BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

http://fit.vut.cz

Project practice
2023/2024

STATIC PERFORMANCE ANALYSIS USING META INFER

Jan Pánek

Abstract
Looper is a static analyzer for determining tight upper bounds on the execution cost of imperative programs. It is developed as a plugin of Meta Infer framework and thereby is implemented in Ocaml and leverages utilities of Infer. Meta infer already provides native program execution cost analyzer COST, however, it's performance on real-world code was not satisfactory at the time. The main idea behind Looper was to provide a different approach for program execution cost analysis which would reflect COST shortcomings. Even tough approach used in Looper seems promising, it has reached it's limits as well. To compete with COST, further extensions which would reflect problematic iteration patterns are required. This report summarizes concepts Looper is build upon and provides motivation behind these concepts. The whole process of cost computation is break down into several steps in order to clearly locate sources of inaccuracies. An outline for future work is attached at the end of the report. All the listed points from the outline are supported by the evaluation of performed precision tests. Additionally, performance comparison with COST is carried out.
Keywords: Static analysis — Complexity analysis — Cost analysis — Difference constraints — Looper

Supplementary Material: Looper GitHub

*xpanek11@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Author of Looper [2] denoted major weaknesses of COST and partially implemented their solution. However, as author admits, some extensions would deserve greater revision, others either lack formal foundation or are not implemented at all. Additionally, Looper as whole is not tested thoroughly and it's behaviour is unpredictable from time to time. Hence, there is space for revision and further development.

The assignment was to study necessary theoretical foundation of Looper in order to locate cases Looper is unable to analyze and to document undefined behaviour. Overall evaluation of Looper and comparison with COST is provided as well. One of the reason for performance tests was to find out whether Looper brings extra information compared to COST and therefore could be used simultaneously with COST.

The backbone of Looper is build upon approach described in [1] and used in LOOPUS analysis tool. Incorporation to Meta Infer framework and extension of concepts from [1] is subject of [2].

The main goal of this report is to break down Looper into more meaningful sub parts which are easier to reason about and to build motivation behind certain steps which are taken during cost computation. This report covers only concepts which are crucial to understand conclusion about future work. These are described to such an extent that it is possible to understand evaluation of test result and proposed modifications. Others are discussed just marginally.

2. How Looper determines cost

Since cycles are the main source of complexity in imperative programs, cycles will be our main interest. Computation of tight upper bound is based on so called *back edge metric* which states upper bound on

how many times a program instruction can be executed. In particular, we are only interested in so called *back jump instructions*, i.e. instructions which cause return of control flow to the loop header. This terminology refers to *jump instructions* used to implemented cycles in assembly languages. To clarify these two concepts two functionally equivalent functions are shown in Figure 1 with highlighted *loop headers* and *back jump instructions*. We assign a cost of 1 to each back jump instruction, others are assigned a cost of 0. Each execution of back jump instruction then increments overall cost by 1.

Next, so called *loop bounds* can be determined. A *loop bound* states how many times a back jump instruction can be executed. Total cost is then defined as a sum of all loop bounds of back edges. In essence, back-edge metric states what is the upper bound on the number of executed elementary operations depending on the magnitude of input. An obvious similarity to asymptotic time complexity can be observed. Determining precise loop bounds seems easy in theory but is very hard in practice. Therefore, in some cases this approach rather aims for close over-approximation of execution cost than precise execution cost computation. This simplification decreases complexity of computation and results are still meaningful in terms of asymptotic time complexity.

Needles to say that cost computation is not performed directly on the source program but rather on a suitable intermediate representation. Used abstraction algorithm presents a challenge itself. Looper leverages LLVM compiler which is part of Infer framework to check lexical, syntactical and semantic aspects of program and to create Infer Control Flow Graph (CFG). Next abstraction step transforms Infer CFG to *labeled transition system* (LTS). Obtained LTS is further transformed into *guarded difference constraint program* and *difference constraint program* (DCP). Listed abstraction steps and the motivation behind them is subject of section 3.

Looper provides full support for *intraprocedural analysis* and partial support for *interprocedural analysis*. Intraprocedural analysis does not take into account side effects and return values of functions called inside currently analyzed function. Hence, no special handling is required for this kind of analysis. On the other hand, interprocedural analysis takes into account function calls and their effect on current program state. It means that order in which functions are analyzed matters. For this purpose Infer framework provides scheduler which determines this order based on call graph. Sink vertices are analyzed first. Subsequently,

```
void select_sort(int arr[], int n){

   int fixed = 0;
   int min = 0;
   int tmp = 0;

   while (fixed < n){ // loop_1_header
      min = fixed;
      tmp = fixed;

      while (tmp < n){ // loop_2_header
         if (arr[min] > arr[tmp]){
            min = tmp;
         }
         tmp++;
      } // back jump instruction

      swap(arr, fixed, min);
      fixed++;
   } // back jump instruction
}
```

```
void select_sort(int arr[], int n){

   int fixed = 0;
   int min = 0;
   int tmp = 0;

   loop_1_header:
   if (fixed >= n) goto loop_1_exit;
      min = fixed;
      tmp = fixed;

      loop_2_header:
      if (tmp >= n) goto loop_2_exit;

         if (arr[min] > arr[tmp]){
            min = tmp;
         }
         tmp++;
         goto loop_2_header; // back jump

      loop_2_exit:

      swap(arr, fixed, min);
      fixed++;
      goto loop_1_header; // back jump

   loop_1_exit:
}
```

Figure 1. Two functionally equivalent implementations of selection_sort. The later implementation emphasizes back jump instructions and loop headers.

scheduler works it's way towards source vertices. By definition, recursive calls are not supported. Other capabilities are described along with limitations of Looper in section 6.

3. Building a way towards a suitable program representation

As was mentioned before, Looper operates on so called DCP which is a result of several abstraction steps. Description of abstraction algorithm in [2] does not provide much motivation behind these transformations.

Therefore, rather than formal description of overall process, I try to build up a meaningfully thought process which can be understood more easily. For completeness, formal definitions are included as well. The thought process is build upon three questions which one should asks at each stage of the abstraction algorithm.

1. What information do we need to preserve?
2. How is the information encoded in current representation of the program?
3. How to express required information in a clearer way?

3.1 Labeled Transition System

What is the information we need to preserve? A number of times a loop body is executed depends solely on a condition in the loop header or conditional jumps inside the loop body. See Figure 1, lets say that if condition of `loop_1_header` is satisfied and the loop body is executed. Statement `goto loop_1_header` causes return of control flow to `loop_1_header` and the condition fulfillment is evaluated again, but this time with a program state which changed as a result of loop body execution. The question is: *How many time is the loop body executed until a program state which does not satisfy condition in loop header is reached?* Naturally, we focus on the statement `fixed++;`, which we know will decrease remaining number of the loop body executions.

Therefore, we are looking for a representation which would clearly encode branching points of a program, corresponding branching conditions and how state of program changes as a result of program execution between any 2 branching point. This is the motivation for construction of LTS, the first abstraction Looper uses.

Definition 3.1 (**Program** [1]).Let Σ be a set of states. A program over Σ is a directed labeled graph $\mathcal{P} = (L, T, l_b, l_e)$, where L is a finite set of locations, $l_b \in L$ is the entry location, $l_e \in L$ is the exit location and $T \subseteq L \times 2^{\Sigma\times\Sigma} \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{\lambda} l_2$ to denote a transition $(l_1, \lambda, l_2) \in T$. We call $\lambda \in 2^{\Sigma\times\Sigma}$ *a transition relation.*

An example function and corresponding LTS are show in Figure 2. Looper obtains LTS from Infer CFG which is automatically generated by Infer framework.

Definition 3.2 (**Infer Control Flow Graph** [2]) Let $\mathcal{C} = (N_\mathcal{C}, E_\mathcal{C}, n_s, n_e)$ be Infer CFG (directed labeled graph), where $N_\mathcal{C}$ is a finite set of nodes, n_s and n_e are the start and exit nodes and $E_\mathcal{C} \subseteq N_\mathcal{C} \times N_\mathcal{C}$ is

```
void select_sort(int arr[], int n){          // l_s

    int fixed = 0;                           // tau_1
    int min = 0;                             // tau_1
    int tmp = 0;                             // tau_1

    while (fixed < n){                       // l_1
        min = fixed;                         // tau_2
        tmp = fixed;                         // tau_2

        while (tmp < n){                     // l_2
            if (arr[min] > arr[fixed]){      // l_3
                min = tmp;                   // tau_5
            }
                                             // l_4
            tmp++;                           // tau_7
        }

        swap(arr, fixed, min);               // tau_4
        fixed++;                             // tau_4
    }
}                                            // l_e
```

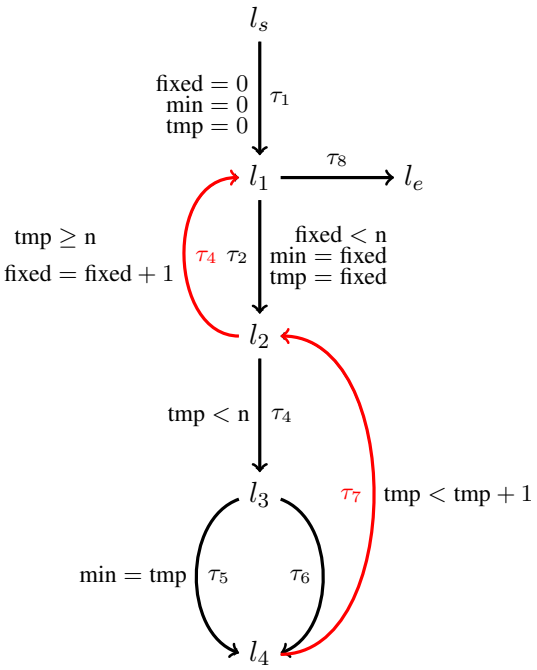

Figure 2. Reference function and corresponding LTS. Note that some transition labels are omitted for the sake of clarity. Back edges are colored red.

a finite set of edges. We write $n_1 \to n_2$ to denote an edge $(n_1, n_2) \in E_\mathcal{C}$.
Let *type(n)* : $N_\mathcal{C} \to$ *{start, exit, prune, join, statement}* be a function which maps the node $n \in N_\mathcal{C}$ to its node type. Additionally, let $\mathcal{I}_S$ be a set of all `SIL` instructions and finally, let $instr(n) : N_\mathcal{C} \to 2^{\mathcal{I}_S}$ be a function which maps the node n to a set of `SIL` instructions contained in this node.

How is the information encoded in the current representation? Infer CFG may be an input for other

kinds of analyzers with different targets. As a result, the structure is very complex in terms of number of nodes presents in the graph and number of node types. Additionally, program instructions are stored in nodes. *How to express required information in clearer way?* As mentioned before, we are only interested in how a program state changes starting from a branching point until another branching point is reached. Thus, it would be sufficient to preserve branching nodes and transform all the nodes on given branch into transition label.

While constructing LTS we mainly leverage two types of nodes, *prune* and *join* nodes. First indicating branching of program control flow, the later indicating merging of 2 or more branches of program control flow. These nodes are used to determine LTS structure, more specifically, program locations and transitions between them. *statement* nodes, on the other hand, contain information about change of program state between 2 branching points. These will be used to label transitions between program locations. This process involves interpretation of `SIL` instructions on according statement nodes and subsequently construction of transition relations. CFG structure and granularity of SIL instructions brings opportunity for further optimizations but also complexity of transition relations construction.

3.2 Construction of Difference Constraint Program

Now, we have a representation of program which emphasizes changes of program state between two branching points. *What information do we need to preserve?* Next, we need to extract information about what limits the number of executions of a loop body and how this limit changes during program execution. This information is contained in conditions of loop headers and conditional jumps in loop bodies. To be able to inspect how this limit changes during program execution we want to express it in a way which would enable it, perhaps, as an expression over program parameters and constants.

How is the information encoded in current representation of the program? First, consider straightforward example in Figure 3. Let loop_reserve be an expression over function parameters and constants. Let increase_reserve() be a function which increase distance from zero of loop_reserve. Let decrease_reserve() be a function which decreases distance from zero of loop_reserve. If the loop_reserve reaches zero, the loop body wont be executed unless the loop_reserve is increased somewhere out of the loop body.

This concept can be generalized for all conditions which contain inequalities. Mentioned loop_reserve can be obtained by a simple transformation of conditions from loop headers and if statements. Every condition of form $a > b$ can be rewritten as $a - b > 0$ where $a - b$ is the reserve. Conditions of form $a \geq b$ can be rewritten as $a - b \geq 0$ or equivalently $a - b + 1 > 0$ where $a - b + 1$ is the reserve. As can be seen in Figure 4, increment of a increases the reserve and decrement of a decreases the reserve. The same applies for b but this time increment of b decreases the reserve and decrement of b increases the reserve. Even tough this step may look intuitive at the first glance, it make is possible to determine how these loop reserves change as a result of function execution.

```
void foo(int loop_reserve){

    for (int i=0;i<10;i++){
        increase_reserve(loop_reserve);
    }

    while (loop_reserve > 0){
        decrease_reserve(loop_reserve);
    }
}
```

Figure 3. Variable loop_reserve limits total number of executions of while loop. The reserve is increased before the loop header is reached for the first time and total number of the loop body executions is increased.

```
void foo(int a, int b){

    for (int i=0;i<10;i++){
        a++;
    }

    while (a > b){ // reserve: a - b > 0
        b++
    }
}
```

Figure 4. Equivalent of example show in Figure 3 but this time mentioned reserve is represented by expression over function parameters $a - b$. The reserve is increased by multiple increments of a. Afterwards the reserve is being decreased by subsequently executing while the loop body where b is incremented.

How to express required information in a clearer way? Approach described in [1] operates on mentioned representation of a program called *difference constraint program* (DCP) which is obtained from LTS by transforming it to a guarded difference constraint program and then to DCP. DCP enhances so called norms (expression over program variables) which are extracted from loop headers and if statements. To model changes of extracted norms it uses *difference constraints*.

Definition 3.3 (**Difference constraints** [1]). A difference constraint over $\mathcal{A}$ is an inequality of form $x \leq y + c$ with $x \in \mathcal{V}$, $y \in \mathcal{A}$, $c \in \mathbb{Z}$. By $DC(\mathcal{A})$ we denote the set of all difference constraints over $\mathcal{A}$.

Definition 3.4 (**Guarded Difference Constraint Program** [1]). A *guarded difference constraint program* (guarded DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta\mathcal{P}_G = (L, E, l_b, l_e)$, where L is a finite set of vertices, $l_b \in L$, $l_e \in L$ and $E \subseteq L \times 2^{\mathcal{V}} \times 2^{\mathcal{DC}(A)} \times L$ is a finite set of edges. We write $l_1 \overset{g,u}{\rightarrow} l_2$ to denote an edge $(l_1, u, g, l_2 \in E)$ labeled by a set of difference constraint $u \in 2^{\mathcal{DC}(\mathcal{A})}$ and guards $g \in 2^{\mathcal{V}}$.

Definition 3.5 (**Regular Difference Constraint Program** [1]). A *difference constraint program* (DCP) is a *guarded DCP* with the finite set of edges E redefined as follows:

$$E \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L,$$

where $u \in 2^{\mathcal{DC}(\mathcal{A})}$ is a set of difference constraints with valuation over natural numbers $\mathbb{N}$.

DCP is obtained from LTS in 4 steps. An example of a DCP is show in Figure 5 and was obtained from LTS shown in 2.

1. **Initial norm selection**: In the first step the initial set of norms is created. We search for loop headers and branching locations, extract conditions of form $a > b$ and $a \geq b$ and transform them into norms. Conditions of form $a > b$ are transformed to $a - b$ and conditions of form $a \geq b$ are transformed to $a - b + 1$. Additionally, only conditions which involve at least 1 variable which is either incremented or decremented in loop body are transformed into norms. In Figure 5, the initial set of norms would consist of n − fixed and n − tmp which were derived from transitions τ_2 and τ_4.
2. **Abstracting transitions**: Initially, all the transitions of DCP contain empty set of *difference constrains*. Subsequently, for each transition τ we symbolically execute τ and generate a new *difference constraint*. Note, that this step can be performed only if all the variables present in norm are defined on according edge. For instance, in Figure 5 a new *difference constraint* n − fixed ≤ n − fixed − 1 on τ_4 would be generated from norm $n - fixed$ by executing fixed = fixed + 1.
3. **Inferring guard**: Guard of a transition is a norm which must be greater than zero in order to execute given transition. Each transition has no guard at the beginning. Subsequently we iterate over all the transitions and try to infer which norms from the set of norms are guards of given transition if any. Looper uses Z3 SMT solver to infer guards. In Figure 5 transition τ_2 is assigned guard n − fixed by checking validity of formula $fixed < n \Rightarrow n - fixed > 0$ by Z3 SMT solver. Transition τ_4 would be assigned guard $n - tmp$. After all the guards are derived we can try to propagate them across the DCP. If g is a guard of all the incoming transitions of program location l and g is not decremented on any of the incoming edges we can infer that g is a guard of all outgoing edges of l.
4. **Creating DCP over Natural Numbers**: Last step is straightforward. Every *difference constraint* $e_1' < e_2 + c$, where e_1, e_2 are norms is transformed depending on c. If $c \geq 0$ (increment) we infer $[e_1]' \leq [e_2] + c$ where $[.] \Leftrightarrow max(., 0)$. For $c < 0$ we infer $[e_1]' \leq [e_2] - 1$ if e_2 is a guard, otherwise we infer $[e_1]' \leq [e_2]$.

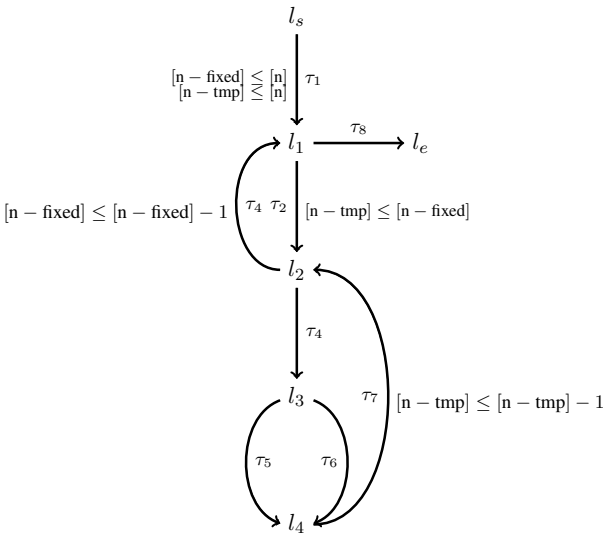

Figure 5. DCP corresponding to function in Figure 1

4. Bound analysis

This sections presents basic form of bound algorithm followed by two extensions build on top. The first extension makes DCP flow sensitive, the later allows more meaningful reasoning about so called reset.

Before we deep dive into the problematic of bound algorithm itself, it is necessary to introduce notion of local bounds. Local bound is a norm which limits number of executions of given transition until another transition is executed. For example in Figure 6 transition τ_1 is assigned a local bound of 1 as it can be executed exactly once. Next, norm tmp − n is a local

bound of τ_7. This transition can be executed only if the value of norm tmp $-$ n is greater than zero. This norm is decremented on every execution of τ_7. Obviously, norm tmp $-$ n approaches zero as a result of subsequent executions of transition τ_7. Note, that norm tmp $-$ n is reset on transition τ_2. Hence, it does not limit total number of executions of τ_7. We can infer that norm tmp $-$ n is a local bound of τ_7. Formally, the local bounds are determined in 3 steps. Let $E' = E$ be a set of transitions without assigned local bound.

1. **Determining loop transitions**: In the first place, *strongly connected components* (SCC) of DCP are computed. All the transitions which do not belong to any SCC are assigned a local bound of 1 and are removed from E'.
2. **Initial local bound mapping**: For every norm e a set of transitions where norm e is decremented $\xi(e) = \{l_1 \xrightarrow{u} l_2 | e' \leq e + c \in u, e < 0\}$ is constructed. Every transition $\tau \in \xi(e)$ is assigned a local bound of e.
3. **Local bound propagation**: For each $\xi(e)$ constructed in previous step we update E' as follows: $E' = E' \setminus \xi(e)$. Then SCCs are computed again with updated E'. By removing these transitions from E' a SCC representing loop may have been destroyed. Therefore, all the transitions which dot not belong to any SCC anymore must have been part of a loop represented by destroyed SCC. Such transitions are assigned a local bound of e.

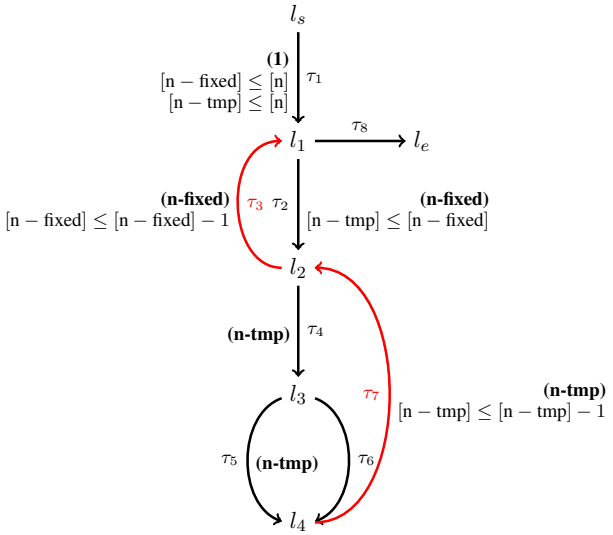

Figure 6. DCP with assigned local bounds and highlighted back edges. Local bounds are depicted above difference constraint in parentheses.

4.1 Basic Algorithm

The core idea is to inspect how often and by how much might the stated local bound change during program execution. Approach described in [1] allows only increments and so called resets of local bounds. Other operators which may increase distance form zero of a local bound such $*, /, <<, \ldots$ are not supported.

The upper bound on execution of a given transition is called *transition bound* (TB). The total execution cost of analyzed function is equal to the sum of total bounds of all back-edges. Cost computation for program represented by DCP from Figure 6 is illustrated in Figure 7.

The transition bound $TB(\tau)$ of a transition τ is defined as follows,

$$\mathrm{TB}(\tau) = \begin{cases} \tau_v, & \text{if } \tau_v \notin \mathcal{V} \\ \mathrm{InctSum}(\tau_v) + \mathrm{RSum}(\tau_v), & \text{else} \end{cases} \quad (1)$$

where τ_v is the local bound of transition τ. TB returns assigned local bound itself if it is a constant as it makes not sense to further inspect how it changes during program. If local bound τ_v is an expression over program variables, the total amount by which it may be increased is defined as $\mathrm{InctSum}(\tau_v) + \mathrm{RtSum}(\tau_v)$. Let $\mathcal{I}(v) = \{(l_1 \xrightarrow{u} l_2, c) \in E \times \mathbb{N} | v' \leq v + c \in u, c > 0\}$ be a set of transition where norm v is increased.

The total amount by which local bound τ_v may increase through increments is defined as follows:

$$\mathrm{InctSum}(\tau_v) = \sum_{(t,c) \in \mathcal{I}(\tau_{\sqsubseteq})} \mathcal{TB}(t) \times c \quad (2)$$

Next, we need to inspect how given local bound may increase through resets.
Let $\mathcal{R}(v) = \{(l_1 \xrightarrow{u} l_2, a, c) \in E \times \mathcal{A} \times \mathbb{N} | v' \leq a + c \in u, c > 0\}$ be a set of transition where norm v is reset. This amount is defined as follows:

$$\mathrm{RSum}(\tau_v) = \sum_{(t,a,c) \in \mathcal{R}(\tau_v)} \mathrm{TB}(t) \times \max(\mathrm{VB}(a) + c, 0) \quad (3)$$

where $VB(a)$ is a *variable bound* which states what is the maximum value of variable a. It is defined as follows:

$$\mathrm{VB}(e) = \begin{cases} e, & \text{if } e \notin \mathcal{V} \\ \mathrm{IncrSum}(e) + \max\limits_{(t,a,c) \in \mathcal{R}(e)} (\mathrm{VB}(a) + c), & \text{else} \end{cases} \quad (4)$$

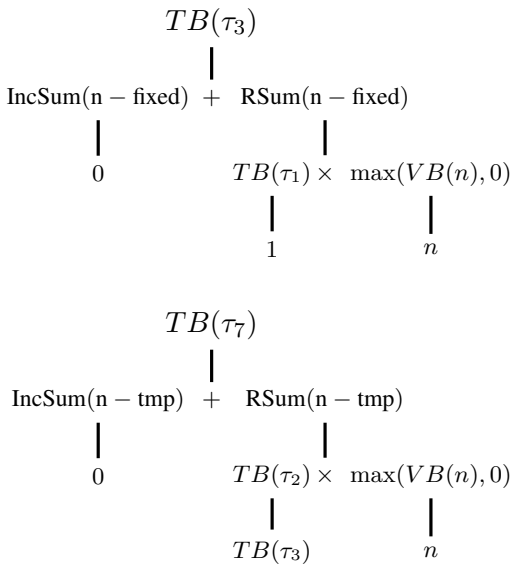

Figure 7. Demonstration of cost computation for function represented by DCP in figure 6. Total cost is calculated as sum of transition bounds of two back edges τ_3 and τ_7. Obtained cost is $TB(\tau_3) + TB(\tau_7) = n + n^2$ which corresponds to $O(n^2)$.

4.2 Adding flow sensitivity

The basic algorithm is *flow-insensitive* by default. This may lead to coarse over approximation. Figure 8 illustrates the problem.

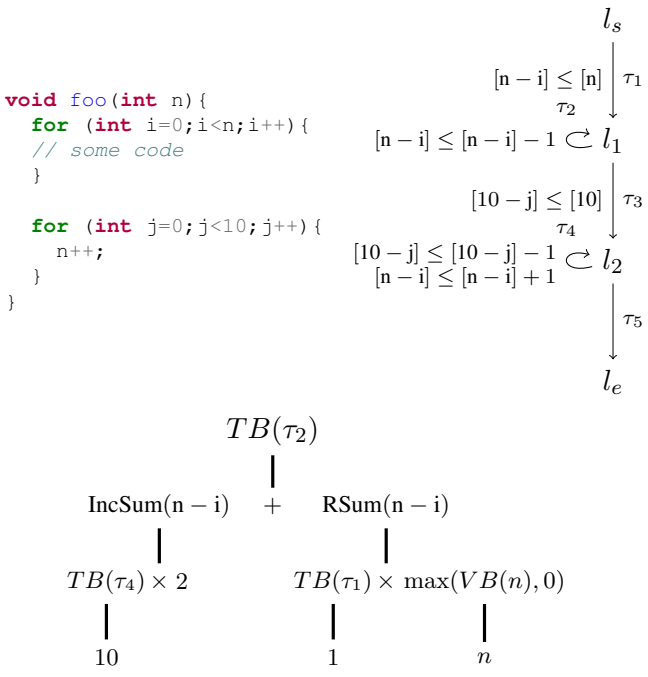

Figure 8. The first for loop is assigned a local bound of $n - i$. The second for loop would increase the local bound if it was executed before the first one. However, basic algorithm does not take this into account obtains cost $n + 20$.

This problem can be solved by quite straightforward pre-processing step which involves renaming of DCP norms based on so called *variable flow graph* (VFG). VFG models how the variables of a program flow from a program location to another. It can be easily obtained from DCP. We first compute SCCs and assign unique number from 0 to k to each of them. All the variables on transitions which belong to a SCC_i is renamed from v to v_i for $i = 0 \dots n$. Variables on transitions which connect two components pointing form SCC_j to SCC_k are renamed as follows. Every DC $x \le y + c$ is transformed to $x_k \le y_j + c$.

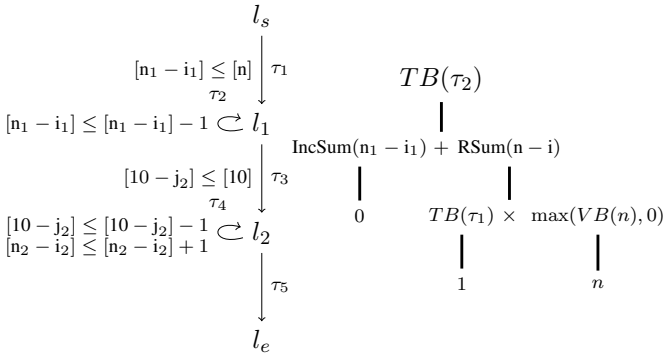

Figure 9. As a result of renaming variables, the second for loop does not affect the local bound $n - i$.

4.3 Reset Chains

The introduced basic version of algorithm handles resets in a way that it assign the greatest possible value which may be propagated through a sequence of resets. However, it does not take into account that some of these sequences of resets may me executed only once. The overapproximation is demonstrated on example in Figure 10 and Figure 11.

This problem can be solved with notion of reset chains. This extensions allows reasoning about sequences of resets. The example in Figure 10 con-

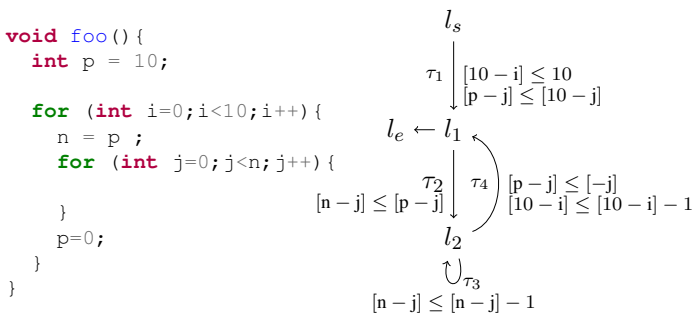

Figure 10. Variable n limits number of executions of inner for loop. It is assigned value 10 during the first iteration of outer for loop and value 0 during remaining 9 iterations of outer for loop. Real cost therefore should be 20. However, the basic algorithm would lead to 110.

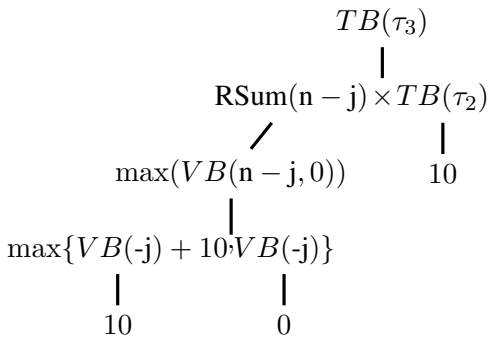

Figure 11. Bound computation for previous example from Figure 10. The overapproximation is caused by $\max\{VB(\text{-j}) + 10, VB(\text{-j})\}$ during reset sum computation.

tains 2 reset chains $k_1 = [10] \xrightarrow{\tau_1} [p] \xrightarrow{\tau_2} [n]$ and $k_2 = [0] \xrightarrow{\tau_4} [p] \xrightarrow{\tau_2} [n]$. Extended version of algorithm now leverages information that value 10 can flow to n exactly once and infers correct cost 20.

5. Comparison with COST

Accuracy tests and comparison with COST from [2] were performed again. Tested subjects are Looper v2 (product of [2]) and COST build from github to date 28.1.2024. Test were performed on 82L5 IdeaPad 5 Pro 16ACH6 with CPU AMD Ryzen 7 5800H with Radeon Graphics (16) @ 3.200GHz and 16 GB of system memory, running Manjaro Linux x86_64.

Table 1. Evaluation results for the accuracy test performed on test-suite from [1] which contains examples from `SPEC CPU`2006 benchmark. Note, that new result are identical with the results from [2].

	Real Bound	Looper	Cost
SingleLinkSimple	n^2	∞	∞
bar	n	$2n$	n^2
foo	$2n$	$2n$	n^2
foobar	$4n$	$5n$	∞
tarjan	$2n$	$2n$	$2n$
twoSCCs	$3n$	$3n$	∞
xnu	$2n$	$2n$	∞
xnuSimple	$2n$	$2n$	∞

Results of the accuracy test from Table 1 prove that there are cases Looper is, in contrast to COST, able to analyze. Note, that this test-suite consists of very limited number of samples which were selected on purpose to prove the statement. Therefore, it cannot be claimed that Looper outperforms COST.

The second test-suite was created by developers of COST analyzer, therefore, it should favor COST. Attached table contains result of Looper and Cost, distinguishing 3 scenarios. A bound is considered to be precise if its equal to the real cost. If big-O is of given bound is equal to the big-O of real cost it is considered to be tight. If neither of previous cases is matched a bound is considered to be imprecise.

Table 2. Evaluation results for the accuracy tests performed on COST test-suite. It originally contained 71 functions. The number of functions was reduced to 65 because it was not possible to assess accuracy of the results of analysis for 6 functions.

	Precise	Tight	Imprecise
COST	6	54	5
Looper	50	5	10

Table 3. Confusion matrix for results from accuracy test performed on COST test-suite.

		Looper		
		Precise	**Tight**	**Imprecise**
COST	**Precise**	6	0	0
	Tight	40	5	9
	Imprecise	4	0	0

Based on the results from Table 2 and Table 3 can be claimed that Looper outperforms COST in terms of cost accuracy. On the other hand, COST leads in the number of correctly analyzed functions in terms of asymptotic time complexity. All the functions which make this difference in performance contain an interation pattern Looper is unable to analyze at this point.

The last test was performed on GNU coreutils. Due to the amount of code it was not possible to annotate each analyzed function with real execution cost. Therefore, this test only distinguishes cases of infinite and not infinite bounds.

Table 4. Evaluation results for test performed on GNU coreutils with total number of functions 2502. The data is based on assumption that analyzed functions should not have an infinite bound.

	Success	Infinite
COST	1934	568
Looper	957	1545

Table 5. The confusion matrix for results of test performed on GNU coreutils. The data is based on assumption that analyzed functions should not have an infinite bound.

		Looper	
		Success	**Infinite**
COST	**Success**	952	982
	Infinite	5	563

Based on results from Table 4 and Table 5 it is obvious that COST is superior in terms of analyzing large scale realworld code bases. Additionally, the set of function Looper is able to analyze is largely a subset of set of functions COST is able to analyze. It means that parallel usage of Looper and COST is not meaningful.

6. Conclusion

Study of theory covered in this report was necessary to understand current limitations of Looper and set objectives for future development.

The poor performance of Looper on real-world codebases is mainly cause by invalid construction of DCP. This involves inability to obtain correct structure LTS or to derive difference constrain due to unsupported operators. The backbone of approach described in [1] seems to work very well and therefore there is no need for major changes. However, some iteration patterns pose a problem for Looper. This may be solved by localization of these patterns and their rearrangement on LTS level to another pattern which Looper can analyze. Other solution would involve intervention to the bound algorithm.

The best way to find other iteration patterns and program structures Looper is unable to analyze is to run Looper on real-world codebases and inspect the results. This requires automated evaluation. Extension of current test sets and completion of script for evaluation will be subject of further development. Additionally, a basic test set which covers cases Looper should be able to analyze will be created. This set will be used to test whether implementation of new feature does not decrease overall performance.

So far, study of theory behind Looper was the main interest. Now it is necessary to delve into the code itself. The documentation is either not finished or not present at all. The goal will be to study implementation of theoretical concepts running under the hood of Looper, review the implementation and finish the documentation if needed.

Even tough intraprocedural analysis was not subject of this report it will also be subject of further study. But intraprocedural analysis will not provide meaningful results unless interprocedural analysis works properly. Therefore, the main intereset is interprocedural analysis for now.

Acknowledgements

I would like to thank my supervisor prof. Ing. Tomáš Vojnar, Ph.D. for his support during study phase and mediation of communication with the author of Looper Ing. Ondřej Pavela.

References

[1] MORIRTZ, S. *Automated complexity analysis for imperative programs*. 2016. Dissertation. Vienna university of technology, Faculty of informatics.

[2] ONDŘEJ, P. *Advanced static analysis using meta infer*. 2023. Master's thesis. Brno university of technology, Faculty of information technology.