau)} \notin V, \text{ else }
$$

$$
TB(\tau) = \text{Incr}(\xi(\tau)) + \sum_{(t, a, c) \in R(\xi(\tau))} TB(t) \times \max(VB(a) + c, 0)
$$

where $\text{Incr}(v) = \sum_{(\tau, c) \in I(v)} TB(\tau) \times c$ (we set $\text{Incr}(v) = 0$ for $I(v) = \emptyset$)

**Discussion** We first explain the subroutine $\text{Incr}(v)$: with $(\tau, c) \in I(v)$ we have that a single execution of $\tau$ increments the value of $v$ by not more than $c$. $\text{Incr}(v)$ multiplies the bound for $\tau$ with the increment $c$ in order to summarize the total amount by which $v$ may be incremented over all executions of $\tau$. $\text{Incr}(v)$ thus computes a bound on the total amount by which the value of $v$ may be incremented during program run.

The function $VB(v)$ computes a variable bound for $v$: after executing a transition $\tau$ with $(\tau, a, c) \in R(v)$, the value of $v$ is bounded by $VB(a) + c$. As long as $v$ is not reset, its value cannot increase by more than $\text{Incr}(v)$.

The function $TB(\tau)$ computes a transition bound for $\tau$ based on the following reasoning: (1) the total amount by which the local bound $\xi(\tau)$ of transition $\tau$ can be incremented is bounded by $\text{Incr}(\xi(\tau))$. (2) We consider a reset $(t, a, c) \in R(\xi(\tau))$; in the worst case, a single execution of $t$ resets the local bound $\xi(t)$ to $VB(a) + c$, adding $\max(VB(a) + c, 0)$.
Table 1: Computation of \( TB(t_5) \) for Example twoSCCs (Fig. 2) by Definition 19

| Call       | Evaluation and simplification                                                                 | Using                            |
|------------|-----------------------------------------------------------------------------------------------|----------------------------------|
| \( TB(t_5) \) | \( \text{Incr}([z]) + TB(t_4) \times \max(VB([x]) + [0], 0) \)                          | \( \xi(t_5) = [z] \).           |
|            | \( 0 + TB(t_4) \times \max(VB([x]) + [0], 0) \)                                           | \( \mathcal{R}([x]) = \{(t_4, [x], 0), \} \). |
|            | \( 0 + 1 \times \max([n] \times 2 + \max([m_1], [m_2]) + [0], 0) \)                     | \( \text{Incr}([z]) \).         |
|            | \( = \max([m_1], [m_2]) + [n] \times 2 \)                                                  | \( TB(t_4) \).                  |
| \( \text{Incr}([z]) \) | \( 0 \)                                                                                  | \( \mathcal{I}([z]) = \emptyset \). |
| \( TB(t_4) \) | \( 1 \)                                                                                  | \( \xi(t_4) = 1 \).             |
| \( VB([x]) \) | \( \text{Incr}([x]) + \max([m_1] + [0], [m_2] + [0]) \)                             | \( \mathcal{R}([x]) = \{(t_1, [m_1], 0), \} \). |
|            | \( = [n] \times 2 + \max([m_1] + [0], [m_2] + [0]) \)                                 | \( \{t_2, [m_2], 0, \} \).      |
| \( \text{Incr}([x]) \) | \( TB(t_3) \times 2 \)                                                                  | \( \mathcal{I}([x]) = \{(t_3, 2), \} \). |
|            | \( [n] \times 2 \)                                                                       | \( TB(t_3) \).                  |
| \( TB(t_3) \) | \( \text{Incr}([y]) + TB(t_0) \times \max([n] + [0], 0) \)                             | \( \xi(t_3) = [y] \).           |
|            | \( 0 + TB(t_0) \times \max([n] + [0], 0) \)                                              | \( \mathcal{R}([y]) = \{(t_0, [n], 0), \} \). |
|            | \( 0 + 1 \times \max([n] + [0], 0) \)                                                   | \( [n] \in \mathcal{C} \).      |
|            | \( = [n] \)                                                                              | \( \text{Incr}([y]) \).         |
| \( \text{Incr}([y]) \) | \( 0 \)                                                                                  | \( \mathcal{I}([y]) = \emptyset \). |
| \( TB(t_0) \) | \( 1 \)                                                                                  | \( \xi(t_0) = 1 \).             |

To the potential number of executions of \( t \); in total all \( TB(t) \) possible executions of \( t \) add up to \( TB(t) \times \max(VB(x) + c, 0) \) to the potential number of executions of \( t \).

**Example** We want to infer a bound for the loop at \( l_3 \) in Fig. 2. We thus compute a transition bound for \( t_5 \) (the single back edge of the loop at \( l_3 \)). See Table 1 for details on the computation. We get \( TB(t_5) = \max([m_1], [m_2]) + [n] \times 2 \). Thus \( \max(m_1, m_2) + 2n \) is a bound for the loop at \( l_3 \) (\( m_1, m_2 \) and \( n \) have type unsigned).

**Termination** Our algorithm does not terminate iff recursive calls cycle, i.e., if a call to \( TB(\tau) \) resp. \( VB(v) \) (indirectly) leads to a recursive call to \( TB(\tau) \) resp. \( VB(v) \). This can be detected easily, we return the expression \( \infty \).

We distinguish three cases of cyclic computation: (1) there is a variable \( v \in V \) such that the computation of \( VB(v) \) ends up calling \( VB(v) \) over a number of recursive calls to \( VB \). (2) There is a transition \( \tau \in E \) such that the computation of \( TB(\tau) \) ends up calling \( TB(\tau) \) over a number of recursive calls to \( TB \). (3) There is a variable \( v \in V \) and a transition \( \tau \in E \) such that the computation of \( TB(\tau) \) calls \( VB(v) \) which in turn ends up calling \( TB(\tau) \) over a number of recursive calls to \( VB \) and \( TB \).

**Case (1)** occurs iff there is a cycle in the reset graph (Definition 20 in Sect. 3.3) of \( \Delta P \). In Sect. 3.4 we discuss a preprocessing that ensures absence of cycles in the reset graph and thus absence of Case (1) by renaming the program variables appropriately.

**Case (2)** occurs iff there is a transition \( \tau_1 \) with local bound \( x \) that increases the local bound \( y \) of a transition \( \tau_2 \) which in turn increases \( x \). We conclude that absence of Case (2) is ensured if for all strongly connected components (SCC) \( SCC \) of \( \Delta P \) we can find an ordering \( \tau_1, \ldots, \tau_n \) of the transitions of \( SCC \) such that the local bound of transition \( \tau_i \) is not increased on any
void foo(uint n) {
    int x = n; int a = 1;
    while(x > 0) {
        x--; int y = a;
        while(y > 0) {
            y--; a++;
        }
    }
}

Complexity: \(2^n\)

(a)

(b)

Fig. 3 a Example with an exponential loop bound, b Example for which we obtain a bound expression of exponential size, transitions \(\tau_1 \leq i \leq \tau_n\) have source- and target-location \(l\) transition \(\tau_j\) with \(n \geq j > i \geq 1\). Note that the existence of such an ordering for each SCC of \(\Delta P\) proves termination of \(\Delta P\): it allows to directly compose a termination proof in form of a lexicographic ranking function by ordering the respective local transition bounds accordingly.

An example for Case (3) is given in Fig. 3a. Let \(\tau_1\) be the transition on which \(y\) is reset to \(a\). Let \(\tau_2\) be the single transition of the inner loop. Assume we want to compute a loop bound for the inner loop, i.e., a transition bound for \(\tau_2\) with local bound \(y\). This triggers a variable bound computation for \(a\) because \(y\) is reset to \(a\). Since \(a\) is incremented on \(\tau_2\), the variable bound computation for \(a\) will in turn trigger a transition bound computation for \(\tau_2\). Note, however, that the loop bound for the inner loop is exponential \(2^n\). We consider exponential loop bounds very rare, we did not encounter an exponential loop bound during our experiments.

Complexity: Our algorithm can be efficiently (polynomial in the number of variables and transitions of the abstract program) implemented using caches (dynamic programming): we set \(\xi(\tau) = TB(\tau)\) after having computed \(TB(\tau)\). Accordingly we introduce a cache to store the result of a \(VB\)-computation. When \(VB(v)\) is called we first check if the result is already in the cache before performing the computation. The computed bound expressions, however, can be of exponential size: consider the \(DCP\) \(\Delta P = ((l_b, l_c), \{\tau_0, \tau_1, \ldots, \tau_n\}, l_b, l_c)\) over variables \(\{x_1, x_2, \ldots, x_n\}\) and constants \(\{m_1, m_2, \ldots, m_n\}\) shown in Fig. 3b. In fact, \(TB(\tau_n) = \sum_{S \subseteq \{1, 2, \ldots, n\}} m_0 \times \prod_{i \in S} m_i\) is precise for Fig. 3b. However, the example is artificial. To our experience the computed bound expressions can, in practice, be reduced to human readable size by applying basic rules of arithmetic.

Theorem 1 (Soundness) Let \(\Delta P(L, E, l_b, l_c)\) be a well-defined and fan-in free \(DCP\) over atoms \(A\). Let \(\xi : E \mapsto Expr(A)\) be a local bound mapping for \(\Delta P\). Let \(a \in A\) and \(\tau \in E\). Let \(TB(\tau)\) and \(VB(a)\) be as defined in Definition 19. We have: (1) \([ TB(\tau) ]\) is a transition bound for \(\tau\). (2) \([ VB(a) ]\) is a variable bound for \(a\).

In the following we describe two straightforward improvements of the algorithm stated in Definition 19.

Improvement I Let \(\tau \in E\). Let \(v \in V\) be a local bound for \(\tau\), i.e., for all runs \(\rho\) of \(\Delta P\) we have that \(\tau(\tau, \rho) \leq \downarrow(v, \rho)\). Let \(c \in \mathbb{N}\). Let \(\downarrow(v, c, \rho)\) denote the number of times that the value of \(v\) decreases on a run \(\rho\) of \(\Delta P\) by at least \(c\) (refines Definition 7). If for all runs \(\rho\) of \(\Delta P\) we have that \(\tau(\tau, \rho) \leq \downarrow(v, c, \rho)\) (refines Definition 9) then \([ [[ TB(\tau) ]/c ]]\) is a bound for \(\tau\)
Improvement II Let \( \tau_1, \tau_2 \in E \) be two transitions with the same local bound, i.e., \( \zeta(\tau_1) = \zeta(\tau_2) \). If \( \tau_1 \) and \( \tau_2 \) cannot be executed without decreasing the common local bound \( \zeta(\tau_1) \) twice, once for \( \tau_1 \) and once for \( \tau_2 \) (e.g., \( \tau_2 \) and \( \tau_5 \) in \texttt{xnuSimple}, Fig. 1), we have that \( \pi(\tau_1, \rho) + \pi(\tau_2, \rho) \leq [TB(\tau_1)](\sigma_0) = [TB(\tau_2)](\sigma_0) \). Thus, \( TB(\tau_1) \) is a bound on the number of times that \( \tau_1 \) and \( \tau_2 \) can be executed on any run of \( \Delta P \). We exploit this observation: assume some \( v \in V \) is incremented by \( c_1 \) on \( \tau_1 \) and by \( c_2 \) on \( \tau_2 \). For computing \( \text{Incr}(v) \) we only add \( TB(\tau_1) \times \max\{c_1, c_2\} \) instead of \( TB(\tau_1) \times c_1 + TB(\tau_2) \times c_2 \). This idea can be generalized to multiple transitions. Further details on the discussed improvement are given in [30].

3.3 Reasoning Based on Reset Chains

Consider Fig. 4. The precise bound for the loop at \( l_3 \) is \( n \): Initially \( r \) has value \( n \), after we have iterated the loop at \( l_3 \), \( r \) is set to 0. Thus the loop can only be executed in at most one iteration of the outer loop. However, our algorithm from Definition 19 infers a quadratic bound for the loop at \( l_3 \); as shown in Table 2 we have \( TB(\tau_3) = [n] \times [n] \). We thus get \( n^2 \) (\( n \) has type \texttt{unsigned}) as bound for the loop at \( l_3 \) in the concrete program.

Our algorithm from Definition 19 does not take into account that \( r \) is reset to 0 after executing the loop at \( l_3 \). In the following we discuss an extension of our algorithm which overcomes this imprecision by taking the context under which a transition is executed into account: we say that a transition \( \tau_3 \) is executed under context \( \tau_1 \) if transition \( \tau_1 \) was executed before the current execution of \( \tau_2 \) and after the previous execution of \( \tau_2 \) (if any).

As an example, consider Fig. 4b, the abstraction of Fig. 4a. We have that \( \tau_2 \) is always executed either under context \( \tau_0 \) or under context \( \tau_4 \). When executing \( \tau_2 \) under context \( \tau_0 \), \( p \) is set to \( n \). But when executing \( \tau_2 \) under context \( \tau_4 \), \( p \) is set to 0. Moreover, \( \tau_2 \) can only be executed once under context \( \tau_0 \) because \( \tau_0 \) is executed only once.

We define the notion of a reset graph as a means to reason systematically about the context under which resets can be executed.

**Definition 20** (Reset Chain Graph) Let \( \Delta P(L, E, l_b, l_e) \) be a DCP over \( A \). The reset chain graph or reset graph of \( \Delta P \) is the directed graph \( G \) with node set \( A \) and edges

void foo(uint n) {
    int x = n;
    int r = n;
    l1 while(x > 0) {
        x = x - 1;
    } l2 if(*) {
        p = r;
    } l3 while(p > 0) {
        p--; r = 0;
    } l4 }

(a) Complexity: \( 2n \)

(b)

(c)

\[ \begin{array}{c}
\text{Fig. 4} \ a \text{ Example, b Abstraction, c Reset Graph}
\end{array} \]

(assuming \( \zeta(\tau) = v \)). In our simple abstract program model, \( c \in \mathbb{N} \) is obtained syntactically from a constraint \( v' \leq v - c \). See Sect. 4 on how we determine relevant constraints. More details on the discussed improvement are given in [30].
\[ \mathcal{E} = \{ (y, \tau, c, x) \mid (\tau, y, c) \in \mathcal{R}(x) \} \subseteq A \times E \times \mathbb{Z} \times \mathcal{V}, \text{ i.e., each edge has a label in} \ E \times \mathbb{Z}. \text{ We call} \ G(A, \mathcal{E}) \text{ a reset chain DAG or reset DAG if} \ G(A, \mathcal{E}) \text{ is acyclic. We call} \ G(A, \mathcal{E}) \text{ a reset chain forest or reset forest if the sub-graph} \ G(\mathcal{V}, \mathcal{E}) \text{ (recall that} \ \mathcal{V} \subseteq \mathcal{A} \text{ is a forest. We call a finite path} \ \kappa = a_n \xrightarrow{t_{n-1},c_{n-1}} a_{n-1} \xrightarrow{t_{n-2},c_{n-2}} \cdots \xrightarrow{t_0,c_0} a_0 \text{ in} \ G \text{ with} \ n > 0 \text{ a reset chain of} \ \Delta \mathcal{P}. \text{ We say that} \ \kappa \text{ is a reset chain from} \ a_n \text{ to} \ a_0. \text{ Let} \ n \geq i \geq j \geq 0. \text{ By} \ k_{i,j} \text{ we denote the sub-path of} \ \kappa \text{ that starts at} \ a_i \text{ and ends at} \ a_j. \text{ We define} \ in(\kappa) = a_n, c(\kappa) = \sum_{i=1}^{n} c_i, \ \text{trn}(\kappa) = \{ t_n, t_{n-1}, \ldots, t_1 \}, \text{ and} \ \text{atm}(\kappa) = \{ a_{n-1}, \ldots, a_0 \}. \ \kappa \text{ is sound if for all} \ 1 \leq i < n \text{ it holds that} \ a_i \text{ is reset on all paths from the target location of} \ t_i \text{ to the source location of} \ t_1 \text{ in} \ \Delta \mathcal{P}. \ \kappa \text{ is optimal if} \ \kappa \text{ is sound and there is no sound reset chain} \ \chi \text{ of length} \ n + 1 \text{ s.t.} \ k_{[n,0]} = \kappa. \text{ Let} \ \nu \in \mathcal{V}, \text{ by} \ \mathcal{R}(\nu) \text{ we denote the set of optimal reset chains ending in} \ \nu. \]

**Example** Figure 4c shows the reset graph of Fig. 4b.

We elaborate on the notions sound and optimal below. Let us first state a basic intuition on how we employ reset chains to enhance the precision of our reasoning:

For a given reset \ (\tau, a, c) \in \mathcal{R}(\nu), \text{ the reset graph determines which atom flows into variable} \ \nu \text{ under which context. For example, consider Fig. 4b and its reset graph in Fig. 4c: when executing the reset} \ (\tau_2, [r], 0) \in \mathcal{R}([p]) \text{ under the context} \ \tau_4, \text{ if the same reset is executed under the context} \ \tau_0, \text{ then} \ [p] \text{ is set to} \ [n]. \text{ Note that the reset graph does not represent increments of variables. We discuss how we handle increments in Sect. 3.3.1.}

Let \ \nu \in \mathcal{V}. \text{ Given a reset chain} \ \kappa \text{ of length} \ n \text{ that ends at} \ \nu, \text{ we say that} \ \text{(trn}(\kappa), \text{ in}(\kappa), c(\kappa)) \text{ is a reset of} \ \nu \text{ with context of length} \ n - 1. \text{ I.e.,} \mathcal{R}(\nu) \text{ from Definition 18 is the set of context-free resets of} \ \nu \text{ (context of length} 0), \text{ because} \ \text{(trn}(\kappa), \text{ in}(\kappa), c(\kappa)) \in \mathcal{R}(\nu) \text{ iff} \ \kappa \text{ ends at} \ \nu \text{ and has length} \ 1. \text{ Our previously defined algorithm from Definition 19 uses only context-free resets, we say that it reasons context free. For reasoning with context, we substitute the term} \]

\[ \sum_{(i,a,c) \in \mathcal{R}(\tau(\kappa))} TB(i) \times \max(VB(a) + c, 0) \]

\[ \text{in Definition 19 by the term} \]

\[ \sum_{\kappa \in \mathcal{R}(\tau(\kappa))} TB(\text{trn}(\kappa)) \times \max(VB(\text{in}(\kappa)) + c(\kappa), 0). \]
Note that we can compute a bound on the number of times that a sequence $\tau_1, \tau_2, \ldots, \tau_n$ of transitions may occur on a run by computing $\min_{1 \leq i \leq n} TB(\tau_i)$.

We now discuss how our algorithm infers the linear bound for $\tau_3$ of Fig. 4 when applying the described modification to Definition 19: the reset graph of Fig. 4b is shown in Fig. 4c.

| Call        | Evaluation and simplification                                                                 | Using                                                                 |
|-------------|-----------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| $TB(\tau_3)$ | $TB(\{\tau_4, \tau_2\}) \times \max([0] + 0, 0)$                                              | $\xi(\tau_3) = [p]$, $\mathcal{R}(\{p\}) = \{[0] \xrightarrow{\tau_4} [r] \xrightarrow{\tau_2} [p]\}$ |
|             | + $TB(\{\tau_0, \tau_2\}) \times \max([n] + 0, 0)$                                            |                                                                       |
|             | + 0 + $TB(\{\tau_0, \tau_2\}) \times \max([n] + 0, 0)$                                        |                                                                       |
|             | $= 0 + \min(1, [n]) \times \max([n] + 0, 0)$                                                   |                                                                       |
|             | $= \min(1, [n]) \times [n]$                                                                 |                                                                       |
| $TB(\tau_2)$ | $\min(TB(\tau_0), TB(\tau_2))$                                                                | $TB(\tau_0)$, $TB(\tau_2)$                                         |
|             | $\min(1, [n])$                                                                               |                                                                       |
| $TB(\tau_0)$ | $1$                                                                                           | $\xi(\tau_0) = 1$                                                   |

Table 3: Computation of $TB(\tau_3)$ for Fig. 4b by Definition 21 (without calls to $\text{Inc}_x$ because $\mathcal{I}(v) = \emptyset$ and thus $\text{Inc}_x(v) = 0$ for Fig. 4b)

Note that we can compute a bound on the number of times that a sequence $\tau_1, \tau_2, \ldots, \tau_n$ of transitions may occur on a run by computing $\min_{1 \leq i \leq n} TB(\tau_i)$.

In the modified program, $\tau_3$ to 0, and the inner loop would have a quadratic loop bound. For the abstract program the described modification replaces the constraint $|r|^+ \leq 0$ on $\tau_4$ in Fig. 4b by $|r|^+ \leq |r|$. In the modified program, $|r|$ is not reset between two executions of $\tau_2$, i.e., the reset chain $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]$ is not sound. Our algorithm therefore reasons based on the reset chain $[r] \xrightarrow{\tau_2} [p]$ and obtains a quadratic bound for $\tau_3$: $TB(\tau_3) = TB(\tau_2) \times \mathcal{V}_B(r) = [n] \times [n]$. I.e., if $r$ is not reset on the outer loop this is modeled in our analysis by considering the reset chain $[r] \xrightarrow{\tau_2} [p]$ rather than the maximal reset chain $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]$. Considering the maximal reset chain $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]$ would be unsound in the described scenario: $\min(TB(\tau_0), TB(\tau_2)) \times [n] = [n]$ is not a valid transition bound for $\tau_3$ if $r$ is not reset to 0.
between two executions of the inner loop. The optimal reset chains are the sound reset chains with maximal context, i.e., those reset chains that are sound and cannot be extended without becoming unsound.

3.3.1 Algorithm Based on Reset Chain Forests

In the presence of cycles in the reset graph we get infinitely many reset chains. Let us for now assume that the given program has a reset forest, i.e., that the sub-graph of the reset graph, which has nodes only in \( V \), is a forest (Definition 20). Then also the complete reset graph is acyclic because \( A = V \cup C \) and the nodes in \( C \) cannot have incoming edges (Definition 20).

**Definition 21** (Bound Algorithm using Reset Chains (reset forest)) Let \( \zeta : E \rightarrow \text{Expr}(A) \) be a local bound mapping for \( \Delta P \). Let \( V_{B} : A \mapsto \text{Expr}(A) \) be as defined in Definition 19. We override the definition of \( T_{B} : E \mapsto \text{Expr}(A) \) in Definition 19 by stating:

\[
T_{B}(\tau) = \zeta(\tau), \text{ if } \zeta(\tau) \notin V, \text{ else }
\]

\[
T_{B}(\tau) = \text{Incr} \left( \bigcup_{\kappa \in \mathcal{R}(\zeta(\tau))} \text{atm}(\kappa) \right) + \sum_{\kappa \in \mathcal{R}(\zeta(\tau))} T_{B}(\text{trn}(\kappa)) \times \max(V_{B}(\text{in}(\kappa)) + c(\kappa), 0)
\]

where \( T_{B}(\{\tau_{1}, \tau_{2}, \ldots, \tau_{n}\}) = \min_{1 \leq i \leq n} T_{B}(\tau_{i}) \) and \( \text{Incr}(\{a_{1}, a_{2}, \ldots, a_{n}\}) = \sum_{1 \leq i \leq n} \text{Incr}(a_{i}) \) with \( \text{Incr}(\emptyset) = 0 \).

**Discussion and Example** We have discussed above why we replace the term \( T_{B}(t) \times \max(V_{B}(a) + c, 0) \) from Definition 19 by the term \( T_{B}(\text{trn}(\kappa)) \times \max(V_{B}(\text{in}(\kappa)) + c(\kappa), 0) \). We further discuss the term \( \text{Incr} \left( \bigcup_{\kappa \in \mathcal{R}(\zeta(\tau))} \text{atm}(\kappa) \right) \) which replaces the term \( \text{Incr}(\zeta(\tau)) \) from Definition 19: consider Example xnuSimple in Fig. 1. Note that \( r \) may be incremented on \( \tau_{1} \) between the reset of \( r \) to 0 on \( \tau_{0} \) resp. \( \tau_{4} \) and the reset of \( p \) to \( r \) on \( \tau_{2} \). The term \( \text{Incr} \left( \bigcup_{\kappa \in \mathcal{R}(\zeta(\tau))} \text{atm}(\kappa) \right) \) takes care of such increments which may increase the value that finally flows into \( \zeta(\tau) \) (in the example \( p \)) when the last transition on \( \kappa \) (in the example \( \tau_{2} \)) is executed. In Table 4 the details of the bound computation are given. We get \( T_{B}(\tau_{3}) = [n] \), i.e., we have the bound \( n \) for the loop at \( \ell_{3} \) in the concrete program (Fig. 1a, \( n \) has type unsigned).

**Soundness** Definition 21 for DCPs with a reset forest is a special case of Definition 23 for DCPs with a reset DAG. We prove soundness of Definition 23 in Electronic Supplementary Material.

**Complexity** The nodes of a reset forest are the variables and constants of the abstract program (the elements of \( A \)). Since the number of paths of a forest is polynomial in the number of nodes, the run time of our algorithm remains polynomial.

3.3.2 Algorithm Based on Reset Chain DAGs

The examples we considered so far had reset forests. (Note that the definition of a reset forest (Definition 20) only requires the sub-graph over the variables, i.e., the reset graph without the nodes that are symbolic constants, to be a forest.) In the following we generalize Definition 21 to reset DAGs. We discuss in Sect. 3.4 how we ensure that the reset graph is acyclic.
Consider the Example shown in Fig. 5. The outer loop (at $l_1$) can be executed $n$ times. The loop at $l_4$ resp. transition $\tau_6$ can be executed $2n$ times, e.g., by executing the program as depicted in Table 5:

Table 4  Computation of $TB(\tau_3)$ for Example xnuSimple (Fig. 1) by Definition 21

| Call       | Evaluation and simplification | Using                                           |
|------------|-------------------------------|-------------------------------------------------|
| $TB(\tau_3)$ | $\rightarrow Incr([r], [p])$ | $\xi(\tau_3) = [p]$, $\mathcal{R}(\{p\}) = \{(0, t_4) \rightarrow [r], t_2 \rightarrow [p]\}$, $[0] \rightarrow [r] \rightarrow [p]$, $[0] \in C$, $Incr([r], [p])$ |
| $Incr([r], [p])$ | $\rightarrow Incr([r]) + Incr([p])$ | $Incr([r]), Incr([p])$ |
| $Incr([p])$ | $\rightarrow 0$ | $I([p]) = \emptyset$, $\xi(\tau_1) = [x]$, $\mathcal{R}(\{x\}) = \{(n, t_0) \rightarrow [x]\}$, $[n] \in C, Incr([x]), TB(t_0)$ |
| $TB(\tau_1)$ | $\rightarrow Incr([x]) + TB(t_0) \times [n]$ | $I([x]) = \emptyset$, $\xi(\tau_1) = [x]$, $\mathcal{R}(\{x\}) = \{(n, t_0) \rightarrow [x]\}$, $[n] \in C, Incr([x]), TB(t_0)$ |
| $Incr([x])$ | $\rightarrow 0$ | $I([x]) = \emptyset$, $\xi(\tau_1) = [x]$, $\mathcal{R}(\{x\}) = \{(n, t_0) \rightarrow [x]\}$, $[n] \in C, Incr([x]), TB(t_0)$ |
| $TB(t_0)$ | $\rightarrow 1$ | $\xi(\tau_0) = 1$ |

```c
void foo(uint n) {  
    int x = n;  
    int y = n;  
    int p = 0;  
    l1 while(x > 0) {  
        x = x - 1;  
        int r = 0;  
    l2 while(y > 0 && *)     
        {  
            y--;  
            r++;  
        }  
    l3 if(*)     
        p = r;  
    l4 while(p > 0)     
        p--;  
        p = r;  
    }  
(a) Complexity: 4n

(b)  

(c)  
```

Fig. 5  a Example, b Abstraction, c Reset Graph
The first row counts the number of iterations of the outer loop, the second row shows the transitions that are executed and in the last two rows the values of $r$ resp. $p$ are tracked. The execution switches between two iteration schemes of the outer loop: an uneven iteration increments $r$ twice (by executing $\tau_2$ twice) and afterward assigns $r$ to $p$ by executing $\tau_5$. We can then execute $\tau_0$ two times. Afterward the value of $r$ is “saved” in $p$ for the next (even) iteration of the outer loop before $r$ is set to 0 on $\tau_1$. Therefore $\tau_6$ can be executed again two times in the next, even iteration though $r$ is not incremented on that iteration.

Consider the abstracted DCP in Fig. 5b and its reset graph in Fig. 5c. We have that $\kappa_2 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_5} [p]$ and $\kappa_3 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_7} [p]$ are two reset chains ending in $[p]$ (see Fig. 5c). Observe that both are sound, i.e., between any two executions of $\tau_7$ resp. $\tau_5$ $[r]$ is reset. However, $[r]$ is not necessarily reset between the execution of $\tau_5$ and $\tau_7$, therefore the accumulated value 2 of $r$ is used twice to increase the local bound $[p]$ of $\tau_6$.

I.e., since there are two paths from $[r]$ to $[p]$ in the reset graph (Fig. 5c) we have to count the increments of $[r]$ twice: once for $\kappa_2$ and once for $\kappa_3$. Definition 22 distinguishes between nodes that have a single resp. multiple path(s) to a given variable in the reset graph. This is used in Definition 23 for a sound handling of the latter case.

**Definition 22** (atm$_1(\kappa)$ and atm$_2(\kappa)$) Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $A$. Let $P(a, v)$ denote the set of paths from $a$ to $v$ in the reset graph of $\Delta P$. Let $v \in V$. Let $\kappa$ be a reset chain ending in $v$. We define $\text{atm}_1(\kappa) = \{a \in \text{atm}(\kappa) \mid |P(a, v)| \leq 1\}$ and $\text{atm}_2(\kappa) = \{a \in \text{atm}(\kappa) \mid |P(a, v)| > 1\}$, where $|S|$ denotes the number of elements in $S$.

**Definition 23** (Bound Algorithm Based on Reset Chains (reset DAG)) Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $A$. Let $\xi : E \mapsto \text{Expr}(A)$ be a local bound mapping for $\Delta P$. Let $VB : A \mapsto \text{Expr}(A)$ be as defined in Definition 19. We override the definition of $TB : E \mapsto \text{Expr}(A)$ in Definition 19 by stating:

$$TB(\tau) = \xi(\tau), \text{if } \xi(\tau) \notin V, \text{ else}$$

$$TB(\tau) = \text{Incr} \left( \bigcup_{\kappa \in \text{atm}_1(\tau)} \right) + \sum_{\kappa \in \text{atm}_2(\tau)} TB(\text{trn}(\tau)) \times \max(VB(\text{trn}(\tau)), 0) + \text{Incr}(\text{atm}_2(\tau))$$

where $TB((\tau_1, \tau_2, \ldots, \tau_n)) = \min_{1 \leq i \leq n} TB(\tau_i)$ and $\text{Incr}([a_1, a_2, \ldots, a_n]) = \sum_{1 \leq i \leq n} \text{Incr}(a_i)$ with $\text{Incr}(\emptyset) = 0$

**Discussion** If atm$_2(\kappa) = \emptyset$ for all reset chains $\kappa$, Definition 23 is equal to Definition 21. This is the case for all DCPs with a reset forest (all examples in this article except Fig. 5). Definition 23 thus is a generalization of Definition 21.

**Example** As shown in Table 6 we get $TB(\tau_6) = [n] + [n]$ for Fig. 5 by Definition 23. I.e., we get the precise bound $2n$ for the loop at $l_4$ in Fig. 5 ($n$ has type unsigned).
Table 6  Computation of $TB(τ_6)$ for Fig. 5 by Definition 23

| Call        | Evaluation and simplification                                                                                                                                                                                                 | Using                                                                 |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| $TB(τ_6)$   | $\rightarrow \text{Incr}([p])$                                                                                                                                                                                                  | $ζ(τ_6) = [p]$.                                                      |
|             | $+ TB([t_0]) \times \max([0] + 0, 0) + 0$                                                                                                                                                                                          | $κ_1 = [0] \xrightarrow{t_0} [p]$.                                   |
|             | $+ TB([t_1, t_7]) \times \max([0] + 0, 0)$                                                                                                                                                                                          | $κ_2 = [0] \xrightarrow{t_1} [r] \xrightarrow{t_7} [p]$.                |
|             | $+ \text{Incr}([r])$                                                                                                                                                                                                                 | $κ_3 = [0] \xrightarrow{t_5} [r] \xrightarrow{t_5} [p]$.                |
|             | $+ TB([t_1, t_5]) \times \max([0] + 0, 0)$                                                                                                                                                                                          | $Ω([p]) = \{κ_1, κ_2, κ_3\}$.                                        |
|             | $+ \text{Incr}([r])$                                                                                                                                                                                                                 | $\text{atm}_1(κ_1) = \{[p]\}$.                                       |
|             | $= \text{Incr}([p]) + 0 + 0 + 0 + \text{Incr}([r])$                                                                                                                                                                               | $\text{atm}_1(κ_2) = \{[p]\}$.                                       |
|             | $+ 0 + \text{Incr}([r])$                                                                                                                                                                                                             | $\text{atm}_1(κ_3) = \{[p]\}$.                                       |
|             | $\rightarrow 0 + 0 + 0 + [n] + 0 + [n]$                                                                                                                                                                                              | $\text{atm}_2(κ_1) = \emptyset$.                                     |
|             | $= [n] + [n]$                                                                                                                                                                                                                       | $\text{atm}_2(κ_2) = \{[r]\}$.                                       |
|             | $\text{Incr}([p])$                                                                                                                                                                                                                  | $\text{atm}_2(κ_3) = \{[r]\}$.                                       |
|             | $\rightarrow 0$                                                                                                                                                                                                                   | $[0] \in C$, $\text{Incr}([p]), \text{Incr}([r])$                     |
| $\text{Incr}([r])$ | $\rightarrow TB(τ_2) \times 1$                                                                                                                                                                                                   | $ζ(τ_2) = \emptyset$.                                                |
|             | $\rightarrow [n] \times 1$                                                                                                                                                                                                       | $I([p]) = \{([τ_2], 1)\}$.                                           |
|             | $= [n]$                                                                                                                                                                                                                             | $TB(τ_2)$                                                           |
| $TB(τ_2)$   | $\rightarrow TB(τ_0) \times \max([n] + 0, 0)$                                                                                                                                                                                     | $ζ(τ_2) = [y]$.                                                      |
|             | $\rightarrow 1 \times \max([n] + 0, 0)$                                                                                                                                                                                           | $Ω([y]) = \{[n] \xrightarrow{τ_0} [y]\}$, $[n] \in C$, $TB(τ_0)$       |
|             | $= [n]$                                                                                                                                                                                                                             | $TB(τ_0)$                                                           |
| $TB(τ_0)$   | $\rightarrow 1$                                                                                                                                                                                                                    | $ζ(τ_0) = 1$                                                        |

**Theorem 2**  (Soundness of Bound Algorithm using Reset Chains) Let $ΔP(L, E, l_p, l_e)$ be a well-defined and fan-in free DCP over atoms $A$. Let $ζ : E \mapsto \text{Expr}(A)$ be a local bound mapping for $ΔP$. Let $TB$ and $VB$ be defined as in Definition 23. Let $τ \in E$ and $a \in A$. If $ΔP$ has a reset DAG then (1) $[TB(τ)]$ is a transition bound for $τ$ and (2) $[VB(a)]$ is a variable bound for $a$.

**Proof**  See Electronic Supplementary Material.

**Complexity**  A DAG can have exponentially many paths in the number of nodes. Thus there can be exponentially many reset chains in $Ω(ν)$ (exponential in the number of variables and constants of the abstract program, i.e., the norms generated during the abstraction process, see Sect. 6). However, in our experiments enumeration of (optimal) reset chains did not affect performance. (See also our discussion on scalability in Sect. 10.1.)

### 3.4 Preprocessing: Transforming a Reset Graph into a Reset DAG

Consider the DCP shown in Fig. 6a. Figure 6a has a cyclic reset graph as shown in Fig. 6b. In the following we describe an algorithm which transforms Fig. 6a into d by renaming the program variables. Figure 6d has an acyclic reset graph (a reset DAG).
\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\multiset \begin{array}{c}
l_b \\
\tau_0 \\
l_1 \xrightarrow{\tau_1} x' \leq n \\
l_1 \\
\tau_0 \\
l_e \\
x' \leq y \\
\end{array} & \begin{array}{c}
\tau_1 \\
y \leq x - 1 \\
\end{array} & \begin{array}{c}
\tau_0 \\
x, 1 \\
y, 2 \\
\end{array} & \begin{array}{c}
l_b \\
\tau_0 \\
l_1 \xrightarrow{\tau_1} z' \leq n \\
l_1 \\
\tau_0 \\
l_e \\
z' \leq z - 1 \\
\end{array} \\
\hline
\end{tabular}
\caption{(a) Example, (b) Reset Graph, (c) Variable Flow Graph, (d) Variables Renamed}
\end{figure}

\textbf{Definition 24} (Variable Flow Graph) Let $\Delta P(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. We call the graph with node set $V \times L$ and edge set

$$\{(y, l_1) \rightarrow (x, l_2) \mid l_1 \xrightarrow{u} l_2 \in E \land x' \leq y + c \in u \text{ with } x, y \in V\}$$

the variable flow graph.

For an example see Fig. 6c.

Let $\Delta P(L, E, l_b, l_e)$ be a DCP. Let $\{\text{SCC}_1, \text{SCC}_2, \ldots, \text{SCC}_n\}$ be the strongly connected components of its variable flow graph. For each SCC $\text{SCC}_i$ we choose a fresh variable $v_i \in V$. Let $\xi : V \times L \mapsto V$ be the mapping $\xi(v, l) = v_i$, where $i$ s.t. $(v, l) \in \text{SCC}_i$. We extend $\xi$ to $\mathcal{A} \times L \mapsto A$ by defining $\xi(s, l) = s$ for all $l \in L$ and $s \in C$.

We obtain $\Delta P'(L, E', l_b, l_e)$ from $\Delta P$ by setting $E' = \{l_1 \xrightarrow{\xi l} l_2 \mid l_1 \xrightarrow{\xi l} l_2 \in E\}$, where $\xi l$ is obtained from $u$ by generating the constraint $\xi(x, l_2)' \leq \xi(y, l_1) + c$ from a constraint $x' \leq y + c \in u$.

\textbf{Examples} Figure 6d is obtained from Fig. 6a by applying the described transformation using the mapping $\xi(x, l_1) = \xi(y, l_2) = z$.

\textbf{Soundness} Soundness of the described variable renaming is obvious if there are no two (different) variables $v_1$ and $v_2$ that are renamed to the same fresh variable at some location $l$. This is the case if in each SCC of the variable flow graph each location $l \in L$ appears at most once, i.e., if there is no SCC $\text{SCC}$ in the variable flow graph of the program such that there is a location $l \in L$ and variables $v_1, v_2 \in V$ with $v_1 \neq v_2$ and $(l, v_1) \in \text{SCC}$ and $(l, v_2) \in \text{SCC}$. In the literature, a program with this property is called stratifiable (e.g., [5]). A fan-in free DCP that is not stratifiable can be transformed into a stratifiable and fan-in free DCP by introducing appropriate case distinctions into the control flow of the program. Details are given in [30]. In the worst-case, however, this transformation can cause an exponential blow up of the number of transitions in the program (the size of the control flow graph).

\section{4 Finding Local Bounds}

In this section we describe our algorithm for finding local bounds.

\textbf{Intuition} Let $\tau \equiv l_1 \xrightarrow{u} l_2 \in E$ and $v \in V$. Clearly, $v$ is a local bound for $\tau$ if $v$ decreases when executing $\tau$, i.e., if $v' \leq v + c \in u$ for some $c < 0$. Moreover, $v$ is a local bound for $\tau$, if every time $\tau$ is executed also some other transition $t \in E$ is executed and $v$ is a local

visited.
bound for \( t \). This is, e.g., the case if \( t \) is always executed either before each execution of \( \tau \) or after each execution of \( \tau \).

**Algorithm** The above intuition can be turned into a simple three-step algorithm. Let \( \Delta_P(L, E, l_b, l_e) \) be a DCP. (1) We set \( \zeta(\tau) = 1 \) for all transitions \( \tau \) that do not belong to a strongly connected component (SCC) of \( \Delta_P \). (2) Let \( v \in V \). We define \( \xi(v) \subseteq E \) to be the set of all transitions \( \tau = l_1 \overset{u}{\rightarrow} l_2 \in E \) such that \( v' \leq v + c \in u \) for some \( c < 0 \). For all \( \tau \in \xi(v) \) we set \( \zeta(\tau) = v \). (3) Let \( v \in V \) and \( \tau \in E \). Assume \( \tau \) was not yet assigned a local bound by (1) or (2). We set \( \zeta(\tau) = v \) if \( \tau \) does not belong to a strongly connected component (SCC) of the directed graph \( (L, E') \) where \( E' = E \setminus \{\xi(v)\} \) (the control flow graph of \( \Delta_P \) without the transitions in \( \xi(v) \)).

If there are \( v_1 \neq v_2 \) s.t. \( \tau \in \xi(v_1) \cap \xi(v_2) \) then \( \zeta(\tau) \) is assigned either \( v_1 \) or \( v_2 \) non-deterministically. An alternative way of handling this case is as follows: we generate two local bound mappings, \( \zeta_1 \) and \( \zeta_2 \) where \( \zeta_1(\tau) = v_1 \) and \( \zeta_2(\tau) = v_2 \). This way we can systematically enumerate all possible choices, finally we apply our bound algorithm once based on \( \zeta_1 \), based on \( \zeta_2 \), etc., and finally take the minimum over all computed bounds. In our implementation, however, we follow the aforementioned greedy approach based on non-deterministic choice.

**Discussion on Soundness** Soundness of Steps (1) and (2) is obvious. We discuss soundness of Step (3): let \( \tau \in E \). If \( \tau \) does not belong to an SCC of \((L, E \setminus \{\xi(v)\})\) we have that some transition in \( \xi(v) \) (which decreases \( v \)) has to be executed in between any two executions of \( \tau \). It remains to ensure that there is a decrease of \( v \) also for the last execution of \( \tau \): for special cases this is unfortunately not the case. Consider Fig. 8b (Sect. 5). The above stated algorithm sets \( \zeta(\tau_1) = [x] \). However, \([x]\) is not a local bound for \( \tau_1 \) of Fig. 8b because there is no decrease of \([x]\) for the last execution of \( \tau_1 \) (before executing \( \tau_2 \)).

It is straightforward to ensure soundness of the algorithm: adding an edge from \( l_e \) to \( l_b \) forces the algorithm to take the last execution of a transition into account. I.e., we set \( E' = E \cup \{l_e \rightarrow l_b\} \setminus \{\xi(v)\} \). Now our algorithm fails to find a local bound for \( \tau_1 \) of Fig. 8b, which is sound. We discuss how we handle the example in Fig. 8 in Sect. 4.1.

**Complexity** Steps (1) and (2): can be implemented in linear time. Step (3): for each \( v \in V \) we need to compute the SCCs of \((L, E \setminus \{\xi(v)\})\). It is well known that SCCs can be computed in linear time (linear in the number of edges and nodes). Since we need to perform one SCC computation per variable, Step (3) is quadratic.

### 4.1 Generalizing Local Bounds to Sets of Local Bounds

Consider the example in Fig. 7. In Fig. 7b the DCP obtained by abstraction (Sect. 6) from the program in Fig. 7a is shown. We have that \( x \) is a local bound for \( \tau_1 \) and \( y \) is a local bound for \( \tau_2 \). However, it is not straightforward to find a local bound for \( \tau_3 \): in order to form a local bound for \( \tau_3 \) we need to combine \( x \) and \( y \) to a linear combination, e.g., \( 2x + y \). It is unclear how to automatically come up with such expressions.

In the following we discuss a simple generalization of our algorithm by which we avoid an explicit composition of local bounds.

We generalize the local bound mapping \( \zeta : E \rightarrow \text{Expr}(A) \) (Definition 17) to a local bound set mapping \( \zeta : E \rightarrow 2^{\text{Expr}(A)} \).
void foo(uint n) {
    int x = n; int y = 0;
    while(x > 0) {
        if(y > 0 && *)
            y = y - 1;
        else {
            x = x - 1;
            y = y + 1;
        }
    }
}

Complexity: 2n

(a)

(b)

Fig. 7 a Example, b DCP obtained by abstraction

Definition 25 (Local Bound Set Mapping) Let \( \Delta P(L, E, l_b, l_e) \) be a DCP over \( A \). Let \( \rho = (l_b, \sigma_0) \xrightarrow{\mu_0} (l_1, \sigma_1) \xrightarrow{\mu_1} \cdots \) be a run of \( \Delta P \). We call a function \( \zeta : E \rightarrow 2^{Expr(A)} \) a local bound set mapping for \( \rho \) if for all \( \tau \in E \) it holds that
\[
\zeta(\tau, \rho) = (\sum_{v \in \zeta(\tau) \cap \nu} \downarrow (v, \rho)) + \sum_{\text{expr} \in \zeta(\tau) \setminus \nu} [\text{expr}] (\sigma_0).
\]
We say that \( \zeta \) is a local bound set mapping for \( \Delta P \) if \( \zeta \) is a local bound set mapping for all runs of \( \Delta P \).

Example For Fig. 7b we have that \( \zeta : E \rightarrow 2^{Expr(A)} \) with \( \zeta(\tau_0) = \{1\} \), \( \zeta(\tau_1) = \{[y]\} \), \( \zeta(\tau_2) = \{[x]\} \) and \( \zeta(\tau_3) = \{[x], [y]\} \) is a local bound set mapping.

We generalize the transition bound algorithm \( TB \) to local bound set mappings by summing up over all \( \text{expr} \in \zeta(\tau) \). We exemplify the generalization by extending Definition 19.

Definition 26 (Bound Algorithm based on Local Bound Sets) Let \( \Delta P(L, E, l_b, l_e) \) be a DCP over \( A \). Let \( \zeta : E \rightarrow 2^{Expr(A)} \). Let \( VB : A \mapsto Expr(A) \) be defined as in Definition 19. We define \( TB : E \mapsto Expr(A) \) as:
\[
TB(\tau) = \sum_{l_b \in \zeta(\tau)} TB(l_b)
\]
\[
TB(l_b) = l_b, \text{ if } l_b \notin \nu, \text{ else}
\]
\[
TB(l_b) = \text{Incr}(l_b) + \sum_{(t, a, c) \in R(l_b)} TB(t) \times \max(VB(a) + c, 0)
\]
where \( \text{Incr}(v) = \sum_{(\tau, c) \in I(v)} TB(\tau) \times c \) (we set \( \text{Incr}(v) = 0 \) for \( I(v) = \emptyset \)).

Example For Fig. 7 we get \( TB(\tau_3) = 2n \), details are shown Table 7 ([n] = n because \( n \) has type unsigned).

Inferring a Local Bound Set Mapping The algorithm for finding local bounds can be easily extended for finding local bound sets: steps (1) and (2) remain unchanged. Step (3) is generalized as follows: let \( v_1, \ldots, v_k \in \nu \) and \( \tau = l_1 \xrightarrow{\mu} l_2 \in E \). We set \( \zeta(\tau) = \{v_1, \ldots, v_k\} \) if it holds that for each execution of \( \tau \) a transition in \( \xi(v_1) \cup \cdots \cup \xi(v_k) \) is executed. This can be implemented by checking, if \( \tau \) does not belong to a strongly connected component (SCC) SCC of the directed graph \((L, E')\) where \( E' = E \cup \{l_e \xrightarrow{\emptyset} l_b\} \setminus (\xi(v_1) \cup \cdots \cup \xi(v_k)) \).

Note that Step (3) is parametrized in the number \( k \in \mathbb{N} \) of variables considered. For obvious reasons it is preferable to find local bound sets of minimal size. Given a transition...
Table 7 Computation of $TB(\tau_3)$ for Fig. 7b by Definition 26

| Call   | Evaluation and simplification                                                                 | Using                                                                 |
|--------|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| $TB(\tau_3)$ | $\rightarrow TB([x]) + TB([y])$                                                             | $\xi(\tau_3) = \{[x], [y]\}$, $TB([x])$, $TB([y])$                  |
|        | $\rightarrow [n] + TB([y])$                                                                     |                                                                      |
|        | $\rightarrow [n] + [n]$                                                                        |                                                                      |
|        | $= 2 \times [n]$                                                                               |                                                                      |
| $TB([x])$        | $\rightarrow 0 + TB(\tau_0) \times \max([n] + 0, 0)$                                             | $\mathcal{R}([x]) = \{[\tau_0, [n], 0]\}$, $\mathcal{I}([x]) = \emptyset$, $[n] \in C$, $TB(\tau_0)$ |
|        | $\rightarrow 1 \times \max([n] + 0, 0)$                                                          |                                                                      |
|        | $= [n]$                                                                                       |                                                                      |
| $TB([y])$        | $\rightarrow TB(\tau_1) \times 1$                                                             | $\mathcal{I}([y]) = \{[\tau_1, 1]\}$, $\mathcal{R}([y]) = \{[\tau_0, [0], 0]\}$, $[0] \in C$, $TB(\tau_1)$ |
|        | $\rightarrow TB(\tau_0) \times \max([0] + 0, 0)$                                               |                                                                      |
|        | $= TB(\tau_1) \times 1 + 0$                                                                    |                                                                      |
|        | $\rightarrow [n] \times 1 + 0$                                                                  |                                                                      |
|        | $= [n]$                                                                                       |                                                                      |
| $TB(\tau_1)$     | $\rightarrow TB([x])$                                                                           | $\xi(\tau_1) = \{[x]\}$, $TB([x])$                                 |
|        | $= [n]$                                                                                       |                                                                      |
| $TB(\tau_0)$     | $\rightarrow TB(1)$                                                                             | $\xi(\tau_0) = \{1\}$                                              |
|        | $\rightarrow 1$                                                                                |                                                                      |

Handling break statements Consider Fig. 8a. The loop (resp. its back-edge) can be executed $n$ times, the skip instruction (a placeholder for some code of interest), however, can be executed $n+1$ times. Consider the abstraction shown in Fig. 8b. Our algorithm for finding local bounds, as we discussed it so far, fails to find a local bound (set) for $\tau_1$ (modeling the skip instruction). We extend the algorithm as follows: we set $\xi(1) = \{\tau \in E \mid \tau$ is not part of any SCC}. I.e., for Fig. 8b we set $\xi(1) = \{\tau_0, \tau_3\}$. We add $1 \in \text{Expr}(A)$ to the set of “variables” $v_1, \ldots, v_k$. I.e., for our example we have $v_1 = [x]$ and $v_2 = 1$. The algorithm now computes $\xi(\tau_3) = \{[x], 1\}$ for $k = 2$ given that $\xi([x]) = \{\tau_2\}$. Based on $\xi(\tau_3) = \{[x], 1\}$ our algorithm from Definition 26 correctly infers $TB(\tau_3) = [n] + 1 = n + 1$ ($n$ has type unsigned).

5 Combined Bound Algorithm

We have developed our algorithm for computing transition bounds and variable bounds on DCPs step-wise in Sect. 3 (Definitions 19, 21, 23), in each step adding new features to the algorithm. In this sense Definition 23 subsumes Definitions 21 and 19. We now combine Definition 23 with the extension to sets of local bounds (Sect. 4.1) and obtain Definition 27.

Definition 27 (Combined Bound Algorithm) Let $\Delta P(L, E, l_b, l_c)$ be a DCP over $A$. Let $\xi : E \rightarrow 2^{\text{Expr}(A)}$. Let $VB : A \mapsto \text{Expr}(A)$ be defined as in Definition 19. We override the definition of $TB : E \mapsto \text{Expr}(A)$ in Definition 19 by stating:
void foo(uint n) {
    int x = n;
    while(1) {
        skip;
        if(x > 0)
            x--;
        else
            break;
    }
}

(a)

(b)

Fig. 8 a Example with a “break”-statement, b DCP obtained by abstraction

\[
TB(\tau) = \sum_{lb \in \xi(\tau)} TB(lb)
\]

\[
TB(lb) = \begin{cases} 
    lb, & \text{if } lb \not\in V, \\
    \text{Incr} \left( \bigcup_{\kappa \in \mathcal{R}(lb)} \text{atm}_1(\kappa) \right) + \sum_{\kappa \in \mathcal{R}(lb)} TB(\text{trn}(\kappa)) \times \max(\text{VB}(\text{in}(\kappa)) + c(\kappa), 0) + \text{Incr}(\text{atm}_2(\kappa)) & \text{else}
\end{cases}
\]

where \(TB([\tau_1, \tau_2, \ldots, \tau_n]) = \min_{1 \leq i \leq n} TB(\tau_i)\) and

\[
\text{Incr}([a_1, a_2, \ldots, a_n]) = \sum_{1 \leq i \leq n} \text{Incr}(a_i) \quad \text{(we set } \text{Incr}(\emptyset) = 0\text{) and}
\]

\[
\text{Incr}(v) = \sum_{(\tau, c) \in I(v)} TB(\tau) \times c \quad \text{(we set } \text{Incr}(v) = 0 \text{ for } I(v) = \emptyset).}
\]

We introduced and discussed the terms from which Definition 27 is composed in Sects. 3 and 4.1.

Soundness Soundness of Definition 27 results from Theorem 2 (proven in Appendix and the discussion in Sect. 4.1. Note that Definition 27 is only sound for DCPs that have a reset DAG.

We have described in Sect. 3.4 how to transform a given DCP into a DCP with a reset DAG.

6 Program Abstraction

In the following we discuss how we abstract a given program to a DCP.

Definition 28 (Difference Constraint Invariants) Let \(P(L, T, le, lb)\) be a program over states \(\Sigma\). Let \(e_1, e_2, e_3\) be norms, i.e., \(e_1, e_2, e_3 : \Sigma \rightarrow \mathbb{Z}\), and let \(c \in \mathbb{Z}\) be some integer. We say \(e_1' \leq e_2 + e_3\) is invariant on a transition \(l_1 \xrightarrow{\lambda} l_2 \in T\), if \(e_1(\sigma_2) \leq e_2(\sigma_1) + e_3(\sigma_1)\) holds for all \((\sigma_1, \sigma_2) \in \lambda\).

Definition 29 (DCP Abstraction of a Program) Let \(P = (L, T, lb, le)\) be a program and let \(N\) be a set of functions from the states to the natural numbers, i.e., \(N \in 2^{\Sigma \rightarrow \mathbb{N}}\). A DCP \(\Delta P = (L, E, lb, le)\) over atoms \(N\) is an abstraction of the program \(P\) iff for each
transition $l_1 \xrightarrow{\lambda} l_2 \in T$ there is a transition $l_1 \xrightarrow{u} l_2 \in E$ s.t. every $e_1' \leq e_2 + c \in u$ is invariant on $l_1 \xrightarrow{\lambda} l_2$.

Our abstraction algorithm proceeds in two steps: we first abstract a given concrete program to a DCP with integer semantics, in a second step we then further abstract the integer-DCP to a DCP over the natural numbers (as defined in Definition 12).

6.1 Abstraction I: DCPs with Integer Semantics

We extend our abstract program model from Definition 12 to the non-well-founded domain $\mathbb{Z}$ by adding guards to the transitions of the program.

Syntax of DCPs with guards The edges $E$ of a DCP with guards $\Delta P_G(L, E, l_b, l_e)$ are a subset of $L \times 2^V \times 2^{DC(A)} \times L$. I.e., an edge of a DCP with guards is of form $l_1 \xrightarrow{g,u} l_2$ with $l_1, l_2 \in L$, $g \in 2^V$ and $u \in 2^{DC(A)}$.

Example See Fig. 10a in Sect. 7.1 for an example.

Semantics of DCPs with guards We extend the range of the valuations $Val_A$ of $A$ from $\mathbb{N}$ to $\mathbb{Z}$. Let $u \in 2^{DC(A)}$. Let $[u]$ be as defined in Definition 13. Let $g \in 2^V$. We define $[g,u] = \{(\sigma_1, \sigma_2) \in [u] | \sigma_1(\nu) > 0 \text{ for all } \nu \in g\}$. A guarded DCP $\Delta P_G = (L, E, l_b, l_e)$ is a program over the set of states $Val_A$ with locations $L$, entry location $l_b$, exit location $l_e$ and transitions $T = \{(l_1, [g,u]) l_2) l_1 \xrightarrow{g,u} l_2 \in E\}$.

I.e., a transition $l_1 \xrightarrow{g,u} l_2$ of a DCP with guards can only be executed if the values of all $\nu \in g$ are greater than 0.

Definition 30 (Guard) Let $\mathcal{P}(L, T, l_e, l_b)$ be a program over states $\Sigma$. Let $e$ be a norm $e : \Sigma \rightarrow \mathbb{Z}$, let $c \in \mathbb{Z}$. We say $e$ is a guard of $l_1 \xrightarrow{\lambda} l_2 \in T$ if $e(\sigma_1) > 0$ holds for all $e(\sigma_1, \sigma_2) \in \lambda$.

We abstract a program $\mathcal{P} = (L, T, l_b, l_e)$ to a DCP with guards $\Delta P_G = (L, E, l_b, l_e)$ as follows:

1. Choosing an initial set of Norms We aim at creating a suitable abstract program for bound analysis. In our non-recursive setting setting complexity evolves from iterating loops. Therefore we search for expressions which limit the number of loop iterations. We consider conditions of form $a > b$ resp. $a \geq b$ found in loop headers or on loop-paths if they involve loop counter variables, i.e., variables which are incremented and/or decremented inside the loop. Such conditions are likely to limit the consecutive execution of single or multiple loop-paths. From each condition of form $a > b$ we create the integer expression $a - b$, from each condition of form $a \geq b$ we create the integer expression $a + 1 - b$. These expressions form our initial set of norms $N$. Note that on those transitions on which $a > b$ holds, $a - b > 0$ must hold, whereas with $a \geq b$ we have $a + 1 - b > 0$.

In $\Delta P_G$ we interpret a norm $e \in N$ from our initial set of norms $N$ as variable, i.e., we have $e \in V$ for all $e \in N$.

2. Abstracting Transitions For each transition $l_1 \xrightarrow{\lambda} l_2 \in T$ we generate a set $u_e$ of difference constraints: initially we set $u_e = \emptyset$ for all transitions $l_1 \xrightarrow{\lambda} l_2 \in T$.

We repeat the following construction until the set of norms $N$ becomes stable: For each $e_1 \in N$ and for each $l_1 \xrightarrow{\lambda} l_2 \in T$, such that all variables in $e_1$ are defined at $l_2$, we check whether there is a difference constraint of form $e_1' \leq e_2 + c$ with $e_2 \in N$ and $c \in \mathbb{Z}$.
in $u_\lambda$. If not, we derive a difference constraint $e'_1 \leq e_2 + c$ as follows: we symbolically execute $\lambda$ for deriving $e'_1$ from $e_1$: e.g., let $e_1 = x + y$ and assume $x$ is assigned $x + 1$ on $l_1 \xrightarrow{\lambda} l_2$ while $y$ stays unchanged. We get $e'_1 = x + 1 + y$ through symbolic execution. In order to keep the number of norms low, we first try

(a) to find a norm $e_2 \in N$ and $c \in \mathbb{Z}$ s.t. $e'_1 \leq e_2 + c$ is invariant on $l_1 \xrightarrow{\lambda} l_2$ (see Definition 28). If we succeed we add the predicate $e'_1 \leq e_2 + c$ to $u_\lambda$. E.g., for $e_1 = x + y$ and $e'_1 = x + 1 + y$ we get the transition invariant $(x + y)' \leq (x + y) + 1$ and will thus add $e'_1 \leq e_1 + 1$ to $u_\lambda$. In general, we find a norm $e_2$ and a constant $c$ by separating constant parts in the expression $e'_1$, using associativity and commutativity, thereby forming an expression $e_3$ over variables and program parameters and an integer constant $c$. E.g., given $e'_1 = 5 + z$ we set $e_3 = z$ and $c = 5$. We then search a norm $e_2 \in N$ with $e_2 = e_3$ where the check on equality is performed modulo associativity and commutativity.

(b) If (a) fails, i.e., no such $e_2 \in N$ exists, we add $e_3$ to $N$ and derive the predicate $e'_1 \leq e_3 + c$. In $\mathcal{AP}_G$ we interpret $e_3$ as atom, i.e., $e_3 \in \mathcal{A}$. We interpret $e_3$ as a symbolic constant, i.e., $e_3 \in \mathcal{C}$, only if $e_3$ is purely built over the program’s input parameters and constants. Note that this step increases the number of norms.

3. *Inferring Guards* For each transition $l_1 \xrightarrow{\lambda} l_2$ we generate a set $g_\lambda$ of guards: initially we set $g_\lambda = \emptyset$ for all transitions $l_1 \xrightarrow{\lambda} l_2$. For each $e \in N$ and each transition $l_1 \xrightarrow{\lambda} l_2$ we check if $e$ is a guard of $l_1 \xrightarrow{\lambda} l_2$. If so, we add $e$ to $g_\lambda$. We use an SMT solver to perform this check. E.g., let $e = x + y$ and assume that $l_1 \xrightarrow{\lambda} l_2$ is guarded by the conditions $x \geq 0$ and $y > x$. An SMT solver supporting linear arithmetic proves that $x \geq 0 \land y > x$ implies $x + y$.

4. We set $E = \{l_1 \xrightarrow{g_3,u_\lambda} l_2 \mid l_1 \xrightarrow{\lambda} l_2 \in T\}$.

Note that SMT reasoning is applied only locally to single transitions to check if an expression is greater than 0 on that transition.

**Propagation of Guards** We improve the precision of our abstraction by propagating guards: consider a transition $l_3 \xrightarrow{g_3,u_\lambda} l_4$. Assume $l_3$ has the incoming edges $l_1 \xrightarrow{g_1,u_1} l_3$ and $l_2 \xrightarrow{g_2,u_2} l_3$. If $y \in g_1 \cap g_2$ (i.e., $y$ is a guard on both incoming edges) and $y$ does not decrease on the corresponding concrete transitions $l_1 \xrightarrow{\lambda} l_3$ and $l_1 \xrightarrow{\lambda} l_3$ (checked by symbolic execution) then $y$ is also a guard on $l_3 \xrightarrow{g_3,u_\lambda} l_4$ and we add $y$ to $g_3$.

**Well-defined and Fan-in free DCPs** generated by our algorithm are always fan-in free by construction: for each transition we get at most one predicate $e' \leq e_2 + c$ for each $e \in N$ because we check whether there is already a predicate for $e$ before a predicate is inferred resp. added. We ensure well-definedness of our abstraction by a final clean-up: we iterate over all $l \in L$ and check if $\text{use}(l) \subseteq \text{def}(l)$ holds. If this check fails we remove all difference constraints $x' \leq y + c$ with $y \in \text{use}(l)\setminus\text{def}(l)$ from all outgoing edges of $l$. We repeat this iteration until well-definedness is established, i.e., until $\text{use}(l) \subseteq \text{def}(l)$ holds for all $l \in L$.

**Termination** We have to ensure the termination of our abstraction procedure, since case (b) in step “2. Abstracting Transitions” triggers a recursive abstraction for the newly added norm: note that we can always stop the abstraction process at any point, getting a sound abstraction.
of the original program. We therefore ensure termination of the abstraction algorithm by limiting the chain of recursive abstraction steps that is triggered by entering case (2.b).

**Non-linear Iterations** We can handle counter updates such as \( x' = 2x \) or \( x' = x/2 \) as follows:

1. We add the expression \( \log x \) to our set of norms.
2. We derive the difference constraint \( (\log x)' \leq (\log x) - 1 \) from the update \( x' = x/2 \) if \( x > 1 \) holds. Symmetrically we get \( (\log x)' \leq (\log x) + 1 \) from the update \( x' = 2x \) if \( x > 0 \) holds.

**Data Structures** In previous publications \([13,24]\) it has been described how to abstract programs with data structures to pure integer programs by making use of appropriate norms such as the length of a list or the number of elements in a tree. In our implementation we follow these approaches using a light-weight abstraction based on optimistic aliasing assumptions (see \([30]\) for details). Once the program is transformed to an integer program, our abstraction algorithm is applied as described above for obtaining a difference constraint program.

### 6.2 Abstraction II: From the Integers to the Natural Numbers

We now discuss how we abstract a DCP with guards \( \Delta P_G = (L, E, l_b, l_e) \) to a DCP \( \Delta P = (L, E', l_b, l_e) \) over \( \mathbb{N} \) (Definition 12):

Let \( e \in \mathbb{N} \). By \( [e] : \Sigma \rightarrow \mathbb{N} \) we denote the function \( [e](\sigma) = \max(e(\sigma), 0) \). Recall that \( e \) is interpreted as atom in \( \Delta P_G \), i.e., \( e \in A \). In \( \Delta P \), we interpret \([e]\) as variable (i.e., \([e] \in V\)) if \( e \in V \). We interpret \([e]\) as symbolic constant (i.e., \([e] \in C\)) if \( e \in C \).

Let \( l_1 \xrightarrow{g,u} l_2 \in E \). We create a transition \( l_1 \xrightarrow{u'} l_2 \in E' \) as follows: let \( e_1' \leq e_2 + c \in u \). If \( c \geq 0 \), we add \( [e_1]' \leq [e_2] + c \) to \( u' \). If \( c < 0 \) and \( e_2 \in g \) we add the constraint \([e_1]' \leq [e_2] - 1\) to \( u' \). If \( c < 0 \) and \( e_2 \notin g \) we add the constraint \([e_1]' \leq [e_2] + 0\) to \( u' \).

**Discussion** Soundness of Abstraction II is due to the following observation: consider a transition \( l_1 \xrightarrow{g,u} l_2 \) of \( \Delta P_G \). Let \( e_1' \leq e_2 + c \in u \), i.e., \( e_1' \leq e_2 + c \) is invariant for the corresponding transition \( \tau \) of the concrete program. Then \([e_1]' \leq [e_2] + 0\) is also invariant for \( \tau \). Further: if \( c \geq 0 \) then \([e_1]' \leq [e_2] + c \) is invariant for \( \tau \) and if \( c < 0 \) and \( e_2 \in g \) (i.e., \( e_2 > 0 \) must hold before executing \( \tau \)), then \([e_1]' \leq [e_2] - 1\) is invariant for \( \tau \).

**Modeling arbitrary Decrements** Consider a transition \( l_1 \xrightarrow{g,u} l_2 \) of \( \Delta P_G \). Assume \( e_1 \in G \) and \( e_1' \leq e_1 - 2 \in u \). Our abstraction procedure, as discussed so far, adds \([e_1]' \leq [e_1] - 1\) to \( u' \), where \( l_1 \xrightarrow{u'} l_2 \) is the corresponding transition of \( \Delta P \). We discuss how we can model a decrease by 2 in \( \Delta P \) (such decreases are handled by Improvement I in Sect. 3.2): let \( \tau = l_1 \Delta \xrightarrow{.} l_2 \) denote the corresponding transition of the concrete program. We make the following observation: since \( e_1 \) is a guard invariant for \( \tau \) \((e_1 > 0 \) before executing \( \tau \)) and \( e_1' \leq e_1 - 2 \) is invariant for \( \tau \) we have that \([e_1 + 1]' \leq [e_1 + 1] - 2\) is invariant for \( \tau \). We thus add the predicate \([e_1 + 1]' \leq [e_1 + 1] - 2\) to \( u' \). Further: let \( \tau' \neq \tau \) be some transition of the concrete program. If \([e_1]' \leq [e_1] + c \) is invariant for \( \tau' \) then \([e_1 + 1]' \leq [e_1 + 1] + c\) is also invariant for \( \tau' \). Let \( e_1 = e_2 \). If \([e_1]' \leq [e_2] + c \) is invariant for \( \tau' \) then \([e_1 + 1]' \leq [e_2] + (c + 1)\) is also invariant for \( \tau' \). We add the corresponding predicates to \( \Delta P \).

We can handle decrements greater than 2 accordingly: e.g., if \( e_1' \leq e_1 - 3 \in u \) we add the predicate \([e_1 + 2]' \leq [e_1 + 2] - 3\) to \( u' \), etc.
void xnu(int len) {
    int beg, end, i = 0;
    while (i < len) {
        i++;
        if (*)
            end = i;
        if (*)
            int k = beg;
        while (k < end)
            end = i;
        beg = end;
    } else if(*) {
        end = i;
        beg = end;
    }
}

Complexity: $2 \times \max(len, 0)$

(a) Example $xnu$  
(b) formal representation of $xnu$ by an LTS

### 7 A Complete Example

Example $xnu$ in Fig. 9a contains a snippet of the source code after which we have modeled Example $xnuSimple$ in Fig. 1. The full version of Example $xnu$ can be found in the SPEC CPU2006 benchmark,\(^1\) in function XNU of 456.hmmer/src/masks.c. The outer loop in Example $xnu$ partitions the interval $[0, \text{len}]$ into disjoint sub-intervals $[\text{beg}, \text{end}]$. The inner loop iterates over the sub-intervals. Therefore the inner loop has an overall linear iteration count. Example $xnu$ is a natural example for amortized complexity: Though a single visit to the inner loop can cost $\text{len}$ (if $\text{beg} = 0$ and $\text{end} = \text{len}$), several visits can also not cost more than $\text{len}$ since in each visit the loop iterates over a disjoint sub-interval. We therefore have: The amortized cost of a visit to the inner loop, i.e., the cost of executing the inner loop within an iteration of the outer loop averaged over all $\text{len}$ iterations of the outer loop, is 1. Here, we refer by cost to the number of consecutive back jumps in the inner loop. But in general, any resource consumption inside the inner loop can, in total, only be repeated up to $\max(len, 0)$ times.

Together with the loop bound $\max(len, 0)$ of the outer loop, our observation yields an overall complexity of $2 \times \max(len, 0)$.

Our experimental results (Sect. 8.3) demonstrate that state-of-the-art bound analyses fail to infer tight bounds for Example $xnu$ and similar problems.

\(^1\) [https://www.spec.org/cpu2006/](https://www.spec.org/cpu2006/)
7.1 Abstraction

We give a formal representation of the *concrete* program semantics of Example \( \exists \lambda \mu \nu \) in form of a *labeled transition system* (LTS) shown in Fig. 9b. Each edge in the LTS is labeled by a formula which encodes the transition relation. Consider, e.g., the edge from \( l_1 \) to \( l_2 \) (\( \tau_1 \)) labeled by the formula \( i < l \land b' = b \land e' = e \land i' = i + 1 \land l' = l \). This formula induces the transition relation \( \lambda_1 = \{(\sigma, \sigma') \in \Sigma \times \Sigma \mid \sigma(i) < \sigma(l) \land \sigma'(b) = \sigma(b) \land \sigma'(e) = \sigma(e) \land \sigma'(i) = \sigma(i) + 1 \land \sigma'(l) = \sigma(l)\} \).

We now discuss how our abstraction algorithm from Sect. 6 abstracts Example \( \exists \lambda \mu \nu \) to the DCP shown in Fig. 10b. Recall abstraction Step I discussed in Sect. 6.1.

1. **Choosing an initial set of Norms** Our described heuristic adds the expressions \( l - i \) and \( e - k \) generated from the conditions \( k < e \) and \( i < l \) to the initial set of norms \( N \). Thus our initial set of norms is \( N = \{l - i, e - k\} \).

2. **Abstracting Transitions**

   - We check how \( l - i \) changes on the transitions \( \tau_0, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6 \):
     - \( \tau_0 \): we derive \( [l - i] < l \) (reset), we add \( l \) to \( N \). Since \( l \) is an input parameter we have \( l \in \mathcal{C} \).
     - \( \tau_1 \): we derive \( [l - i'] \leq [l - i] - 1 \) (decrement)
     - \( \tau_2 \), \( \tau_3 \), \( \tau_4 \), \( \tau_5 \), \( \tau_6 \): \( l - i \) unchanged
   
   - We check how \( e - k \) changes on the transitions \( \tau_3, \tau_4 \) (\( k \) is only defined at \( l_4 \)):
     - \( \tau_3 \): we derive \( [e - k]' \geq [e - b] \) (reset), we add \( e - b \) to \( N \)
     - \( \tau_4 \): we derive \( [e - k]' \leq [e - k] - 1 \) (decrement)

Fig. 10 a Abstraction I: DCP with guards for Example \( \exists \lambda \mu \nu \). b Abstraction II: DCP for Example \( \exists \lambda \mu \nu \), assuming \( p, q, r, x \in \mathcal{V} \) and \( [e - k] = p, [e - b] = q, [i - b] = r, [l - i] = x \)
We check how \( e - b \) changes on the transitions \( \tau_0, \tau_1, \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_{3c}, \tau_4, \tau_5, \tau_6 \):

- \( \tau_0 \): we derive \( [e - b]' \leq 0 \) (reset), we add 0 to \( N \). Since 0 is a constant we have \( 0 \in C \).
- \( \tau_{2a} \): we derive \( [e - b]' \leq [i - b] \), we add \( i - b \) to \( N \).
- \( \tau_{3c} \): we derive \( [e - b]' \leq 0 \) (reset)
- \( \tau_5 \): we derive \( [e - b]' \leq 0 \) (reset)
- \( \tau_1, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_4, \tau_6 \): \( e - b \) unchanged

We check how \( i - b \) changes on the transitions \( \tau_0, \tau_1, \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_{3c}, \tau_4, \tau_5, \tau_6 \):

- \( \tau_0 \): we derive \( [i - b]' \leq 0 \) (reset)
- \( \tau_1 \): we derive \( [i - b]' \leq [i - b] + 1 \) (increment)
- \( \tau_{3c} \): we derive \( [i - b]' \leq 0 \) (reset)
- \( \tau_5 \): we derive \( [i - b]' \leq 0 \) (reset)
- \( \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_4, \tau_6 \): unchanged

We have processed all norms in \( N \).

3. **Inferring Guards** We add the guard \( [l - i] \) to \( \tau_1 \) in \( \Delta P_G \) because \( l - i \) is a guard of \( \tau_1 \) in \( P \) (due to the condition \( i < l \)), we add the guard \( [e - k] \) to \( \tau_4 \) in \( \Delta P_G \) because \( e - k \) is a guard of \( \tau_4 \) in \( P \) (due to the condition \( k < e \)).

4. The resulting DCP with guards is shown in Fig. 10a.

Applying abstraction Step II discussed in Sect. 6.2 gives us the DCP shown in Fig. 10b. In the depiction of the abstraction we assume \( p, q, r, x \in V \) and \( [e - k] = p, [e - b] = q, [i - b] = r, [l - i] = x \).

### 7.2 Bound Computation

In Fig. 11 the reset graph of Fig. 10b is shown. Table 8 shows how our bound algorithm from Sect. 3 infers the linear bound \( \max(l, 0) \) for the inner loop at \( l_4 \) of Example xnu by computing \( TB(\tau_4) = [l] \) on the abstraction shown in Fig. 10b. Recall that the abstract variable \( [l] \) represents the expression \( \max(l, 0) \) in the concrete program.

Note that the DCP in Fig. 10b has a reset forest. Therefore \( atm_1(\kappa) = atm(\kappa) \) for all reset paths \( \kappa \) of Fig. 10b, as discussed in Sect. 3.3.2. The computation traces of Definitions 21 and 23 are thus equivalent for Fig. 10b.
Table 8  Computation of $TB(\tau_4)$ for Example $xnu$ (Fig. 9) by Definition 21 resp. Definition 23

| Call       | Evaluation and simplification | Using |
|------------|-------------------------------|-------|
| $TB(\tau_4)$ | $\rightarrow \text{Incr}((p, q, r))$ | $\xi(\tau_4) = p$, $\eta(p) = \emptyset$, $\rho_0$ |
|            | $+ TB((\tau_5, \tau_2, \tau_3)) \times \max([0] + 0, 0)$ | $[0] \xrightarrow{\tau_0} r \xrightarrow{\tau_2} q \xrightarrow{\tau_3} p$, $[0] \xrightarrow{\tau_5} r \xrightarrow{\tau_2} q \xrightarrow{\tau_3} p$, $[0] \xrightarrow{\tau_5} q \xrightarrow{\tau_3} p$, $[0] \xrightarrow{\tau_5} q \xrightarrow{\tau_3} p$, $[0] \in C$, $\text{Incr}((p, q, r))$ |
|            | $+ TB((\tau_1, \tau_3)) \times \max([0] + 0, 0)$ | |
|            | $+ TB((\tau_3, \tau_3)) \times \max([0] + 0, 0)$ | |
|            | $= \text{Incr}((p, q, r)) + 0 + 0 + 0 + 0 + 0 + 0 + 0$ | |
|            | $\rightarrow [l] + 0 + 0 + 0 + 0 + 0 + 0 + 0$ | |
|            | $= [l]$ | |

| $\text{Incr}((p, q, r))$ | $\rightarrow \text{Incr}(p) + \text{Incr}(q) + \text{Incr}(r)$ | $\text{Incr}(p)$, $\text{Incr}(q)$, $\text{Incr}(r)$ |
| $\text{Incr}(p)$ | $\rightarrow 0$ | $\mathcal{I}(p) = \emptyset$ |
| $\text{Incr}(q)$ | $\rightarrow 0$ | $\mathcal{I}(q) = \emptyset$ |
| $\text{Incr}(r)$ | $\rightarrow TB(\tau_1) \times 1$ | $TB(\tau_1)$ |
|            | $\rightarrow [l] \times 1$ | $\mathcal{I}(r) = ([\tau_1, 1])$ |
|            | $= [l]$ | |

| $TB(\tau_1)$ | $\rightarrow \text{Incr}(x) + TB(t_0) \times [l]$ | $\xi(\tau_1) = x$, $\eta(x) = [l] \rightarrow x$, $[l] \in C$, $\text{Incr}(x), TB(t_0)$ |
|            | $\rightarrow 0 + TB(t_0) \times [l]$ | |
|            | $\rightarrow 0 + 1 \times [l]$ | |
|            | $= [l]$ | |

| $\text{Incr}(x)$ | $\rightarrow 0$ | $\mathcal{I}(x) = \emptyset$ |
| $TB(t_0)$ | $\rightarrow 1$ | $\xi(t_0) = 1$ |

8 Experiments

Implementation The presented analysis defines the core ideas and techniques of our implementation loopus. A complete description of the implemented techniques, including a path-sensitive extension of our bound algorithm, is given in [30]. Our implementation is open-source and available at [18]. loopus reads in the LLVM [23] intermediate representation and performs an intra-procedural analysis. It is capable of computing bounds for loops as well as analyzing the complexity of non-recursive functions.

In the following we discuss three experimental setups and tool comparisons. Our first experiment, which we discuss in Sect. 8.1 is performed on a benchmark of open-source C programs. For our second experiment (Sect. 8.2), we assembled a benchmark of challenging programs from the literature on automatic bound analysis. The third experiment was performed on a set of interesting loop iteration patterns that we found in real source code.
Table 9 Tool results on analyzing the complexity of 1659 functions in the cBench benchmark, none of the tools infers $\log$ bounds

| Tool      | Succ. | $1$ | $n$ | $n^2$ | $n^3$ | $n^{>3}$ | $2^n$ | Total time | # Time outs |
|-----------|-------|-----|-----|-------|-------|----------|-------|------------|-------------|
| loopus'15 | 806   | 205 | 489 | 97    | 13    | 2        | 0     | 15 m       | 6           |
| loopus'14 | 431   | 200 | 188 | 43    | 0     | 0        | 0     | 40 m       | 20          |
| KoAT      | 430   | 253 | 138 | 35    | 2     | 0        | 2     | 5.6 h      | 161         |
| CoFloCo   | 386   | 200 | 148 | 38    | 0     | 0        | 0     | 4.7 h      | 217         |

8.1 Evaluation on Real World C Code

Experimental Setup We base our experiment on the program and compiler optimization benchmark Collective Benchmark [17] (cBench), which contains a total of 1027 different C files (after removing code duplicates) with 211,892 lines of code. We set up the first comparison of complexity analysis tools on real-world code. For comparing the tool (loopus’15) we chose the three most promising tools from recent publications: the tool KoAT implementing the approach of [6], the tool CoFloCo implementing [10] and our own earlier implementation loopus’14 [29]. Note that we compared against the most recent versions of KoAT and CoFloCo (download 01/23/15). We were not able to evaluate RANK (implementing [2]) and C4B (implementing [7]) on our benchmark because both tools support only a limited subset of C. The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor and 16 GB memory. The task was to perform a complexity analysis on function level. We used the following experimental set up:

1. We compiled all 1027 C files in the benchmark into the LLVM intermediate representation using clang.
2. We extracted all 1751 functions which contain at least one loop using the tool llvm-extract (comes with the LLVM tool suite). Extracting the functions to single files guarantees an intra-procedural setting for all tools.
3. We used the tool llvm2kittel [20] to translate the 1751 LLVM modules into 1751 text files in the integer transition system (ITS) format that is read in by KoAT.
4. We used the transformation described in [10] to translate the ITS format of KoAT into the cost equations representation that is read in by CoFloCo. This last step is necessary because there exists no direct way for translating C or the LLVM intermediate representation into the CoFloCo input format.
5. We decided to exclude the 91 recursive functions from the benchmark set because we were not able to run CoFloCo on these examples (the transformation tool does not support recursion), KoAT was not successful on any of them, and loopus does not support recursion. In total our example set thus comprises 1659 functions.

Evaluation Table 9 shows the results of all four tools on our benchmark using a time out of 60 s (Table 10 shows the results on the subset of those functions on which no tool timed out). The first column shows the number of functions which were successfully bounded by the respective tool, the last column shows the number of time outs, on the remaining examples (not shown in the table) the respective tool did not time out but was also not able to compute a bound. The column Time shows the total time used by the respective tool to
process the benchmark. loopus’15 obtains results for about twice as many functions as KoAT, CoFloCo, and loopus’14 while needing an order of magnitude less time than KoAT and CoFloCo, and significantly less time than loopus’14. We conclude that our implementation is both more scalable and, on real C code, more successful than implementations of other state-of-the-art approaches. However, while the experiment clearly demonstrates that our implementation outperforms the competitors with respect to scalability, it does not allow to compare the strengths of the different bound analyses conclusively: we observed that llvm2kittel, the only tool available for translating C-code resp. the LLVM intermediate representation into the ITS format of KoAT, looses information that is kept by our analysis. As a result, it is unclear if a failure to compute a bound is due to different analysis strength or due to information loss during translation (we have not seen such an information loss for our second and third experiment, on which we report in Sects. 8.2 and 8.3, where the considered benchmarks consist of rather small, pure integer programs for which llvm2kittel works well).

We hope that our experiment motivates the development of better tools for bound analysis of real world-code and drives the research towards solving realistic complexity analysis problems. We want to add that to our experience, working with C programs instead of integer transition systems is very helpful for developing and debugging a complexity analysis tool: looking at C code, we can use our own intuition as programmers about the expected complexity of the analyzed code and compare it to the complexity reported by the tool.

**Pointers and Shapes** Even loopus’15 computed bounds for only about half of the functions in the benchmark. Studying the benchmark code we concluded that for many functions pointer alias and/or shape analysis is needed for inferring functional complexity. In our experimental comparison such information was not available to the tools. Using optimistic (but unsound) assumptions on pointer aliasing and heap layout, our tool loopus’15 was able to compute the complexity for in total 1185 out of the 1659 functions in the benchmark, using 28 min total time. A discussion of our optimistic pointer aliasing and heap layout assumption and on the reasons of failure can be found in [30].

The benchmark and more details on our experimental results can be found on [18] where our tool is also offered for download.

### 8.2 Evaluation on Examples from the Literature

In order to evaluate the precision of our approach on a number of known challenges to bound analysis, we performed a tool comparison on 110 examples from the literature. Our example set comprises those examples from the tool evaluation in [6,29] that were available as imperative code (C or pseudo code, in total 89 examples), and additionally the examples used for the evaluation of Ref. [7] (15 examples) as well as the running examples of Ref. [27]
(6 examples). We added the tools Rank (implementing [2]) and C4B (implementing [7]) to the comparison, because we were able to formulate the examples over the restricted C subset that is supported by these two tools (this was not possible for our experiment on real-world code).

The results of our evaluation are shown in Table 11. Our two tools loopus’15 and loopus’14 compute the highest number of linear bounds and are also significantly faster than the other tools, in particular than KoAT and CoFloCo. On the other hand, KoAT computes the highest number of bounds in total (4 more than loopus). CoFloCo computes, in total, 1 bound more than our tool. The comparable low number of bounds computed by C4B is also due to the fact that the approach implemented in C4B is limited to linear bounds.

In summary, our second evaluation shows that our approach is not only successful on the class of problems on which we focused in this article, but solves also many other bound analysis problems from the literature. Note, that in contrast to our first evaluation, our second benchmark contains small examples from academia (1293 LOC, in average 12 lines per file). On these examples our implementation is comparable in strength to the implementation of other state-of-the-art approaches to bound analysis. Given that our tool is a prototype implementation, there is room for improvement, concrete suggestions are discussed in [30]. More details on the results computed by each tool can be found on [19].

8.3 Evaluation on Challenging Iteration Patterns from Real Code

Scanning through two C-code benchmarks (cBench [17] and SPEC CPU 2006 [21]), we found a number of 23 different loop iteration patterns which we consider to be particular challenging for state-of-the-art bound analyses. The 23 patterns have the following property in common: (1) there is an inner loop L with loop counter c, such that c is increased on an outer loop of L. (2) Nevertheless, the amortized cost of L (the overall worst-case cost of executing L, averaged over the number of executions of its outer loop) is lower than the worst-case cost of a single execution (a single instance of consecutive iterations) of L.

Example xnu (discussed in Sect. 7) is a natural example for the described behaviour.

The complete benchmark is available at [19]. For each pattern we link its origin in the header of the respective file. Note that for some patterns we found several instances.

Table 12 states the results that were obtained by loopus’15, loopus’14, CoFloCo, KoAT, Rank and C4B: ‘✓’ denotes that the bound computed by the respective tool is tight (in the same asymptotic class as the precise bound, see Definition 5), ‘O(n^x)’ denotes that the respective tool did not infer a tight bound but a bound in the asymptotic class O(n^x), ‘✗’ denotes that no bound was inferred, ‘TO’ denotes that the tool timed out (the time

| Tool         | Succ. | n | n^2 | n^3 | n^>3 | 2^n | Time w/o TO | # Time outs |
|--------------|-------|---|-----|-----|------|-----|-------------|------------|
| loopus’15    | 86    | 2 | 51  | 27  | 1    | 5   | 0           | 4 s        | 0          |
| loopus’14    | 86    | 2 | 50  | 28  | 2    | 4   | 0           | 4 s        | 0          |
| CoFloCo      | 87    | 3 | 45  | 34  | 2    | 3   | 0           | 1 min 40 s | 1          |
| KoAT         | 78    | 3 | 49  | 21  | 3    | 2   | 0           | 20 s       | 0          |
| C4B          | 36    | 0 | 36  | 0   | 0    | 0   | 0           | 6 s        | 0          |
| KoAT         | 90    | 3 | 43  | 36  | 3    | 5   | 0           | 3 min 50 s | 3          |

The time out was 120 s. A higher time out did not yield additional results
Table 12  Tool results on 23 challenging loop iteration patterns from cBench and SPEC CPU 2006 Benchmarks

|                  | loopus’15 | loopus’14 | CoFloCo | KoAT | Rank | C4B |
|------------------|-----------|-----------|---------|------|------|-----|
| cf_decode_eol    | O(n)      | ✓         | X       | O(n²) | ✓    | X   |
| cryptRandWriteFile| O(n)      | ✓         | O(n²)   | ✓    | ✓    | ✓   |
| encode_mcu_AC_refine | O(n)    | ✓         | X       | O(n²) | ✓    | ✓   |
| hc_compute      | O(n²)     | ✓         | O(n³)   | O(n³) | TO   | ✓   |
| inflated_stored | O(n)      | ✓         | ✓       | X    | X    | X   |
| PackBitsEncode  | O(n)      | ✓         | X       | TO   | O(n²) | O   |
| s_SFD_process   | O(n)      | ✓         | X       | O(n²) | X    | X   |
| send_tree       | O(n)      | ✓         | ✓       | O(n²) | ✓    | O   |
| sendMTFValues   | O(n)      | ✓         | ✓       | O(n²) | ✓    | O   |
| set_color_ht    | O(n²)     | ✓         | O(n³)   | ✓    | X    | X   |
| subsetdump      | O(n)      | ✓         | ✓       | O(n²) | O    | X   |
| zwritehexstring_at | O(n)   | ✓         | O(n²)   | ✓    | O(n²) | ✓   |
| analyse_other   | O(n³)     | O(n⁴)    | O(n⁴)   | X    | O(n¹⁵) | X   |
| ApplyBndRobin   | O(n³)     | ✓         | ✓       | X    | ✓    | O   |
| asctoeg         | O(n²)     | ✓         | ✓       | O(n³) | ✓    | X   |
| Configure       | O(n)      | O(n²)    | O(n²)   | O(n²) | ✓    | X   |
| load_mems       | O(n)      | ✓         | O(n³)   | X    | O(n³) | ✓   |
| local_alloc     | O(n)      | ✓         | O(n²)   | ✓    | O(n²) | ✓   |
| ParseFile       | O(n)      | ✓         | ✓       | TO   | O(n³) | O   |
| Perl_scan_vstring | O(n) | ✓         | O(n²)   | X    | O(n²) | ✓   |
| SingleLinkCluster | O(n²) | ✓         | ✓       | X    | TO   | X   |
| xdr3dfcoord     | O(n)      | ✓         | ✓       | O(n²) | O(n²) | O   |
| XNU              | O(n)      | ✓         | O(n²)   | O(n²) | O(n²) | ✓   |

Total tight 21 12 6 2 7 6
Total Over-approx. 2 10 7 18 2 0
Total fail 0 1 8 1 10 16
Total timed Out 0 0 2 2 0 0
Total time 2 s 1 s 41 m 74 m 28 s 20 m
Total time w/o TO 2 s 1 s 1 m 34 m 28 s 20 m

The time out was 20 min, a longer time out did not yield additional results
a bound, but returned within the time out limit of 20 min. ‘Total Timed Out’ states the total number of examples on which the respective tool timed out (the time out limit was 20 min). ‘Total Time’ states the overall time consumed by the respective tool for processing the complete benchmark. ‘Total Time w/o TO’ states the overall time consumed by the respective tool on those examples on which the tool did not time out.

loopus'15 fails to infer a tight bound only for Configure and analyse_other. For both examples a precise bound can be obtained by an improvement of our variable bound function (VB) described in [30] which is not yet implemented into our tool. loopus'15 is far more successful in inferring tight bounds for the examples than any of the competitors. loopus'15 infers 21, loopus'14 12, Rank 7, C4B 6, CoFloCo 6 and KoAT 2 tight bounds. There are 9 examples for which only our tool loopus (loopus’15 and loopus’14) infers a tight bound: cf_decode_eol, PackBitsEncode, s_SFD_process, send_tree, subsetdump, ParseFile, SingleLinkCluster, xdr3dfcoord, and XNU.

The experiment demonstrates, that our bound analysis complements the state-of-the-art, by inferring tight bounds for a class of real-world loop iterations, on which existing techniques mostly fail or obtain coarse over-approximations.

Technical remarks (1) We counted the time needed by the tool Aspic (a preprocessor for Rank which performs invariant generation) into the time of the bound analysis performed by Rank. (2) Rank reported an unsound bound and an error message for the examples s_SFD_process.c, load_mems.c and SingleLinkCluster.c. On these examples we therefore assessed Rank’s return value as fail (‘✗’).

9 Amortized Complexity Analysis

In the following we discuss how our approach relates to amortized complexity analysis as introduced by Tarjan in his influential paper [32]. We recall Tarjan’s idea of using potential functions for amortized analysis in Sect. 9. In Sect. 9.1 we explain how our approach can be viewed as an instantiation of amortized analysis via potential functions.

Amortized Analysis using Potential Functions Amortized complexity analysis [32] aims at inferring the worst-case average cost over a sequence of calls to an operation or function rather than the worst-case cost of a single call. In (resource) bound analysis the difference between the single worst-case cost and the amortized cost is relevant, e.g., if a function $f$ is called inside a loop: assume the loop bound is $n$ and the single worst-case cost of a call to $f$ is also $n$. The cost of a single call to $f$ amortized over all $n$ calls might, however, be lower than $n$, e.g., 2. In this case the total worst-case cost of iterating the loop is $2n$ rather than $n^2$. Note that in our non-recursive setting function calls can always be inlined. The amortized analysis problem thus boils down to the problem of inferring the cost of executing an inner loop averaged over all executions of the outer loop.

Tarjan [32] motivates amortized complexity analysis on the example of a program which executes $n$ stack operations StackOp. Each StackOp operation consists of a push instruction, adding an element to the stack, followed by a pop instruction, removing an arbitrary number of elements from the stack. Initially the stack is empty. The cost of a single push is 1 and the cost of a single pop is the number of elements removed from the stack. Tarjan points out that the worst-case cost of a single pop is $n$: the $n$th pop instruction may pop $n$ elements (cost $n$) from the stack, if the previous pop instructions did not remove any elements from the stack. i.e., the worst-case cost of a single StackOp operation is $n + 1$. Nevertheless
all $n$ operations $StackOp$ cannot cost more than $2n$ in total since we cannot remove more elements from the stack than have been added to the stack and thus the overall cost of the $pop$ instructions is bounded by the total number of $push$ instructions ($n$ by assumption). The amortized cost of $StackOp$, i.e., the cost of $StackOp$ averaged over the sequence of all $n$ operations, is therefore $2$.

**Potential Function** As a means to reason about the amortized cost of an operation or a sequence of operations, Tarjan introduces the notion of a potential function. A potential function is a function $\Phi : \Sigma \rightarrow \mathbb{Z}$ from the program states to the integers. Let $C_{op}(\sigma)$ denote the cost of executing operation $op$ at program state $\sigma \in \Sigma$. Let $\Phi$ be a potential function. Tarjan defines the amortized cost $C_{op}^A(\sigma)$ as $C_{op}^A(\sigma) = C_{op}(\sigma) + \Phi(\sigma') - \Phi(\sigma)$ where $\sigma$ denotes the program state before and $\sigma'$ denotes the program state after executing $op$. I.e., the amortized cost is the cost plus the decrease resp. increase in the value of the potential.

Consider a sequence of $n$ operations, let $op_i$ denote the $i$th operation in the sequence. Let $\sigma_i$ denote the program state before executing operation $op_i$, $\sigma_{i+1}$ is the program state after executing $op_i$. In general, the total cost of executing all $n$ operations is:

$$\sum_{i=1}^{n} C_{op_i}(\sigma_i) = \sum_{i=1}^{n} C_{op_i}^A(\sigma_i) - \Phi(\sigma_{i+1}) + \Phi(\sigma_i) = \Phi(\sigma_1) - \Phi(\sigma_{n+1}) + \sum_{i=1}^{n} C_{op_i}^A(\sigma_i) \quad (1)$$

That is, the total time of the operations equals the sum of their amortized times plus the net decrease in potential from the initial to the final configuration. [...] In most cases of interest, the initial potential is zero and the potential is always non-negative. In such a situation the total amortized time is an upper bound on the total time \[32\]. I.e., if $\Phi_i \geq 0$ and $\Phi_0 = 0$ then

$$\sum_{i=1}^{n} C_{op_i}(\sigma_i) \leq \sum_{i=1}^{n} C_{op_i}^A(\sigma_i) \quad (2)$$

Reconsider Tarjan’s previously discussed example of a sequence of $n$ executions of operation $StackOp$. Let $j$ denote the stack size, i.e., $\sigma_i(j)$ is the size of the stack in program state $\sigma_i$. The cost of executing $StackOp$ in program state $\sigma_i$ is $C_{StackOp}(\sigma_i) = 1 + (\sigma_i(j) + 1 - \sigma_i(j))$ (where $\sigma_i(j) + 1 - \sigma_i(j)$ is the cost of the $pop$ operation). Tarjan proposes to use the stack size $j$ as a potential function, i.e. we choose $\Phi(\sigma_i) = \sigma_i(j)$. We have

$$C_{StackOp}^A(\sigma_i) = C_{StackOp}(\sigma_i) + \Phi(\sigma_{i+1}) - \Phi(\sigma_i)$$
$$= C_{StackOp}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j)$$
$$= 1 + (\sigma_i(j) + 1 - \sigma_{i+1}(j)) + \sigma_{i+1}(j) - \sigma_i(j)$$
$$= 2$$

With (2) we get:

$$\sum_{i=1}^{n} C_{StackOp_i}(\sigma_i) \leq \sum_{i=1}^{n} C_{StackOp_i}^A(\sigma_i) = \sum_{i=1}^{n} 2 = 2n$$

### 9.1 Amortized Analysis in our Algorithm

Example tarjan in Fig. 12 is a model of Tarjan’s motivating example (discussed above): variable $j$ models the stack size. The $push$ instruction is modeled by increasing the stack size $j$ by 1. The $pop$ instruction is modeled by decreasing the stack size. Further all calls
void tarjan(uint n) {
    uint i = n, j = 0;
    l1: while (i > 0) {
        i--;
        j++; //push
    }
    l2: while (j > 0 && *) //pop
    j--;
}
Complexity: 2n

(a)

Fig. 12  a Example tarjan  b LTS of Example tarjan  c DCP obtained by abstraction from tarjan

to StackOp, push and pop are inlined. Consider the labeled transition system of Example tarjan shown in Fig. 12b. We have that transition $\tau_1$ models the push instruction, increasing the stack size $j$ by 1, a sequence of transitions $\tau_2$ models the pop instruction, decreasing the stack by an arbitrary number of elements. A complete run $\rho$ of Example tarjan can be decomposed into the initial transition $\tau_0$ and a number of sub-runs $\rho[i_k,i_{k+1}]$ with $1 \leq i_1 < i_2 \ldots$ s.t. each $\rho[i_k,i_{k+1}]$ consists of a single transition $\tau_1$ (push) followed by a sequence of transitions $\tau_2$ (pop), followed by a single execution of transition $\tau_3$. Each sub-run $\rho[i_k,i_{k+1}]$ models Tarjan’s StackOp operation. We thus have that the amortized cost of a sub-run $\rho[i_k,i_{k+1}]$ is 2. Given that $\tau_1$ cannot be executed more than $n$ times and each $\rho[i_k,i_{k+1}]$ contains exactly one $\tau_1$, we get that the overall cost of executing Example tarjan is bounded by $n \times 2 = 2n$.

In the following we argue that our transition bound algorithm $TB$ is an instantiation of amortized analysis using potential functions. We base our discussion on the concrete semantics of Example tarjan given by the LTS in Fig. 12b. Note, however, that our algorithm runs on the abstracted DCP in Fig. 12c where the same reasoning applies: suppose we want to compute the transition bound of transition $\tau_2$ in order to compute the total cost of the pop instructions. Let $\rho = (\sigma_0, l_0) \xrightarrow{l_0} (\sigma_1, l_1) \xrightarrow{l_1} \ldots$ be a run of Example tarjan. Let $len(\rho)$ denote the length of $\rho$ (i.e., total number of transitions on $\rho$). We define the cost of executing $\tau_2$ in program state $\sigma_i$ as $C_{\tau_2}(\sigma_i) = 1$ and the cost of executing $\tau_1$ and $\tau_3$ as $C_{\tau_1}(\sigma_i) = C_{\tau_3}(\sigma_i) = 0$ since we are only interested in $\tau_2$. We have

$$z(\tau_2, \rho) = \sum_{i=1}^{len(\rho)-1} C_{\rho(i)}(\sigma_i)$$

where $\rho(i)$ denotes the $i + 1$th transition $l_i \xrightarrow{u_i} l_{i+1}$ on $\rho$. Our algorithm reduces the question “how often can $\tau_2$ be executed?” to the question “how often can the local bound ’$j$’ of $\tau_2$ be increased on $\tau_1$?”. This reasoning uses the local bound $j$ of $\tau_2$ as a potential function, as we show next: we get the following amortized costs for executing $\tau_1$, $\tau_2$ and $\tau_3$ respectively:

$$C_{\tau_2}^A(\sigma_i) = C_{\tau_2}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 1 + \sigma_{i+1}(j) - \sigma_i(j) = 0$$

$$C_{\tau_1}^A(\sigma_i) = C_{\tau_1}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 0 + \sigma_{i+1}(j) - \sigma_i(j) = 1$$

$$C_{\tau_3}^A(\sigma_i) = C_{\tau_3}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 0 + \sigma_{i+1}(j) - \sigma_i(j) = 0$$
With $\sigma_i(j) \geq 0$, $\sigma_1(j) = 0$ and (2) we have:

$$\sharp(\tau_2, \rho) = \sum_{i=1}^{\text{len}(\rho) - 1} C_{\rho(i)}(\sigma_i) \leq \sum_{i=1}^{\text{len}(\rho) - 1} C^A_{\rho(i)}(\sigma_i) = \sharp(\tau_1, \rho) \times 1$$

We point out that choosing the local bound $j$ of $\tau_2$ as potential function causes the amortized cost of executing $\tau_2$ to be 0 and reduces the question how often $\tau_2$ can be executed to how often the potential $j$ can be incremented on $\tau_1$.

Using $\sharp(\tau_1, \rho) \leq \sharp(\tau_0, \rho) \times n = n$ one obtains the upper bound $n$ for the total cost of the pop instructions.

## Conclusion

We presented a new approach to (resource) bound analysis. Our approach complements existing approaches in several aspects, as discussed in Sect. 2.3. Our analysis handles bound analysis problems of high practical relevance which current approaches cannot handle: current techniques [6,7,10,29] fail on Example xnux and similar problems. We have argued that such problems, e.g., occur naturally in parsing and string matching routines. During our experiments on real-world source code, we found 23 different iteration patterns that pose a challenge for similar reasons as Example xnux: in these patterns, the worst-case cost of a single inner loop execution is lower than the worst-case cost of the inner loop averaged over the iterations of the outer loop. Our implementation obtains tight bounds for 21 out of these 23 iteration patterns (Sect. 8.3).

Our algorithm (Sect. 3) obtains invariants by means of bound analysis and does not rely on external techniques for invariant generation. This is in contrast to current bound analysis techniques (see discussion on related work in Sect. 2). We have compared our algorithm to classical invariant analysis and argued that we can efficiently compute invariants which are difficult to obtain by standard abstract domains such as octagon or polyhedra (Sect. 2). We have demonstrated that the limited form of invariants (upper bound invariants) that our algorithm obtains is sufficient for the bound analysis of a large class of real-world programs.

We have demonstrated that difference constraints are a suitable abstract program model for automatic complexity and resource bound analysis. Despite their syntactic simplicity, difference constraints are expressive enough to model the complexity-related aspects of many imperative programs. In particular, difference constraints allow to model amortized complexity problems such as the bound analysis challenge posed by Example xnux (discussed in Sect. 7). We developed appropriate techniques for abstracting imperative programs to DCPs (Sect. 6): we described how to extract norms (integer-valued expressions over the program state) from imperative programs and showed how to use these norms as variables in DCPs.

Our approach deals with many challenges bound analysis is known to be confronted: in Sect. 8.2 we compared our tool on a benchmark of challenging problems from publications on bound analysis. The results show that our prototype implementation can handle most of these problems. Here, our implementation, while comparable in terms of strengths to other implementations of state-of-the-art bound analysis techniques, performs the task significantly faster than the competitors. The results obtained by our prototype tool could be further enhanced by extending our implementation with additional techniques discussed in [30].

We stress that our approach is more scalable than existing approaches. We presented empirical evidence of the good performance characteristics of our analysis by a large exper-
iment and tool comparison on real source code in Sect. 8.1. We discuss the main technical reasons for scalability of our analysis in Sect. 10.1.

We think that the abstract program model of difference constraint programs is worth further investigation: given that difference constraints can model standard counter manipulations (counter increments, decrements and resets), a further research on complexity analysis of difference constraint programs is of high value. We consider DCPs to be a very suitable program model for studying the principle challenges of automated complexity and resource bound analysis for imperative programs.

10.1 Discussion on the Scalability of Our Analysis

In the following we state what we consider to be the main technical reasons that make our analysis scale:

First of all, we achieve scalability by local reasoning: note that our abstraction procedure relies on purely local information, i.e., information that is available on single program transitions. In particular, we do not apply global invariant analysis. Further, the sets $S(v)$ and $R(v)$, by which our main algorithm is parametrized, are built by categorizing the difference constraints on single (abstract) program transitions based on simple syntactic criteria. Our algorithm for computing the local bound mapping $\xi$ (Sect. 4) is polynomial even in the generalized case (Sect. 4.1).

We use bound analysis to infer bounds on variable values (variable bounds). Unlike classical invariant analysis this approach is demand-driven and does not perform a fixed point iteration (see discussion in Sect. 2.3).

Note that the only general purpose reasoner we employ is an SMT solver. Further, the SMT solver is only employed in the program abstraction phase. In terms of size, the problems we feed to the SMT solver are small, namely simple linear arithmetic formulas, composed of the arithmetic of single transitions. Our approach instruments the SMT solver only for yes/no answers, no optimal solution (e.g., minimum or minimal unsatisfiable core) is required.

Our basic bound algorithm (Definition 19) runs in polynomial time. The reasoning based on reset chains (Definition 23), however, has exponential worst-case complexity, resulting from the potentially exponential number of paths in the program (exponential in the number of program transitions). We did not experience this to be an issue in practice because the simplicity of our abstract program model allows to take straightforward engineering measures: program slicing reduces the number of paths in the program significantly, further, merging of similar paths can be applied (details are given in [28]).

Acknowledgements  Open access funding provided by TU Wien.

Open Access  This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. 413(1), 142–159 (2012)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14–16, 2010. Proceedings. Springer Berlin Heidelberg, Berlin (2010)
3. Bagnara, R., Mesnard, F., Pescetti, A., Zaffanella, E.: A new look at the automatic synthesis of linear ranking functions. Inf. Comput. 215, 47–67 (2012)

4. Ben-Amram, A.M.: Size-change termination with difference constraints. ACM Trans. Program. Lang. Syst. TOPLAS, 30(3), (2008). doi:10.1145/1353445.1353450

5. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst. TOPLAS 29(1), 5 (2007)

6. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. 38(4), 13:1–13:50 (2016)

7. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: PLDI (2015)

8. Colcombet, T., Daviaud, L., Zuleger, F.: Size-change abstraction and max-plus automata. In: MFCS, pp. 208–219 (2014)

9. Coppa, E., Demetrescu, C., Finocchi, I.: Input-sensitive profiling. In: PLDI, pp. 89–98 (2012)

10. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: APLAS, pp. 275–295 (2014)

11. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)

12. Gulwani, S., Juvekar, S.: Bound analysis using backward symbolic execution. Technical Report MSR-TR-2004-95, Microsoft Research (2009)

13. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL, pp. 239–251 (2009)

14. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: POPL, pp. 127–139 (2009)

15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI, pp. 292–304 (2010)

16. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. 34(3), 14 (2012)

17. http://ctuning.org/wiki/index.php/CTools:CBench

18. http://forsyte.at/software/loopus/

19. http://forsyte.at/static/people/sinn/loopusJAR/

20. https://github.com/s-falke/llvm2kittel

21. https://www.spec.org/cpu2006/

22. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. In: PLDI, pp. 77–88 (2012)

23. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88 (2004)

24. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL, pp. 211–222 (2010)

25. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI, pp. 239–251 (2004)

26. Seidl, H., Gawlitza T.M., Schwarz, M.: Parametric strategy iteration. In: Kutsia, T., Voronkov, A. (eds.) SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science, Volume 30 of EPiC Series in Computing, pp. 62–76. EasyChair (2014)

27. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: an adequate abstraction for complexity analysis of imperative programs. In: FMCAD, pp. 144–151 (2015)

28. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. CoRR, abs/1401.5842 (2014)

29. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: CAV, pp. 745–761. Springer (2014)

30. Sinn, M.: Automated complexity analysis for imperative programs. Ph.D. thesis, TU Wien, Faculty of Informatics, Wien (2016)

31. Smith, G.: On the foundations of quantitative information flow. In: FOSSACS, pp. 288–302 (2009)

32. Tarjan, R.E.: Amortized computational complexity. SIAM J. Algebraic Discrete Methods 6(2), 306–318 (1985)

33. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Theising, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puata, I., Pushcher, P., Stachulkat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7(3), 36 (2008). doi:10.1145/1347375.1347389

34. Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. In: PLDI, pp. 67–76 (2012)

35. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: SAS, pp. 280–297 (2011)