MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Symbolic Loop Bound Analysis

DIPLOMA THESIS

Pavel Čadek

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Pavel Čadek

Advisor: doc. RNDr. Jan Strejček, Ph.D.

Acknowledgement

I would like to thank my supervisor, doc. RNDr. Jan Strejček, Ph.D. for discussions about the topic and advice he has provided during writing of this thesis. I would also like to thank Mgr. Marek Trtík, PhD., who helped me with the implementation and regularly discussed the topic with me. I must express my gratitude to closer friends, who stood next to me in difficult moments. Last, but not the least, I would like to thank my family for their unconditional support and patience throughout my whole studies.

Abstract

We present a new method for computation of upper bounds on the number of visits of given program locations. These bounds are expressed as functions over input variable symbols. A description of the algorithm is the core of this thesis. We have implemented our method in a prototype tool `Looperman` and evaluated it on benchmarks from the literature. Besides the evaluation results, we provide also a detailed description of two other tools, `Loopus` and `KoAT`, which we used for the comparison with our tool.

Keywords

static analysis, loop bound analysis, symbolic bound computation, worst case execution time, time complexity, size complexity, program termination, reachability bound problem, symbolic execution, abstracting path conditions

Contents

1 Introduction

Static program analysis plays an important role in code optimization and verification. Demands for tools that are able to automatically prove certain program properties grow over time. One of the most important program property is its time complexity and termination. Indeed, for a lot of real systems, like embedded systems in cars or aircraft, there is a necessity to prove that they terminate on any input within some time limit. However, time complexity analysis is useful for any systems working with large data. Consider, for example, sorting algorithms: some of them are practically useless for very large arrays, because they could spend hours sorting them, while other finish the computation within seconds. Most of the current static analysis tools in this area compute only termination or an asymptotic complexity of a program. But one can see that there is a significant difference between time complexities n^2 and $1000 \cdot n^2$, while the asymptotic complexity is the same. We present an algorithm for computing more precise time bounds. The need for the precision can be seen in the embedded systems: hardware which has to perform n operations within certain time limit is more expensive, than hardware which has to perform just $\frac{n}{2}$ operations. Note that there is no simple way of deriving the real time estimation out of the theoretical complexity given in abstract time units, because some instruction in the source code can be more time-demanding than another one. Therefore only the upper bounds on the number of loop iterations ("loop bounds", for short) are usually computed. With them, the real time estimation can be derived. We propose a more general goal: We want to compute an upper bound on the number of visits of any given program location as a function over its input variables (we call such a bound a "symbolic bound"). This formulation allows us to infer the time complexity for every costly instruction separately and it can result into different bounds for locations on different branches of the same loop. Note that an instruction can be costly also concerning computational space (e.g. a memory allocation). Symbolic bounds can be useful, for example, for various kinds of schedulers. With them, the schedulers can compute the resources (e.g. space or processor time), which a function demands, in advance, with just the function's input values. Other usage can be in program verification, or code optimization.

We have implemented a prototype tool, that computes symbolic bounds on the number of visits of given program locations. We present our algorithm in Chapter 2. Its computational steps are explained there on the well-

known sorting algorithm Bubble Sort used as input. In the next chapter, we present two recently proposed alternative approaches for computing the bounds as well as a brief overview of tools in the area of termination and complexity analysis. Finally, in Chapter 4, we present the results of evaluating our experimental tool on a set of scientific benchmarks and its comparison with other four tools for computing loop bounds.

2 Symbolic Bound Computation Algorithm

2.1 Preliminaries

This section defines terms and notation employed by the algorithm. Particularly the terms *backbone*, *induced flowgraph*, and *upper bound for a transition* should be noticed, because they are not commonly used. Most of the definitions come from [18].

2.1.1 Control Flow of the Program

We represent programs by labelled transition systems. We consider only instructions operating on *scalar variables* $\mathtt{a}, \mathtt{b}, \ldots$ of type $\mathtt{Int}$ and multi-dimensional *array variables* $\mathtt{A}, \mathtt{B}, \ldots$ of type $\mathtt{Int}^k \to \mathtt{Int}$. We use two kinds of instruction: an assignment and an assumption. The assignment instruction is either of the form $\mathtt{a} := e$ for some (integer) expression e and some scalar variable $\mathtt{a}$, or $\mathtt{A}[e_1, \ldots, e_k] := e$ for some (integer) expressions $e, e_1 \ldots e_k$ and some array variable $\mathtt{A} : \mathtt{Int}^k \to \mathtt{Int}$. The assumption instructions are of the form $\mathtt{assume}(\gamma)$ for some quantifier-free formula γ over program variables. We often omit $\mathtt{assume}$ from the assumption instructions.

Definition 2.1. *Let I be a set of instructions. A* flowgraph *is a tuple $P = (L, T, l_s, l_e)$, where L is a finite set of program* locations, *$T \subseteq L \times I \times L$ is a finite set of program* transitions, *and $l_s, l_e \in L$ are different* start *and* exit *locations respectively. A location is* branching *if its out-degree is 2. All other locations have out-degree at most 1. Outgoing transitions of any branching location are labelled with assumptions $\mathtt{assume}(\gamma)$ and $\mathtt{assume}(\neg\gamma)$ for some γ. We assume that every location is reachable from l_s and l_s has no predecessor. Analogically, l_e is reachable from all locations and l_e has no successor.*

Definition 2.2. *A* path *in a flowgraph is a finite sequence $l_1 l_2 \cdots l_k$ of locations such that $k > 0$ and for all $1 \leq i < k$ there is a transition from l_i to l_{i+1}. A path from the start to the exit location is called a* complete *path. A* backbone *is a complete acyclic path.*

Definition 2.3. *Let π be a backbone with a prefix αv. There is a* loop *C with an* entry location *v along π, if there exists a path $v\beta v$ such that no location of β appears in α. The loop C is then the smallest set containing all locations of all such paths $v\beta v$.*

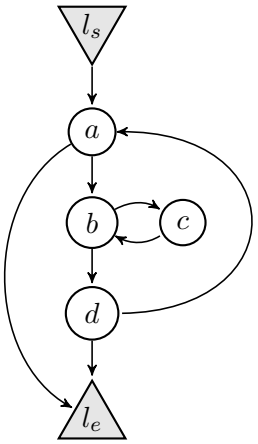

Figure 2.1: Example of nested loops. Instruction labels are omitted here.

For example, the flowgraph in Figure 2.1 has two backbones: $\pi_1 = l_s a l_e$ and $\pi_2 = l_s a b d l_e$. Both of them contain the loop entry a to the loop $\{a, b, c, d\}$, but only π_2 contains the loop entry b to the loop $\{b, c\}$. This is a simplified example of a typical program with nested loops and a possible return statement inside the outer one.

Remark 2.4. *We can assign exactly one backbone π to each complete path π' by the following procedure: If π' is acyclic, then the backbone is directly π'. Otherwise, we find the leftmost repeating node in π', remove the part of π' between the first and the last occurrence of this node (including the last occurrence), and repeat the procedure. In other words, the backbone π arises from π' by removing all loop iterations. We say that π is the* backbone of the path π'.

Definition 2.5. *For a loop C with an entry location v, a* flowgraph induced by the loop, *denoted as $P(C, v)$, is derived from the subgraph of the original flowgraph induced by C, where v is marked as the start location, a fresh location v' is added and marked as the exit location, and every transition $(u, \iota, v) \in T$ leading to v is replaced by a transition (u, ι, v').*

A loop path *is a backbone of a flowgraph induced by a loop.*

Figure 2.2 shows the flowgraph induced by the loop $\{a, b, c, d\}$ from our previous example. It has one loop path (backbone) $abda'$. There is the loop $\{b, c\}$ with the loop entry b along this loop path.

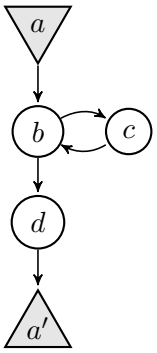

Figure 2.2: The flowgraph induced by the loop $\{a, b, c, d\}$ from Figure 2.1. Instruction labels are omitted here.

2.1.2 Symbolic Execution

The core idea of symbolic execution (as described in [13]) is that instead of supplying the normal inputs to a program (e.g. numbers), one supplies symbols representing arbitrary values. The execution proceeds as a normal execution except that values may be symbolic expressions over the input symbols. When a conditional branching appears, the execution continues on each branch separately and keeps the condition, which the inputs must satisfy in order for an execution to follow the associated path. More precisely, the condition is altered every time an `assume` instruction is executed. If it is not satisfiable after that, there is no program run which could follow the particular path and we can stop the execution.

Definition 2.6. *By* symbolic expressions *we mean all expressions built with integers, standard integer operations and functions, and*

- symbols $\underline{a}, \underline{a}', \underline{a}'', \ldots$ *for each scalar variable* `a`,
- function symbols $\underline{A}, \underline{A}', \ldots$ *for each array variable* `A`, *where arity of* $\underline{A}$ *corresponds to the dimension of array* `A`,
- *a* lambda expression $\lambda(x_1, \ldots, x_k).e$ *of arity* $k \geq 1$, *where* e *is a symbolic expression over* $x_1, \ldots, x_k$,
- *a countable set* $\{\kappa_1, \kappa_2, \ldots\} \cup \{K\}$ *of variables called* path counters, *and*
- *a special symbol* $\star$ *called* unknown.

Remark 2.7. *Among the operations and functions allowed in the symbolic expression are also* $\mathbf{max}, \mathbf{min}$ *and* $\mathbf{ite}$*, where* $\mathbf{ite}(\psi, e_1, e_2)$ *equals* e_1 *if* ψ *holds and* e_2 *otherwise.*

Remark 2.8. *We use the notation* $e[x/e_x]$ *for denoting the expression* e*, where all occurrences of a variable symbol or a path counter* x *are replaced by* e_x*. Similarly, we use the notation* $e[x_i/e_i \mid i \in I]$ *for denoting the expression* e*, where all occurrences of* x_i *are replaced by* e_i*, for each* $i \in I$*.*

Definition 2.9. *A* symbolic memory *is a function* θ *assigning to each scalar variable* `a` *a symbolic expression and to each array variable* `A` *of arity* $k \geq 1$ *a symbol* $\underline{A}$ *or a lambda expression of arity* k*.*

A path condition *is a boolean formula over the symbols used in symbolic expressions, which the inputs must satisfy in order for an execution to follow the corresponding path.*

A symbolic state *is a couple* (θ, φ)*, where* θ *is a symbolic memory and* φ *is a* path condition*.*

Remark 2.10. *We can extend the symbolic memory function from variables to whole expressions in a natural way. For example, when* $\theta(\mathtt{a}) = \underline{a}$ *and* $\theta(\mathtt{b}) = \underline{c}+1$*, then* $\theta(\mathtt{a} + \mathtt{b} - 3) = \underline{a} + \underline{c} - 2$*.*

2.1.3 Bound Computation

Finally, we need to define the terminology for our main purpose: We want to get an upper bound on the number of executions of some transition in a flowgraph. There is a term "ranking function" often repeated in various ways in the literature (e.g. [17, 5, 2]), which is used for that purpose. We use the term "upper bound for a transition" instead, because we find it more suitable for our approach.

Definition 2.11. An upper bound for a transition t *(resp. an* upper bound for a loop C with a loop entry v*) in a flowgraph* P *is a symbolic expression* ρ *with the following properties:*

- *The only symbols in* ρ *(except of integers, lambda expressions, and standard integer operators and functions) are symbols for variables appearing in* P*.*
- *If* P *is executed on any input, then* t *is executed at most* ρ' *times, where* ρ' *is the expression that we get by replacing each scalar variable symbol* $\underline{a}$ *by the initial value of the variable* `a` *and each array variable symbol* $\underline{A}$ *by the initial function of the variable* `A`*.*

Remark 2.12. *A lower bound could be defined analogically. Because we usually work with upper bounds, by terms "bound for a transition" we always mean the upper bound.*

Remark 2.13. *Note that the loop bound computation is a subproblem of our aim, so our method can be applied for it too. In fact, our experimental tool* `Looperman` *works in two modes: computation of bounds for given transitions or computation of bounds for all loops in a program.*

2.2 Basic Idea

The algorithm uses the idea of loop summaries described in the article *Abstracting Path Conditions* [18], which uses symbolic execution to find a necessary condition for reaching a given program location. However our goal is different: we want to find upper bounds for the number of executions of given transitions.

When a program is executed on some input, the execution follows some complete path. From Remark 2.4, we know that each complete path has exactly one backbone, so all runs can be divided into finitely many classes according to the backbones of the paths they follow. Therefore we analyse each backbone separately and put the results together in the end. Assume we analyse a backbone π. We perform the symbolic execution along π until we come to a loop entry. Before we continue with the next transition along π, the loop is processed and the values of variables are changed according to the inferred effect of the loop. The bounds are computed together with the symbolic execution. We can look on a bound for a transition as a counter of its executions. Hence we start with the bound 0 for every transition and increment it during the analysis.

Suppose some transition lies within a simple loop with just one loop path and no loop nested within it. Then the amount, by which its bound must be incremented after execution of the loop, is exactly the number of the loop iterations. To find it, we first treat the loop like a standalone program and compute the effect of executing it once. Then we associate a path counter κ to the loop path. We can look on a path counter as an imaginary variable, which is 0 before entering the loop and it is increased by 1 at the end of each loop iteration following the particular loop path. In this case, κ corresponds to the number of finished iterations. Knowing the effect of executing the loop once, we try to compute the variable values after κ iterations. By substituting these values into the looping condition, we can infer

```
int i:=0;
do
  i:=i+1;
while (i<x);
```

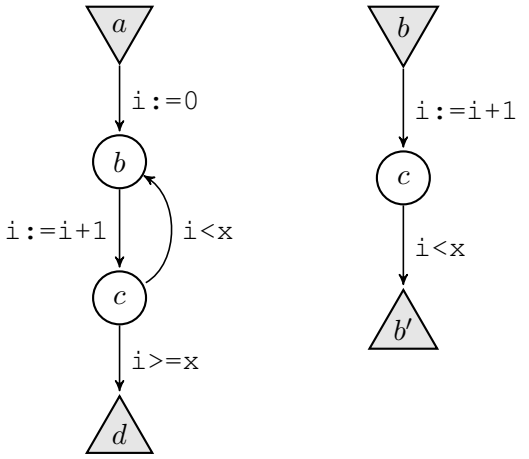

Figure 2.3: Example of a simple C-like code (on the left) with its flowgraph (in the middle) and the flowgraph induced by the loop $\{b, c\}$ with the loop entry b (on the right).

a bound on the size of κ, which is also a bound on the number of the loop iterations.

Let us explain the concept on a simple example from Figure 2.3. We see that the program has only one backbone $\pi = abcd$. The symbolic execution starts with the symbolic values $\underline{x}$ for the variable `x` and $\underline{i}$ for `i`. All the bounds are 0 at the beginning. After the first transition $(a, \texttt{i:=0}, b)$ its bound is incremented to 1 and the value of the variable `i` is updated to 0. Now we are at location b that is a loop entry to the loop $\{b, c\}$. So we pause the symbolic execution and process the loop.

The flowgraph induced by the loop $\{b, c\}$ with the entry b is depicted in Figure 2.3 on the right. We first compute the effect of executing it once. It has one backbone bcb' with no loop entry. We start a new symbolic execution at the new start location b. To avoid confusion with the values from the main flowgraph, we start with the value $\underline{x}'$ for `x` and $\underline{i}'$ for `i`. After executing both transitions along the backbone, we end with values $\underline{i}' + 1$ for `i` and $\underline{x}'$ for `x` and the path condition $\underline{i}' + 1 < \underline{x}'$. From this we can conclude that after κ iterations, the values are $\underline{i}' + \kappa$ for `i` and $\underline{x}'$ for `x` and thus the condition to perform an iteration of the loop after κ iterations is $\underline{i}' + \kappa + 1 < \underline{x}'$. Now we return back to the context of our main flowgraph. We entered the loop with the value 0 for `i` and $\underline{x}$ for `x`. Hence we can substitute these values for $\underline{i}'$ and $\underline{x}'$ in the condition $\underline{i}' + \kappa + 1 < \underline{x}'$, getting $\kappa + 1 < \underline{x}$. The maximal

κ satisfying it is $\underline{x} - 2$, so before the last iteration, κ is at most $\underline{x} - 2$. Thus, after including the fact that the number of iterations cannot be negative, we get the bound $\mathbf{max}(0, \underline{x} - 1)$. We increment by that amount the (currently zero) bounds for transitions $(b, \texttt{i:=i+1}, c)$ and $(c, \texttt{i<x}, b)$. The last thing we need to do before we leave the loop, is updating the values of variables. We can see that the variable `x` stays unchanged, no matter how many iterations were taken. However, the size of `i` depends on the number of iterations. For the simplicity of explanation, we assign the value $\star$ (standing for unknown) to `i`, which is always a safe approximation.[1]

After the loop procedure, we continue with the next transition along the main backbone, which is $(b, \texttt{i:=i+1}, c)$. We raise its current bound $\mathbf{max}(0, \underline{x} - 1)$ by 1, getting $\mathbf{max}(1, \underline{x})$. The new value for `i` is $\star + 1$ which equals $\star$. The last transition $(c, \texttt{i>=x}, e)$ is of the `assume` type, so we should alter the path condition. By substituting the values of `x` and `i` we get $\star \geq \underline{x}$. Such a condition is evaluated to *true* (because of the $\star$ symbol), so the path condition stays the same. The bound for the transition is incremented from 0 to 1.

Because we have reached the exit location, the analysis is finished with the following bounds: 1 for transitions $(a, \texttt{i:=0}, b)$ and $(c, \texttt{i>=x}, e)$, $\mathbf{max}(1, \underline{x})$ for $(b, \texttt{i:=i+1}, c)$ and $\mathbf{max}(0, \underline{x} - 1)$ for $(c, \texttt{i<x}, b)$.

We provide a detailed description of the algorithm in the next section, including the treatment of more backbones in the main flowgraph, multi-path loops, or nested loops.

2.3 The Algorithm

This section introduces the algorithm as a pseudo-code. We explain all the computation steps on Bubble Sort (see Figure 2.4) used as input for our algorithm. It is a typical example of a program, where our method provides better precision than the state-of-the-art tools.

2.3.1 The Main Procedure

The main procedure `ExecuteProgram` takes a flowgraph of a program as input and returns the set of backbones of the flowgraph, the result of symbolic execution of these backbones and the overall bounds for transitions in the flowgraph. In the notation of the pseudo-code, β stays for a function

[1] In practise, we use more sophisticated method for computing variable values after loops.

Algorithm 1: `ExecuteProgram(`G`)`

Input:

G *// a flowgraph of a program*

Output:

$(\{(\pi_1, \theta_1, \varphi_1), \ldots, (\pi_k, \theta_k, \varphi_k)\}, \beta)$ *// the set of feasible backbones of the flowgraph with symbolic memories and path conditions after their execution; the overall bounds for all transitions*

1 $\{\pi_1, \ldots, \pi_k\} \longleftarrow$`DetermineBackbones(`$G$`)`
2 `result` $\longleftarrow \emptyset$
3 **foreach** $i = 1, \ldots, k$ **do**
4 Initialize β_i to return $\{0\}$ for each transition.
5 Initialize θ_i to return $\underline{a}$ for each scalar variable `a` and $\underline{A}$ for each array variable `A`.
6 Initialize φ_i to $true$.
7 Let $\pi_i = v_1 \ldots v_n$.
8 **foreach** $j = 1, \ldots, n-1$ **do**
9 **if** v_j is a loop entry **then**
10 Let C be the loop with the loop entry v_j along π_i.
11 Compute the flowgraph $P(C, v_j)$ induced by the loop.
12 $(\beta_i, \theta_i) \longleftarrow$`ProceedLoop(`$P(C, v_j), \theta_i, \varphi_i, \beta_i$`)`
13 Let t be the transition (v_j, ι, v_{j+1}).
14 **if** ι *has the form* `assume`(ψ) *and* $\theta_i(\psi)$ *contains no* $\star$ **then**
15 $\varphi_i \longleftarrow \varphi_i \wedge \theta_i(\psi)$
16 **if** φ_i *is not satisfiable* **then**
17 Continue at line 24.
18 **if** ι *has the form* `a` $:= e$ **then**
19 $\theta_i($`a`$) \longleftarrow \theta_i(e)$
20 **if** ι *has the form* `A`$[i_1, \ldots, i_m] := e$ **then**
21 $\theta_i($`A`$)$
 $\longleftarrow \lambda(x_1, \ldots, x_m)$. **ite**$(\bigwedge_{n=1}^{m} x_n = \theta_i(i_n), \theta_i(e), \theta_i($`A`$)(x_1, \ldots, x_m))$
22 **foreach** $\rho \in \beta_i(t)$ **do**
23 $\rho \longleftarrow \rho + 1$
24 Insert $(\pi_i, \theta_i, \varphi_i)$ into `result`.
25 **foreach** *transition* t **do**
26 $\beta(t) \longleftarrow \emptyset$
27 **foreach** $(\rho_1, \ldots, \rho_k), \rho_i \in \beta_i(t)$ *for each* $i \in \{1, \ldots, k\}$ **do**
28 Insert $\mathbf{max}(\rho_1, \ldots, \rho_k)$ into $\beta(t)$.
29 **return** (`result`, β)

```
void bubble_sort(int n, int* A){
  for(int i = 0; i < n - 1; i++){
    for(int j = 0; j < n - i - 1; j++){
      if(A[j+1] < A[j]){
        int tmp = A[j + 1];
        A[j + 1] = A[j];
        A[j] = tmp;
}}}}
```

Figure 2.4: Bubble Sort written in C.

that maps each transition to a set of its bounds (in opposite to the previous section, we allow more bounds for each transition). Further, π, θ, φ stay for a backbone, a symbolic memory and a path condition, respectively. Note that in practice, we do not compute bounds for all the transitions. If we are interested only in the loop bounds, one transition per loop is enough.

At first, the algorithm finds the backbones of the input flowgraph. The outer `foreach` loop iterates over the backbones and do the analysis for each of them independently. This provides the symbolic memory and the path condition resulting from our (modified) symbolic execution along the backbones. In the end, the overall bounds are computed from the local bounds of the backbones.

Running Example 2.3.1. *The flowgraph of Bubble Sort is depicted in Figure 2.5. There is only one backbone $\pi_1 = abk$ with one loop entry b to the loop $\{b, c, d, e, f, g, h, i, j\}$.*

Let us have a look now at one iteration of the outer `foreach` loop (lines 4 to 24). Assume we analyse the backbone π_i. We initialize its local bounds β_i to return $\{0\}$ for each transition. If some transition remains unvisited at the end of the analysis of π_i, it means it is unreachable during an execution along π_i, and 0 is a correct bound for it. If, at some point of the analysis, we are not able to infer any bound for some transition, we set β_i to return $\emptyset$ for it.[2] Next, the initialization of local symbolic memory θ_i and local path condition φ_i takes place. Note that we always denote an initial value of some variable x by $\underline{x}$ (respectively $\underline{x}', \underline{x}'', \ldots$ at loops).

Running Example 2.3.2. *In our example, we will compute bounds only for transitions t_1, t_2, t_3, and t_4. The initial values of local symbolic memory, path condition*

[2] An empty set of bounds can be understood as a possibly infinite number of executions.

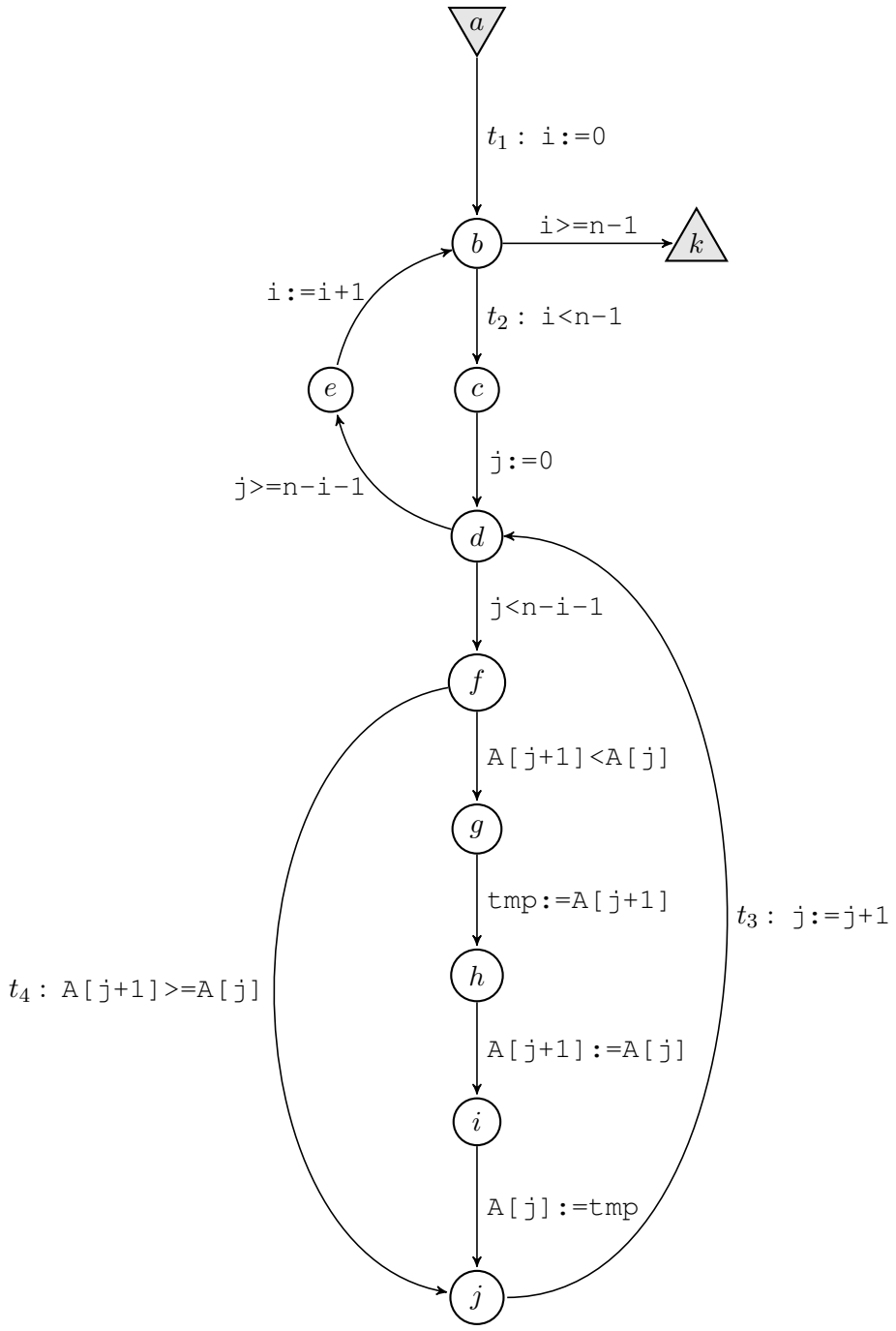

Figure 2.5: Flowgraph of the program in Figure 2.4. t_1, t_2, t_3, and t_4 mark the corresponding transitions (they are not part of the instructions).

and bounds for the only backbone π_1 are:

$$\theta_1 = \{\mathtt{n} \mapsto \underline{n}, \mathtt{A} \mapsto \underline{A}, \mathtt{i} \mapsto \underline{i}, \mathtt{j} \mapsto \underline{j}, \mathtt{tmp} \mapsto \underline{tmp}\}$$
$$\varphi_1 \equiv \mathit{true}$$
$$\beta_1(t_1) = \beta_1(t_2) = \beta_1(t_3) = \beta_1(t_4) = \{0\}$$

In the inner `foreach` loop (lines 8 to 23), the backbone is symbolically executed step by step (with a special treatment of loops). In every iteration we move to the next location on the backbone. The current location is denoted by v_j. If it is not a loop entry, we directly execute the next transition along the backbone and include the effect into the symbolic state, i.e. either the path condition φ_i (for `assume` instructions) or the symbolic memory θ_i (for assignments) is altered. If the transition is of the `assume` type and the new path condition is not satisfiable, no further transitions along the backbone can be visited. Hence we finish the execution with the current bounds, save the computed results for the backbone and then continue with the next cycle of the outer `foreach` loop at line 3. Otherwise, we raise all bounds for the executed transition by 1 and continue further with the execution of the backbone.

Running Example 2.3.3. *In our example, the symbolic memory after the execution of the transition $(a, \mathtt{i} := 0, b)$ is $\theta_1 = \{\mathtt{n} \mapsto \underline{n}, \mathtt{A} \mapsto \underline{A}, \mathtt{i} \mapsto 0, \mathtt{j} \mapsto \underline{j}, \mathtt{tmp} \mapsto \underline{tmp}\}$. The set of bounds for t_1 changes from $\{0\}$ to $\{1\}$. The path condition remains unchanged.*

If the current location is a loop entry, we need first to detect and take into consideration the effect of the loop before we continue with the next transition along the backbone. For that we compute the flowgraph induced by the loop and use it as an input to the procedure `ProceedLoop`. It alters not only the bounds for the transitions inside the loop, but also values of all variables modified inside the loop.

Running Example 2.3.4. *The flowgraph induced by the loop with the loop entry b from our example is depicted in Figure 2.6. After the loop procedure, we have the following values:*

$$\theta_1 = \{\mathtt{n} \mapsto \underline{n}, \mathtt{A} \mapsto \lambda x.\star, \mathtt{i} \mapsto \star, \mathtt{j} \mapsto \star, \mathtt{tmp} \mapsto \star\}$$
$$\varphi_1 \equiv \mathit{true}$$
$$\beta(t_1) = \{1\}, \beta(t_2) = \{\mathbf{max}(0, \underline{n} - 1)\},$$
$$\beta(t_3) = \beta(t_4) = \{\mathbf{ite}(\underline{n} < 2, 0, \tfrac{(\underline{n}-1)\cdot n}{2})\}$$

The steps leading to this result are described at Running Example 2.3.5 in Subsection 2.3.2.

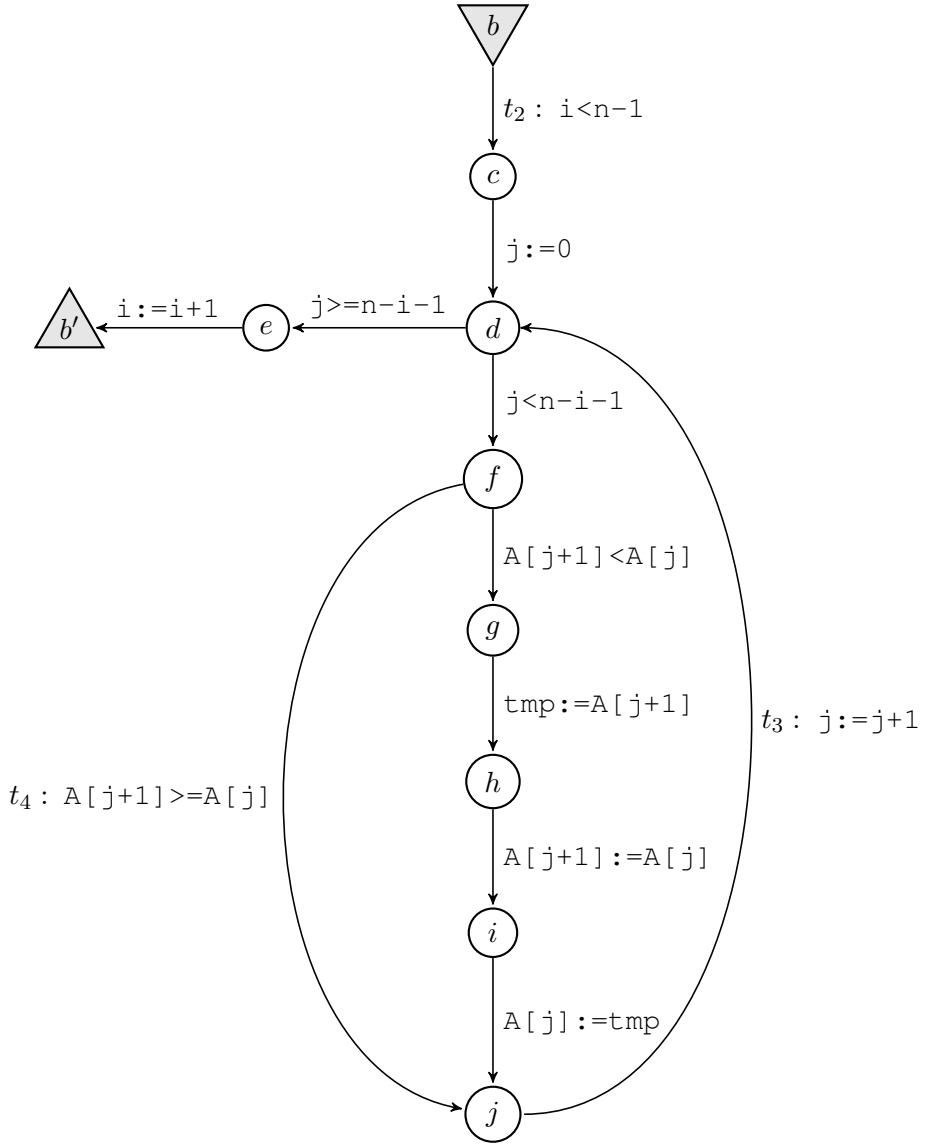

Figure 2.6: The flowgraph induced by the loop from Figure 2.5.

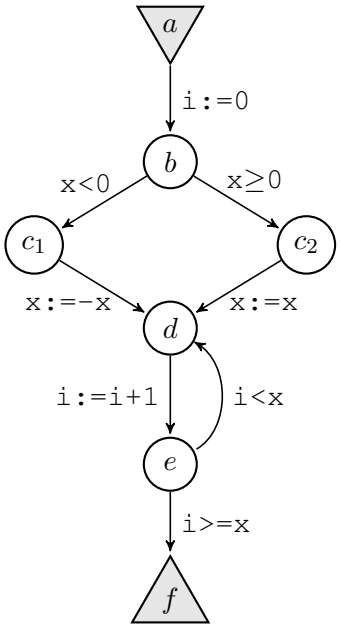

Figure 2.7: Example to show computing of the overall bounds out of the local ones.

When we are done with all the backbones, the last remaining thing is to put the bounds together. For every transition t, we first initialize $\beta(t)$ to return $\emptyset$. Then for each combination of bounds holding for t on the separate backbones, we need to take the maximum of them. This is done at lines 27 to 28. Because all the resulting bounds are correct, the most precise one is the minimum of them. Note that we would get the same result by the following expression: $\mathbf{max}(\mathbf{min}(\beta_i(t)) \mid i \in \{1, \ldots, k\})$. The reason why we do not use it is that we need all the separate bounds as simple as possible when we process nested loops, as we will see later in Subsection 2.3.4.

Let us explain the idea on an example with three feasible backbones with their local bounds $\beta_1(t) = \{\underline{a}, \underline{b}\}, \beta_2(t) = \{\underline{c}, \underline{b} + 1\}$ and $\beta_3(t) = \{\underline{a}\}$. The overall bounds are then $\beta(t) = \{\mathbf{max}(\underline{a}, \underline{c}, \underline{a}), \mathbf{max}(\underline{a}, \underline{b} + 1, \underline{a}), \mathbf{max}(\underline{b}, \underline{c}, \underline{a}), \mathbf{max}(\underline{b}, \underline{b} + 1, \underline{a})\} = \{\mathbf{max}(\underline{a}, \underline{c}), \mathbf{max}(\underline{a}, \underline{b} + 1), \mathbf{max}(\underline{b}, \underline{c}, \underline{a})\}$. Hence for input values 1 for the variable `a`, 3 for `b` and 2 for `c`, the most precise bound is $\mathbf{max}(\underline{a}, \underline{c}) = \mathbf{max}(1, 2) = 2$. Thus the transition t is executed at most 2 times for this particular input values.

Another example, that is more realistic, is the flowgraph depicted in Figure 2.7. It is an altered version of the example from the previous section:

we added the update of `x` to its absolute value before entering the loop. The flowgraph has two backbones: $\pi_1 = abc_1def$ and $\pi_2 = abc_2def$. Let $t = (d, \texttt{i:=i+1}, e)$. While the bound set $\beta_2(t) = \{\mathbf{max}(1, \underline{x})\}$ for the second backbone (as we inferred in Section 2.2), the first backbone enters d with the value $-\underline{x}$ of `x`, resulting in the bound set $\beta_1(t) = \{\mathbf{max}(1, -\underline{x})\}$. Thus the overall set of bounds for t is $\beta(t) = \{\mathbf{max}(\mathbf{max}(1, -\underline{x}), \mathbf{max}(1, \underline{x}))\} = \{\mathbf{max}(1, -\underline{x}, \underline{x})\}$.

2.3.2 The Loop Procedure

The procedure `ProceedLoop` takes a flowgraph induced by a loop as input, together with the symbolic memory θ_{in}, the path condition φ_{in}, and the bound function β_{in}, which all hold while entering the loop. It returns the bound function β_{out} and the symbolic memory after the loop.

For computing bounds inside the loop, we do not take into account the input bounds at first. We introduce a new bound function β_{loop}, which stores only the bounds on the number of executions during looping. In opposite to the main function, we initialize it to return $\emptyset$ for each transition inside the loop. The bounds for each transition are then added only if we are able to infer the bound on the number of iterations going through it. The same holds for β_{out}, but β_{out} must return the same value as β_{in} for the transitions outside the loop.[3]

After the initialization of bound functions, we first compute the effect of one iteration of the loop (line 3), without any dependence to the input symbolic memory, path condition or bound function. Thus we get symbolic memories and path conditions for all loop paths and the bound function β_{inner}. If there is no other loop inside the currently processed loop, β_{inner} returns only bound sets $\{1\}$ or $\{0\}$ for each transition, because no transition could be visited twice during one iteration. However, there can be some non-trivial bounds for some transitions, if there is a nested loop inside. To distinguish the variable values resulting from line 3 from the input variable values, we use the primed symbols for variables ($\underline{a}', \underline{a}'', \dots$).

We continue with the computation of a loop summary, which is the effect of iterating the loop several times on the symbolic memory. More precisely, if we have loop paths $\pi_1, \dots, \pi_k$ and path counters $\kappa_1, \dots \kappa_k$, the symbolic memory $\theta^{\vec{\kappa}}$ keeps values of variables after κ_1 iterations of π_1, κ_2 iterations of π_2 etc. Note that we abstract from the order in which the loop paths are taken during the looping. This part of the analysis is described in more

[3] The domain of β_{out} contains also transitions, which are not in G_{in}.

Algorithm 2: `ProceedLoop` $(G_{in}, \theta_{in}, \varphi_{in}, \beta_{in})$

Input:

$(G_{in}, \theta_{in}, \varphi_{in}, \beta_{in})$ *// a flowgraph induced by a loop; an input symbolic memory and path condition; input bounds for all transitions*

Output:

$(\beta_{out}, \theta_{out})$ *// the bounds and symbolic memory after the loop*

1 Initialize β_{loop} to return $\emptyset$ for each transition t in G_{in}.
2 Initialize β_{out} to return $\emptyset$ for each transition t in G_{in} and $\beta_{in}(t)$ otherwise.
3 $(\{(\pi_1, \theta_1, \varphi_1), \ldots, (\pi_k, \theta_k, \varphi_k)\}, \beta_{inner}) \longleftarrow$ `ExecuteProgram` (G_{in})
4 $\theta^{\vec{\kappa}} \longleftarrow$ `ComputeSummary` $(\{(\pi_1, \theta_1), \ldots, (\pi_k, \theta_k)\})$
5 **if** $\beta_{inner}(t') = \emptyset$ *for some transition* t' *in* G_{in} **then**
6 Continue at line 27.
7 **foreach** $i = 1, \ldots, k$ **do**
8 $\varphi_i^{\vec{\kappa}} \longleftarrow$ `UpdateWithKappa` $(\varphi_i, \theta^{\vec{\kappa}})$
9 $\varphi_i^{\vec{\kappa}} \longleftarrow$ `UpdateWithInput` $(\varphi_i^{\vec{\kappa}}, \theta_{in})$
10 $\varphi_i^{\vec{\kappa}} \longleftarrow \varphi_i^{\vec{\kappa}} \wedge \varphi_{in}$
11 **foreach** *transition* t *in* G_{in} **do**
12 **if** $\beta_{inner}(t) = \{0\}$ **then**
13 $\beta_{out}(t) \longleftarrow \{0\}$
14 Continue at line 10.
15 **if** t *is not a part of any loop in* G_{in} **then**
16 Let $X = \{j \mid t$ connects two subsequent locations from $\pi_j\}$
17 $\beta_{loop}(t) \longleftarrow$ `ComputeBounds` $(X, \bigvee_{j \in X} \varphi_j^{\vec{\kappa}})$
18 **else**
19 Let $X = \{j \mid t$ lies on a loop, the loop entry of which is on $\pi_j\}$
20 **foreach** $\rho_{inner} \in \beta_{inner}(t)$ **do**
21 $\rho_{inner}^{\vec{\kappa}} \longleftarrow$ `UpdateWithKappa` $(\rho_{inner}, \theta^{\vec{\kappa}})$
22 $\rho_{inner}^{\vec{\kappa}} \longleftarrow$ `UpdateWithInput` $(\rho_{inner}^{\vec{\kappa}}, \theta_{in})$
23 $\beta_{loop}(t) \longleftarrow \beta_{loop}(t) \cup$ `NestedBounds` $(\rho_{inner}^{\vec{\kappa}}, X, \varphi_1^{\vec{\kappa}}, \ldots, \varphi_k^{\vec{\kappa}})$
24 **foreach** $\rho_{loop} \in \beta_{loop}(t)$ **do**
25 **foreach** $\rho_{in} \in \beta_{in}(t)$ **do**
26 Insert $\rho_{loop} + \rho_{in}$ into $\beta_{out}(t)$.
27 $\theta_{out} \longleftarrow$ `MemoryAfterLoop` $(\theta^{\vec{\kappa}})$
28 **return** $(\beta_{out}, \theta_{out})$

detail in Subsection 2.3.3.

For the following steps, we want to be sure, that the execution will not "get stuck" in the middle of an iteration of the currently processed loop, i.e. either a whole iteration is performed, or the loop is exited. Such situation can occur only if there is an unbounded loop nested in the currently processed one. Thus to be on the safe side, we leave all the bounds for all transitions inside G_{in} empty in such case and continue to the end of the procedure. This is done at lines 5 to 6.

Remark 2.14. *In the following, by an* iteration along a loop path π_i *we mean an iteration of the loop, that follows a path, the backbone of which is* π_i *(see Remark 2.4), and by an* iteration along a set of loop paths Π *we mean an iteration along a loop path from* Π*. By* $\vec{\kappa}$ *iterations we mean* κ_1 *iterations along* π_1*,* κ_2 *iterations along* π_2 *etc. By a* bound for a loop path π (resp. a set of loop paths Π) *we mean an upper bound on the number of iterations along* π *(resp.* Π*).*

For each loop path π_i, we need to compute the condition for performing an iteration along π_i after $\vec{\kappa}$ iterations. In the path condition φ_i for a single iteration along π_i (independent on the input), we first substitute the value $\theta^{\vec{\kappa}}(\mathtt{x})$ for each symbol $\underline{x}'$ in φ_i, and then we substitute the value $\theta_{in}(\mathtt{x})$ for each symbol $\underline{x}'$ in φ_i. In the end, we add the path condition φ_{in}, with which we entered the loop. The condition for the loop path π_i after $\vec{\kappa}$ iterations is denoted by $\varphi_i^{\vec{\kappa}}$.

Let us show the idea on a loop with one loop path π_1, a condition $\varphi_1 \equiv \underline{x}' > 0$ and the symbolic memory resulting from the loop summary $\theta^{\kappa_1}(\mathtt{x}) = \underline{x}' - \kappa_1$. Roughly speaking, the meaning of the condition φ_1 is that if $\underline{x}'$ is the value of x at the loop entry and $\underline{x}' > 0$ holds, an iteration of the loop can be performed. So if we want to infer the condition for entering the loop after κ_1 iterations, we need to substitute the value of x after κ_1 iterations for $\underline{x}'$ in the condition φ_1. Assume $\underline{x}$ is the input value for x. Then $\underline{x} - \kappa_1$ is the value of x after κ_1 iterations and we get the condition $\underline{x} - \kappa_1 > 0 \ \equiv \ \kappa_1 \leq \underline{x} - 1$. Hence the number of iterations before the last iteration is at most $\underline{x} - 1$, so the upper bound for the number of iterations is $\mathbf{max}(0, \underline{x})$.

At this point, we can start to compute the bounds. In every iteration of the `foreach` loop at lines 11 to 26, the resulting set of bounds is computed for one transition t from the induced flowgraph G_{in}. If $\beta_{inner}(t) = \{0\}$, t cannot be visited during any loop iteration, so $\beta_{out}(t) = \beta_{in}(t)$. The computation is simple, if t is not a part of any nested loop in G. Then t must lie on at least one loop path. The set X contains the indices of loop paths, on which t lies, so we want to compute bounds on the number of iterations

```
while (x>0)
  if (y>0)
    y:=y-1;
  x:=x-1;
```

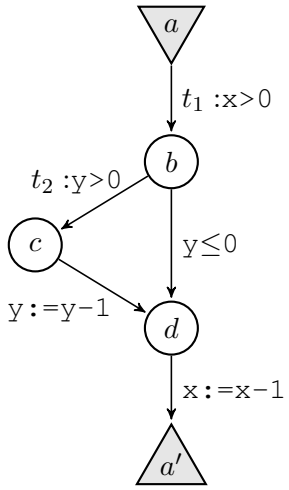

Figure 2.8: Code with a multi-path loop on the left and the flowgraph induced by the loop on the right.

along loop paths with indices from X. Every time t is visited, at least one of the conditions for those loop paths must be satisfied, which means their disjunction must hold. We use these conditions as input for the procedure `ComputeBounds` (see Subsection 2.3.5).

Let us have a look at the loop in Figure 2.8. The induced flowgraph has two loop paths: $\pi_1 = abcda'$ and $\pi_2 = abda'$ with conditions $\varphi_1 \equiv (\underline{x}' > 0) \wedge (\underline{y}' > 0)$ and $\varphi_2 \equiv (\underline{x}' > 0) \wedge (\underline{y}' \leq 0)$. The procedure `ComputeSummary` infers $\theta^{\kappa_1,\kappa_2}(\mathtt{x}) = \underline{x}' - \kappa_1 - \kappa_2$ and $\theta^{\kappa_1,\kappa_2}(\mathtt{y}) = \underline{y}' - \kappa_1$. Suppose $\theta_{in}(\mathtt{x}) = \underline{x}$, $\theta_{in}(\mathtt{y}) = \underline{y}$ and $\varphi_{in} \implies (\underline{x} > 0) \wedge (\underline{y} > 0)$. Then $\varphi_1^{\kappa_1,\kappa_2} \equiv (\underline{x} - \kappa_1 - \kappa_2 > 0) \wedge (\underline{y} - \kappa_1 > 0) \wedge \varphi_{in}$ and $\varphi_2^{\kappa_1,\kappa_2} \equiv (\underline{x} - \kappa_1 - \kappa_2 > 0) \wedge (\underline{y} - \kappa_1 \leq 0) \wedge \varphi_{in}$. The transition t_1 lies on both π_1 and π_2, so after κ_1 iterations of π_1 and κ_2 iterations of π_2 it is visited $\kappa_1 + \kappa_2$ times. Hence we want to compute the upper bound on the size of $\kappa_1 + \kappa_2$. Note that after each iteration along $\{\pi_1, \pi_2\}$ the sum $\kappa_1 + \kappa_2$ is increased by 1 and it is 0 at the beginning. Moreover, after $\vec{\kappa}$ iterations, the condition to enter the loop along $\{\pi_1, \pi_2\}$ is $\varphi_1^{\kappa_1,\kappa_2} \vee \varphi_2^{\kappa_1,\kappa_2} \equiv \underline{x} - \kappa_1 - \kappa_2 > 0 \wedge \varphi_{in}$. From that we conclude $\kappa_1 + \kappa_2 \leq \underline{x} - 1$, so the biggest size of $\kappa_1 + \kappa_2$ to iterate the loop along $\{\pi_1, \pi_2\}$ is $\underline{x} - 1$. At the end of that iteration, the sum $\kappa_1 + \kappa_2$ is increased by 1 for the last time. Thus the bound for t_1 is $\underline{x} - 1 + 1 = \underline{x}$. The transition t_2 lies only on π_1, so we compute the bound for κ_1 with the condition $\varphi_1^{\kappa_1,\kappa_2} \equiv (\underline{x} - \kappa_1 - \kappa_2 > 0) \wedge (\underline{y} - \kappa_1 > 0) \wedge \varphi_{in}$. From $\underline{x} - \kappa_1 - \kappa_2 > 0$ we infer

$\kappa_1 + \kappa_2 \leq \underline{x} - 1$ and thus $\kappa_1 \leq \underline{x} - 1$ (all path counters are non-negative, so we can subtract κ_2 from the left side of the inequation), and from $\underline{y} - \kappa_1 > 0$ we infer $\kappa_1 \leq \underline{y} - 1$. Thus we have inferred two bounds for t_2: $\underline{x}$ and $\underline{y}$.

The situation is more complicated if the transition t, for which we compute the bounds, lies inside a loop nested in the currently processed one. The set X then contains indices of loop paths, from which the nested loop can be entered. Note that at this point of the analysis, the nested loop on which t lies was already processed during the procedure `ExecuteProgram` at line 3, which means that its inner bounds (during one iteration of the outer loop) are already stored in β_{inner}. However, these bounds are still independent on θ_{in} and $\theta^{\vec{\kappa}}$. Hence, we transform them at lines 21 to 22 in the same way like we transformed the conditions at lines 8 to 9. From every inner bound for t we can infer several overall bounds. Here, by overall bounds we mean the maximum number of executions of t during all iterations of the currently processed loop. The procedure `NestedBounds` is explained in detail in Subsection 2.3.4.

Let us look now at lines 24 to 26. All bounds for t during the looping are stored in $\beta_{loop}(t)$. We need to add them to the input bounds $\beta_{in}(t)$. Note that each combination $\rho_{in} + \rho_{loop}$ is a correct bound, because $\rho_{in} \in \beta_{in}(t)$ is a correct bound for t before the looping and $\rho_{loop} \in \beta_{loop}$ is a correct bound for t during the looping. All such inferred bounds are added to the final result $\beta_{out}(t)$.

Finally, we need to compute the symbolic memory after the loop, that assigns only symbolic expressions without path counters to the variables. For simplicity, we can set all variables, which are changed inside the loop, to the symbol $\star$ (resp. $\lambda(x_1, \ldots, x_k).\star$ for array variables of arity k) and keep the values from θ_{in} for the unchanged ones. More sophisticated method is proposed in Subsection 2.3.6.

Running Example 2.3.5. *Let us go through the loop procedure once again with Bubble Sort. We start with the loop from Figure 2.6. The input symbolic memory is* $\theta_{in} = \{\mathtt{n} \mapsto \underline{n}, \mathtt{A} \mapsto \underline{A}, \mathtt{i} \mapsto 0, \mathtt{j} \mapsto \underline{j}, \mathtt{tmp} \mapsto \underline{tmp}\}$*, the input path condition* $\varphi_{in} \equiv$ *true and the input bounds* $\beta_{in}(t_1) = \overline{\{1\}}, \beta_{in}(t_2) = \beta_{in}(t_3) = \beta_{in}(t_4) = \{0\}$*. After initialization of* β_{loop} *and* β_{out} *we call* `ExecuteProgram` *on the flowgraph. The procedure first determines its only backbone* $\pi_1 = bcdeb'$*. During the symbolic execution along* π_1 *we come to the (nested) loop entry* d *and call* `ProceedLoop` *again, this time with the flowgraph induced by the nested loop, that is depicted in Figure 2.9.*

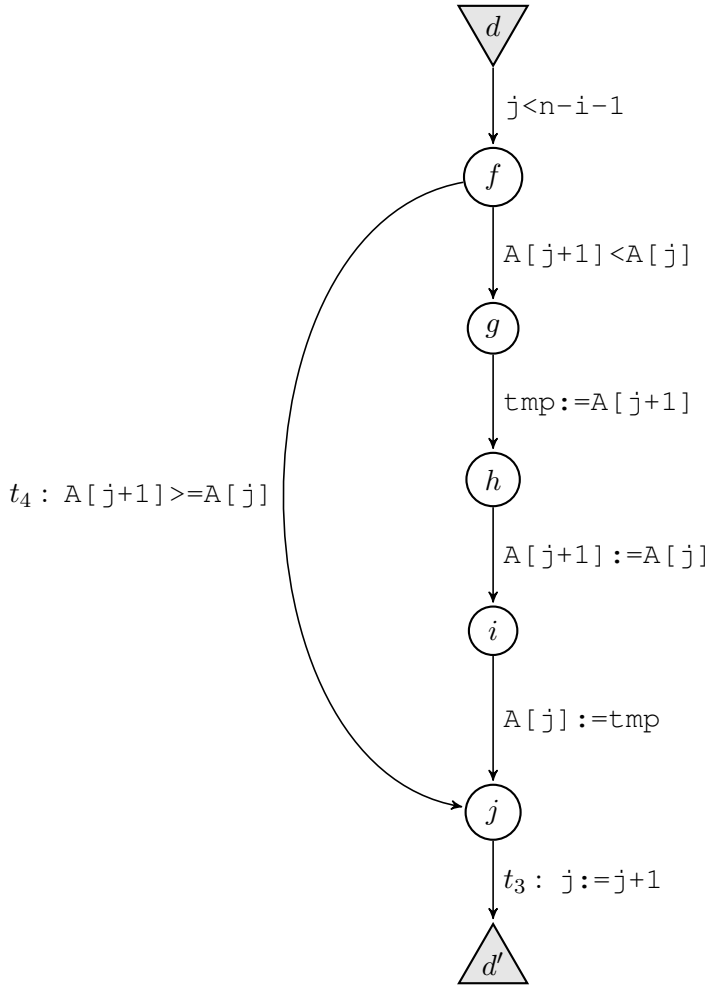

Figure 2.9: The flowgraph induced by the loop from Figure 2.6.

Let us go through `ProceedLoop` *with the nested loop in Figure 2.9. The input symbolic memory is* $\theta_{in} = \{\mathtt{n} \mapsto \underline{n}', \mathtt{A} \mapsto \underline{A}', \mathtt{i} \mapsto \underline{i}', \mathtt{j} \mapsto 0, \mathtt{tmp} \mapsto \underline{tmp}'\}$*, the input path condition* $\varphi_{in} \equiv \underline{i}' < \underline{n}' - 1$ *and the input bounds* $\beta_{in}(t_2) = \{1\}, \beta_{in}(t_3) = \beta_{in}(t_4) = \{0\}$*. After initialization of* β_{loop} *and* β_{out} *we call* `ExecuteProgram`*. There are two backbones with no loop entry in the flowgraph:* $\pi_1 = dfjd'$ *and* $\pi_2 = dfghijd'$*. The procedure returns the following results:*

$$
\begin{aligned}
&\theta_1 = \{\mathtt{n} \mapsto \underline{n}'', \mathtt{A} \mapsto \underline{A}'', \mathtt{i} \mapsto \underline{i}'', \mathtt{j} \mapsto \underline{j}'' + 1, \mathtt{tmp} \mapsto \underline{tmp}''\} \\
&\varphi_1 \equiv \underline{j}'' < \underline{n}'' - \underline{i}'' - 1 \ \wedge \ \underline{A}''(\underline{j}'' + 1) \geq \underline{A}''(\underline{j}'') \\
&\theta_2 = \{\mathtt{n} \mapsto \underline{n}'', \mathtt{A} \mapsto \lambda x.\, \mathbf{ite}(x = \underline{j}'', \underline{A}''(\underline{j}'' + 1), \mathbf{ite}(x = \underline{j}'' + 1, \\
&\qquad \underline{A}''(\underline{j}''), \underline{A}''(x))), \mathtt{i} \mapsto \underline{i}'', \mathtt{j} \mapsto \underline{j}'' + 1, \mathtt{tmp} \mapsto \underline{A}''(\underline{j}'' + 1)\} \\
&\varphi_2 \equiv \underline{j}'' < \underline{n}'' - \underline{i}'' - 1 \ \wedge \ \underline{A}''(\underline{j}'' + 1) < \underline{A}''(\underline{j}'') \\
&\beta_{inner}(t_3) = \beta_{inner}(t_4) = \{1\}
\end{aligned}
$$

We continue at line 4 of `ProceedLoop`*. The symbolic memory resulting from the loop summary computation (as described in Subsection 2.3.3) is* $\theta^{\kappa_1,\kappa_2} = \{\mathtt{n} \mapsto \underline{n}'', \mathtt{A} \mapsto \lambda x.\star, \mathtt{i} \mapsto \underline{i}'', \mathtt{j} \mapsto \underline{j}'' + \kappa_1 + \kappa_2, \mathtt{tmp} \mapsto \star\}$*. We do not take the* `if` *branch at line 5, because no inner bound set is empty. After updating* φ_1 *and* φ_2 *at lines 7 to 10 we get:*

$$
\begin{aligned}
\varphi_1^{\kappa_1,\kappa_2} &\equiv \kappa_1 + \kappa_2 < \underline{n}' - \underline{i}' - 1 \ \wedge \ \star \geq \star \ \wedge \ \varphi_{in} \\
&\equiv \kappa_1 + \kappa_2 < \underline{n}' - \underline{i}' - 1 \ \wedge \ \underline{i}' < \underline{n}' - 1 \\
\varphi_2^{\kappa_1,\kappa_2} &\equiv \kappa_1 + \kappa_2 < \underline{n}' - \underline{i}' - 1 \ \wedge \ \underline{i}' < \underline{n}' - 1
\end{aligned}
$$

Now we can compute the bounds for the transitions t_3 *and* t_4*. Neither of them has the inner bound* $\{0\}$ *and neither is a part of a loop inside the currently processed loop (there is no such loop). Let us start with* t_3*: It lies on both* π_1 *and* π_2*, so* $X = \{1, 2\}$ *and we compute the upper bound on the size of* $\kappa_1 + \kappa_2$ *after the last iteration from the looping condition* $\varphi_1^{\kappa_1,\kappa_2} \vee \varphi_2^{\kappa_1,\kappa_2} \equiv \kappa_1 + \kappa_2 < \underline{n}' - \underline{i}' - 1 \ \wedge \ \underline{i}' < \underline{n}' - 1$*, from which we infer the bound* $\beta_{loop}(t_3) = \{\underline{n}' - \underline{i}' - 1\}$ *(see Subsection 2.3.5). Because* $\beta_{in}(t_3) = \{0\}$*, from the loops at lines 24 to 26 we get* $\beta_{out}(t_3) = \{\underline{n}' - \underline{i}' - 1 + 0\}$*. So as for* t_4*, it lies only on* π_1*, so we compute the bound on the size of* κ_1*. Similarly like in the previous transition, we get the looping condition* $\kappa_1 + \kappa_2 < \underline{n}' - \underline{i}' - 1$*. We can safely subtract* κ_2 *from the left side (because path counters are non-negative) and get* $\kappa_1 < \underline{n}' - \underline{i}' - 1$*. Hence* $\beta_{out}(t_4) = \beta_{out}(t_3) = \{\underline{n}' - \underline{i}' - 1\}$*. For the transition* t_2 *outside the currently processed loop we have* $\beta_{out}(t_2) = \{1\}$*. Finally, we compute the output symbolic memory* $\theta_{out} = \{\mathtt{n} \mapsto \underline{n}', \mathtt{A} \mapsto \lambda(x.\star), \mathtt{i} \mapsto \underline{i}', \mathtt{j} \mapsto \star, \mathtt{tmp} \mapsto \star\}$*.*

Now we continue with the outer loop (the flowgraph in Figure 2.6). After adding the effect of the inner loop and finishing the symbolic execution along π_1*,*

`ExecuteProgram` returns the following:

$$\theta_1 = \{\mathtt{n} \mapsto \underline{n}', \mathtt{A} \mapsto \lambda x.\star, \mathtt{i} \mapsto \underline{i}' + 1, \mathtt{j} \mapsto \star, \mathtt{tmp} \mapsto \star\}$$
$$\varphi_1 \equiv (\underline{i}' < \underline{n}' - 1) \ \wedge \ (\star \geq \underline{n}' - \underline{i}' - 1) \ \equiv \ \underline{i}' < \underline{n}' - 1$$
$$\beta_{inner}(t_2) = \{1\}, \beta_{inner}(t_3) = \beta_{inner}(t_4) = \{0, \underline{n}' - \underline{i}' - 1\}$$

`ComputeSummary` returns $\theta^{\kappa_1} = \{\mathtt{n} \mapsto \underline{n}', \mathtt{A} \mapsto \lambda x.\star, \mathtt{i} \mapsto \underline{i}' + \kappa_1, \mathtt{j} \mapsto \star, \mathtt{tmp} \mapsto \star\}$. The `if` branch at line 5 is not taken. After the loop at lines 7 to 10, we have $\varphi_1^{\kappa_1} \equiv \kappa_1 < \underline{n} - 1 \ \wedge \ \varphi_{in} \equiv \kappa_1 < \underline{n} - 1$. The transition t_2 does not have the inner bound $\{0\}$, it is not a part of the nested loop and it lies on π_1, so we infer the bound $\mathbf{max}(0, \underline{n} - 1)$ for it. The transitions t_3 and t_4 lie in the nested loop. As we infer in Subsection 2.3.4 (Running Example 2.3.6), the resulting bounds for both are $\{\mathbf{ite}(\underline{n} < 2, 0, \frac{(\underline{n}-1)\cdot n}{2})\}$. At the end we get the result $\beta_{out}(t_1) = \{1\}, \beta_{out}(t_2) = \{\mathbf{max}(0, \underline{n} - 1)\}, \beta_{out}(t_3) = \beta_{out}(t_4) = \{\mathbf{ite}(\underline{n} < 2, 0, \frac{(\underline{n}-1)\cdot n}{2})\}$. The symbolic memory after the loop is $\theta_{out} = \{\mathtt{n} \mapsto \underline{n}, \mathtt{A} \mapsto \lambda x.\star, \mathtt{i} \mapsto \star, \mathtt{j} \mapsto \star, \mathtt{tmp} \mapsto \star\}$. With the output values we continue the analysis on the main flowgraph (Figure 2.5). However, no bound is further changed.

We have described the two main parts of our algorithm: `ExecuteProgram` and `ProceedLoop`. We have shown them step by step on Bubble Sort. In the following subsections, we explain the smaller, but non-trivial parts of the algorithm.

2.3.3 Computation of a Loop Summary

The goal of the procedure `ComputeSummary` is to compute the effect of a loop on the symbolic memory in dependence on the number of iterations of each of its loop paths. That means that we abstract from the order, in which the various loop paths were executed. This excludes loops like:

```
while (x<n)
  if (random)
    x:=x+1;
  else
    x:=x*2;
```

The reason why we cannot compute the effect of the loop is that taking the `if` branch in the first iteration and `else` branch in the second may result in different values than taking the `else` branch in the first and `if` branch in the second iteration. However the order, in which the loop paths are taken, is not usually important for the number of iterations in real programs.

Algorithm 3: `ComputeSummary` $(\{(\pi_1, \theta_1), \ldots, (\pi_l, \theta_l)\})$

Input:

$\{(\pi_1, \theta_1), \ldots, (\pi_l, \theta_l)\}$ *// results from single execution of loop paths*

Output:

$(\theta^{\vec{\kappa}})$ *// the computed summary*

1 Introduce fresh path counters $\vec{\kappa} = (\kappa_1, \ldots, \kappa_l)$ for loop paths $\pi_1, \ldots, \pi_l$, respectively.
2 Initialize $\theta^{\vec{\kappa}}$ to return $\star$ for each scalar variable and $\lambda(x_1, \ldots, x_k.\star)$, for each array variable of arity k.
3 **repeat**
4 `change` $\longleftarrow$ *false*
5 **foreach** scalar variable a *such that* $\theta^{\vec{\kappa}}(\mathtt{a}) = \star$ **do**
6 Compute an improved value e for the variable a from symbolic memories $\theta_1, \ldots, \theta_l$ and $\theta^{\vec{\kappa}}$.
7 **if** $e \neq \star$ **then**
8 $\theta^{\vec{\kappa}}(\mathtt{a}) \longleftarrow e$
9 `change` $\longleftarrow$ *true*
10 **until** `change` $=$ *false*
11 **foreach** array variable `A` **do**
12 **if** $\theta_i(\mathtt{A}) = \underline{A}$ *for all* $i \in \{1, \ldots, l\}$ **then**
13 $\theta^{\vec{\kappa}}(\mathtt{A}) \longleftarrow \underline{A}$
14 **return** $(\theta^{\vec{\kappa}})$

The input for the procedure are loop paths $\pi_1, \ldots, \pi_l$ and symbolic memories $\theta_1, \ldots, \theta_l$ denoting the effect of executing them once and the output is the symbolic memory $\theta^{\vec{\kappa}}$ after κ_1 iterations of π_1, κ_2 iterations of π_2, ..., and κ_l iterations of π_l.

After introducing the path counters $\kappa_1, \ldots, \kappa_l$, we safely initialize $\theta^{\vec{\kappa}}$ to return $\star$ for each scalar variable and $\lambda(x_1, \ldots, x_k).\star$ for each array variable of arity k. During the loop at lines 3 to 10, we improve the precision of $\theta^{\vec{\kappa}}$ until we reach a fix point (no value of any scalar variable is further changed). The crucial step is the computation of an improved value e for a scalar variable a (line 6). For that purpose, we use the definition from [18]. In the following, we use the notation $\theta^{\vec{\kappa}}\langle d \rangle$ for denoting the expression d, where every occurrence of each variable symbol $\underline{a}$ is replaced by $\theta^{\vec{\kappa}}(\mathtt{a})$. The improved value e is defined as $\star$, except in the following cases:

1. For each loop path π_i, we have $\theta_i(\mathtt{a}) = \underline{a}$. In other words, the value of a is not changed in any iteration of the loop. This case is trivial. We set $e = \underline{a}$.

2. For each loop path π_i, either $\theta_i(\mathtt{a}) = \underline{a}$ or $\theta_i(\mathtt{a}) = \underline{a} + d_i$ for some symbolic expression d_i such that $\theta^{\vec{\kappa}}\langle d_i \rangle$ contains neither $\star$ nor any path counters. Let us assume that the latter possibility holds for loop paths $\pi_1, \ldots, \pi_m$. The condition on $\theta^{\vec{\kappa}}\langle d_i \rangle$ guarantees that the value of d_i is constant during all iterations over the loop. In this case, we set $e = \underline{a} + \sum_{1 \leq i \leq m} \theta^{\vec{\kappa}}\langle d_i \rangle \cdot \kappa_i$.

3. There exists a symbolic expression d such that $\theta^{\vec{\kappa}}\langle d \rangle$ contains neither $\star$ nor any path counters. For each loop path π_i, either $\theta_i(\mathtt{a}) = \underline{a}$ or $\theta_i(\mathtt{a}) = d$. Let us assume that the latter possibility holds for loop paths $\pi_1, \ldots, \pi_m$. In other words, the value of a is set to d in each iteration with loop path π_i for $1 \leq i \leq m$, while it is unchanged in any other iteration. Hence, we set $e = \mathbf{ite}(\sum_{1 \leq i \leq m} \kappa_i > 0, \theta^{\vec{\kappa}}\langle d \rangle, \underline{a})$.

4. For one loop path, say π_i, $\theta_i(\mathtt{a}) = d$ for some symbolic expression d such that $\theta^{\vec{\kappa}}\langle d \rangle$ contains neither $\star$ nor any path counters except κ_i. Further, for each loop path π_j such that $i \neq j$, $\theta_j(\mathtt{a}) = \underline{a}$. That is, only iterations along loop path π_i modify a and they set it to a value independent on other path counters than κ_i. Note that if we assign d to a in the κ_i-th iteration with loop path π_i, then the actual assigned value of d is the value after $\kappa_i - 1$ iterations along the paths. Therefore we set $e = \mathbf{ite}(\kappa_i > 0, (\theta^{\vec{\kappa}}\langle d \rangle)[\kappa_i / \kappa_i - 1], \underline{a})$.

Note that one can add another cases covering other situations where the value of a can be expressed precisely, e.g. the case capturing geometric progressions (like `while (x<n) x:=2*x;`).

In Bubble Sort, we had only instances of the first and the second case and we have seen its application in practise in Subsection 2.3.2. Let us show the other cases on the following examples:

```
while (x>y)
  x:=y;
```

```
while (x>0)
  x:=y;
  y:=y-1;
```

The variable x from the left example falls into the third case, while in the right example it falls into the fourth case. The variable y falls into the first case in the left example and second case in the right example. Note that if the algorithm tries to compute the improved value for x before y, then after the first iteration of the loop at lines 3 to 10 of `ComputeSummary`, the symbolic memory $\theta^{\vec{\kappa}}$ returns $\star$ for x and $\underline{y}$ for y and the resulting value $\mathbf{ite}(\kappa_1 > 0, \underline{y}, \underline{x})$ (resp. $\mathbf{ite}(\kappa_1 > 0, \underline{y} - (\kappa_1 - 1), \underline{x})$ for the second example) is known after two iterations. However, the order of the variables, in which the summary is computed, does not influence the result.

For a simplicity of the algorithm we improve values of array variables only if they remain constant during the loop.

2.3.4 Dealing with Nested Loops

The goal of the procedure `NestedBounds` is to infer bounds for a transition t inside a loop C_{inner} nested in a loop C_{outer}. Let us describe the input values: The outer loop has k loop paths: $\pi_1, \ldots, \pi_k$. Further, $\varphi_i^{\vec{\kappa}}$ is the condition that after $\vec{\kappa}$ iterations the next iteration of the loop can be along π_i, and $\rho_{inner}^{\vec{\kappa}}$ is the bound on the number of executions of t during the next iteration after $\vec{\kappa}$ iterations. X is the set of indices of loop paths, along which t can be visited, i.e. they contain the loop entry to the nested loop, inside which t lies.

At first, we safely initialize B_{res} to $\emptyset$. We are able to infer the overall bounds only if the inner bound is in the specific form defined at line 2 of the procedure. Thus if the procedure fails to transform the bound into such form, it returns $\emptyset$. Let $\Pi^X = \{\pi_i \mid i \in X\}$. The set Y introduced at line 3 contains indices of loop paths, which are not in Π^X, but they can influence the inner bound. We want to have the inner bound expressed with just one path counter, so we introduce a fresh path counter K, which

Algorithm 4: `NestedBounds`$(\rho^{\vec{\kappa}}_{inner}, X, \varphi^{\vec{\kappa}}_1, \ldots, \varphi^{\vec{\kappa}}_k)$

Input:

$(\rho^{\vec{\kappa}}_{inner}, X, \varphi^{\vec{\kappa}}_1, \ldots, \varphi^{\vec{\kappa}}_k)$ *// a bound on the number of executions of a transition inside a nested loop in the next iteration after $\vec{\kappa}$ iterations of the outer loop; a set of indices of the outer loop paths with a loop entry to the inner loop; path conditions of all outer loop paths after $\vec{\kappa}$ iterations of the outer loop*

Output:

B_{res} *// the set of overall bounds*

1 $B_{res} \longleftarrow \emptyset$

2 Try to transform $\rho^{\vec{\kappa}}_{inner}$ to the form $\mathbf{max}(c, e + a_1\kappa_1 + \ldots + a_k\kappa_k)$, where $e, c, a_1, \ldots, a_k$ are symbolic expressions without path counters. If it is not possible, **return** $\emptyset$.

3 Let $Y = \{j \mid j \notin X \wedge a_j \neq 0\}$

4 $B^Y_{outer} \longleftarrow$ `ComputeBounds`$(Y, \bigvee_{j \in Y} \varphi^{\vec{\kappa}}_j)$

5 Introduce a fresh path counter K. // $K = \sum_{j \in X} \kappa_j$

6 $a^X_{max} \longleftarrow \mathbf{max}(a_j \mid j \in X)$

7 $a^Y_{max} \longleftarrow \mathbf{max}(a_j \mid j \in Y)$ // *Here* $\mathbf{max}(\emptyset) = \mathbf{min}(\emptyset) = 0$.

8 $\rho^K_{inner} \longleftarrow \mathbf{max}(c, e + a^X_{max}K + \mathbf{max}(0, a^Y_{max})\mathbf{min}(B^Y_{outer}))$

9 $B^X_{outer} \longleftarrow$ `ComputeBounds`$(X, \bigvee_{j \in X} \varphi^{\vec{\kappa}}_j)$

10 **foreach** $\rho^X_{outer} \in B^X_{outer}$ **do**

11 $\quad \rho_{res} \longleftarrow \sum_{K=0}^{\rho^X_{outer}-1} \rho^K_{inner}$

12 $\quad$ *Insert ρ_{res} into B_{res}*

13 **return** B_{res}

counts the number of iterations along Π^X. The nested loop is visited only during iterations along Π^X. To be on the safe side, we transform the inner bound into its maximal value after K iterations along Π^X: We replace the subexpression $\sum_{j \in X} a_j \kappa_j$ by $a^X_{max} K$, where $a^X_{max} = \mathbf{max}(a_j \mid j \in X)$. The subexpressions $a_j \kappa_j$, where $j \in Y$, are still there. Let $\Pi^Y = \{\pi_i \mid i \in Y\}$. The maximal increase of the inner bound in each iteration along Π^Y is $a^Y_{max} = \mathbf{max}(a_j \mid j \in Y)$. The set of upper bounds on the number of iterations along Π^Y is B^Y_{outer} declared at line 4. If $a^Y_{max} < 0$, i.e. each iteration along Π^Y decreases the inner bound, we must transform the inner bound as if there was no iteration along Π^Y to obtain the maximal value. Therefore, we replace the subexpression $\sum_{j \in Y} a_j \kappa_j$ by $\mathbf{max}(0, a^Y_{max})\mathbf{min}(B^Y_{outer})$. In this way we get the improved inner bound ρ^K_{inner}, that contains only one path counter K.

At line 9, we compute the bounds on the number of iterations along Π^X. Note that in the first iteration $K = 0$ and it is increased by 1 after each iteration along Π^X. Hence if ρ^X_{outer} is a (non-zero) bound on the number of iterations along Π^X, $K = \rho^X_{outer} - 1$ in the last iteration. Thus the resulting overall bound is $\rho_{res} = \sum_{K=0}^{\rho^X_{outer}-1} \rho^K_{inner}$. Note that if $\rho^X_{outer} = 0$, the inner loop cannot be visited and $\sum_{K=0}^{-1} = 0$ is a correct bound for t.

Running Example 2.3.6. *In Bubble Sort, the transition t_3 is a part of the nested loop. The bound on the number of executions of t_3 during one iteration of the outer loop (Figure 2.6) is $\underline{n}' - \underline{i}' - 1$ (see Running Example 2.3.5). We know that after κ_1 iterations of the only loop path in the outer loop, the variable* `i` *has value κ_1 and* `n` *has value $\underline{n}$. Hence the bound on the number of executions of t_3 in the $(\kappa_1 + 1)$-th iteration of the outer loop is $\rho^{\kappa_1}_{inner} = \underline{n} - \kappa_1 - 1$. Moreover $X = \{1\}$ and $\varphi^{\kappa_1}_1 \equiv \kappa_1 < \underline{n} - 1$. We transform $\rho^{\kappa_1}_{inner}$ to the form $\mathbf{max}(0, \underline{n} - 1 + (-1) \cdot \kappa_1)$, so $c = 0, e = \underline{n} - 1$, and $a_1 = -1$. Because $Y = \emptyset$, $a^Y_{max} = 0$ and $B^Y_{outer} = \emptyset$. Moreover $a^X_{max} = -1$, so we get $\rho^K_{inner} = \mathbf{max}(0, \underline{n} - 1 - K)$. Further, we compute $B^X_{outer} = \{\mathbf{max}(0, \underline{n} - 1)\}$. Hence the result is $\sum_{K=0}^{\mathbf{max}(-1, \underline{n}-2)} \mathbf{max}(0, \underline{n} - K - 1)$. Suppose the number of iterations of the outer loop is greater than 0 (which means $\underline{n} \geq 2$). Then $\underline{n} - K - 1$ is always greater or equal 0, because the biggest size of K is $\underline{n} - 2$. Hence we can simplify the sum to $\sum_{K=0}^{\underline{n}-2} \underline{n} - K - 1$. After applying the well known formula for the sum of arithmetic progression, we get $\rho_{res} = \frac{(\underline{n}-0-1+\underline{n}-\underline{n}+2-1)\cdot(\underline{n}-1)}{2} = \frac{\underline{n}\cdot(\underline{n}-1)}{2}$. After adding the possibility of zero iterations ($\underline{n} < 2$), we get the resulting bound $\mathbf{ite}(\underline{n} < 2, 0, \frac{\underline{n}\cdot(\underline{n}-1)}{2})$. Because there is no other outer bound from which we could infer the overall bound, the resulting set of bounds is $\{\mathbf{ite}(\underline{n} < 2, 0, \frac{\underline{n}\cdot(\underline{n}-1)}{2})\}$.*

Let us explain the algorithm on two more complicated examples. In Fig-

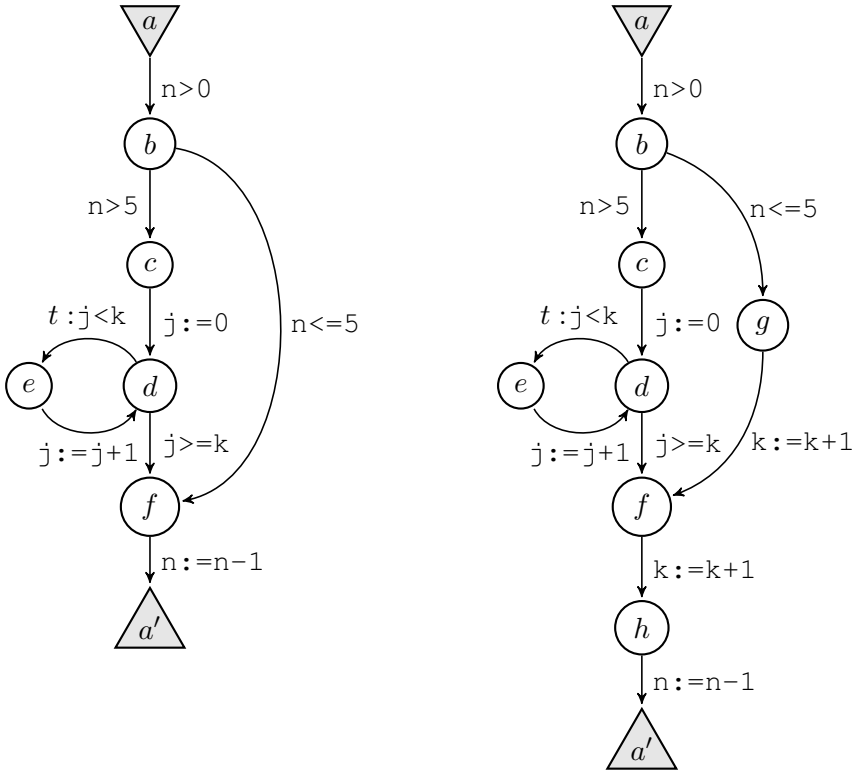

Figure 2.10: Induced flowgraphs by loops with two loop paths and a nested loop.

ure 2.10, there are two flowgraphs induced by loops with a nested loop. In both of them we want to infer bounds for the transition t. We begin with the left one. There are two loop paths: $\pi_1 = abcdfa'$ and $\pi_2 = abfa'$. Assume we entered the loop with the value $\underline{n}$ for `n` and $\underline{k}$ for `k`. After κ_1 iterations of π_1 and κ_2 iterations of π_2, the variable `n` has value $\underline{n} - \kappa_1 - \kappa_2$ and `k` has value $\underline{k}$. The nested loop with t has an entry only on π_1. Thus $X = \{1\}$. The number of executions of t in each iteration of π_1 is $\rho_{inner}^{\kappa_1,\kappa_2} = \mathbf{max}(0, \underline{k})$ and it is unvisited if the loop path π_2 is executed. Assume $\varphi_{in} \equiv true$. From the condition $\varphi_1^{\kappa_1,\kappa_2} \equiv \underline{n} - \kappa_1 - \kappa_2 > 0 \quad \wedge \quad n - \kappa_1 - \kappa_2 > 5$ we infer $B_{outer}^X = \texttt{ComputeBounds}(X, \varphi_1^{\kappa_1,\kappa_2}) = \{\mathbf{max}(0, \underline{n} - 5)\}$. Further, $\rho_{inner}^K = \mathbf{max}(0, \underline{k})$, so the only resulting bound is $\rho_{res} = \sum_{K=0}^{\mathbf{max}(-1,\underline{n}-6)} \mathbf{max}(0, \underline{k})$. Suppose the number of iterations of the outer loop is greater than 0 (i.e. $\underline{n} \geq 6$). Then $\rho_{res} = \sum_{K=0}^{\underline{n}-6} \mathbf{max}(0, \underline{k}) = (\underline{n} - 5) \cdot \mathbf{max}(0, \underline{k})$. After including the possibility of zero iterations, we get the resulting bound $\mathbf{ite}(\underline{n} < 6, 0, (\underline{n} - 6) \cdot \mathbf{max}(0, \underline{k}))$.

The example on the right is slightly more complicated. The flowgraph has two loop paths: $\pi_1 = abcdfha'$ and $\pi_2 = abgfha'$. Assume we entered the loop with the value $\underline{n}$ for `n` and 0 for `k`. After κ_1 iterations of π_1 and κ_2 iterations of π_2, the variable `n` has value $\underline{n} - \kappa_1 - \kappa_2$ and `k` has value $\kappa_1 + 2 \cdot \kappa_2$. The nested loop with t has an entry only on π_1. Thus $X = \{1\}$. The number of executions of t after $\vec{\kappa}$ iterations is bounded by $\rho_{inner}^{\kappa_1,\kappa_2} = \mathbf{max}(0, \kappa_1 + 2 \cdot \kappa_2)$. Thus $Y = \{2\}$. Further, $B_{outer}^Y = \{\mathbf{max}(0, \underline{n})\}$, $a_{max}^Y = 2$, and $a_{max}^X = 1$, so $\rho_{inner}^K = \mathbf{max}(0, K + 2 \cdot \mathbf{min}(\{\mathbf{max}(0, \underline{n})\})) = \mathbf{max}(0, K + 2 \cdot \mathbf{max}(0, \underline{n}))$. Finally, we compute $B_{outer}^X = \{\mathbf{max}(0, \underline{n} - 5)\}$ and get the only resulting bound $\rho_{res} = \sum_{K=0}^{\mathbf{max}(-1,\underline{n}-6)} \mathbf{max}(0, K + 2 \cdot \mathbf{max}(0, \underline{n})) = \mathbf{ite}(\underline{n} < 6, 0, \sum_{K=0}^{\underline{n}-6} K + 2\underline{n}) = \mathbf{ite}(\underline{n} < 6, 0, \frac{(\underline{n}-5)\cdot(5\underline{n}-6)}{2})$. Note that we could not avoid some over-approximation here: in fact, π_2 cannot influence the number of executions of t, because the nested loop is entered only during the first $\underline{n} - 5$ iterations. Thus the correct bound is $\sum_{K=0}^{\mathbf{max}(-1,\underline{n}-6)} K = \mathbf{ite}(\underline{n} < 6, 0, \frac{(\underline{n}-6)\cdot(\underline{n}-5)}{2}$.

2.3.5 Computation of Bounds

The procedure `ComputeBounds` infers bounds on the number of iterations of given loop paths out of the necessary condition, which must hold in each iteration of any of the loop paths. The input is the set of indexes of these loop paths and the condition. Output is the set of inferred bounds.

Let $\Pi = \{\pi_i \mid i \in I\}$ be the set of loop paths with indexes from I. At first, we check if it is possible to iterate along Π at least once. Before the first

iteration along Π each path counter $\kappa_i, i \in I$ equals 0. Let φ' be the formula, which we gain by substituting each occurrence of $\kappa_i, i \in I$ in φ by 0. If φ' is not satisfiable, then the number of iterations along Π must be 0. In fact, the number of iterations along Π must be 0 if the number of iterations along any superset $\Pi' \supseteq \Pi$ is 0. Testing the case of 0 iterations of a proper superset $\Pi' \supset \Pi$ of Π requires including the conditions for all loop paths to the input of `ComputeBounds`. We will omit it here, for simplicity. Let us have a look at examples a) and b) from Figure 2.11. The loop in the first one has always 0 iterations, while the loop in the second one does not have to have always 0 iterations, but it has a loop path along which no iteration is possible.

Algorithm 5: `ComputeBounds`(I, φ)

Input:

(I, φ) *// a non-empty set of indexes of loop paths; a disjunction of path conditions for them*

Output:

B_{res} *// a set of derived bounds on the number of iterations along the loop paths with indexes in* I

1 **if** $\varphi[\kappa_i/0 \mid i \in I]$ *is not satisfiable* **then**
2 **return** *{0}*
3 Replace all occurrences of $\mathbf{ite}(\sum_{i \in M} > 0, e_1, e_2), M \supseteq I$ in φ by e_1.
4 **if** φ *is not satisfiable* **then**
5 **return** *{1}*
6 $B_{res} \longleftarrow \emptyset$
7 Transform φ to the (possibly most simplified) CNF.
8 **foreach** *clause* ψ *of* φ **do**
9 Try to transform ψ to the form $b_1\kappa_1 + \ldots + b_k\kappa_k < e$, where $b_j > 0$ for all $j \in \{1, \ldots, k\}$, e is a symbolic expression without path counters and there is some b_i for each $i \in I$. If the attempt fails, continue with the next clause at line 8.
10 $b_{min} = \mathbf{min}(b_i \mid i \in I)$
11 **if** $\varphi \implies e \geq 0$ **then**
12 Add $\lceil \frac{e}{b_{min}} \rceil$ into B_{res}.
13 **else**
14 Add $\mathbf{max}(0, \lceil \frac{e}{b_{min}} \rceil)$ into B_{res}.
15 If there are two bounds $\rho_1, \rho_2 \in B_{res}$, for which $\rho_1 \leq \rho_2$ always holds, keep just ρ_1 in B_{res} and delete the other.
16 **return** B_{res}

As we have seen in Subsection 2.3.3, some variables may have value $\mathbf{ite}(\sum_{i\in M}\kappa_i > 0, e_1, e_2)$ for some set of loop path indexes M. This form is not practical for computing the bounds, so we would like to simplify it, if possible. Suppose $M \supseteq I$. At this point of the algorithm, we already know, that there might be more than zero iterations along Π. Let us assume then, there was already exactly one iteration along Π. Then $\sum_{i\in I}\kappa_i > 0$, which implies $\sum_{i\in M}\kappa_i > 0$, which means we can replace every occurrence of $\mathbf{ite}(\sum_{i\in M}\kappa_i > 0, e_1, e_2)$ in φ by e_1. If φ is not satisfiable after the substitution, it means there can be no more than 1 iteration along Π and thus 1 is a correct bound. Let us have a look at the examples c) and d) from Figure 2.11. The number of iterations of the loop in the first one is at most 1. In the second example, the number of iterations following the `else` branch is at most 1. Without this information we would not even know, that the program terminates.

Let us continue at line 6 of the algorithm. To be on the safe side, we initialize B_{res} to $\emptyset$. Then we transform the condition φ into the possibly most simplified conjunctive normal form. Note that each clause of φ must be satisfied in every iteration along Π, so we can infer a bound out of each one separately. For example if $\varphi \equiv \kappa_1 < \underline{n} \ \wedge \ \kappa_1 < \underline{m}$, then we can infer two bounds $\underline{n}$ and $\underline{m}$ on the number of iterations along π_1, i.e. one bound out of each clause.

The only type of clause, from which we are able to infer bound at this point of the analysis, is a size comparison of two symbolic expressions. We try to transform it in such a way, that we have a sum of path counters multiplied by some positive integers on the left side, a symbolic expression without path counters on the right side and a comparison symbol $<$ between them (if we have $l \leq r$, we transform it into $l < r + 1$). Recall that after each iteration along Π, the sum of path counters $\sum_{i\in I}\kappa_i$ is increased by 1 and in the beginning $\sum_{i\in I}\kappa_i = 0$. Hence if we have $\sum_{i\in I}\kappa_i < e$ at the beginning of each iteration and $e \geq 0$, the sum can be increased by 1 at most $\lceil e \rceil$ times. Thus the number of iterations is bounded by $\lceil e \rceil$. If it does not hold that $\varphi \implies e \geq 0$, we have to include the possibility that $e < 0$, so the resulting bound is then $\mathbf{max}(0, \lceil e \rceil)$. To apply this approach, we need to transform the clause to have just $\sum_{i\in I}\kappa_i$ on the left side and a symbolic expression e without path counters on the right. Note that we can subtract any path counter from the left side while keeping the inequation true, because all path counters are non-negative. Thus we get the inequation $\sum_{i\in I} b_{min} \cdot \kappa_i < e$, from which we can infer $\sum_{i\in I} \cdot \kappa_i < \frac{e}{b_{min}}$, because $b_{min} > 0$. In this way we get the resulting bound $\lceil \frac{e}{b_{min}} \rceil$ (resp.

a)
```
x:=-1;
while(x>0)
  x:=x-1
```

c)
```
while (x<y)
  x:=y;
```

e)
```
while(x>0)
  x:=x-k;
```

b)
```
s:=1;
while (x>0)
  if (s==1)
    x:=x-1;
  else
    x:=x+1;
```

d)
```
while (x>0)
  if (s==1)
    x:=x-1;
  else
    x:=x+1;
    s:=1;
```

f)
```
i:=0;
while (i<10)
  if (i<5)
    i:=i+1;
  else
    i:=i+2;
```

Figure 2.11: Challenging cases in the bound computation.

$\mathbf{max}(0, \lceil \frac{e}{b_{min}} \rceil))$.

There are some special cases, which we did not cover in the algorithm, but which our implementation supports. The first one is the case of a clause of the form $\sum_{i \in I} \kappa_i < e + \kappa_j, j \notin I$. It can be seen in the example d) from Figure 2.11: Let us assume π_1 is the loop path with the condition $(s = 1)$ and π_2 the other one. If we compute the bounds just for π_1, we get $\underline{x} - \kappa_1 + \kappa_2 > 0$, which is the same as $\kappa_1 < \underline{x} + \kappa_2$. In this case it is sufficient to compute the bound on the size of κ_2, which is the number of iterations along π_2, so in this case we get $\kappa_1 < \underline{x} + 1$ resulting in the correct bound $\mathbf{max}(0, \underline{x} + 1)$ on the number of iterations along π_1.

Let us stay at the example d). Assume we want to compute the bounds for the whole loop. We know the bounds $\mathbf{max}(0, \underline{x} + 1)$ for π_1 and 1 for π_2, but we are not able to get a clause of the form $\kappa_1 + \kappa_2 < e$. Because in every iteration of the loop we iterate along π_1 or π_2, we can infer the bound for it by adding the bounds for π_1 and π_2 together, getting $\mathbf{max}(0, \underline{x} + 2)$. Thus during the computation of bounds B for Π, we can compute the bounds B' for $\Pi' \subset \Pi$ and B'' for $\Pi'' \subset \Pi$ such that $\Pi' \cup \Pi'' = \Pi, \Pi' \cap \Pi'' = \emptyset$ and insert all $\rho' + \rho''$ such that $\rho' \in B', \rho'' \in B''$ into B.

Another issue are path counters multiplied by variables or other path counters, like in the example e), where the value of x after κ_1 iterations is $\underline{x} - \underline{k} \cdot \kappa_1$. Thus the number of the loop iterations is computed using the condition $\underline{x} - \underline{k} \cdot \kappa_1 > 0 \equiv \underline{k} \cdot \kappa_1 < \underline{x}$. If $\underline{k}$ is positive, the result is $\lceil \frac{\underline{x}}{\underline{k}} \rceil$, but

a)
```
i:=0;
while(i<n)
  i:=i+1;
```

b)
```
i:=0;
while(i<n)
  if(nondet())
    i:=i+1;
  else
    i:=i+2;
```

c)
```
i:=0;
j:=n;
while(i<n)
  i:=i+1;
  j:=j-1;
while (j>0)
  j:=j-1;
```

Figure 2.12: Examples for the computation of memory after a loop.

the program may not terminate if $\underline{k}$ is negative or zero. Note that we would have to extend our definition of bounds to expressions like $\mathbf{ite}(\psi, e_1, \infty)$.

The last special case is depicted in the example f). Let π_1 be the path with the `if` branch and π_2 the other one. We can get the correct bound 5 for π_1, but from the condition $\kappa_1 + 2 \cdot \kappa_2 < 10 \wedge \kappa_1 + 2 \cdot \kappa_2 \geq 5$ we infer just the bound $\lceil \frac{10}{2} \rceil = 5$ for π_2. We get the correct bound $\lceil \frac{10-5}{2} \rceil = 3$ by including the lower bound 5 on the number of iterations along π_1.

2.3.6 Computation of a Memory After a Loop

A challenging task is to infer the correct symbolic values without path counters for variables after a loop. Until now, we kept only values for variables that did not change inside the loop. All other values contain a path counter (see Subsection 2.3.3). The problem is that if we do not know the precise number of iterations of a loop path, we cannot usually substitute any expression for its path counter.

Look at the example a) from Figure 2.12. The value of variable `i` after κ_1 iterations is κ_1. We can infer the upper bound $\mathbf{max}(0, \underline{n})$ for the loop, which is also the lower bound for it. Therefore we can assign the value $\mathbf{max}(0, \underline{n})$ to `i` after the loop. The situation becomes more complicated in the example b). Let us keep aside the problem of extending our approach to function calls. Let π_1 be the loop path following the `if` branch and π_2 the other one. The value of variable `i` after κ_1 iterations of π_1 and κ_2 iterations of π_2 is $\kappa_1 + 2 \cdot \kappa_2$. The precise value of `i` after the loop is tricky, because it can be both $\underline{n}$ or $\underline{n} + 1$. A special reasoning must be added to our method to infer that the lower bound on the size of `i` after the loop is $\mathbf{max}(0, \underline{n})$ and the upper bound is $\mathbf{max}(0, \underline{n} + 1)$. The symbolic execution does not work on intervals, but we can overcome this problem by assigning `i` a new value $\underline{i_l}$

and add the condition $\underline{i_l} \leq \mathbf{max}(0, \underline{n} + 1) \ \wedge \ \underline{i_l} \geq \mathbf{max}(0, \underline{n})$ into the path condition after the loop.

So as for the variables with values of the form $\mathbf{ite}(\sum_{i \in M} \kappa_i > 0, e_1, e_2)$, we have to infer the sufficient condition ψ for iterating along $\Pi = \{\pi_i \mid i \in M\}$ at least once and a sufficient condition ψ' for iterating along Π zero times. Then the resulting value is $\mathbf{ite}(\psi, e_1, \mathbf{ite}(\psi', e_2, \star))$. The situation gets even more complicated if e_1 or e_2 contains a path counter. However, if we include the input path condition φ_{in}, with which we entered the loop, and it holds that $\varphi_{in} \implies \psi$ (resp.$\varphi_{in} \implies \neg\psi$), then all expressions of the form $\mathbf{ite}(\psi, e_1, e_2)$ can be replaced by e_1 (resp. e_2).

We can utilize the knowledge of variable values after a loop in cases of two subsequent loops, where the number of iterations of the second one depends on the size of a variable changed in the first one. Consider the example c) from Figure 2.12. We would not be able to infer the correct bound 0 for the second loop, if we did not compute the value of `j` after the first loop.

As we have seen, there are many ways to improve precision of our approach. We have implemented the algorithm in a prototype tool `Looperman`. Some of the extra methods proposed in the last two subsections are included in our implementation. The results of evaluating `Looperman` on a set of scientific benchmarks are described in Chapter 4.

3 Alternative Approaches

Loop bound analysis is currently an active research area. However, much of the work concerns only with termination proving and all loop bounds are just side products of the analysis. No requirement on precision gives here an opportunity to a greater robustness. There are many termination provers in the literature, such as T2 [4], which replaced the original TERMINATOR project, ARMC [16] (also built upon the idea of TERMINATOR), APROVE [10] (the base for the tool KoAT described below), or KITTeL [9].

Loopus [17] came from the idea of the former tool SPEED [11]. It uses lexicographic combination of ranking functions to infer symbolic bounds for nested loops. Also the tools Rank [2] and KoAT [5] are based on the principle of combination of ranking functions, but KoAT adds methods for computation of the sizes of variables after loops. The tool ABC [3] computes symbolic bounds for nested loops, but it does not treat sequences of loops. PUBS [1], implemented by the research group COSTA, computes symbolic bounds via recurrence relations and uses an input generated from Java bytecode. r-TuBound [14] uses recurrence solving to compute precise bounds even for nested loops, but it is restricted only to some specific loop patterns. There are also some tools based on the abstract interpretation on the interval domain, such as SWEET [8] or AiT [6], but they do not compute symbolic bounds.

Because loop bound computation is a part of the worst case execution time analysis (WCET), it is possible to follow the research via the WCET community [12].

In the next two sections, I explain in more detail methods of two of the tools, Loopus and KoAT, which are similar to our approach and which show good results on the benchmark set used for the evaluation.

3.1 Loopus

Loopus is a tool developed at TU Vienna. It is the first tool, which can derive amortized complexity of the analysed programs. It is implemented as an intraprocedural analysis based on the LLVM [15] compiler framework. In the next subsections, we explain informally the basic idea of the approach. For more details, see [17] or [19].

```
   void main(uint n) {
     int a = n, b = n;
l1:  while (a > 0) {
       a--;
l2:    for (int i = n-1; i > 0; i--)
l3:      if (b > 0 && nondet()) {
           a++; b--;
     } } }
```

Figure 3.1: Example for `Loopus` written in C.

3.1.1 Basic Idea

The program works in four separate steps:

1. program abstraction to Vector Addition System with States (VASS)
2. control flow abstraction
3. ranking function generation
4. bound computation

Let us show all of them on the example from Figure 3.1: We want to compute upper bounds on the number of iterations of both loops. The example is challenging in the way that the variable `a` important for termination of the outer loop is altered in the inner loop. Particularly, the number of executions of the `if` branch in the inner loop influences the number of iterations of the outer loop.

1. Program Abstraction: First, the analysis abstracts the program to the Vector Addition System with States (VASS). It is a directed graph similar to the flowgraph of a program (see 2.1), but the transition labels represent the increase of program variables. Every transition label can be seen as a conjunction of formulas of the form $x' \leq x + d$, meaning that the transition increases the variable `x` by at most d, where d is an integer or a symbol for some constant. Note that x (resp. x') denotes the value of the variable `x` before (resp. after) the transition. The VASS of our example is depicted in Figure 3.2. For the exact definition, see [17].

An important property of the VASS is that all variables are non-negative. Hence, for example, if a transition decrements a variable `x` by 1 and the value of `x` is already 0, we cannot proceed the execution through that transition. Some heuristics used for transforming programs in order to satisfy

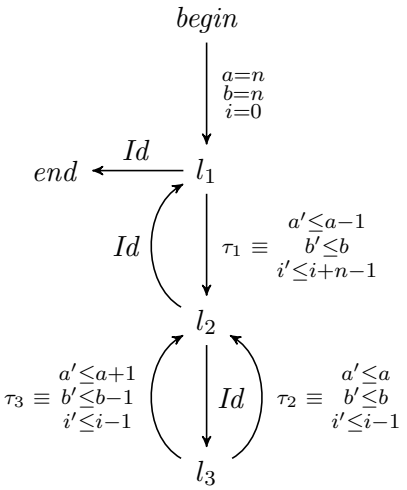

Figure 3.2: VASS for the example from Figure 3.1

.

this condition are proposed in [19].

Assuming that all variables are non-negative, we can use the following rules for rewriting program statements into the VASS transitions:

$\mathtt{x} = \mathtt{x} + c \quad \rightsquigarrow \quad x' \leq x + c$, for c constant
$\mathtt{x} = c \quad \rightsquigarrow \quad x' \leq x + c$, for c constant
$\mathtt{x} = \mathtt{y} \quad \rightsquigarrow \quad x' \leq x + b$, if b is an upper bound on the size of $\mathtt{y}$

The constant c can be an integer or some symbol for a variable, that stays unchanged (like the variable `n` from our example).

2. Control Flow Abstraction: The second phase presents a new abstraction for bound analysis and it is the core of the approach. Part of the VASS, which corresponds to nested loops, is rewritten into a transition system with just one location and several looping transitions. Basically it means that we abstract from the control flow (inner and outer loop hierarchy) and make just one big loop with several independent paths. For that we need to specify the notion of *loop path*. Let us follow the terminology of Section 2.1 (one could notice, that the definition corresponds to our notion of loop path).

Definition 3.1. *A* loop path *is a path, which*

1. *starts and ends at some loop entry l,*

2. *visits only locations inside the loop of l,*
3. *does not visit any location twice except for the start and end location.*

In our example from Figure 3.2, we can see that $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_1$ is a loop path, as well as $l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_2} l_2$. However $l_2 \xrightarrow{Id} l_1 \xrightarrow{\tau_1} l_2$ is not a loop path, because it starts at l_2, which is not the loop entry for the loop containing l_1 (note, that we work only with reducible graphs, which means, that each loop has a unique entry). The path $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_3} l_2 \xrightarrow{Id} l_1$ is not a loop path, because it visits the location l_2 twice and l_2 is not its start and end location.

There are two basic principles for Control Flow Abstraction:

1. We replace every loop path with one transition denoting the overall effect of the path.
2. We merge all nested loops with their parent loop.

The abstraction steps are shown in Figure 3.3: The first picture is the VASS of our example before the abstraction. In the second picture, the loop path $l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_2} l_2$ is replaced with the transition ρ_2 with the effect of τ_2 after Id (which is the same as τ_2). Similarly, we get the transition ρ_3 in the third picture. When we proceed the loop path $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_1$, we see that it contains a nested loop entry l_2 and thus we merge l_1 with l_2 and replace the aforementioned loop path with ρ_1 at the same time. The result for the loop in our example is the following set of transitions:

$$\rho_1 \equiv a' \leq a - 1 \ \wedge \ b' \leq b \ \wedge \ i' \leq i + n - 1$$
$$\rho_2 \equiv a' \leq a \ \wedge \ b' \leq b \ \wedge \ i' \leq i - 1$$
$$\rho_3 \equiv a' \leq a + 1 \ \wedge \ b' \leq b - 1 \ \wedge \ i' \leq i - 1$$

Let us show the idea of the abstraction on a path in the original VASS: Let $\pi = l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_2} l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_3} l_2 \xrightarrow{Id} l_1$. It consists of the loop path $\pi_1 = l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_1$ divided into two parts (one at the beginning and one at the end) and loop paths $\pi_2 = l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_2} l_2$ and $\pi_3 = l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_3} l_2$ inside. In the abstracted model, passing π can be modelled as a sequence of transitions $\rho_1 \circ \rho_2 \circ \rho_3$ (see picture 4. in Figure 3.3), which is the same as passing $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{Id} l_1$ after $l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_2} l_2$ after $l_2 \xrightarrow{Id} l_3 \xrightarrow{\tau_3} l_2$. Because we use only the commutative '$+$' operator on invariant expressions, we can rearrange the transitions of π in any order. Thus the simulation is correct. For the proof, see [17].

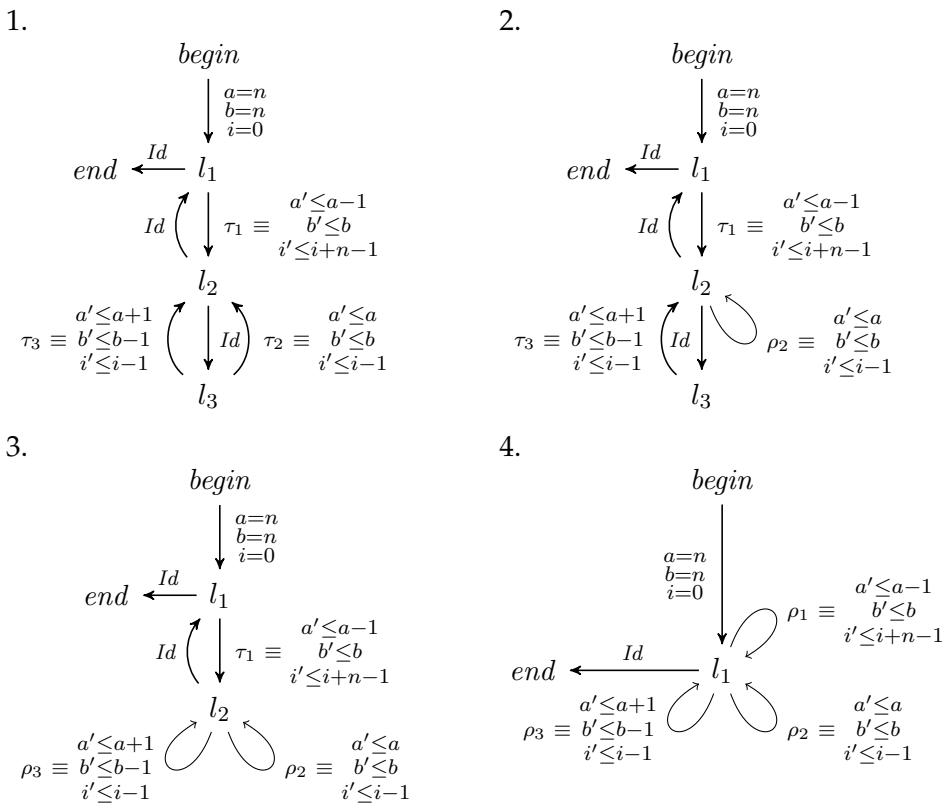

Figure 3.3: Control Flow Abstraction of the VASS from Figure 3.2.

3. Ranking Function Generation: Before the bound computation, transitions from the previous step are ordered according to their ranking functions from the definition below.

Definition 3.2. *We call a variable* $\mathtt{x}$ *a* local ranking function *for a transition* ρ*, if* $\rho \models x' < x$.

A tuple of variables $l = (\mathtt{y}_1, \mathtt{y}_2, \cdots, \mathtt{y}_k)$ *is a* lexicographic ranking function *for a transition system* T *if and only if for each transition* ρ *in* T *there is a ranking function component* $\mathtt{y}_i$ *that is a local ranking function for* ρ *and* $\rho \models y'_j \leq y_j$ *for all* $j < i$.

The algorithm first determines the local ranking functions of the transitions. For our example it is:

$$\begin{array}{l|l} \rho_1 \equiv a' \leq a-1 \ \wedge \ b' \leq b \ \wedge \ i' \leq i+n-1 & \texttt{a} \\ \rho_2 \equiv a' \leq a \ \wedge \ b' \leq b \ \wedge \ i' \leq i-1 & \texttt{i} \\ \rho_3 \equiv a' \leq a+1 \ \wedge \ b' \leq b-1 \ \wedge \ i' \leq i-1 & \texttt{b}, \texttt{i} \end{array}$$

For the next step, the algorithm chooses non-deterministically exactly one local ranking function for each transition. We can see that the transition ρ_3 has two local ranking functions, so let us choose `b` for now. We build the lexicographic ranking function from left to right. On the leftmost position, there must be a variable (local ranking function), that is not increased by any transition. For our example it can be only the local ranking function `b` for the transition ρ_3. We cannot put `i` on the second position and `a` on the third one, because the transition ρ_1, the local ranking of which would be the third component `a`, would increase the second component `i`. Thus we get the lexicographic ranking function $(\texttt{b}, \texttt{a}, \texttt{i})$.

Note that if we chose `i` as a local ranking function for ρ_3, there would be only two candidates for lexicographic ranking functions: $(\texttt{a}, \texttt{i})$ and $(\texttt{i}, \texttt{a})$. Because ρ_3 with the local ranking function `i` increases `a` and ρ_1 with the local ranking function `a` increases `i`, neither of those two candidates is a lexicographic ranking function. In such cases, we say there is a cyclic dependency among the transitions.

4. Bound Computation: We compute the overall bounds for the transitions in the order arising from the previous step. For our example we start with ρ_3. We know that its local ranking function `b` is decreased by one each time ρ_3 is taken and stays unchanged otherwise, which implies that $\texttt{Bound}(\rho_3) = \texttt{InitialValue}(\texttt{b}) = n$. Transition ρ_1 decreases its local ranking function `a` by one and only ρ_3 increases it by one. However, we already know that ρ_3 can be taken at most n times, so altogether it can increase `a` only by n. So $\texttt{Bound}(\rho_1) = \texttt{InitialValue}(\texttt{a}) + \texttt{Bound}(\rho_3) = 2 \cdot n$. Finally, ρ_2 decreases its local ranking function `i` by one and only ρ_1 increases it by $n-1$ (we must assume that $n-1$ is non-negative), which gives $\texttt{Bound}(\rho_2) = \texttt{InitialValue}(\texttt{i}) + \texttt{Bound}(\rho_1) \cdot (n-1) = 2 \cdot n \cdot (n-1)$. Let us return back to Figure 3.1: the `if` branch of the inner loop corresponds to the transition ρ_3 and the `else` branch corresponds to ρ_2, so the number of iterations of the inner loop is bounded from above by the sum of bounds for ρ_2 and ρ_3, which is $2 \cdot n \cdot (n-1) + n$. In the same way we get the bound $2n$ on the number of iterations of the outer loop.

From the bound for ρ_3, we can see how the amortized complexity is achieved. Despite there can be $O(n)$ iterations of ρ_3 during one of $O(n)$ iterations of ρ_1, the overall bound for ρ_3 is in $O(n)$, not $O(n^2)$.

There are some rules to improve the precision of the bound computation. The first one is simple: when the local ranking function is decremented by some $k > 1$, than the resulting bound is divided by k, like in `while (i>0) i:=i-2;` The second rule is more complicated. Consider the following program:

```
x:=n;
while (x>0)
  if (?)
    t:=t+1;
  x:=x-1;
```

The second phase of the analysis returns transitions $\rho_1 = x' \leq x - 1 \wedge t' \leq t + 1$ and $\rho_1 = x' \leq x - 1 \wedge t' \leq t$. The bound n is computed for each one of them, but the bound for the whole loop is n, not $n + n$. To avoid such unnecessary over-approximations, the transitions are merged, when they have the same local ranking functions and decrement them by the same amount. In this case, we would get just one transition $\rho = x' \leq x - 1 \wedge t' \leq t + 1$ (all the other variables are incremented by the maximum value of the two merged transitions). However, this improvement does not work on our main example from Figure 3.1, because the local ranking functions of ρ_3 and ρ_2 are different. The bound for the inner loop is established as $2 \cdot n \cdot (n-1) + n$ while it is in fact just $2 \cdot n \cdot (n-1)$ (i.e. the maximum, not sum of the two bounds). Methods from [11] could be used to solve this problem.

3.1.2 Limitations

Despite the tool shows good results on the benchmarks, it has still some limitations. The first problem is in the program transformation to VASS. Instructions like `x:=x/2` are problematic, because they cannot be easily transformed into the arithmetic with just '+' operator (in some special cases, taking $log(x)$ except of x would be enough). Instructions of the form `x:=y`, where the algorithm fails to find an upper bound on the size of `y` are problematic too. There are also some technical issues, like bitwise operations, external functions, function pointers etc., which are not yet supported. The algorithm can also fail to transform the program into the VASS, where all variables must be non-negative. Other problems can arise during the ranking function generation. In some cases, the tool either does not find a local

```
void main(int x){
   y=0;
   while(x>0){
      x--;
      y=y+x;
   }
   while(y>0){
      for(int z=y-1;z>0;z--);
      y--;
   }
}
```

Figure 3.4: Example for KoAT written in C.

ranking function for some transition, or there is a cyclic dependency among the transitions. However, the latter case appears very sparsely in practice.

3.2 KoAT

KoAT is a prototype tool built on top of a larger AProVE project [10] from RWTH Aachen University. It uses a modular approach with combining size and time analysis to infer the asymptotic complexity of an input program. It can also be used to obtain symbolic loop bounds, but due to the principle of variable size analysis, it usually greatly over-approximates the complexity. The tool KITTeL [9] can translate C programs into transition rewrite systems, which are used as input for KoAT. Like in the previous section, I provide just an informal explanation of how the tool works. For more details, see [5].

3.2.1 Basic Idea

We will work with the program from Figure 3.4. It has two subsequent loops, where the number of iterations of the second one depends on the number of iterations of the first one. If we omitted the first loop, the asymptotic complexity would be $O(y^2)$. However after the first loop, the size of y is asymptotically quadratic with respect to x. Hence the overall asymptotic complexity is $O(x^4)$. This example is challenging in two ways: (1) The number of iterations of the first loop influences the size of the variable y, which is crucial for the number of iterations of the second loop. So we need to combine size and time bound analysis to infer a correct bound for the

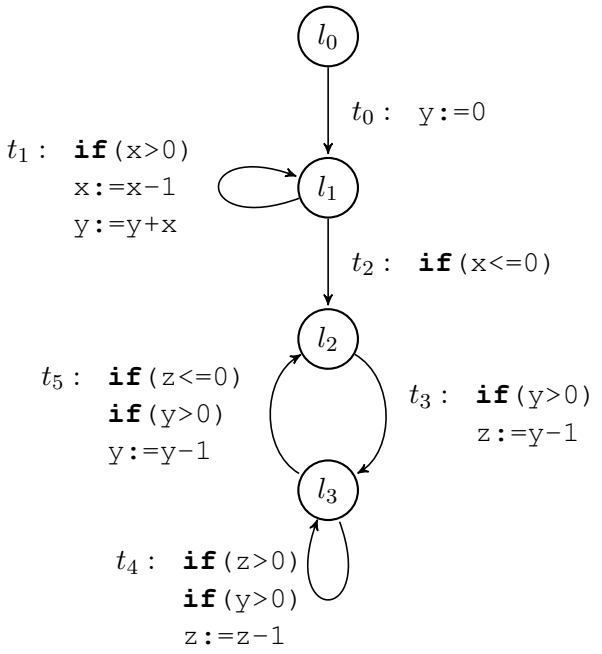

Figure 3.5: Transition rewrite system for the example from Figure 3.4.

second loop. (2) We must handle non-trivial increments of variables, such as `y:=y+x` in the first loop.

First, we transform the input program into the transition rewrite system (see Figure 3.5). As before, nodes are called *locations* and labelled edges are called *transitions*. Compared to our definition of a flowgraph, here the transitions can represent a condition and an assignment at once (which does not make any principal difference).

The algorithm iteratively switches between time bound analysis and size bound analysis. Let us fix the set of variables $\mathcal{V} = \{v_1, \ldots, v_n\}$. We define the set of *(upper) bounds* $\mathcal{C}$ as weakly monotonic functions $\mathbb{Z}^n \to \mathbb{N}_0$ and $?$, where $?(\vec{m}) = \omega$ for all $\vec{m} \in \mathbb{Z}^n$. We have $\omega > n$ for all $n \in \mathbb{N}_0$. Here the condition on the *weak monotonicity* means that an increase of the absolute value of any input variable implies an increase of the bound.

There are two important functions for time and size bound analysis:

- *runtime approximation* $\mathcal{R} : \mathcal{T} \to \mathcal{C}$
- *size approximation* $\mathcal{S} : RV \to \mathcal{C}$

Here $\mathcal{T}$ is the set of transitions and RV is the set of *result variables* defined

by $RV = \{(t, v') \mid t \in \mathcal{T}, v \in \mathcal{V}\}$. Roughly speaking, if the analysed program is executed with input $m_1, \ldots, m_n$, $\mathcal{R}(t) = f$ means, that transition t can be executed at most $f(m_1, \ldots, m_n)$ times during the whole program run and $\mathcal{S}(t, \mathtt{x}') = g$ means, that the size of variable $\mathtt{x}$ after every execution of t is at most $g(m_1, \ldots, m_n)$. On the example, we explain the runtime and size parts of the analysis separately.

Computing Runtime Bounds: Runtime bounds are computed with the use of *polynomial ranking functions* (PRF). A PRF Pol assigns an integer polynomial $Pol(l)$ over the program variables to each location l such that during any execution of any transition $t = (l_1, \tau, l_2)$, at least one of the following holds:

1. t does not increase the PRF, i.e. $\tau \Rightarrow Pol(l_1) \geq Pol(l_2)$
2. t decreases the PRF and the PRF is always positive before the execution of t, i.e. $\tau \Rightarrow Pol(l_1) > Pol(l_2) \ \wedge \ \tau \Rightarrow Pol(l_1) \geq 1$.

Additionally, the set $\mathcal{T}_{\succ}$ of transitions satisfying the second property must be non-empty.

The constraints on a PRF Pol implies, that the transitions from $\mathcal{T}_{\succ}$ can be used only a limited number of times, because they decrease the measure, which is always positive before their execution, and no other transition increases it. Suppose l_0 is the start location. If the program is executed with input variable sizes $m_1, \ldots, m_n$ and $(Pol(l_0))(m_1, \ldots, m_n) \geq 0$, no transition from $t \in \mathcal{T}_{\succ}$ can be used more often than $(Pol(l_0))(m_1, \ldots, m_n)$ times. Consequently $\mathbf{max}(0, Pol(l_0))$ is the runtime bound for t. In the example from Figure 3.5 we could use PRF Pol with $Pol(l) = \mathtt{x}$ for all locations l. We can see that t_1 decreases Pol and it is executed only if $\mathtt{x}{>}0$, so it satisfies the second property and $\mathcal{T}_{\succ} = \{t_1\}$. We can see that $\mathbf{max}(0, Pol(l_0)) = \mathbf{max}(0, \mathtt{x})$ is really the bound for t_1. On the other hand, we can see that t_5 decreases $\mathtt{y}$, but if we chose $Pol'(l) = \mathtt{y}$ for all locations l, transition t_1 could increase it and thus Pol' is not a PRF.

One can see that we cannot simply set $\mathcal{R}(t) = Pol(l_0)$ for some transition t. For example, if $Pol(l_0) = -\mathtt{x}$, the bound would be negative for positive value of $\mathtt{x}$. Neither a PRF $\mathbf{max}(0, \mathtt{x} - \mathtt{y})$ can be used, because it does not satisfy the condition on the weak monotonicity. Therefore the authors introduce the function $[Pol(l)]$, which results from $Pol(l)$ by replacing all variables and coefficients by their absolute values (e.g., $[\mathtt{x} - 3 \cdot \mathtt{y}] = |\mathtt{x}| + 3 \cdot |\mathtt{y}|$). Now we can safely set $\mathcal{R}(t) = [Pol(l_0)]$.

The basic methods for finding PRFs only succeed on simple examples. It often fails for programs with non-linear runtime. The problem is that the

	t_0	t_1	t_2	t_3	t_4	t_5
$\mathcal{R}_0$	?	?	?	?	?	?
$\mathcal{R}_1$	1	?	?	?	?	?
$\mathcal{R}_2$	1	?	1	?	?	?
$\mathcal{R}_3$	1	$\lvert\mathtt{x}\rvert$	1	?	?	?
$\mathcal{R}_4$	1	$\lvert\mathtt{x}\rvert$	1	?	?	$\lvert\mathtt{x}\rvert^2$
$\mathcal{R}_5$	1	$\lvert\mathtt{x}\rvert$	1	$1 + \lvert\mathtt{x}\rvert^2$	?	$\lvert\mathtt{x}\rvert^2$
$\mathcal{R}_6$	1	$\lvert\mathtt{x}\rvert$	1	$1 + \lvert\mathtt{x}\rvert^2$	$\lvert\mathtt{x}\rvert^4 + \lvert\mathtt{x}\rvert^2$	$\lvert\mathtt{x}\rvert^2$

Figure 3.6: Runtime approximations computed during the analysis of the example from Figure 3.5.

PRF considers all transitions at once. Therefore KoAT uses a new modular technique that only considers isolated program parts $\mathcal{T}' \subseteq \mathcal{T}$. On our example, we can see that if $\mathcal{T}' = \{t_3, t_4, t_5\}$, we can use $Pol(l) = \mathtt{y}$ for all locations l as a PRF.

The algorithm starts with the runtime bound ? for all transitions. It iteratively improves the runtime approximation and if no more PRF is found, it deletes transitions, which do not have bound ? and continues the analysis on the rest.

For our example we start with $\mathcal{T}_0 = \mathcal{T}$ and $\mathcal{R}_0(t) =?$ for all t. The runtime approximations computed during the analysis are stated in Figure 3.6. The algorithm finds a PRF $Pol_0(l_0) = 1$ and $Pol_0(l) = 0$ for all other locations l. $\mathcal{T}_\succ = \{t_0\}$, hence $\mathcal{R}_1(t_0) = [Pol_0(l_0)] = 1$ and $\mathcal{R}_1(t) = \mathcal{R}_0(t)$ otherwise. Another PRF is $Pol_1(l_0) = Pol_1(l_1) = 1, Pol_1(l_2) = Pol_1(l_3) = 0$ with $\mathcal{T}_\succ = \{t_2\}$, which results into $\mathcal{R}_2(t_2) = [Pol_1(l_0)] = 1$ and $\mathcal{R}_2(t) = \mathcal{R}_1(t)$ otherwise. The last PRF for $\mathcal{T}_0$ is $Pol_3(l) = \mathtt{x}$ for all l, where $\mathcal{T}_\succ = \{t_1\}$. Thus $\mathcal{R}_3(t_1) = [Pol_3(l_0)] = \lvert\mathtt{x}\rvert$ and $\mathcal{R}_3(t) = \mathcal{R}_1(t)$ otherwise.

At this moment, there is no more trivial (linear or constant) polynomial ranking function, which could be found, so we continue with a submodule $\mathcal{T}_1 = \{t_3, t_4, t_5\}$. The algorithm finds PRF $Pol_4(l_2) = Pol_4(l_3) = \mathtt{y}$ with $\mathcal{T}_\succ = \{t_5\}$. Now $Pol_4(l_0)$ is not defined, so we can infer only a local bound for the module $\mathcal{T}_1$. The only entry to $\mathcal{T}_1$ is l_2, through transition t_2. So the local bound for t_5 is $[Pol_4(l_2)] = \lvert\mathtt{y}\rvert$. At this moment, we have $\mathcal{S}(t_2, \mathtt{y}') = \lvert\mathtt{x}\rvert^2$ from the size part of the analysis (explained later). Moreover, we know that we can visit $\mathcal{T}_1$ only through t_2 and t_2 can be executed only $\mathcal{R}_3(t_2) = 1$ times. The overall runtime approximation for t_5 is then $\mathcal{R}_4(t_5) = 1 \cdot \lvert\mathtt{x}\rvert^2 = \lvert\mathtt{x}\rvert^2$. Again, $\mathcal{R}_4(t) = \mathcal{R}_3(t)$ for the other transitions.

We continue with $\mathcal{T}_2 = \{t_3, t_4\}$. The entry location is again l_2. We find

PRF $Pol_5(l_2) = 1, Pol_5(l_3) = 0$ with $\mathcal{T}_\succ = \{t_3\}$. Hence the local runtime bound for t_3 is 1. It does not contain any variable, so we do not have to consider size approximations. However, this time $\mathcal{T}_2$ is reachable through two transitions, t_2 and t_5. Because $\mathcal{R}_4(t_2) = 1$ and $\mathcal{R}_4(t_5) = |\mathtt{x}|^2$, we infer $\mathcal{R}_5(t_3) = (1 + |\mathtt{x}|^2) \cdot 1 = 1 + |\mathtt{x}|^2$ and $\mathcal{R}_5(t) = \mathcal{R}_4(t)$ for the rest. Note that if there was more than one entry location, we would compute the runtime approximations separately and sum them in the end.

We cannot use PRF $Pol_5(l_2) = Pol_5(l_3) = \mathtt{z}$ for $\mathcal{T}_2$, because it is not clear, whether t_3 increments or decrements $\mathtt{z}$. Hence, we have to work with $\mathcal{T}_3 = \{t_4\}$ and $Pol_5(l_3) = \mathtt{z}$ with $\mathcal{T}_\succ = \{t_4\}$. The entry is l_3 via the transition t_3. The size approximation $\mathcal{S}(t_3, \mathtt{z}') = \mathtt{x}^2$ is available at the moment. Thus the final runtime approximation is $\mathcal{R}_6(t_4) = \mathcal{R}_5(t_3) \cdot [Pol_5(l_3)[\mathtt{z}/\mathcal{S}(t_3, \mathtt{z})]] = (|\mathtt{x}|^2+1) \cdot |\mathtt{x}|^2 = |\mathtt{x}|^4 + |\mathtt{x}|^2$. As usual, $\mathcal{R}_6(t) = \mathcal{R}_5(t)$ for the other transitions. Because there are no more transitions with bound $?$, $\mathcal{R}_6$ is the result runtime approximation.

Computing Size Bounds: We could see that for a successful runtime analysis we sometimes need size bounds. To find them, we first infer *local size bounds* $\mathcal{S}_l$, that approximate the effect of a single transition on the sizes of variables. More precisely, $\mathcal{S}_l(t, v')$ describes how the size of the post-variable v' is related to the sizes of pre-variables of the transition t. So in our example we have $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + |\mathtt{x}|$. The absolute values here are again for the reason of weak monotonicity. We could also infer $\mathcal{S}_l(t_1, \mathtt{x}') = \mathbf{max}(|\mathtt{x}| - 1, 0)$, but in this case we use a simpler expression $\mathcal{S}_l(t_1, \mathtt{x}') = |\mathtt{x}|$, which is also a safe size approximation, because of the condition `x>0`.

With the local size bounds for each result variable we proceed to the next step, which is a construction of a *result variable graph* (RVG). Its nodes are the result variables and there is an edge from a result variable (t_1, v_1') to (t_2, v_2') if t_1 can be used directly before t_2 and the variable v_1 appears in the local size bound of $\mathcal{S}_l(t_2, v_2')$. The RVG for our example is depicted in Figure 3.7 (note that for better understanding, the nodes here are labelled with $\mathcal{S}_l(t_1, \mathtt{x}') \geq (t_1, \mathtt{x}')$). For example, the result variable $(t_1, \mathtt{y}')$ has four predecessors: $(t_0, \mathtt{y}')$ and $(t_0, \mathtt{x}')$, because t_0 can directly precede t_1 (in some program run) and both $\mathtt{x}$ and $\mathtt{y}$ appear in the local size bound $|\mathtt{y}| + |\mathtt{x}|$, and $(t_1, \mathtt{y}')$ together with $(t_1, \mathtt{x}')$, because t_1 can also directly precede itself and both $\mathtt{x}$ and $\mathtt{y}$ appear in the bound. Another interesting node is $(t_3, \mathtt{z}')$. Its local bound does not contain the variable $\mathtt{z}$ and thus it is connected only to nodes with the variable $\mathtt{y}$. As we can observe from the graph, the initial value of $\mathtt{z}$ is thus unimportant for transitions t_3, t_4, and t_5.

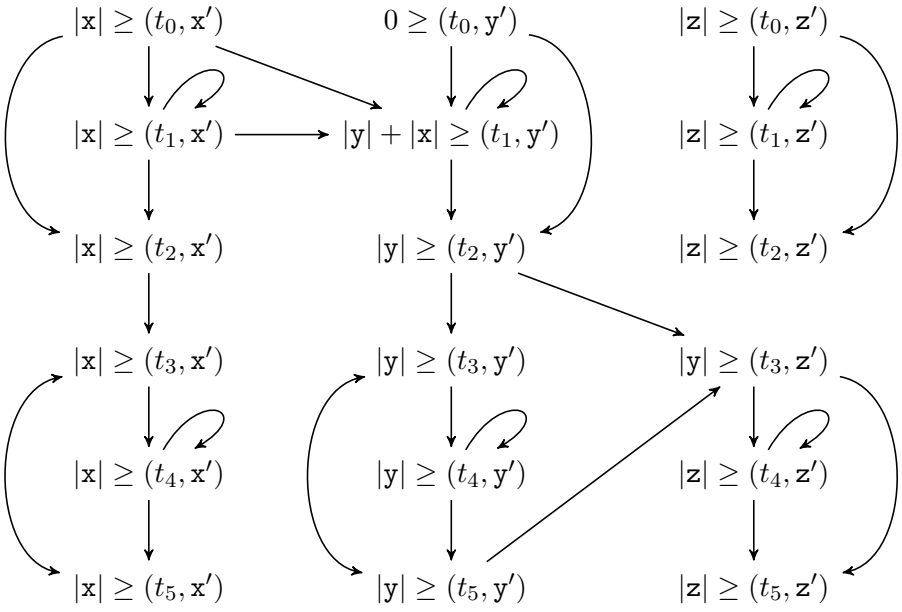

Figure 3.7: The RVG for the example from Figure 3.5.

After the construction of the RVG, we compute the global size bounds. Each strongly connected component (SCC) represents a set of result variables, which may influence each other. Recall that SCC is a maximal subgraph with a path from each node to every other node. We call an SCC *trivial* if it consists of only one node without a self-loop. We treat trivial and non-trivial SCCs differently. The analysis starts with $\mathcal{S}(\alpha) =?$ for every result variable α. We assume that the start location is not a loop entry, so (t_0, v') forms always a trivial SCC and its local bound is also the global bound. Hence, for our example $\mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|, \mathcal{S}(t_0, \mathtt{y}') = 0$ and $\mathcal{S}(t_0, \mathtt{z}') = |\mathtt{z}|$. Consider another trivial SCC $(t_3, \mathtt{z}')$. The local bound contains only $\mathtt{y}$, so we substitute it by the maximal possible value of $\mathtt{y}$, which can appear before executing the transition t_3. From the RVG we see that there are two incoming edges from the nodes with variable $\mathtt{y}$: $|t_2, \mathtt{y}'|$ and $|t_5, \mathtt{y}'|$. Thus to infer the global size bound for $(t_3, \mathtt{z}')$, we need values $\mathcal{S}(t_2, \mathtt{y}')$ and $\mathcal{S}(t_5, \mathtt{y}')$, which we do not have yet. Hence we must proceed first some of the non-trivial SCCs.

Each non-trivial SCC corresponds to a loop and thus each of its local changes can be applied several times. We classify the result variables α from the non-trivial SCC depending on their local size bounds to the following

classes:

1. α is an "equality": α is not larger than its pre-variables or a constant.
2. α "adds a constant": some constant value is added to the variable in each iteration.
3. α "adds variables": some other variable values are added to the variable in each iteration.

So for our example, the result variables $(t_1, \mathtt{x}')$ and $(t_1, \mathtt{z}')$ are in the "equality" class. Because t_1 does not change their sizes, the global size bounds are constrained only by the preceding transition t_0. Thus $\mathcal{S}(t_1, \mathtt{x}') = \mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|$ and $\mathcal{S}(t_1, \mathtt{z}') = \mathcal{S}(t_0, \mathtt{z}') = |\mathtt{z}|$.

The result variable $(t_1, \mathtt{y}')$ is in the last class, as the value of variable x is added to y in each iteration. From the runtime analysis we know that the maximal number of executing the looping edge t_1 is $\mathcal{R}(t_1) = |\mathtt{x}|$. The maximal increase of y in each iteration is $\mathbf{max}(\mathcal{S}(t_0, \mathtt{x}'), \mathcal{S}(t_1, \mathtt{x}')) = |\mathtt{x}|$. Hence, the global size bound $\mathcal{S}(t_1, \mathtt{y}')$ equals $\mathcal{S}(t_0, \mathtt{y}') + |\mathtt{x}| \cdot |\mathtt{x}| = 0 + |\mathtt{x}|^2$. Note that if there was a cyclic dependency between $(t_1, \mathtt{x}')$ and $(t_1, \mathtt{y}')$, we would fail to infer a size bound. An example of such a cyclic dependency between variables (taken from [5]) is `while (z>0) { x:=x+y; y:=x; z:=z-1}`, that increase the size of x exponentially. The global bounds for result variables from the second class are inferred analogically to the result variables from the third class.

Let us continue with our example: $(t_2, \mathtt{x}'), (t_2, \mathtt{y}')$ and $(t_2, \mathtt{z}')$ are trivial SCC, so the global bounds are $\mathcal{S}(t_2, \mathtt{x}') = \mathbf{max}(\mathcal{S}(t_0, \mathtt{x}'), \mathcal{S}(t_1, \mathtt{x}')) = \mathbf{max}(|\mathtt{x}|, |\mathtt{x}|) = |\mathtt{x}|$, $\mathcal{S}(t_2, \mathtt{y}') = \mathbf{max}(\mathcal{S}(t_0, \mathtt{y}'), \mathcal{S}(t_1, \mathtt{y}')) = \mathbf{max}(0, |\mathtt{x}|^2) = |\mathtt{x}|^2$ and $\mathcal{S}(t_2, \mathtt{z}') = \mathbf{max}(\mathcal{S}(t_0, \mathtt{z}'), \mathcal{S}(t_1, \mathtt{z}')) = \mathbf{max}(|\mathtt{z}|, |\mathtt{z}|) = |\mathtt{z}|$.

Next, we have SCC $\{(t_3, \mathtt{x}'), (t_4, \mathtt{x}'), (t_5, \mathtt{x}')\}$. The method allows a cyclic dependency between result variables with the same variable, like in this case. Thus, we can find out that all result variables from this SCC are in the first class and compute global size bounds $\mathcal{S}(t_3, \mathtt{x}') = \mathcal{S}(t_4, \mathtt{x}') = \mathcal{S}(t_5, \mathtt{x}') = \mathcal{S}(t_2, \mathtt{x}') = |\mathtt{x}|$. In the same way we get $\mathcal{S}(t_3, \mathtt{y}') = \mathcal{S}(t_4, \mathtt{y}') = \mathcal{S}(t_5, \mathtt{y}') = |\mathtt{x}|^2$. The only remaining nodes are the result variables with z. $(t_3, \mathtt{z}')$ forms a trivial SCC, so its global bound is $\mathcal{S}(t_3, \mathtt{x}') = \mathbf{max}(\mathcal{S}(t_2, \mathtt{y}'), \mathcal{S}|t_5, \mathtt{y}'|) = |\mathtt{x}|^2$. $(t_4, \mathtt{z}')$ forms a non-trivial SCC and it is in the first class. Thus $\mathcal{S}(t_4, \mathtt{z}') = \mathcal{S}(t_3, \mathtt{z}') = |\mathtt{x}|^2$. Finally, $(t_5, \mathtt{z}')$ forms a trivial SCC and thus $\mathcal{S}(t_5, \mathtt{z}') = \mathbf{max}(\mathcal{S}(t_3, \mathtt{z}'), \mathcal{S}(t_4, \mathtt{z}')) = |\mathtt{x}|^2$.

The overall procedure combines the runtime and size analysis as long as there is any improvement of the bounds. As each step improves the runtime bounds, we can stop the procedure at any time and get correct bounds (however, some of them may be '?').

3.2.2 Limitations

One limitation is that the method only generates polynomial bounds. For an exponential and logarithmic complexity bounds, ranking functions like $log_a(v)$ could be added. Another limitation is that the method allows only certain forms of local size bounds in non-trivial SCCs. For example, assignments like `x:=2*x` inside a loop are not currently handled, but the procedure could be extended in this way too. The method also over-approximates the sizes of variables, which are both incremented and decremented in the same loop. The main problem is that we are restricted only to weakly monotonic bounds, which causes big over-approximations (for example, bounds like $\mathtt{x} - \mathtt{y}$ are transformed into $|\mathtt{x}| + |\mathtt{y}|$). Due to all these imprecisions, the approach sometimes infers bounds, that are asymptotically larger than necessary.

4 Experimental Evaluation

4.1 Implementation

We have implemented our algorithm from Chapter 2 as a part of the symbolic execution tool suite `Bugst`[1], which can work with input programs in LLVM format [15]. Our tool `Looperman` computes symbolic upper bounds on the number of iterations of program loops as well as bounds on the number of executions of any program location. At the moment, it performs only intraprocedural analysis. We use the Z3 SMT solver [7] for checking satisfiability of conditions and simplifying expressions. We have implemented our own inequation solver for computing the bounds. A guide for running the analysis on C programs is given in Appendix B.

4.2 Experimental Results

We have compared `Looperman` against the tools `Loopus` [17], `KoAT` [5], `PUBS` [1], and `Rank` [2] on 199 benchmarks from the scientific benchmark suite used to evaluate `KoAT` and `Loopus`.[2] Because `KoAT` expects an internal representation of programs as input, we used a translation of these benchmarks to C programs provided by the authors of `Loopus`.[3] We excluded the benchmarks used to evaluate the tool `T2` [4], because they are specialized in termination proving rather than loop bound analysis. We also excluded the functional `RAML` programs and programs with recursive function calls, because our tool does not support them. We took the evaluation results for `KoAT`, `PUBS`, and `Rank` from the report about the tool `KoAT`.[4] For running the benchmarks on `Loopus`, we used the version of `Loopus` provided online by the authors.[5] The results of evaluating the tools on the set of 199 benchmarks are stated in Table 4.2. Because some of the tools do not output loop bounds, we compared only asymptotic complexities inferred by the tools (for example, we take the bound $\mathbf{max}(0, \underline{a}^2 + \underline{b} - 1)$ as $O(n^2)$). Columns 2-5 state the number of programs, for which the respective tool

(2) found an asymptotically correct bound,
(3) failed to compute a bound,

[1] http://sourceforge.net/projects/bugst
[2] http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity
[3] http://forsyte.at/static/people/sinn/loopus/CAV14
[4] http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity/report.html
[5] http://forsyte.at/static/people/sinn/loopus

	Correct Bound	Failed	Time-out	Incorrect Bound
Looperman	104	95	8	0
Loopus	163	36	0	0
KoAT	140	57	22	2
PUBS	85	85	1	29
Rank	26	171	0	2

Table 4.1: Analysis results for the benchmark suite.

Looperman vs.	F. / S.	S. / F.	S. / S. (better)	S. (better) / S.	S. / S. (same)
Loopus	61	2	8	0	94
KoAT	54	18	3	3	80
PUBS	28	47	2	3	52
Rank	10	88	0	0	16

Table 4.2: Detailed comparison against the other tools. (In the first row, "F." stands for "failed" and "S." for "Succeeded" and the value for `Looperman` is on the left of the slash.)

(4) did not complete the analysis within 60 seconds,
(5) inferred an asymptotically incorrect bound.

The last column shows that, except of `Loopus`, all of the other tools derived at least one incorrect bound. Especially the tool `PUBS` showed itself unreliable for the loop bound analysis. This is in contrast with results from the literature, where only the number of inferred bounds (correct or incorrect) is measured.

Table 4.2 provides more detailed comparison of `Looperman` against the other tools. The columns 2-6 state the number of programs, on which the respective tool

(2) succeeded to compute a correct bound while `Looperman` failed,
(3) failed to compute a correct bound while `Looperman` succeeded,
(4) inferred asymptotically better result than `Looperman`,
(5) inferred asymptotically worse result than `Looperman`,
(6) inferred asymptotically the same result as `Looperman`.

The last three columns cover the cases, when both `Looperman` and the respective tool succeeded to infer a correct bound within the time limit. We can see from the table that no tool overrun `Looperman` on all benchmarks (and vice versa).

To sum up the results, `Looperman` was significantly better than `PUBS` and `Rank`, but it did not manage to infer more bounds than `Loopus` and

	Correct Bound	Failed	Incorrect Bound
Looperman	191	120	0
Loopus	265	43	3

Table 4.3: Analysis results for the benchmark suite by means of the number of bounded loops.

Loopus succeeded, Looperman failed	81
Loopus failed, Looperman succeeded	7
Both succeeded, Loopus asymptotically better	7
Both succeeded, Looperman asymptotically better	0
Both succeeded, asymptotically the same, Loopus more precise	2
Both succeeded, asymptotically the same, Looperman more precise	33
Both succeeded, the same bounds	142

Table 4.4: Detailed Comparison of `Looperman` against `Loopus`.

`KoAT`. However, as we can see in the next subsections, our tool provides more precise results than `Loopus` in many cases and `KoAT` returns only bounds for the whole program run, so unlike `Looperman`, it is practically useful only for the asymptotic complexity. For the complete table of results, see the electronic attachments described in Appendix A.

4.2.1 Detailed Comparison with Loopus

`Loopus` infers bounds for all loops separately, so we could compare it with our tool on each loop from the benchmarks. The columns 2-4 in Table 4.2.1 state the number of loops, for which the respective tool:

(2) found an asymptotically correct bound,
(3) failed to compute a bound within the time limit,
(4) inferred an asymptotically incorrect bound.

As we can see from the table, `Looperman` was able to correctly bound 191 out of 311 loops contained in the benchmarks. `Loopus` inferred more bounds, but three of them were incorrect. The reason why the tools show better results than in Table 4.2 is that they could fail to find all loop bounds for some program, but they were still able to bound some loops contained in it.

Table 4.2.1 states more detailed comparison on the benchmark loops. It shows that `Loopus` bounded many loops where `Looperman` failed. However, there is a large set of loops, for which both tools inferred asymptotically the same bounds, but our tool was more precise. Bubble Sort is a typi-

a)
```
while (x>0)
  if (a>0)
    a:=a-1;
  else
    a:=n;
    x:=x-1;
```

b)
```
a:=n;
while (a>0)
  while (n>0 && ?)
    n:=n-1;
  a:=a-1;
```

c)
```
 while (x>0)
   while (y<n)
     y:=y+1;
   while (y>0)
     y:=y-1;
   x:=x-1;
```

Figure 4.1: Some representative programs for the comparison of `Looperman` against `Loopus`.

cal example of such programs. The reason why `Looperman` achieves more precise bounds is the technique of computing bounds for nested loops as a sum of arithmetic progression. `Loopus` would in this case just multiply the bound for the outer loop with the maximum number of iterations of the inner loop within one iteration of the outer loop.

Another advantage of `Looperman` is that it can infer more than one bound for each loop. Thus on loops like `while(x>0 && y>0){x:=x-1; y:=y-1}` it provides better result, because both $\mathbf{max}(0, \mathtt{x})$ and $\mathbf{max}(0, \mathtt{y})$ are correct bounds for the loop. The reason why `Loopus` covers only one of the bounds is that it chooses only one local ranking function for each transition. However, it is not a fundamental advantage of our approach, because `Loopus` could be easily extended to provide the results for all possible selections of local ranking functions.

The loop in Figure 4.1a) represents a typical situation, where `Loopus` infers a bound, while `Looperman` fails. The reason is that we are not able to compute the loop summary value for the variable `a`. `Loopus`, on the other hand, first computes the local ranking functions `a` for the `if` branch and `x` for the `else` branch, then it creates the lexicographic function $(\mathtt{x}, \mathtt{a})$ and infers the bounds x for the `else` branch and $x \cdot n + a$ for the `if` branch.

The situation where `Loopus` infers asymptotically more precise bound is shown in Figure 4.1b). `Loopus` computes asymptotically linear bound for the inner loop, while `Looperman` infers a quadratic bound. The reason for that touches the functionalities, which are not covered in the basic algorithm (however, they are proposed in Subsection 2.3.6): the value of the variable `n` after each iteration of the outer loop is not known, but it cannot be more than $\underline{n}$, so it is set to a new symbol $\underline{n_l}$ with a constraint $\underline{n_l} \leq \underline{n}$. Thus `Looperman` over-approximates the bound $\underline{n_l}$ for the inner loop to $\underline{n}$ in every iteration of the outer loop, which results in the asymptotically quadratic

a)
```
while (x>0 && t>0)
  x:=x-t;
```

b)
```
while (x>y)
  x:=x-1;
```

Figure 4.2: Some representative programs for the comparison of `Looperman` against `KoAT`.

bound with respect to $\underline{n}$.

Figure 4.1c) represents a program, where `Looperman` succeeds to find a bound, while `Loopus` fails. `Looperman` infers that the value of the variable y is $\underline{n}$ after the first loop and 0 after the second (as a result of the extra functionalities proposed in Subsection 2.3.6) and thus it can compute bounds for both inner loops. On the other hand, `Loopus` merges the two inner loops and loses the information that the first loop precedes the second one. More precisely, it gains two transitions with a cyclic dependency among them, so it cannot infer any bound for them.

To sum up the results, `Loopus` is more scalable than `Looperman`, but it provides less precise results on a considerably large set of loops. Moreover unlike our tool, `Loopus` does not support computation of bounds on the number of visits of any given program location.

4.2.2 Detailed Comparison with KoAT

A big disadvantage of `KoAT` is that it does not output bounds for each loop separately. Thus we cannot provide the detailed comparison on loops, like in the previous subsection. However, an extension to compute bounds for each loop could easily be implemented. Let us show the general cases, where one tool overruns the other:

When analysing the program in Figure 4.2a), `Looperman` cannot infer any bound from the condition $\underline{x} - \underline{t} \cdot \kappa_1 > 0$. An extension to cope with programs of this type is left for future research. On the other hand, `KoAT` assigns the polynomial ranking function x to each location. Let t be the transition corresponding to the instruction `x:=x-t`. It decreases the polynomial ranking function and $\mathtt{x} \geq 1$ holds every time before t is executed and thus the runtime bound $|\mathtt{x}|$ is inferred for t. This runtime bound is also an upper bound for the number of the loop iterations. At the same moment, this example shows the unnecessary over-approximations in `KoAT`, because if the initial value of x is negative, the number of loop iterations is 0, not $|\mathtt{x}|$.

Figure 4.2b) shows another source of the over-approximations. `KoAT`

Type	1	2	3	4	5	6	7	8	9	X
Count	10	4	6	11	5	1	3	10	8	141

Table 4.5: Number of recognized types of failure on the benchmarks.

finds the correct polynomial ranking function $Pol = \mathtt{x} - \mathtt{y}$, but it sets the bound to $[Pol] = |\mathtt{x}| + |\mathtt{y}|$. `Looperman`, on the other hand, infers the correct bound $\mathbf{max}(0, \underline{x} - \underline{y})$.[6]

4.3 Observations and Future Work

After closer examination of the benchmarks on which our tool failed, we have classified them by the reason of failure to ten classes. A representative for each of the first eight types is given in Figure 4.3. The type for each particular benchmark on which `Looperman` failed, is stated in the complete table of results (see Appendix A). The following overview briefly introduces all the types and suggests the particular idea for future work.

Type 1: Some inequation during the bound computation is linear with respect to all path counters, but it contains a path counter multiplied by a variable. During the analysis of the loop in Figure 4.3a), `Looperman` derives the inequation $\underline{x} - \underline{t} \cdot \kappa_1 > 0$, but it cannot infer bounds from inequations containing path counters multiplied by variables. A possible solution is proposed in Subsection 2.3.5.

Type 2: Some inequation during the bound computation is not linear with respect to some path counter. During the analysis of the loop in Figure 4.3b), `Looperman` infers the inequation $(\underline{x} - \kappa_1) \cdot (\underline{y} - \kappa_1) > 0$. The solution is in improving our inequation solver for the procedure `ComputeBounds` (Subsection 2.3.5) or connecting our algorithm to an external one.

Type 3: One variable is incremented by a value of another variable, which changes in each iteration. We can see in the representative program that the value of x is asymptotically quadratic with respect to y after the loop. The problem could be solved by extending the procedure `ComputeSummary` (Subsection 2.3.3), but only some special cases could be covered, like in Figure 4.3c), where there is a linear growth of y.

[6] Do not take into account the different notation for initial values of variables, like $\underline{x}$ in `Looperman` and x in `KoAT`.

a) Type 1:

```
while (x>0)
  x:=x-t;
```

b) Type 2:

```
while (x*y>0)
  x:=x-1;
  y:=y-1;
```

c) Type 3:

```
y:=0;
x:=0;
while(x<n)
  y:=y+1;
  x:=x+y;
```

d) Type 4:

```
x:=1;
while (x<n)
  x:=2*x;
```

e) Type 5:

```
while (x>0)
  if (a>0)
    a:=a-1;
  else
    a:=n;
    x:=x-1;
```

f) Type 6:

```
while(x>0 && x<n)
  if (a>0)
    x:=x+1;
  else
    x:=x-1;
```

g) Type 7:

```
while(x>0 && x<n)
  if (a>0)
    x:=x+1;
    a:=a-1;
  else
    x:=x-1;
```

h) Type 8:

```
b:=0;
c:=0;
while (c<n)
  tmp:=c;
  c:=b+1;
  b:=tmp;
```

Figure 4.3: Representative programs for the types of failure.

Type 4: There is an exponential growth of some variable in the loop. As before, the problem requires an extension of the loop summary computation.

Type 5: There is a loop with two loop paths π_1 and π_2. `Looperman` is not able to infer any bound for the loop, because it considers the effect of iterating along both loop paths at once when computing the loop summary. However, we could infer a bound ρ_1 for the number of iterations solely on π_1 (without any inference of π_2), as well as ρ_2 for the iterations solely on π_2. This type covers the cases where it holds that the resulting bound is $\rho_1 \cdot \rho_2$. The representative example is identical to the example in Figure 4.1c). This type, as well as the types 6 and 7, requires a different approach to bound computation. One of the possible solutions is proposed in the article *Reachability Bound Problem* [11].

Type 6: This type is similar to type 5, but the difference is that the resulting bound is the maximum of ρ_1 and ρ_2 (not a multiple). In Figure 4.3f) either the `if` branch is executed in each iteration of the loop and the `else` branch is never visited, or the other way round.

Type 7: There is a loop with two loop paths and the first n iterations follow only one path, while the remaining ones follow only the second.

Type 8: This type covers some unusual and interesting programs, where the loop typically requires an ad hoc approach. We can see in Figure 4.3i) that the variable `c` is incremented by 1 every second iteration. We would get this information, if we computed the effect of two subsequent iterations at once. There is no weighty reason to implement any extension to cover these programs, because they are very rare.

Type 9: `Looperman` was not able to compute the bounds before time-out. This problem comes when the input C program contains a lot of branches, because our algorithm is exponential with respect to the branching count. Moreover, the LLVM compiler increases the number of branches even more, but many of them are infeasible. Some heuristics and refining the input could overcome the problem in many cases.

Type X: This type covers programs, which we were not able to classify to the previous types and for which we see no simple way of extending our tool.

Table 4.3 states the number of recognized occurrences of each of the types of failure in the benchmarks. From the types, the best candidates for future extensions are 1, 4, and 9, because they are easy to cope with and frequent at the same time. On the other hand, dealing with some of the loops of types 3, 8 or X could be a challenging research topic for the future.

5 Conclusion

We have presented a new approach to symbolic loop bound computation, which is based on the symbolic execution. The description of the algorithm in Chapter 2 is the core of this thesis. We have implemented the algorithm in a prototype tool `Looperman` built upon the tool suite `Bugst`. The experimental results given in Chapter 4 show that our tool is less robust than two of the four tools used for the comparison, but it provides more precise results in many cases. Moreover, unlike the others, it supports computing bounds on the number of visits of any given program location.

An important part of the thesis is also the description of two alternative approaches in Chapter 3. They are implemented in the tools that proved the best results in the experimental evaluation: `Loopus` and `KoAT`. Introduction of these algorithms allowed us to provide more detailed comparison of `Looperman` against both of the tools. It may also be an inspiration for eventual combination of the techniques.

In the electronic attachments, we provide the tool together with all components needed to run it, the set of benchmarks we used for evaluation, and the detailed table of results on those benchmarks. All the attachments are described in Appendix A and B.

We believe this thesis brings new ideas to the area of symbolic loop bound analysis. We hope that it will bring an inspiration for future research and that one day tools deriving complexity in terms of symbolic bounds will be both robust and precise at the same time.

Bibliography

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis (SAS)*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer Berlin Heidelberg, 2008.

[2] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis (SAS)*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer Berlin Heidelberg, 2010.

[3] Régis Blanc, Thomas. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic Bound Computation for Loops. In EdmundM. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 103–118. Springer Berlin Heidelberg, 2010.

[4] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better Termination Proving through Cooperation. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 413–429. Springer Berlin Heidelberg, 2013.

[5] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin Heidelberg, 2014.

[6] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[7] Leonardo de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963

of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[8] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET Bounds by Abstract Execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*. Austrian Computer Society (OCG), 2011.

[9] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 261–277. Springer Berlin Heidelberg, 2012.

[10] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving Termination of Integer Term Rewriting. In Ralf Treinen, editor, *Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 2009.

[11] Sumit Gulwani and Florian Zuleger. The Reachability-bound Problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 292–304, New York, NY, USA, 2010. ACM.

[12] Reinhard Von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm (eds, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Raimund Kirner, Laura Kovács, Felix Krause, Adrian Prantl, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET Tool Challenge 2011: Report. In *In Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2011.

[13] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[14] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics (PSI)*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer Berlin Heidelberg, 2012.

[15] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Andreas Podelski and Andrey Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In Michael Hanus, editor, *Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin Heidelberg, 2007.

[17] Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761. Springer International Publishing, 2014.

[18] Jan Strejček and Marek Trtík. Abstracting Path Conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 155–165, New York, NY, USA, 2012. ACM.

[19] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In Eran Yahav, editor, *Static Analysis (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer Berlin Heidelberg, 2011.

A Electronic Attachments

There are three folders in the attachment of this thesis:

Benchmarks: This folder contains all the benchmarks used to evaluate our tool as well as the complete table of results "`Benchmark Comparison.xlsx`"[1]. There are several rows for each benchmark. The first one, written in bold font, contains the name of the benchmark, the number of loops the benchmark contains, the number of loops, which `Looperman` successfully bounded, and in each of the next columns, the asymptotic complexity inferred by the respective tool or the symbol "F" standing for "Failed" or "T/O" for time-out. In the next rows, there are the results of `Looperman` and `Loopus` on the respective loop (the loops are ordered by the lines, on which their loop entries appear). For `Looperman`, there are also added the types of errors (see Section 4.3). In the whole table, red colour denotes incorrect bounds, green (resp. brown) colour means that `Looperman` inferred a more precise (resp. less precise) bound than `Loopus`.

Bugst: This folder contains the source code of `Bugst`. The part corresponding to `Looperman` is in subfolders "bounds" and "tools/looperman".

Looperman: This folder contains everything needed to run `Looperman` on Windows (tested on Windows 7 and Windows 8). The manual is given in Appendix B.

[1] The folder contains the same table also in `.csv` and `.xml` format.

B Running Looperman on Windows

Preparing the Input: The input for `Looperman` is a program in lonka format, which is a special flowgraph representation of programs used within `Bugst`. A C program `a.c` is converted to the lonka format by the following steps:

1. Convert `a.c` into `a.llvm` by the tool `Clang`, which is part of the LLVM compiler infrastructure[1]:
   ```
   clang.exe -emit-llvm -g -S a.c -o a.llvm
   ```
2. Convert `a.llvm` into `a.cel.llvm` by the tool `llvm2celllvm`, that is a part of the `Bugst` suite.
   ```
   llvm2celllvm_Win32_Release.exe a.llvm -O a.cel.llvm
   ```
3. Convert `a.cel.llvm` into `a.lonka` by the tool `celllvm2lonka`, that is a part of the `Bugst` suite.
   ```
   celllvm2lonka_Win32_Release.exe a.cel.llvm -O a.lonka
   ```

The easiest way to convert a set of C programs into lonka format is running the script `!buildAll.bat`. The script converts all files with the extension `.c` in the folder `testPrograms` and the resulting lonka files are saved to the same folder.

Options for Running Looperman: The program supports the following options:

`-h` Prints a help message.
`-v` Prints the currect version of the tool.
`-I` ⟨file⟩ A lonka program to be analysed.
`-X` ⟨n⟩ Sets the time-out in seconds. The default value is 60.
`--verbose` Enables commented output.
`--functionCallBounds` If the program is run with this option, bounds are not computed for loops, but for all transitions with function calls. Hence, to compute an upper bound on the number of executions of some line in a program, one has to insert into the C code a call of some function with no effect (like `void bound(){}`) just before that line.

The easiest way to run `Looperman` on a set of lonka files is to run the script `!boundAll.bat`, which calls `Looperman` on each file `a.lonka` from the folder `testPrograms` with the following options:

```
looperman_Win32_Release.exe -I a.lonka -X 60
```

[1] http://llvm.org