Difference Constraints: An adequate Abstraction for Complexity Analysis of Imperative Programs

Moritz Sinn, Florian Zuleger, Helmut Veith
TU Wien, Austria

arXiv:1508.04958v1 [cs.PL] 20 Aug 2015

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 $c \in \mathbb{Z}$. In this paper, we argue that the complexity of imperative programs typically arises from counter increments and resets, which can be modeled naturally by difference constraints. We present the first practical algorithm for the analysis of difference constraint programs and describe how C programs can be abstracted to difference constraint programs. Our approach contributes to the field of automated complexity and (resource) bound analysis by enabling automated amortized complexity analysis for a new class of programs and providing a conceptually simple program model that relates invariant- and bound analysis. We demonstrate the effectiveness of our approach through a thorough experimental comparison on real world C code: our tool Loopus computes the complexity for considerably more functions in less time than related tools from the literature.**

I. 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 [14], C# [13], C [5], [20], [16], Java [4] and Integer Transition Systems [4], [7], [10].

Difference constraints (DCs) have been introduced by Ben-Amram for termination analysis in [6], 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).

In this paper, we advocate the use of DCs for program complexity and (resource) bounds analysis. Our key insight is that DCs provide a *natural abstraction* of the standard manipulations of counters in imperative programs: counter *increments/decrements* $x := x + c$ resp. *resets* $x := y$, can be modeled by the DCs $x' \leq x + c$ resp. $x' \leq y$ (see Section IV on program abstraction). In contrast, previous approaches to bound analysis can model either only resets [13], [5], [20], [4], [7], [10] or increments [16]. For this reason, we are able to design a more powerful analysis: In Section II-A we discuss that our approach achieves *amortized analysis* for a new class of programs. In Section II-B we describe how our approach performs *invariant analysis* by means of bound analysis.

In this paper, we establish the practical usefulness of DCs for bound (and complexity) analysis of imperative programs: 1) We propose the first algorithm for bound analysis of $DCPs$. Our algorithm is based on the dichotomy between increments and resets. 2) We develop appropriate techniques for abstracting C programs to $DCPs$: we describe how to extract *norms* (integer-valued expressions on the program state) from C programs and how to use them as variables in $DCPs$. We are not aware of any previous implementation of $DCPs$ for termination or bound analysis. 3) We demonstrate the effectiveness of our approach through a thorough experimental evaluation. We present the first comparison of bound analysis tools on source code from real software projects (see Section V). Our implementation performs significantly better in time and success rate.

II. MOTIVATION AND RELATED WORK

A. Amortized Complexity Analysis

Example 1 stated in Figure 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, we refer the reader to Example 3 stated in Appendix A, which contains a snippet of the source code after which we have modeled Example 1. Example 1 has the linear *complexity* $2n$, because the inner loop as well as the outer loop can be iterated at most n times (as argued in the next paragraph). However, previous approaches to bound analysis [13], [5], [20], [16], [4], [7], [10] are only able to deduce that the inner loop can be iterated at most a *quadratic* number of times (with loop bound n^2) by the following reasoning: (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. We note that inferring the linear complexity $2n$ for Example 1, 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* [18].

In the following, we give an overview how our approach infers the linear complexity for Example 1:

1. Program Abstraction. We abstract the program to a DCP over $\mathbb{Z}$ as shown in Figure 1. We discuss our algorithm for abstracting imperative programs to DCPs based on symbolic

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

abstracted DCP of Example 1

l_b

$\tau_0 \equiv \begin{array}{l} x' \le n; \\ r' \le 0; \end{array}$

$l_1 \longrightarrow l_e$

$\tau_1 \equiv \begin{array}{l} x > 0, \\ x' \le x - 1 \\ r' \le r + 1 \end{array}$

l_2

$\tau_5 \equiv \begin{array}{l} r' \le r \\ x' \le x \end{array}$

$\tau_{2a} \equiv \begin{array}{l} x' \le x \\ r' \le r \\ p' \le r \end{array}$

$\tau_{2b} \equiv \begin{array}{l} x' \le x \\ r' \le r \end{array}$

$l_4 \longleftarrow l_3$

$\tau_4 \equiv \begin{array}{l} x' \le x \\ r' \le 0 \end{array}$

$\tau_3 \equiv \begin{array}{l} p > 0, \\ x' \le x \\ r' \le r \\ p' \le p - 1 \end{array}$

Complexity: $TB(\tau_5) + TB(\tau_3) = n + n = 2n$

Example 1

```
foo(uint n, uint m1,
    uint m2) {
    int y = n;
    int x;
l1  if(*)
        x = m1;
    else
        x = m2;
l2  while(y > 0) {
        y--;
        x = x + 2; }
    int z = x;
l3  while(z > 0)
        z--; }
```

abstracted DCP of Example 2

l_b

$\tau_0 \equiv y' \le n$

l_1

$\tau_{0a} \equiv \begin{array}{l} y' \le y \\ x' \le m1 \end{array}$

$\tau_{0b} \equiv \begin{array}{l} y' \le y \\ x' \le m2 \end{array}$

l_2 $\tau_1 \equiv \begin{array}{l} y > 0, \\ y' \le y - 1 \\ x' \le x + 2 \end{array}$

$\tau_2 \equiv z' \le x;$

$l_3 \longrightarrow l_e$

$\tau_3 \equiv \begin{array}{l} z > 0, \\ z' \le z - 1 \end{array}$

Complexity: $TB(\tau_1) + TB(\tau_3) = \max(m_1, m_2) + 3n$
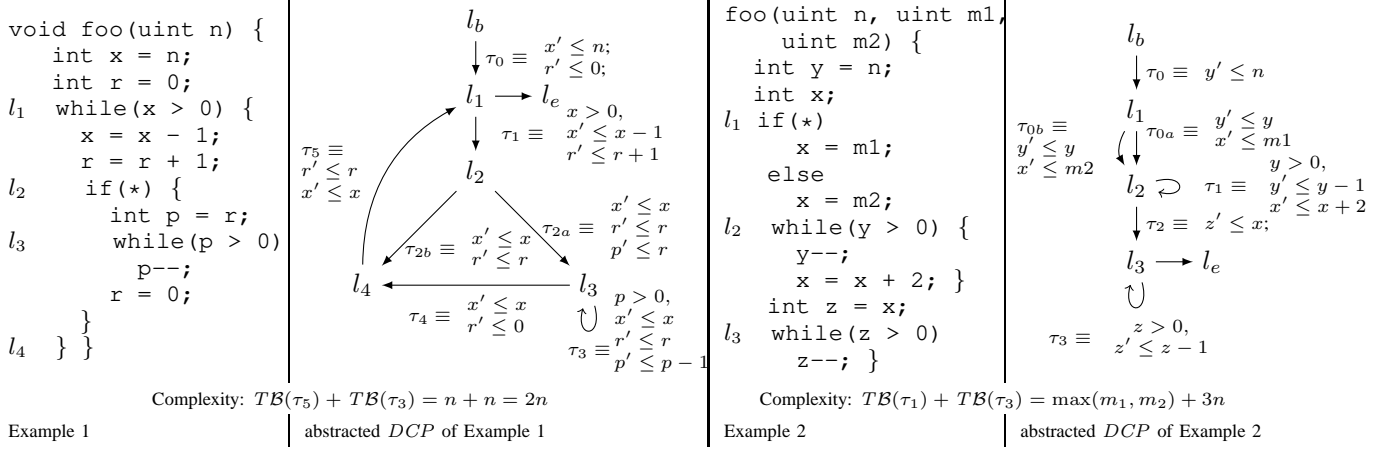
Example 2

Fig. 1. Running Examples, * denotes non-determinism (arising from conditions not modeled in the analysis)

execution in Section IV.

2. Finding Local Bounds. We identify p as a variable that limits the number of executions of transition τ_3: We have the guard $p > 0$ on τ_3 and p decreases on each execution of τ_3. We call p a *local bound* for τ_3. Accordingly we identify x as a *local bound* for transitions $\tau_1, \tau_{2a}, \tau_{2b}, \tau_4, \tau_5$.

3. Bound Analysis. Our algorithm (stated in Section III) 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 $VB(v)$) and transition bound analysis ("how often", function $TB(\tau)$). Next, we give an intuition how our algorithm computes transition bounds: Our algorithm computes $TB(\tau) = n$ for $\tau \in \{\tau_1, \tau_{2a}, \tau_{2b}, \tau_4, \tau_5\}$ because the local bound x is initially set to n and never increased or reset. Our algorithm computes $TB(\tau_3)$ (τ_3 corresponds to the loop at l_3) as follows: τ_3 has local bound p; p is reset to r on τ_{2a}; our algorithm detects that before each execution of τ_{2a}, r is reset to 0 on either τ_0 or τ_4, which we call the *context* under which τ_{2a} 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 $TB(\tau_3) = 0 + TB(\tau_1) \times 1 = n$. We give an example for the mutual recursion between TB and VB in Section II-B.

We contrast our approach for computing the loop bound of l_3 of Example 1 with classical invariant analysis: Assume 'c' counting the number of inner loop iterations (i.e., c is initialized to 0 and incremented in the inner loop). For inferring $c \le n$ through invariant analysis the invariant $c + x + r \le n$ is needed for the outer loop, and the invariant $c + x + p \le n$ for the inner loop. Both relate 3 variables and cannot be expressed as (parametrized) octagons (e.g., [11]). 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.

B. Invariants and Bound Analysis

We explain on Example 2 in Figure 1 how our approach performs *invariant analysis* by means of bound analysis. We first motivate the importance of invariant analysis for bound analysis. 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 $x \le \text{expr}(n, m_1, m_2)$. Here, the most precise invariant $x \le \max(m_1, m_2) + 2n$ 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 $x \le m_1 + m_2 + 2n$. Unfortunately, it is well-known that the polyhedra abstract domain does not scale to larger programs and needs to rely on heuristics for termination. Next, we explain how our approach computes invariants using bound analysis and discuss how our reasoning is substantially different from invariant analysis by abstract interpretation.

Our algorithm computes a transition bound for the loop at l_3 by $TB(\tau_3) = TB(\tau_2) \times VB(x) = 1 \times VB(x) = VB(x) = TB(\tau_1) \times 2 + \max(m_1, m_2) = (n \times TB(\tau_0)) \times 2 + \max(m_1, m_2) = (n \times 1) \times 2 + \max(m_1, m_2) = 2n + \max(m_1, m_2)$. We point out the mutual recursion between TB and VB: $TB(\tau_3)$ has called $VB(x)$, which in turn called $TB(\tau_1)$. We highlight that the variable bound $VB(x)$ (corresponding to the invariant $x \le \max(m_1, m_2) + 2n$) has been established during the computation of $TB(\tau_3)$.

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, $VB(x)$ infers the invariant $x \le \max(m1, m2) + 2n$ by modular reasoning: *local information* about the program (i.e., increments/resets of variables, local bounds of transitions) 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

2

will find applications outside of bound analysis.

C. Related Work

In [6] it is shown that termination of $DCPs$ is undecidable in general but decidable for the natural syntactic subclass of *deterministic DCPs* (see Definition 3), which is the class of $DCPs$ we use in this paper. It is an open question for future work whether there is a complete algorithm for bound analysis of deterministic $DCPs$.

In [16] 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. The resulting abstract program model is strictly less powerful than $DCPs$. In [20] a bound analysis based on so-called *size-change constraints* $x' \lhd y$ is proposed, where $\lhd \in \{<, \leq\}$. Size-change constraints form a strict syntactic subclass of DCs. However, termination is decidable even for non-deterministic size-change programs and a complete algorithm for deciding the complexity of size-change programs has been developed [9]. Because the constraints in [20], [16] are less expressive than DCs, the resulting bound analyses cannot infer the linear complexity of Example 1 and need to rely on external techniques for invariant analysis.

In Section V we compare our implementation against the most recent approaches to automated complexity analysis [10], [7], [16]. [10] extends the COSTA approach by control flow refinement for cost equations and a better support for multi-dimensional ranking functions. The COSTA project (e.g. [4]) computes resource bounds by inferring an upper bound on the solutions of certain recurrence equations (so-called *cost equations*) relying on external techniques for invariant analysis (which are not explicitly discussed). The bound analysis in [7] uses approaches for computing polynomial ranking functions from the literature to derive bounds for SCCs in isolation and then expresses these bounds in terms of the function parameters using invariant analysis (see next paragraph).

The powerful idea of expressing locally computed loop bounds in terms of the function parameters by alternating between loop bound analysis and variable upper bound analysis has been explored in [7], [16] (as discussed in the extended version [17]) and [12]. We highlight some important differences to these earlier works. [7] computes upper bound invariants only for the *absolute* values of variables; this does, for example, not allow to distinguish between variable increments and decrements during the analysis. [17] and [12] do not give a general algorithm but deal with specific cases.

[19] discusses automatic parallelization of loop iterations; the approach builds on summarizing inner loops by multiplying the increment of a variable on a single iteration of a loop with the loop bound. The loop bounds in [19] are restricted to simple syntactic patterns.

The recent paper [8] 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. Interestingly, [8] is able to compute the linear bound for l_3 of Example 1 but fails to deduce the bound for the original source code (provided in Appendix A).Moreover,

[8] is restricted to linear bounds, while our approach derives polynomial bounds (e.g., Example B in Figure 2) which may also involve the maximum operator. An experimental comparison was not possible as [8] was developed in parallel.

III. 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 variable (variable bound). We base our algorithm on the abstract program model of DCPs stated in Definition 3. In Section III-B we generalize DCPs and our algorithm to the non-well-founded domain $\mathbb{Z}$.

Definition 1 (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. $\mathcal{A} = \mathcal{V} \cup \mathcal{C} \cup \mathbb{N}$ is the set of* atoms.

Definition 2 (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 $c \in \mathbb{Z}$. We denote by $\mathcal{DC}(\mathcal{A})$ the set of all difference constraints over $\mathcal{A}$.*

Definition 3 (Difference Constraint Program). *A difference constraint program (DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta \mathcal{P} = (L, T, l_b, l_e)$, 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 $T \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{u} l_2$ to denote a transition $(l_1, u, l_2) \in T$ labeled by a set of difference constraints $u \in 2^{\mathcal{DC}(\mathcal{A})}$. Given a transition $\tau = l_1 \xrightarrow{u} l_2 \in T$ of $\Delta \mathcal{P}$ we call l_1 the source location of τ and l_2 the target location of τ. A path of $\Delta \mathcal{P}$ is a sequence $l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \cdots$ with $l_i \xrightarrow{u_i} l_{i+1} \in T$ for all i. The set of valuations of $\mathcal{A}$ is the set $Val_{\mathcal{A}} = \mathcal{A} \to \mathbb{N}$ of mappings from $\mathcal{A}$ to the natural numbers with $\sigma(\mathtt{a}) = \mathtt{a}$ if $\mathtt{a} \in \mathbb{N}$. A run of $\Delta \mathcal{P}$ is a sequence $(l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ such that $l_b \xrightarrow{u_0} l_1 \xrightarrow{u_1} \cdots$ is a path of $\Delta \mathcal{P}$ and for all i it holds that (1) $\sigma_i \in Val_{\mathcal{A}}$, (2) $\sigma_{i+1}(x) \leq \sigma_i(y) + c$ for all $x' \leq y + c \in u_i$, (3) $\sigma_i(s) = \sigma_0(s)$ for all $s \in \mathcal{C}$. Given $\mathtt{v} \in \mathcal{V}$ and $l \in L$ we say that $\mathtt{v}$ is defined at l and write $\mathtt{v} \in \mathcal{D}(l)$ if $l \neq l_b$ and for all incoming transitions $l_1 \xrightarrow{u} l \in T$ of l it holds that there are $\mathtt{a} \in \mathcal{A}$ and $c \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{a} + c \in u$.*
$\Delta \mathcal{P}$ is deterministic *(fan-in-free in the terminology of [6]), if for every transition $l_1 \xrightarrow{u} l_2 \in T$ and every $\mathtt{v} \in \mathcal{V}$ there is at most one $\mathtt{a} \in \mathcal{A}$ and $c \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{a} + c \in u$.*

Our approach assumes the given DCP to be *deterministic*. We further assume that DCPs are *well-defined*: Let $\mathtt{v} \in \mathcal{V}$ and $l \in L$, if $\mathtt{v}$ is *live* at l then $\mathtt{v} \in \mathcal{D}(l)$. Our abstraction algorithm from Section IV generates only deterministic and well-defined DCPs.

In Definitions 4 to 11 we assume a $DCP \Delta \mathcal{P}(L, T, l_b, l_e)$ over $\mathcal{A}$ to be given.

Definition 4 (Transition Bound). *Let $\tau \in T$, τ is bounded iff τ appears a finite number of times on any run of $\Delta \mathcal{P}$. An expression $\mathtt{expr}$ over $\mathcal{C} \cup \mathbb{Z}$ is a transition bound for τ iff τ is bounded and for any finite run $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \dots (l_e, \sigma_n)$ of $\Delta \mathcal{P}$ it holds that τ appears not more than $\sigma_0(\mathtt{expr})$ often on ρ. We say that a transition bound*
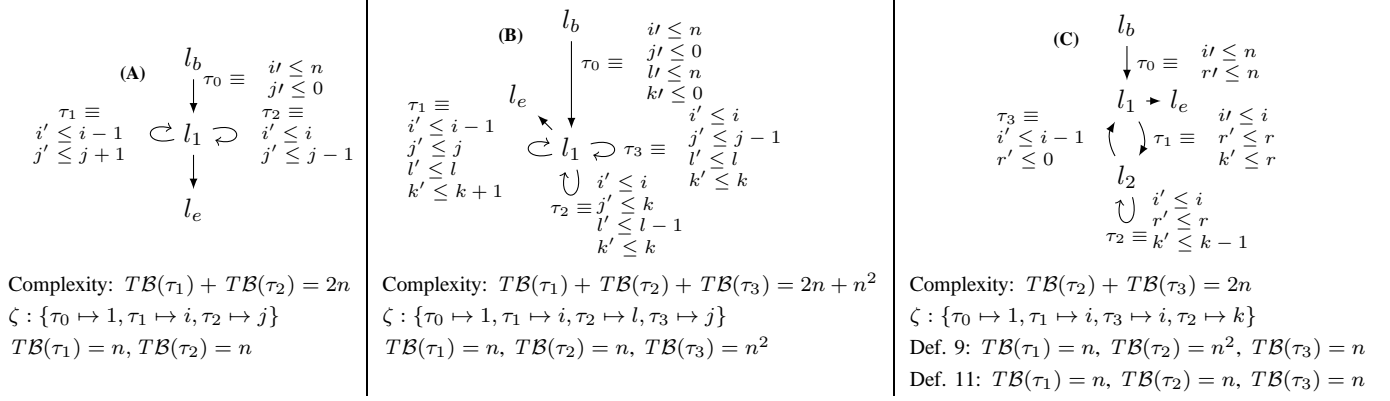
Fig. 2. Example DCP's (A), (B), (C)

(A) Complexity: $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_2) = 2n$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_2 \mapsto j\}$
$T\mathcal{B}(\tau_1) = n, T\mathcal{B}(\tau_2) = n$

(B) Complexity: $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_2) + T\mathcal{B}(\tau_3) = 2n + n^2$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_2 \mapsto l, \tau_3 \mapsto j\}$
$T\mathcal{B}(\tau_1) = n, T\mathcal{B}(\tau_2) = n, T\mathcal{B}(\tau_3) = n^2$

(C) Complexity: $T\mathcal{B}(\tau_2) + T\mathcal{B}(\tau_3) = 2n$
$\zeta : \{\tau_0 \mapsto 1, \tau_1 \mapsto i, \tau_3 \mapsto i, \tau_2 \mapsto k\}$
Def. 9: $T\mathcal{B}(\tau_1) = n, T\mathcal{B}(\tau_2) = n^2, T\mathcal{B}(\tau_3) = n$
Def. 11: $T\mathcal{B}(\tau_1) = n, T\mathcal{B}(\tau_2) = n, T\mathcal{B}(\tau_3) = n$

expr of τ is precise *iff there is a run ρ of $\Delta\mathcal{P}$ s.t. τ appears $\sigma_0(\mathtt{expr})$ times on ρ.*

We want to infer the complexity of the examples in Figure 2 (Examples A, B, C), i.e., we want to infer how often location l_1 can be visited during an execution of the program. We will do so by computing a bound on the number of times transitions τ_0, τ_1, τ_2 and τ_3 may be executed. In general, the complexity of a given program can be inferred by summing up the transition bounds for the back edges in the program.

Definition 5 (Counter Notation). *Let $\tau \in T$ and $\mathtt{v} \in \mathcal{V}$. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots (l_e, \sigma_n)$ be a finite run of $\Delta\mathcal{P}$. By $\sharp(\tau, \rho)$ we denote the number of times that τ occurs on ρ. By $\downarrow(\mathtt{v}, \rho)$ we denote the number of times that the value of $\mathtt{v}$ decreases on ρ, i.e. $\downarrow(\mathtt{v}, \rho) = |\{i \mid \sigma_i(\mathtt{v}) > \sigma_{i+1}(\mathtt{v})\}|$.*

Definition 6 (Local Transition Bound). *Let $\tau \in T$ and $\mathtt{v} \in \mathcal{V}$. $\mathtt{v}$ is a local bound for τ iff on all finite runs $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ it holds that $\sharp(\tau, \rho) \leq \downarrow(\mathtt{v}, \rho)$. We call a complete mapping $\zeta : T \to \mathcal{V} \cup \{\mathbf{1}\}$ a local bound mapping for $\Delta\mathcal{P}$ if $\zeta(\tau)$ is a local bound of τ or $\zeta(\tau) = \mathbf{1}$ and τ can only appear at most once on any path of $\Delta\mathcal{P}$.*

Example A: i *is a local bound for* τ_1, j *is a local bound for* τ_2. *Example C:* i *is a local bound for* τ_1 *and for* τ_3.

A variable $\mathtt{v}$ is a *local transition bound* if on any run of $\Delta\mathcal{P}$ we can traverse τ not more often than the number of times the value of $\mathtt{v}$ decreases. I.e., a local bound $\mathtt{v}$ limits the potential number of executions of τ as long as the value of $\mathtt{v}$ does not increase. In our analysis, *local transition bounds* play the role of *potential functions* in classical *amortized complexity analysis* [18]. Our bound algorithm is based on a mapping which assigns each transition a local bound. We discuss how we find local bounds in Section III-C.

Definition 7 (Variable Bound). *An expression expr over $\mathcal{C} \cup \mathbb{Z}$ is a variable bound for $\mathtt{v} \in \mathcal{V}$ iff for any finite run $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \ldots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ and all $1 \leq i \leq n$ with $\mathtt{v} \in \mathcal{D}(l_i)$ it holds that $\sigma_i(\mathtt{v}) \leq \sigma_0(\mathtt{expr})$.*

Let $\mathtt{v} \in \mathcal{V}$. Our algorithm is based on a *syntactic* distinction between transitions which *increment* $\mathtt{v}$ or *reset* $\mathtt{v}$.

Definition 8 (Resets and Increments). *Let $\mathtt{v} \in \mathcal{V}$. We define the resets $\mathcal{R}(\mathtt{v})$ and increments $\mathcal{I}(\mathtt{v})$ of $\mathtt{v}$ as follows:*
$$\mathcal{R}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in T \times \mathcal{A} \times \mathbb{Z} \mid$$
$$\mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u, \mathtt{a} \neq \mathtt{v}\}$$
$$\mathcal{I}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in T \times \mathbb{Z} \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u, \mathtt{c} > 0\}$$
Given a path π of $\Delta\mathcal{P}$ we say that $\mathtt{v}$ is reset on π if there is a transition τ on π such that $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ for some $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$.

Example B: $\mathcal{I}(k) = \{(\tau_1, 1)\}$ and $\mathcal{R}(k) = \{(\tau_0, n, 0)\}$.
I.e., we have $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ if variable $\mathtt{v}$ is reset to a value $\leq \mathtt{a} + \mathtt{c}$ when executing the transition τ. Accordingly we have $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ if variable $\mathtt{v}$ is incremented by a value $\leq \mathtt{c}$ when executing the transition τ.

Our algorithm in Definition 9 is build on a *mutual recursion* between the two functions $V\mathcal{B}(\mathtt{v})$ and $T\mathcal{B}(\tau)$, where $V\mathcal{B}(\mathtt{v})$ infers a *variable bound* for $\mathtt{v}$ and $T\mathcal{B}(\tau)$ infers a *transition bound* for the transition τ.

Definition 9 (Bound Algorithm). *Let $\zeta : T \to \mathcal{V} \cup \{\mathbf{1}\}$ be a local bound mapping for $\Delta\mathcal{P}$. We define $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ and $T\mathcal{B} : T \mapsto Expr(\mathcal{A})$ as:*
$$V\mathcal{B}(\mathtt{a}) = \mathtt{a}, \text{ if } \mathtt{a} \in \mathcal{A} \setminus \mathcal{V}, \text{ else}$$
$$V\mathcal{B}(\mathtt{v}) = \mathtt{Incr}(\mathtt{v}) + \max_{(\_, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})}(V\mathcal{B}(\mathtt{a}) + \mathtt{c})$$

$$T\mathcal{B}(\tau) = \mathbf{1}, \text{ if } \zeta(\tau) = \mathbf{1}, \text{ else}$$
$$T\mathcal{B}(\tau) = \mathtt{Incr}(\zeta(\tau))$$
$$+ \sum_{(\mathtt{t}, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} T\mathcal{B}(\mathtt{t}) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$$

where
$$\mathtt{Incr}(\mathtt{v}) = \sum_{(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})} T\mathcal{B}(\tau) \times \mathtt{c} \ (\mathtt{Incr}(\mathtt{v}) = 0 \text{ for } \mathcal{I}(\mathtt{v}) = \emptyset)$$

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

The function $V\mathcal{B}(\mathtt{v})$ computes a variable bound for $\mathtt{v}$: After executing a reset transition $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$, the value of $\mathtt{v}$ is

bounded by $VB(\mathtt{a}) + \mathtt{c}$. As long as $\mathtt{v}$ is not *reset*, its value cannot increase by more than $\mathtt{Incr}(\mathtt{v})$.

The function $TB(\tau)$ computes a transition bound for τ based on the following reasoning: (1) The total amount by which the local bound $\zeta(\tau)$ of transition τ can be *incremented* is bounded by $\mathtt{Incr}(\zeta(\tau))$. (2) We consider a reset $(\mathtt{t}, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))$; in the worst case, a single execution of $\mathtt{t}$ resets the local bound $\zeta(\mathtt{t})$ to $VB(\mathtt{a}) + \mathtt{c}$, adding $\max(VB(\mathtt{a}) + \mathtt{c}, 0)$ to the potential number of executions of $\mathtt{t}$; in total all $TB(\mathtt{t})$ possible executions of $\mathtt{t}$ add up to $TB(\mathtt{t}) \times \max(VB(\mathtt{a}) + \mathtt{c}, 0)$ to the potential number of executions of $\mathtt{t}$.

Example A, ζ as defined in Figure 2: j is *reset* to 0 on τ_0 and incremented by 1 on τ_1. i is *reset* to n on τ_0. Our algorithm computes $TB(\tau_2) = TB(\tau_1) \times 1 + TB(\tau_0) \times 0 = TB(\tau_1) = TB(\tau_0) \times n = n$. Thus the overall complexity of Example A is inferred by $TB(\tau_1) + TB(\tau_2) = 2n$.

Example B, ζ as defined in Figure 2: i and l are *reset* to n on τ_0. Our algorithm computes $TB(\tau_1) = TB(\tau_0) \times n = n$ and $TB(\tau_2) = TB(\tau_0) \times n = n$. j is *reset* to 0 on τ_0 and *reset* to k on τ_2. Our algorithm computes $TB(\tau_3) = TB(\tau_0) \times 0 + TB(\tau_2) \times VB(k)$. Since k is *reset* to 0 on τ_0 and incremented by 1 on τ_1, our algorithm computes $VB(k) = TB(\tau_1) \times 1 = n \times 1 = n$. Thus $TB(\tau_3) = TB(\tau_2) \times VB(k) = n \times n = n^2$. Thus the overall complexity of Example B is inferred by $TB(\tau_1) + TB(\tau_2) + TB(\tau_3) = n + n + n^2 = 2n + n^2$.

Example 2 (Figure 1): $\zeta = \{\tau_0, \tau_{0_a}, \tau_{0_b}, \tau_2 \mapsto 1, \tau_1 \mapsto y, \tau_3 \mapsto z\}$, $\mathcal{R}(z) = \{(\tau_2, x, 0)\}$, $\mathcal{I}(x) = \{(\tau_1, 2)\}$, $\mathcal{R}(x) = \{(\tau_{0_a}, m1, 0), (\tau_{0_b}, m2, 0)\}$, $\mathcal{R}(y) = \{(\tau_0, n, 0)\}$. We have stated the computation of $TB(\tau_3)$ in Section II-B.

Termination: Our algorithm does not terminate if recursive calls cycle, i.e., if a call to $TB(\tau)$ resp. $VB(\mathtt{v})$ (indirectly) leads to a recursive call to $TB(\tau)$ resp. $VB(\mathtt{v})$. This can be easily detected, we return the value $\bot$ (undefined).

Theorem 1 (Soundness). *Let* $\Delta\mathcal{P}(L, T, l_b, l_e)$ *be a well-defined and deterministic DCP over atoms* $\mathcal{A}$, $\zeta : T \mapsto \mathcal{V} \cup \{1\}$ *be a* local bound mapping *for* $\Delta\mathcal{P}$, $\mathtt{v} \in \mathcal{V}$ *and* $\tau \in T$. *Either* $TB(\tau) = \bot$ *or* $TB(\tau)$ *is a* transition bound *for* τ. *Either* $VB(\mathtt{v}) = \bot$ *or* $VB(\mathtt{v})$ *is a* variable bound *for* $\mathtt{v}$.

A. Context-Sensitive Bound Analysis

So far our algorithm reasons about resets occurring on single transitions. In this section we increase the precision of our analysis by exploiting the context under which resets are executed through a refined notion of resets and increments.

Definition 10 (Reset Graph). *The* Reset Graph *for* $\Delta\mathcal{P}$ *is the graph* $\mathcal{G}(\mathcal{A}, \mathcal{E})$ *with* $\mathcal{E} \subseteq \mathcal{A} \times T \times \mathbb{Z} \times \mathcal{V}$ *s.t.* $\mathcal{E} = \{(x, \tau, \mathtt{c}, y) \mid (\tau, y, \mathtt{c}) \in \mathcal{R}(x)\}$. *We call a* finite *path* $\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \ldots \mathtt{a}_0$ *in* $\mathcal{G}$ *with* $n > 0$ *a* reset path *of* $\Delta\mathcal{P}$. *We define* $in(\kappa) = \mathtt{a}_n$, $c(\kappa) = \sum_{i=1}^{n} c_i$, $trn(\kappa) = \{\tau_n, \tau_{n-1} \ldots, \tau_1\}$, *and* $atm(\kappa) = \{\mathtt{a}_n, \mathtt{a}_{n-1} \ldots, \mathtt{a}_0\}$. κ *is* sound *if for all* $1 \leq i < n$ *it holds that* $\mathtt{a}_i$ *is reset on all paths from the target location of* τ_1 *to the source location of* τ_i *in* $\Delta\mathcal{P}$. κ *is* optimal *if* κ *is sound and there is no sound reset path* $\hat{\kappa}$ *s.t.* κ *is a suffix of* $\hat{\kappa}$, *i.e.,* $\hat{\kappa} = \mathtt{a}_{n+k} \xrightarrow{\tau_{n+k}, c_{n+k}} \mathtt{a}_{n+k-1} \xrightarrow{\tau_{n+k-1}, c_{n+k-1}} \ldots \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \ldots \mathtt{a}_0$
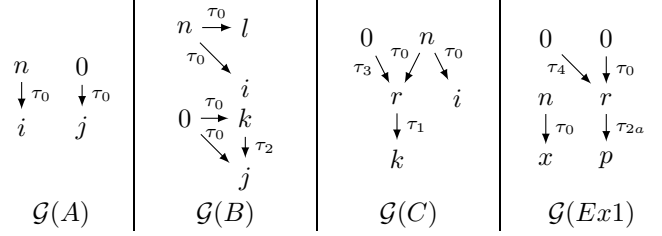

Fig. 3. Reset Graphs, increments by 0 are not depicted

with $k \geq 1$. *Let* $\mathtt{v} \in \mathcal{V}$, *by* $\mathfrak{R}(\mathtt{v})$ *we denote the set of optimal reset paths ending in* $\mathtt{v}$.

We explain the notions *sound* and *optimal* in the course of the following discussion. Figure 3 shows the reset graphs of Examples A, B, C and Example 1 from Figure 1. For a given reset $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$, the reset graph determines which atom flows into variable $\mathtt{v}$ under which context. For example, consider $\mathcal{G}(C)$: When executing the reset $(\tau_1, r, 0) \in \mathcal{R}(k)$ under the context τ_3, k is set to 0, if the same reset is executed under the context τ_0, k is set to n. Note that the reset graph does not represent *increments* of variables. We discuss how we handle increments below.

We assume that the reset graph is a DAG. We can always force the reset graph to be a DAG by abstracting the DCP: we remove all program variables which have cycles in the reset graph and all variables whose values depend on these variables. Note that if the reset graph is a DAG, the set $\mathfrak{R}(\mathtt{v})$ is finite for all $\mathtt{v} \in \mathcal{V}$.

Let $\mathtt{v} \in \mathcal{V}$. Given a reset path κ of length k that ends in $\mathtt{v}$, we say that $(trn(\kappa), in(\kappa), c(\kappa))$ is a reset of $\mathtt{v}$ with context of length $k - 1$. I.e., $\mathcal{R}(\mathtt{v})$ from Definition 8 is the set of *context-free* resets of $\mathtt{v}$ (context of length 0), because $(trn(\kappa), in(\kappa), c(\kappa)) \in \mathcal{R}(\mathtt{v})$ iff κ ends in $\mathtt{v}$ and has length 1. Our algorithm from Definition 9 reasons *context free* since it uses only *context-free* resets.

Consider Example C. The precise bound for τ_2 is n because we can iterate τ_2 only in the first iteration of the loop at l_1 since r is reset to 0 on τ_3. But when reasoning context-free, our algorithm infers a *quadratic* bound for τ_2: We assume ζ to be given as stated in Figure 2. In $\mathcal{G}(C)$ $\kappa = r \xrightarrow{\tau_1, 0} k$ is the only reset path of length 1 ending in k. Thus $\mathcal{R}(k) = \{(\tau_1, r, 0)\}$. Our algorithm from Definition 9 computes: $TB(\tau_1) = TB(\tau_0) \times n = n$, $VB(r) = TB(\tau_0) \times n + TB(\tau_3) \times 0 = n$, $TB(\tau_2) = TB(\tau_1) \times VB(r) = n \times n = n^2$.

We show how our algorithm infers the *linear* bound for τ_2 when using *resets with context*: If we consider κ with contexts, we get $\kappa_1 = 0 \xrightarrow{\tau_3, 0} r \xrightarrow{\tau_1, 0} k$ and $\kappa_2 = n \xrightarrow{\tau_0, 0} r \xrightarrow{\tau_1, 0} k$. Note that κ_1 and κ_2 are *sound* by Definition 10 because r is reset on all paths from the target location l_2 of τ_1 to the source location l_1 of τ_1 in Example C (namely on τ_3). Thus $\mathfrak{R}(k) = \{(\{\tau_3, \tau_1\}, 0, 0), (\{\tau_0, \tau_1\}, n, 0)\}$. 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 $\min_{1 \leq i \leq n} TB(\tau_i)$. Thus, basing our analysis on $\mathfrak{R}(k)$ rather than $\mathcal{R}(k)$ we compute: $TB(\tau_2) = \min(TB(\tau_3), TB(\tau_1)) \times 0 + \min(TB(\tau_0), TB(\tau_1)) \times n = \min(n, 1) \times n = n$.

We have demonstrated that our analysis gains precision when adding context to our notion of resets. It is, however, not sound to base the analysis on maximal reset paths (i.e., resets with maximal context) only: Consider Example B with ζ as stated in Figure 2. There are 2 maximal reset paths ending in j (see $\mathcal{G}(B)$): $\kappa_1 = 0 \xrightarrow{\tau_0,0} j$ and $\kappa_2 = 0 \xrightarrow{\tau_0,0} k \xrightarrow{\tau_2,0} j$. Thus $\mathfrak{R}(j)' = \{(\{\tau_0, \tau_2\}, 0, 0), (\{\tau_0\}, 0, 0)\}$ is the set of resets of j with *maximal* context. Using $\mathfrak{R}(j)'$ rather than $\mathcal{R}(j)$ our algorithm computes: $T\mathcal{B}(\tau_3) = \min(T\mathcal{B}(\tau_0), T\mathcal{B}(\tau_2)) \times 0 + T\mathcal{B}(\tau_0) \times 0 + T\mathcal{B}(\tau_1) \times 1 = T\mathcal{B}(\tau_1) \times 1 = n$, but n is not a transition bound for τ_3. The reasoning is unsound because κ_2 is *unsound* by Definition 10: k is *not* reset on all paths from the target location l_1 of τ_2 to the source location l_1 of τ_2 in Example B: e.g., the path $\tau_2 = l_1 \xrightarrow{u_2} l_1$ of Example B does not reset k.

We base our *context sensitive* algorithm on the set $\mathfrak{R}(v)$ of *optimal* reset paths. The optimal reset paths are those that are maximal within the *sound* reset paths (Definition 10).

Definition 11 (Bound Algorithm with Context). *Let $\zeta : T \rightarrow \mathcal{V} \cup \{1\}$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ be as defined in Definition 9. We override the definition of $T\mathcal{B} : T \mapsto Expr(\mathcal{A})$ in Definition 9 by stating:*

$$T\mathcal{B}(\tau) = \mathbf{1} \text{ if } \zeta(\tau) = \mathbf{1} \text{ else}$$
$$T\mathcal{B}(\tau) = \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0)$$
$$+ \sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$$

where
$$T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$$

Discussion and Example: The main difference to the definition of $T\mathcal{B}(\tau)$ in Definition 9 is that the term $\mathtt{Incr}(\zeta(\tau))$ is replaced by the term $\sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$. Consider the abstracted DCP of Example 1 in Figure 1. We have discussed in Section II-A that r may be incremented on τ_1 between the reset of r to 0 on τ_0 resp. τ_4 and the reset of p to r on τ_{2a}. The term $\sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a})$ 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 κ (in the example τ_{2a}) is executed: We use the local bound mapping $\zeta = \{\tau_0 \mapsto 1, \tau_1 \mapsto x, \tau_{2a} \mapsto x, \tau_{2b} \mapsto x, \tau_4 \mapsto x, \tau_5 \mapsto x, \tau_3 \mapsto p\}$ for Example 1. The reset graph of Example 1 is shown in Figure 3. We have $\mathfrak{R}(p) = \{0 \xrightarrow{\tau_0} r \xrightarrow{\tau_{2a}} p, 0 \xrightarrow{\tau_4} r \xrightarrow{\tau_{2a}} p\}$. Thus our algorithm computes $T\mathcal{B}(\tau_3) = \sum_{\kappa \in \mathfrak{R}(p)} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a} \in atm(\kappa)} \mathtt{Incr}(\mathtt{a}) = T\mathcal{B}(\{\tau_0, \tau_{2a}\}) \times \max(V\mathcal{B}(0), 0) + \mathtt{Incr}(r) + T\mathcal{B}(\{\tau_4, \tau_{2a}\}) \times \max(V\mathcal{B}(0), 0) + \mathtt{Incr}(r) = 2 \times \mathtt{Incr}(r) = 2 \times T\mathcal{B}(\tau_1) \times 1 = 2 \times n$ (with $T\mathcal{B}(\tau_1) = n$).
Complexity: In theory there can be exponentially many resets in $\mathfrak{R}(v)$. In our experiments this never occurred, enumeration of (optimal) reset paths did not affect performance.
Further Optimization: We have shown in Section II that transitions τ_3 of Example 1 has a *linear* bound, precisely n. The Bound $2n$ that is computed by our bound algorithm

from Definition 11 is *linear* but not precise. We compute $2n$ because r appears on both reset paths of p and therefore $\mathtt{Incr}(r) = n$ is added twice. However, there is only one transition (τ_{2a}) on which p is reset to r and between any two executions of τ_{2a} r will be reset to 0. For this reason each increment of r can only contribute once to the increase of the local bound p of τ_3, and not twice. We thus suggest to further optimize our algorithm from Definition 11 by distinguishing if there is more than one way how $\mathtt{a} \in atm(\kappa)$ may flow into the target variable of κ or not. We divide $atm(\kappa)$ into two disjoint sets $atm_2(\kappa) = \{\mathtt{a} \in atm(\kappa) \mid$ more than 1 path from $\mathtt{a}$ to target variable of κ in $\mathcal{G}(\Delta\mathcal{P})\}$, $atm_1(\kappa) = atm(\kappa) \setminus atm_2(\kappa)$. We define

$$T\mathcal{B}(\tau) = (\sum_{\substack{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)}} \mathtt{Incr}(\mathtt{a})) +$$
$$\sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0)$$
$$+ \sum_{\mathtt{a} \in atm_2(\kappa)} \mathtt{Incr}(\mathtt{a})$$

for $\zeta(\tau) \neq 1$. Note that for Example 1 $atm_1(\kappa) = \{r\}$ and $atm_2(\kappa) = \emptyset$ for both $\kappa \in \mathfrak{R}(p)$. Therefore $T\mathcal{B}(\tau_3) = \mathcal{I}(r) = n$ with the optimization.

Theorem 2 (Soundness of Bound Algorithm with Context). *Let $\Delta\mathcal{P}(L, T, l_b, l_e)$ be a well-defined and deterministic DCP over atoms $\mathcal{A}$, $\zeta : T \mapsto \mathcal{V} \cup \{1\}$ be a* local bound mapping *for $\Delta\mathcal{P}$, $v \in \mathcal{V}$ and $\tau \in T$. Let $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\mathtt{a})$ be defined as in Definition 11. Either $T\mathcal{B}(\tau) = \bot$ or $T\mathcal{B}(\tau)$ is a transition bound for τ. Either $V\mathcal{B}(v) = \bot$ or $V\mathcal{B}(v)$ is a variable bound for v.*

B. DCPs over non-well-founded domains

In real world code, many data types are not well-founded. The abstraction of a concrete program is much simpler and more information is kept if the abstract program model is not limited to a well-founded domain. Below we extend our program model from Definition 3 to the non-well-founded domain $\mathbb{Z}$ by adding guards to the transitions in the program. Interestingly our bound algorithm from Definition 9 resp. Definition 11 remains sound for the extended program model, if we adjust our notion of a *local transition bound* (Definition 12).
We extend the range of the *valuations* $Val_{\mathcal{A}}$ of $\mathcal{A}$ from $\mathbb{N}$ to $\mathbb{Z}$ and allow constants to be integers, i.e., we define $\mathcal{A} = \mathcal{V} \cup \mathcal{C} \cup \mathbb{Z}$. We extend Definition 3 as follows: The transitions T of a *guarded DCP* $\Delta\mathcal{P}(L, T, l_b, l_e)$ are a subset of $L \times 2^{\mathcal{V}} \times 2^{\mathcal{DC}(\mathcal{A})} \times L$. A sequence $(l_b, \sigma_0) \xrightarrow{g_0, u_0} (l_1, \sigma_1) \xrightarrow{g_1, u_1} \cdots$ is a *run* of $\Delta\mathcal{P}$ if it meets the conditions required in Definition 3 and additionally $\sigma_i(x) > 0$ holds for all $x \in g_i$. For examples see Figure 1.

Definition 12 (Local Transition Bound for DCPs with guards). *Let $\Delta\mathcal{P}(L, T, l_b, l_e)$ be a DCP with guards over $\mathcal{A}$. Let $\tau \in T$ and $v \in \mathcal{V}$. v is a* local bound *for τ if for all finite runs $\rho = (l_b, \sigma_0) \xrightarrow{\tau_0} (l_1, \sigma_1) \xrightarrow{\tau_1} \cdots (l_e, \sigma_n)$ of $\Delta\mathcal{P}$ it holds that $\sharp(\tau, \rho) \leq \downarrow(\max(v, 0), \rho)$.*

The algorithms in Sections III-C and IV are based on the extended program model over $\mathbb{Z}$, it is straightforward to adjust them for DCPs without guards.

C. Determining Local Bounds

We call a path of a DCP $\Delta\mathcal{P}(L, T, l_b, l_e)$ *simple and cyclic* if it has the same start- and end-location and does not visit a location twice except for the start- and end-location. Given a transition $\tau \in T$ we assign it $\mathsf{v} \in \mathcal{V}$ as local bound if for all simple and cyclic paths $\pi = l_1 \xrightarrow{g_1, u_1} l_2 \xrightarrow{g_2, u_2} \dots l_n \ (l_n = l_1)$ of $\Delta\mathcal{P}$ that traverse τ it holds that (1) $\exists 0 < i < n$ s.t. $\mathsf{v} \in g_i$ and (2) $\exists 0 < i < n$ s.t. $\mathsf{v}' \leq \mathsf{v} + \mathsf{c} \in u_i$ for some $\mathsf{c} < 0$. Our implementation avoids an explicit enumeration of the simple and cyclic paths of $\Delta\mathcal{P}$ by a simple data flow analysis.

IV. PROGRAM ABSTRACTION

In this section we present our concrete program model and discuss how we abstract a given program to a DCP.

Definition 13 (Program). *Let Σ be a set of* states. *The set of transition relations $\Gamma = 2^{\Sigma \times \Sigma}$ is the set of relations over Σ. A program is a directed labeled graph $\mathcal{P} = (L, E, l_b, l_e)$, 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 $E \subseteq L \times \Gamma \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{\rho} l_2$ to denote a transition (l_1, ρ, l_2). A norm $e \in \Sigma \to \mathbb{Z}$ is a function that maps the states to the integers.*

Programs are labeled transition systems over some set of states, where each transition is labeled by a transition relation that describes how the state changes along the transition. Note, that a DCP (Definition 3) is a program by Definition 13.

Definition 14 (Transition Invariants). *Let $e_1, e_2, e_3 \in \Sigma \to \mathbb{Z}$ be norms, and let $c \in \mathbb{Z}$ be some integer. We say $e_1' \leq e_2 + e_3$ is invariant for $l_1 \xrightarrow{\rho} l_2$, if $e_1(s_2) \leq e_2(s_1) + e_3(s_1)$ holds for all $(s_1, s_2) \in \rho$. We say $e_1 > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$, if $e_1(s_1) > 0$ holds for all $(s_1, s_2) \in \rho$.*

Definition 15 (Abstraction of a Program). *Let $\mathcal{P} = (L, E, l_b, l_e)$ be a program and let N be a finite set of norms. A DCP $\Delta\mathcal{P} = (L, E', l_b, l_e)$ with atoms N is an abstraction of the program $\mathcal{P}$ iff for each transition $l_1 \xrightarrow{\rho} l_2 \in E$ there is a transition $l_1 \xrightarrow{u, g} l_2 \in E'$ s.t. every $e_1' \leq e_2 + c \in u$ is invariant for $l_1 \xrightarrow{\rho} l_2$ and for every $e_1 \in g$ it holds that $e_1 > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$.*

We propose to abstract a program $\mathcal{P} = (L, E, l_b, l_e)$ to a DCP $\Delta\mathcal{P} = (L, E', l_b, l_e)$ as follows: Let N be some initial set of norms.
1) For each transition $l_1 \xrightarrow{\rho} l_2 \in E$ we generate a set of difference constraints $\alpha(\rho)$: Initially we set $\alpha(\rho) = \emptyset$ for all transitions $l_1 \xrightarrow{\rho} l_2$. We then repeat the following construction until the set of norms N becomes stable: For each $e_1 \in N$ and $l_1 \xrightarrow{\rho} l_2 \in E$ we check whether there is a difference constraint of form $e_1' \leq e_2 + c$ for e_1 in $\alpha(\rho)$. If not, we try to find a norm e_2 (possibly not yet in N) and a constant $c \in \mathbb{Z}$ s.t. $e_1' \leq e_2 + c$ is invariant for ρ. If we find appropriate e_2 and c, we add $e_1' \leq e_2 + c$ to $\alpha(\rho)$ and e_2 to N. I.e., our transition abstraction algorithm performs a fixed point computation which might not terminate if new terms keep being added (see discussion in next section).
2) For each transition $l_1 \xrightarrow{\rho} l_2$ we generate a set of guards

$G(\rho)$: Initially we set $G(\rho) = \emptyset$ for all transitions $l_1 \xrightarrow{\rho} l_2$. For each $e \in N$ and each transition $l_1 \xrightarrow{\rho} l_2$ we check if $e > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$. If so, we add e to $G(\rho)$.
3) We set $E' = \{l_1 \xrightarrow{G(\rho), \alpha(\rho)} l_2 \mid l_1 \xrightarrow{\rho} l_2 \in E\}$.
In the following we discuss how we implement the above sketched abstraction algorithm.

A. Implementation

0. Guessing the initial set of Norms.: We aim at creating a suitable abstract program for bound analysis. In our non-recursive setting, complexity evolves from iterating loops. Therefore we search for expressions which limit the number of loop iterations. For this purpose 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 such condition we form the integer expression $b - a$ and add it to our initial set of norms. Note that on those transitions on which $a > b$ holds, $b - a > 0$ must hold.

1. Abstracting Transitions.: For a given norm $e \in N$ and a transition $l_1 \xrightarrow{\rho} l_2$ we derive a transition predicate $e' \leq e_2 + c \in \alpha(\rho)$ as follows: We symbolically execute ρ for deriving e' from e. In order to keep the number of norms low, we first try
i) to find a norm $e_2 \in N$ s.t. $e' \leq e_2 + e_3$ is invariant for ρ where e_3 is some integer valued expression. If $e_3 = c$ for some integer $c \in \mathbb{Z}$ we derive the transition predicate $e' \leq e_2 + c$. Else we use our bound algorithm (Section III) for over-approximating e_3 by a constant expression $k \geq e_3$ and infer the transition predicate $e' \leq e_2 + k$ where we consider k to be a symbolic constant.
ii) If i) fails, we form a norm e_4 s.t. $e' \leq e_4 + c$ by separating constant parts in the expression e' using associativity and commutativity of the addition operator. E.g., given $e' = \mathsf{v} + 5$ we set $e_4 = \mathsf{v}$ and $c = 5$. We add e_4 to N and derive the predicate $e' \leq e_4 + c$.
Since case ii) triggers a recursive abstraction for the newly added norm we have to ensure the termination of our abstraction procedure: Note that we can always stop the abstraction process at any point, getting a sound abstraction of the original program. We therefore enforce termination of the abstraction algorithm by limiting the chain of recursive abstraction steps triggered by entering case ii) above: In case this limit is exceeded we remove all norms from the abstract program which form part of the limit exceeding chain of recursive abstraction steps. This also ensures well-definedness of the resulting abstract program.
Further note that the DCPs generated by our algorithm are always *deterministic*: For each transition, we get at most one predicate $e' \leq e_2 + c$ for each $e \in N$.
2. Inferring Guards: Given a transition $l_1 \xrightarrow{\rho} l_2$ and a norm e, we use an SMT solver to check whether $e > 0$ is invariant for $l_1 \xrightarrow{\rho} l_2$. If so, we add e to $G(\rho)$.
Non-linear Iterations.: We handle counter updates such as $x' = 2x$ or $x' = x/2$ as discussed in [16].

	Succ.	1	n	n^2	n^3	$n^{>3}$	2^n	Time	TO
Loopus'15	806	205	489	97	13	2	0	15m	6
Loopus'14	431	200	188	43	0	0	0	40m	20
KoAT	430	253	138	35	2	0	2	5,6h	161
CoFloCo	386	200	148	38	0	0	0	4.7h	217

Fig. 4. Tool Results on analyzing the complexity of 1659 functions in the cBench benchmark, none of the tools infers *log* bounds.

V. EXPERIMENTS

Implementation: We have implemented the presented algorithm into our tool Loopus [1]. Loopus reads in the LLVM [15] 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.

Experimental Setup: For our experimental comparison we used the program and compiler optimization benchmark *Collective Benchmark* [2] (cBench), which contains a total of 1027 different C files (after removing code duplicates) with 211.892 lines of code. In contrast to our earlier work we did not perform a loop bound analysis but a complexity analysis on function level. We set up the first comparison of complexity analysis tools on real world code. For comparing our new tool (Loopus'15) we chose the 3 most promising tools from recent publications: the tool KoAT implementing the approach of [7], the tool CoFloCo implementing [10] and our own earlier implementation (Loopus'14) [16]. Note that we compared against the most recent versions of KoAT and CoFloCo (download 01/23/15).[1] The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor and 16 GB memory. 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 [3] to translate the 1751 llvm modules into 1751 text files in the Integer Transition System (ITS) format read in by KoAT.

4) We used the transformation described in [10] to translate the ITS format of KoAT into the ITS format of CoFloCo. This last step is necessary because there exists no direct way of translating C or the llvm intermediate representation into the CoFloCo input format.

5) We decided to exclude the 91 recursive functions in the 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 4 shows the results of the 4 tools on our benchmark using a time out of 60 seconds. 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

---

[1]https://github.com/s-falke/kittel-koat, https://github.com/aeflores/CoFloCo

---

not able compute a bound. The column *Time* shows the total time used by the tool to process the benchmark. Loopus'15 computes the complexity 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 approach is both scalable and more successful than existing approaches.

Pointer and Shape Analysis: 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 minutes total time).

Amortized Complexity: During our experiments, we found 15 examples with an amortized complexity that could only be inferred by the approach presented in this paper. These examples and further experimental results can be found on [1] where our new tool is offered for download.

REFERENCES

[1] http://forsyte.at/software/loopus/.

[2] http://ctuning.org/wiki/index.php/CTools:CBench.

[3] https://github.com/s-falke/llvm2kittel.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[5] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133, 2010.

[6] A. M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[7] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, page to appear, 2014.

[8] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. PLDI, 2015.

[9] T. Colcombet, L. Daviaud, and F. Zuleger. Size-change abstraction and max-plus automata. In *MFCS*, pages 208–219, 2014.

[10] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS*, pages 275–295, 2014.

[11] T. M. Gawlitza, M. D. Schwarz, and H. Seidl. Parametric strategy iteration. *arXiv preprint arXiv:1406.5457*, 2014.

[12] S. Gulwani and S. Juvekar. Bound analysis using backward symbolic execution. Technical Report MSR-TR-2004-95, Microsoft Research, 2009.

[13] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

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

[15] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[16] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, pages 745–761. Springer, 2014.

[17] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. *CoRR*, abs/1401.5842, 2014.

[18] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, Apr. 1985.

[19] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Languages and Compilers for Parallel Computing*, pages 427–441. Springer, 2003.

[20] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.
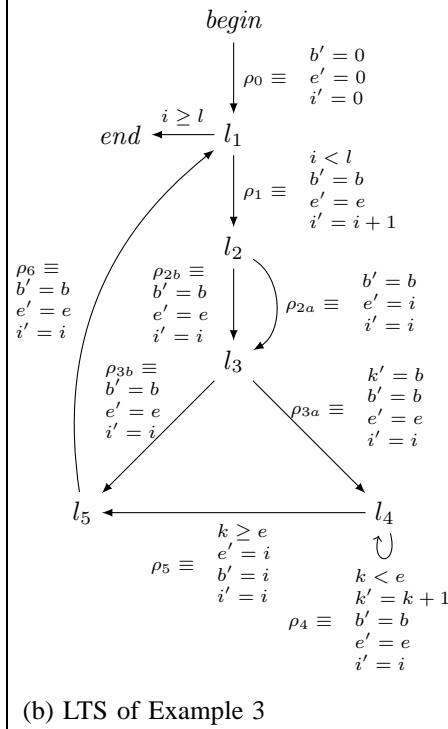
Fig. 5

(a) Example 3:

```
xnu(int len) {
    int beg,end,i = 0;
l₁  while(i < len) {
        i++;
l₂      if (*)
            end = i;
l₃      if (*) {
            int k = beg;
l₄          while (k < end)
                k++;
            end = i;
            beg = end;
        }
l₅  }
}
```
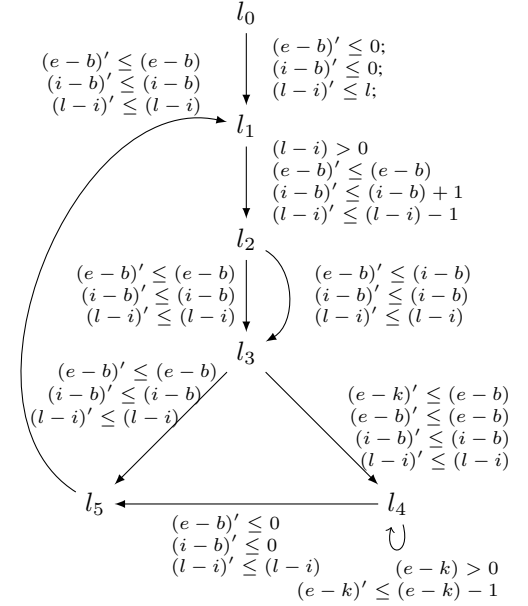
(b) LTS of Example 3

$$\rho_0 \equiv \begin{array}{l} b' = 0 \\ e' = 0 \\ i' = 0 \end{array}$$

$i \ge l$: $end \leftarrow l_1$

$$\rho_1 \equiv \begin{array}{l} i < l \\ b' = b \\ e' = e \\ i' = i + 1 \end{array}$$

$$\rho_{2b} \equiv \begin{array}{l} b' = b \\ e' = e \\ i' = i \end{array} \qquad \rho_{2a} \equiv \begin{array}{l} b' = b \\ e' = i \\ i' = i \end{array}$$

$$\rho_6 \equiv \begin{array}{l} b' = b \\ e' = e \\ i' = i \end{array}$$

$$\rho_{3b} \equiv \begin{array}{l} b' = b \\ e' = e \\ i' = i \end{array} \qquad \rho_{3a} \equiv \begin{array}{l} k' = b \\ b' = b \\ e' = e \\ i' = i \end{array}$$

$$\rho_5 \equiv \begin{array}{l} k \ge e \\ e' = i \\ b' = i \\ i' = i \end{array} \qquad \rho_4 \equiv \begin{array}{l} k < e \\ k' = k + 1 \\ b' = b \\ e' = e \\ i' = i \end{array}$$

(c) Abstracted DCP for Example 3

$$l_0 \xrightarrow{\begin{array}{l}(e-b)' \le 0; \\ (i-b)' \le 0; \\ (l-i)' \le l;\end{array}} l_1$$

$$\begin{array}{l}(e-b)' \le (e-b) \\ (i-b)' \le (i-b) \\ (l-i)' \le (l-i)\end{array}$$

$l_1 \to l_2$: $\begin{array}{l}(l-i) > 0 \\ (e-b)' \le (e-b) \\ (i-b)' \le (i-b)+1 \\ (l-i)' \le (l-i)-1\end{array}$

l_2: left $\begin{array}{l}(e-b)' \le (e-b) \\ (i-b)' \le (i-b) \\ (l-i)' \le (l-i)\end{array}$ right $\begin{array}{l}(e-b)' \le (i-b) \\ (i-b)' \le (i-b) \\ (l-i)' \le (l-i)\end{array}$

l_3: left $\begin{array}{l}(e-b)' \le (e-b) \\ (i-b)' \le (i-b) \\ (l-i)' \le (l-i)\end{array}$ right $\begin{array}{l}(e-k)' \le (e-b) \\ (e-b)' \le (e-b) \\ (i-b)' \le (i-b) \\ (l-i)' \le (l-i)\end{array}$

$l_5 \leftarrow l_4$: $\begin{array}{l}(e-b)' \le 0 \\ (i-b)' \le 0 \\ (l-i)' \le (l-i)\end{array}$ l_4 loop: $\begin{array}{l}(e-k) > 0 \\ (e-k)' \le (e-k)-1\end{array}$

(a) Example 3   (b) LTS of Example 3   (c) Abstracted DCP for Example 3

Fig. 5. Example 3 shows the code after which we have modeled Example 1, * denotes non-determinism (arising from conditions not modeled in the analysis)

APPENDIX

A. Full Example

Example 3 in Figure 5 contains a snippet of the source code after which we have modeled Example 1 in Figure 1. Example 3 can be found in the SPEC CPU2006 benchmark[2], in function XNU of 456.hmmer/src/masks.c. The outer loop in Example 3 partitions the interval $[0, len]$ into disjoint sub-intervals $[beg, end]$. The inner loop iterates over the sub-intervals. Therefore the inner loop has an overall linear iteration count. Example 3 is a natural example for amortized complexity: Though a single visit to the inner loop can cost len (if $beg = 0$ and $end = len$), several visits can also not cost more than len since in each visit the loop iterates over a disjoint sub-interval. I.e., the total cost len of the inner loop is the *amortized cost* over all visits to the inner loop. To the best of our knowledge our new implementation Loopus'15 (available at [1]) is the only tool that infers the linear complexity of Example 3 without user interaction.

1) Abstraction: In Figure 5 (b) the labeled transition system for Example 3 is shown. We discuss how our abstraction algorithm from Section IV abstracts the example to the DCP shown in Figure 5 (c).

Our heuristics add 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 $N = \{l - i, e - k\}$.

- We check how $l - i$ changes on the transitions $\rho_0, \rho_1, \rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_5, \rho_6$:
  - ρ_0: we derive $(l-i)' \le l$ (reset), we add l to N

- ρ_1: we derive $(l-i)' \le (l-i)-1$ (negative increment)
  - $\rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_5, \rho_6$: $l - i$ unchanged
- We check how l changes on the transitions $\rho_0, \rho_1, \rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_5, \rho_6$:
  - unchanged on all transitions
- We check how $e - k$ changes on the transitions ρ_{3a}, ρ_4 (k is only defined at l_4):
  - ρ_{3a}: we derive $(e-k)' \le (e-b)$ (reset), we add $(e-b)$ to N
  - ρ_4: we derive $(e-k)' \le (e-k)-1$ (negative increment)
- We check how $e - b$ changes on the transitions $\rho_0, \rho_1, \rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_5, \rho_6$::
  - ρ_0: we derive $(e-b)' \le 0$ (reset)
  - ρ_{2a}: we derive $(e-b)' \le (i-b)$, we add $(i-b)$ to N
  - ρ_5: we derive $(e-b)' \le 0$ (reset)
  - $\rho_1, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_6$:: $e - b$ unchanged
- We check how $i - b$ changes on the transitions $\rho_0, \rho_1, \rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_5, \rho_6$:
  - ρ_0: we derive $(i-b)' \le 0$ (reset)
  - ρ_1: we derive $(i-b)' \le (i-b)+1$ (increment)
  - ρ_5: we derive $(i-b)' \le 0$ (reset)
  - $\rho_{2a}, \rho_{2b}, \rho_{3a}, \rho_{3b}, \rho_4, \rho_6$:: unchanged
- We have processed all norms in N

We infer that $\rho_1 \models (l-i) > 0$ and $\rho_4 \models (e-k) > 0$. The resulting DCP is shown in Figure 5(c).

2) Bound Computation: We discuss how our bound algorithm from Section III infers the *linear* bound for the inner loop at
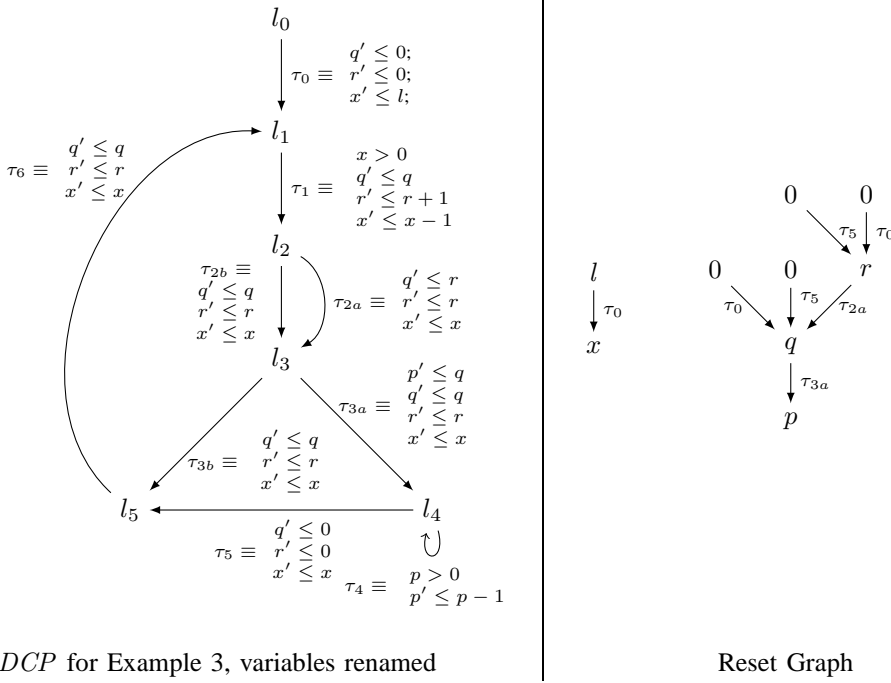
9

l_0

$\tau_0 \equiv \begin{array}{l} q' \le 0; \\ r' \le 0; \\ x' \le l; \end{array}$

l_1

$\tau_6 \equiv \begin{array}{l} q' \le q \\ r' \le r \\ x' \le x \end{array}$

$\tau_1 \equiv \begin{array}{l} x > 0 \\ q' \le q \\ r' \le r+1 \\ x' \le x-1 \end{array}$

l_2

$\tau_{2b} \equiv \begin{array}{l} q' \le q \\ r' \le r \\ x' \le x \end{array}$

$\tau_{2a} \equiv \begin{array}{l} q' \le r \\ r' \le r \\ x' \le x \end{array}$

l_3

$\tau_{3a} \equiv \begin{array}{l} p' \le q \\ q' \le q \\ r' \le r \\ x' \le x \end{array}$

$\tau_{3b} \equiv \begin{array}{l} q' \le q \\ r' \le r \\ x' \le x \end{array}$

$l_5 \longleftarrow l_4$

$\tau_5 \equiv \begin{array}{l} q' \le 0 \\ r' \le 0 \\ x' \le x \end{array}$

$\tau_4 \equiv \begin{array}{l} p > 0 \\ p' \le p-1 \end{array}$

DCP for Example 3, variables renamed

Reset Graph

Fig. 6.

l_4. For ease of readability, we state the abstracted DCP of Example 3 in Figure 6 renaming the variables by the following scheme: $\{\mathbf{p} = (e-k), \mathbf{q} = (e-b), \mathbf{r} = (i-b), \mathbf{x} = (l-i)\}$. On the right hand side the reset graph is shown. Our Algorithm from Definition 11 now computes a bound for the example by the following reasoning:

1) Our algorithm for determining the local bound mapping (Section III-C) assigns the following local bounds to the respective transitions $\zeta(\tau_0) = 1$, $\zeta(\tau_1) = \zeta(\tau_{2a}) = \zeta(\tau_{2b}) = \zeta(\tau_{3a}) = \zeta(\tau_{3b}) = \zeta(\tau_5) = \zeta(\tau_6) = x$, $\zeta(\tau_4) = p$.

2) $\mathfrak{R}(p) = \{0 \xrightarrow{\tau_0,0} r \xrightarrow{\tau_{2a},0} q \xrightarrow{\tau_{3a},0} p, 0 \xrightarrow{\tau_5,0} r \xrightarrow{\tau_{2a},0} q \xrightarrow{\tau_{3a},0} p, 0 \xrightarrow{\tau_0,0} q \xrightarrow{\tau_{3a},0} p, 0 \xrightarrow{\tau_5,0} q \xrightarrow{\tau_{3a},0} p\}$

3) We get: $TB(\tau_1)$ resp. $TB(\tau_{2a})$ resp. $TB(\tau_{2b})$ resp. $TB(\tau_{3a})$ resp. $TB(\tau_{3b})$ resp. $TB(\tau_5)$ resp. $TB(\tau_6) = TB(\tau_0) \times l = l$ (Definition 11) with $TB(\tau_0) = 1$

4) For τ_4 we get: $TB(\tau_4) = TB(\tau_0, \tau_{2a}, \tau_{3a}) \times 0 + TB(\tau_1) \times 1 + TB(\tau_5, \tau_{2a}, \tau_{3a}) \times 0 + TB(\tau_1) \times 1 + TB(\tau_0, \tau_{3a}) \times 0 + TB(\tau_5, \tau_{3a}) \times 0 = n \times 1 + n \times 1 = 2n$ (Definition 11) with $TB(\tau_1) = n$

5) We get the precise bound n for τ_4 when applying the optimization presented in the discussion under Definition 11: For all $\kappa \in \mathfrak{R}(p)$ we have $atm_1(\kappa) = \{r, q\}$ and $atm_2(\kappa) = \emptyset$. Therefore $TB(\tau_4) = TB(\tau_1) \times 1 + TB(\tau_0, \tau_{2a}, \tau_{3a}) \times 0 + TB(\tau_5, \tau_{2a}, \tau_{3a}) \times 0 + TB(\tau_0, \tau_{3a}) \times 0 + TB(\tau_5, \tau_{3a}) \times 0 = n \times 1 = n$ with $TB(\tau_1) = n$.

10