FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Bounds for Variables and Loops: Better Together

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Fabian Souczek
Matrikelnummer 0728541

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Mitwirkung: Univ.Ass. Moritz Sinn MSc

Wien, 02.12.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Bounds for Variables and Loops: Better Together

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Fabian Souczek
Registration Number 0728541

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Assistance: Univ.Ass. Moritz Sinn MSc

Vienna, 02.12.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Fabian Souczek
Hauptstraße 44, 2126 Ladendorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagungen

Mein großer Dank gebührt meinen Betreuern Moritz Sinn und Florian Zuleger. Erst durch ihren Glauben an mich, bekam ich die Möglichkeit eine Arbeit im Bereich der Programmanalyse zu schreiben. Die Erfahrungen meiner beiden Betreuer in diesem Gebiet halfen mir enorm, eine starke und praxisrelevante Programmanalyse zu entwickeln.

Florian, deine Betreuung ließ keine Wünsche offen. Du hast immer das Big-Picture vor Augen, und es war somit ein Leichtes für mich den richtigen Fokus beizubehalten. Ich freute mich auf jedes der zahlreichen Treffen mit dir, da ich immer wusste, ich würde gestärkt und mit neuen Impulsen herauskommen.

Moritz, du weißt selbst, wie viel Dank dir gehört. Die unzähligen Probleme und Problemchen von der Konzeptualisierung, der Implementierung bis hin zur Formalisierung, die wir besprochen haben, könnten eine zweite Diplomarbeit füllen. Danke!

Ich hoffe, dass all die zukünftige wissenschaftliche Arbeit meiner beiden Betreuer die Anerkennung und Aufmerksamkeit bekommt, die sie verdient hat.

Ladendorf, Dezember 2014
Fabian Souczek

Abstract

A great part of today's used software has one point in common: software programs are running in an environment inherently constrained by physical resources such as power, network-traffic, time and memory. A mobile device has a limited amount of energy; a real-time system has to fulfill its tasks within a fixed time; and an embedded system is equipped with minimal memory.

Automatic resource-usage analyses have become an attractive area of research. Such an analysis can be started periodically within software development phases and is able to detect performance problems early and hidden to the engineers. Resource-usage analysis tools can help to mitigate the problem that mobile app stores are flooded by apps disregarding any fair resource usage and cloud providers may use them to predict the resource consumption of a served program.

How much of a resource a program may consume can often be reduced to the question how many times a loop is executed that contains a program location using this resource.

Recent techniques aim to compute loop bounds in a modular way. At first, a bound is computed for each loop that is expressed in terms of program variables defined immediately before the loop. To estimate the runtime complexity of the entire procedure, the next task is to translate each such bound into an expression over the procedure inputs. This task can be reduced to the problem of computing for each variable within that bound a specific kind of invariant that upper (or lower) bounds the possible value at a given location.

As experience shows, state-of-the-art invariant generation techniques are mostly insufficient for the purpose of bound analysis; they do not scale to non-linear or disjunctive invariants, which are required to shape bounds in presence of nested loops or loops with complex control flow.

We propose a bound analysis that connects loop and variable bound computation. The underlying observation is that loop and variable bounds can be expressed in terms of each other - they build a mutual recursion.

For example, if a program variable is incremented in every iteration of a loop by two, the overall increase of the variable's value due to that loop is bounded by the loop bound multiplied by two.

We demonstrate the efficacy of our new bound analysis by a thorough experimental evaluation. We implement a bound algorithm for the existing tool Loopus, which is developed at TU Vienna by Sinn and Zuleger and computes loop bounds of C programs.

We show that our method works effectively on patterns typically occurring in real-world programs and on examples posed in related literature. We show in comparison with other recent approaches that our method achieves promising results and outperforms existing techniques.

Kurzfassung

Ein Großteil heutzutage eingesetzter Software läuft in Umgebungen, die nur einen eingeschränkten Verbrauch von physischen Ressourcen (z.B. Speicher, Energie, Netzwerkverkehr, Zeit) erlauben. Mobilgeräte haben eine beschränkte Kapazität an Energie; Echtzeitsysteme führen Aufgaben in beschränkter Zeit aus; und eingebettete Systeme erhalten minimalen Speicher.

Die automatische Analyse von Software bezüglich ihres Ressourcenverbrauchs ermöglicht eine periodische Überprüfung der Einhaltung bekannter Schranken, sodass Performanceprobleme bereits früh im Entwicklungsprozess festgestellt werden. Weiters könnten Ressourcenanalysen von mobilen App-Store- und Cloud-Providern eingesetzt werden, um auf Apps mit hohem Ressourcenverbrauch hinzuweisen bzw. um genügend Ressourcen für Cloud-Programme bereitzustellen.

Die Frage nach dem Ressourcenverbrauch eines Programms kann oft auf das Problem reduziert werden, wie oft eine Schleife, die eine gegebene Ressource verbraucht, durchlaufen wird.

Jüngste Methoden berechnen Schleifenschranken in modularer Art. Zuerst werden Schranken über Programmvariablen ausgedrückt, die am Schleifenkopf definiert sind. Um eine Aussage über die Laufzeitkomplexität des ganzen Programms treffen zu können, werden diese Schranken anschließend in Ausdrücke über den Programmparametern zurückübersetzt. Diese Aufgabe kann wiederum auf die Berechnung von Invarianten zurückgeführt werden, welche die möglichen Werte von Variablen nach oben hin beschränken.

Aktuell eingesetzte Methoden der Invariantengenerierung sind unzureichend für die Analyse von Schranken; sie skalieren nicht für nicht-lineare und disjunktive Invarianten, welche notwendig sind im Falle von verschachtelten Schleifen und Schleifen mit komplexem Kontrollfluss.

Wir empfehlen Schranken für Schleifen und Variablen gemeinsam zu berechnen. Die zugrundeliegende Beobachtung ist, dass Schleifenschranken mittels Variablenschranken ausgedrückt werden können und umgekehrt - sie bilden eine wechselseitige Rekursion.

Zum Beispiel kann der Wertzuwachs einer Variable in einer Schleife, die in jeder Wiederholung die Variable um den Wert zwei inkrementiert, abgeschätzt werden, indem man die Schleifenschranke mit zwei multipliziert.

Wir demonstrieren die Leistungsfähigkeit unseres Ansatzes durch umfassende Experimente. Dazu wurde unsere Schrankenanalyse in das Analysetool LOOPUS integriert, welches an der TU Wien von Sinn und Zuleger entwickelt wird, um Schleifenschranken zu berechnen.

Unsere Methode ist für Programme effektiv, die in der Praxis eingesetzt werden oder in verwandter Literatur zu finden sind. Wir zeigen im Vergleich mit anderen Schrankenanalysen, dass unsere Entwicklung vielversprechende Resultate erreicht und andere Techniken an Leistungsfähigkeit übertrifft.

Contents

CHAPTER 1

Introduction

Computer programs consume physical resources like power, bandwidth, memory and CPU time. In many situations during software development and software verification, quantitative information about the resource usage provides helpful feedback for developers.

Memory bounds can ensure the reliability of embedded systems [17, 20]. Prior information about the power consumption is critical for mobile devices and cloud platforms [19, 25]. Real-time multimedia streaming requires bandwidth-intensive delivery, which happens nowadays often over mobile wireless networks that are typically bandwidth-limited. The amount of leaked data and the accuracy of programs running on unreliable hardware depends on the number of times a certain operation is used [16, 54]. The validation of real-time systems requires an upper bound of the execution time in the worst-case [63].

In general, many problems appearing in quantitative program analysis can be reduced to the question how many times a loop is executed that operates on a resource of special interest.

1.1 Problem Definition

Loop Bound A loop bound expresses the maximal number of iterations of a loop during execution of the program.

Variable Bound A variable bound is an expression that upper- (or lower-) bounds the possible value of a variable when the control flow reaches a specific program location.

A formal problem definition is stated in Chapter 2.

1.2 Variable Bounds for Loop Bound Computation

We demonstrate the need for variable bounds during loop bound computation on the examples depicted in Figure 1.1. We are looking in each example for a loop bound in terms of procedure inputs for the bottommost loop. We deduce that the value of variable x at the header of each bottommost loop is a loop bound for each bottommost loop. As we want to estimate the runtime complexity of the entire procedure and x is not a procedure parameter, we have to compute a variable bound for x in terms of procedure inputs.

1.3 Variable Bounds as Program Invariants

Finding a variable bound is an instance of the invariant generation problem, where invariants [29] are of the structure $variable \leq expression$. Although the automatic construction of invariants is provably intractable in the general case [11], methods exist that are able to obtain at least partial or conservative solutions. For example, techniques based on abstract interpretation [22] attempt to find invariants within a given domain like conjunctions of linear inequalities. However, as argued below, invariants generated by state-of-the-art tools are mostly insufficient for the purpose of bound analysis.

1.3.1 Insufficiency of State-of-the-Art Invariant Generation Tools

Applying abstract interpretation over the octagon abstract domain [56], which computes inequality relationships between two variables with unit coefficients, in example $Ex2$ reveals the invariant $x \leq j \wedge j \leq n$ at location 5 (i.e., header of the inner loop), which suffices to infer the upper bound n for x. Though this analysis scales well, it is not able to bound any of the remaining examples. In procedure $Ex3$, the polyhedron abstract domain [24], which consists of linear inequalities between multiple variables, establishes the invariant $x - m \leq 2i \wedge i \leq n$ at location 3, which leads to the bound $m + 2n$ for the variable x. However, this analysis does not scale and is restricted to linear relationships between variables. Non-linear invariants are needed in procedures $Ex1$ and $Ex6$. They can be generated by an abstract interpretation based analysis [31], but requires the user to list non-linear expressions that should be tracked. Other approaches supporting non-linearity are based on Gröbner basis computation and only scale to small programs [58, 59]. A further challenge is the need for disjunctive invariants. For example in $Ex4$, the disjunctive invariant $(0 \leq y < 50 \wedge x = 50) \vee (50 \leq y \leq 100 \wedge x = y)$ at location 3 suffices to deduce the bound $x \leq 100$. Unfortunately, standard abstract interpretation-based techniques support only conjunctive invariants and techniques capable to infer disjunctive invariants are mostly complex and inefficient or require user input [23, 33, 34, 39].

1.4 Loop and Variable Bounds: Better Together

A principal challenge in invariant inference is the reasoning about loops. Loop constructs introduce notorious difficulties as they enable indefinite iterations of loops and infinite ranges of values a variable may take. Fortunately, our analysis has one advantage: loops are bounded (if the analysis is able to prove it). So, we can compute variable bounds by using loop bounds to approximate the value a variable may possess after execution of a loop. The same scheme can also be applied for loop bounds, which paves the way to an integrated loop and variable bound computation.

```
1  Ex1(int n) {
2  int x=0,i,j;
3  for(i=0;i<n;++i){
4     for(j=0;j<n;++j)
5        x++;
6  }
7  while(x>0)
8     x--;
9  }
```

$x \leq n^2$

```
1  Ex2(int n) {
2  int x,i,j;
3  for(i=0;i<n;++i){
4     x=0;
5     for(j=0;j<n;++j)
6        x++;
7  }
8  while(x>0)
9     x--;
10 }
```

$x \leq n$

```
1  Ex3(int n,m){
2  int x=m, i;
3  for(i=0;i<n;
4        ++i){
5     x=x + 2;
6  }
7  while(x>0)
8     x--;
9  }
```

$x \leq m + 2n$

```
1  Ex4() {
2  int y=0,x=50;
3  while(y<100) {
4     if(y<50)
5        y++;
6     else {
7        y++;
8        x++;
9     }
10 }
11 while(x>0)
12    x--;
13 }
```

$x \leq 100$

```
1  Ex5(int n) {
2  int x=0,i,j=0;
3  for(i=0;i<n;++i){
4     j++;
5     while(j>0 && ?){
6        j--;
7        x++;
8     }
9  }
10 while(x>0)
11    x--;
12 }
```

$x \leq n$

```
1  Ex6(int n,m){
2  int x,y,i,j;
3  if (?) x=n;
4  else    x=m;
5  for(i=0;i<n;
6        ++i){
7     y=x + i;
8     for(j=0;j<n;
9           ++j)
10       y=y + 1;
11    x=y;
12 }
13 while(x>0)
14    x--;
15 }
```

$x \leq max(m,n) + 2n^2$

Figure 1.1: Examples illustrating the challenging task of computing upper bounds of variables. The sign '?' denotes non-determinism. Examples 1-3 are motivating examples of [35]. Example 4 is a prominent example for disjunctive invariants [34]. Example 5 shows why we model increments not as resets. If we approximate the increment of j at line 4 by an assignment to n (as n is an upper bound for j), we would get the quadratic invariant $x \leq n^2$. Example 6 is hard as the variable x is assigned the value of variable y (line 11) and vice versa (line 7), such that a naive recursive bound computation would fail.

1.4.1 Loop Bounds for Variable bound Computation

We have already discussed the need for variable bounds during loop bound computation in Section 1.2. We close the circle by integrating loop bounds into variable bound computation. The mutual recursive dependencies between loop and variable bounds are best explained on an example. Consider function $Ex6$ depicted in Figure 1.1.

We see that the loop at line 11 can run as often as the actual value of variable x at location 11. We write $LB(l_{11})$ to denote the loop bound for the loop at line 11 and $VB(x)$ to denote the (upper) variable bound for variable x.	$LB(l_{11}) = VB(x)$
To formulate the variable bound of x, we take into account that x is either initialized to n at line 3 or to m at line 4 by summarizing those two initializations with a maximum-expression. We spot a cyclic dependency between the variables x and y, as variable y is assigned the value of x at line 6 (plus the value of variable i) and at line 8 variable x is assigned 'reversely' the value of y. Nevertheless, the variable bound of x can be defined without requiring the variable bound of y by adding the increment of variable y at line 8 directly to the variable bound of x and by regarding the assignment at line 6 as an increment of x by i.	$VB(x) =$ $max(n, m)$ $+LB(l_7) * 1$ $+LB(l_5) * VB(i)$
The bound of the loop at line 7 depends on how often j is set back to 0 by the outer loop at line 5, which enables the loop to iterate n more times.	$LB(l_7) = LB(l_5) * n$
The loop at line 5 iterates at most n times.	$LB(l_5) = n$
The variable bound of i depends solely on how often the increment of i by 1 at line 5 can be executed as i is initialized to 0.	$VB(i) = LB(l_5) * 1$

We can easily resolve all remaining recursive definitions:

$$VB(i) = n,\ LB(l_7) = n^2, \text{ and } LB(l_{11}) = VB(x) = max(n, m) + n^2 + n^2.$$

1.4.2 Main Steps of our Bound Analysis

We envision the main steps of our analysis on procedure $Ex6$ depicted in Figure 1.1:

1. Program Abstraction: We abstract a given program to a difference-constraint program (DCP) [8]. We introduce DCPs in Chapter 2. The main characteristic of a DCP is that transitions are annotated by constraints of the form $x \leq y + c$, where x, y are variables and c is a constant. We explain in Chapter 4 how to extract DCPs from a C program.

   A DCP representing procedure $Ex6$ is depicted in Figure 1.2.

2. Termination Analysis: We compute an order on all loops of the DCP such that there is a variable for every loop, which decreases towards 0 on that loop and which does not increase on the loops that are lower in the order. As a result we get a lexicographic ranking function (as pioneered by Bradley et al. [12]). We give an algorithm that computes a lexicographic ranking function in Section 4.3.

   In our running example, we enumerate following loops: $\pi_0 = l_5 \to l_7 \to l_5$[1], $\pi_1 = l_7 \to l_7$ and $\pi_2 = l_{11} \to l_{11}$. We compute the lex. ranking function $\langle z_{n-i}, z_{n-j}, x\rangle$ for the loops π_0, π_1, π_2.

3. Bound Analysis: We compose a bound for a loop π or for variable x by adding for every loop τ how often and by how much τ increases the variable of π resp. the variable x. If this increase is expressed in terms of variables that are not procedure parameters, we compute an upper bound for them. A formal algorithm that computes loop and variable bounds is given in Chapter 3.

   We deduce that variable x is incremented during execution of the outer loop π_0 $(= l_5 \to l_7 \to l_5)$ by i plus the increment of y within the inner loop π_1 $(= l_7 \to l_7)$ by 1. As i is not constant, we compute recursively an upper bound, i.e., n. Assuming that we have already computed the bound n for π_0 and n^2 for π_1, we conclude the upper bound for x to be $max(n, m) + 2n^2$, which is simultaneously the loop bound for π_2 $(= l_{11} \to l_{11})$.

1.4.3 Comparison to State-of-the-Art Invariant Generation Tools

In Section 1.3.1, we noted the insufficiency of state-of-the-art invariant generation tools for bound analysis. In summary, we identify following two main challenges that are not solved by standard tools.

(C1) *Nested loops*. If a variable is modified inside a nested loop, then non-linear invariants are required. For example, in $Ex1$ variable x is incremented by 1 on the path of the nested loop going from location 5 to 5. The corresponding upper bound for x is non-linear, i.e., n^2.

(C2) *Multi-path loops*. If a variable is modified inside a loop with multiple paths (arising from conditional statements), then an invariant generation tool has to compute path-sensitive disjunctive invariants. In example $Ex4$, variable x is only incremented by the loop at location 3 when taking the else-branch, which happens only in 50 out of the total 100 iterations of that loop. The disjunctive invariant $(0 \le y < 50 \wedge x = 50) \vee (50 \le y \le 100 \wedge x = y)$ at location 8 enables the deduction of bound $x \le 100$ at line 8.

In contrast, our bound analysis solves the challenges (C1) and (C2): We obtain non-linear bounds by integrating already computed bounds with arithmetic operations like $*$ and max (defined as in common over the natural numbers). As we enumerate all paths of a loop and ask how every loop influences a bound, we accomplish path-sensitive bounds without having to compute explicitly disjunctive invariants.

[1] We write $l_i \to l_j$ for a path from location at line i to the location at line j.

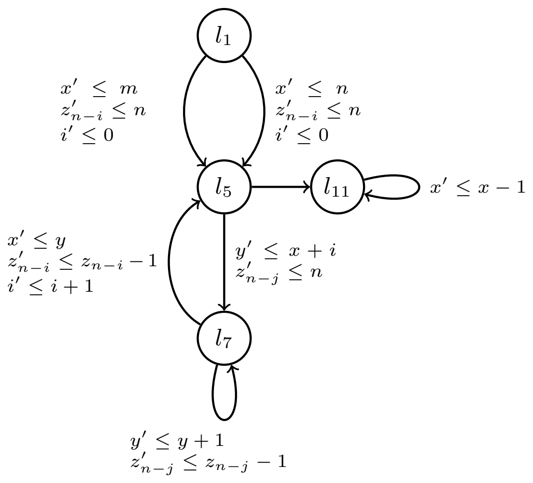


Figure 1.2: We illustrate the difference-constraint program obtained by abstraction from example $Ex6$ in Figure 1.1. We heuristically determine variables (e.g, x) and expressions (e.g., $n - i$ and $n - j$) that are in consequence represented in the DCP (e.g., z_{n-i} models $n - i$). Nodes correspond to the loop-headers and are labeled by the corresponding line number. Each transition is annotated by a set of difference constraints. If a variable v is not part of such a set, then that set contains implicitly $v' \leq v$. A transition predicate $y' \leq y + 1$ means that the value of variable y after taking the transition is incremented by 1, stays equal or is less than the value of y before taking the transition.

1.4.4 Comparison to other Related Work

The idea to harness loop bounds to compute an upper bound of a variable is already applied by other bound analysis tools [15, 35].

In contrast to our approach, [15] is only able to deduce the quadratic bound n^2 for the bottommost loop at location 10 in example $Ex5$, because the increment of j at line 4 is approximated by an assignment to n (as n is an upper bound for j). An experimental comparison between our tool and the tool of [15] can be found in Chapter 5.

The method in [35] concentrates on the definition of a variable bound algorithm and lacks therefore a solution incorporating variable and loop bound computation. Further, the resulting bounds are more conservative than those established by our method. In detail, the tool of [35] computes the asymptotically greater bound $x \leq m+n+n^2+n^3$. They do not incorporate a max-operator and they multiply the sum of the bounds for the loops modifiying x (i.e., $\ell_5 \to \ell_7 \to \ell_5$ with bound n and $\ell_7 \to \ell_7$ with bound n^2) with a unified increment (i.e., n as $n \geq i$ and $n \geq 1$) instead of multiplying each loop bound with the increment actually happening on that loop (i.e., n for $\ell_5 \to \ell_7 \to \ell_5$ and 1 for $\ell_7 \to \ell_7$).

1.5 Contributions

We list our main contributions to the area of bound analysis [5, 7, 15, 35, 37, 60, 64].

- We provide a program analysis that integrates loop and variable bound computation in a simple abstract program model with decidable properties.

- We state a method to compute program invariants of the form $variable \leq expression$, which outperforms existing invariant generation tools in presence of nested loops and loops with complex control flow.

- In line with earlier instances of Loopus but in contrast to other approaches [5, 7, 37, 64], our method does not rely on elaborate computations like abstract interpretation, computer algebra, linear optimization or software model checking.

- We thoroughly experiment with our new tool on several benchmarks with more than 200.000 lines of code consisting of real-world applications written in C and of examples found in related literature. We show the scalability of our method, that we can bound more loops and that we get preciser bounds than related tools. The experimental results can be found in Chapter 5.

1.6 Previous and Related Work

Recent work of Sinn and Zuleger [60] constitutes the basis of this work. They propose a well separated analysis consisting of four phases: program abstraction, control-flow abstraction, generation of a lexicographic ranking function and bound computation. Their ambitious program abstraction results in a system called lossy vector addition system with states (lossy VASS) that does not allow any interaction between variables (i.e., the value of a variable is never reset and cannot be added to another variable). A lossy VASS is further simplified during control-flow abstraction by a transition system that consists only of one location and contains for each simple loop-path one transition. So, in contrast to former work [38, 64] the handling of inner-loops is much easier as it is not necessary to add summaries of inner loops on the outer loop. Next, the termination algorithm creates a lexicographic ranking function consisting of one variable for each transition, such that the variable decreases on the transition towards 0. Finally, the bound algorithm computes a bound based on the lexicographic order. They extract lossy VASSs from programs using invariant generation based on proof-rules and symbolic execution techniques. We elaborate those proof-rules and present an integrated solution for the generation of run-time bounds and invariants.

Former work by Sinn and Zuleger encompasses the set-up of a theory for the use in bound analysis called size-change abstraction [64]. They name it 'size-change' as the abstraction models a possible change of the size of a variable after taking a transition by means of inequality constraints. First, they compute a disjunctive transition system for a given location. The main obstacle here results from the circumstance that inner loops are handled by computing their transitive hull. So, the hull of an inner loop is inserted to the path of the outer loop. Due to the disjunctiveness of the hull this leads to an exponential blow-up. Second, they compute for each strongly connected component of the control flow graph a local bound and finally compose them to a global bound. As a local bound can contain variables that are modified during run-time, abstract interpretation is applied to substitute such variables with global invariants.

Gulwani and Zuleger came up with a proof-rule based approach [38] tackling the problem of finding a bound on the number of times a specific program location can be reached during runtime - they call it accordingly the reachability-bound problem. Their algorithm computes first a transition system for a given location that over-approximates the state relations between any two consecutive visits to that location. Inner loops are inserted on a transition by their transitive closure computed by abstract interpretation. They use a powerset abstract domain lifted from a conjunctive abstract domain like octagon or polyhedra, which introduces termination problems as such a domain is of infinite height. The next step is to find a ranking function for each transition by looking for patterns commonly used by programmers that disallow an infinite trace. The final step is to compose the ranking functions to a bound by means of proof rules, which take into account the different ways how two transitions can interact with each other.

1.6.1 Related Work

Abstract-interpretation based solutions

Several publications corresponding to symbolic program complexity were worked out in the course of the SPEED project initiated by Microsoft Research. In [31], Gulavani and Gulwani describe an abstract-interpretation based timing analysis. They instrument the program with a monitor variable that increases in each loop iteration and is then bounded by abstract interpretation. To extend the precision, they lift a given linear abstract domain (e.g. difference constraints, polyhedra, ...) such that non-linear and disjunctive (incorporating a max-operator) bounds can be computed. Limitations are that it requires the user or some not specified heuristic to describe the kind of non-linear expressions to track.

In Gulwani's next approach [37], counter variable instrumentation is elaborated in such a way that the need for disjunctive and non-linear invariants is avoided making abstract interpretation an easier task. The trick is to add multiple counter variables such that the individual bound on each counter is linear. A global bound is composed from that individual bounds and can be non-linear or disjunctive. Their strategy where to place counter variables is provable optimal for achieving a precise bound. They state that fewer counters and less dependencies between them lead to more precise bounds. Additionally, they can also handle non-arithmetic programs, if the user provides quantitative functions for abstract data-structures.

In Gulwani's next work [33], two new concepts useful for bound analysis are presented. The work does not describe how to compute a loop bound (it requires an external function to do that), but provides helpful hints how to handle more and achieve more precise bounds for practical patterns. For a better handling of multi-path loops they suggest to refine the control flow making interleavings more explicit, which often results that the analyzed object is much simpler as infeasible paths are eliminated. Our tool LOOPUS applies a transformation with a similar effect called contextualization, which is out of scope of this thesis. For more information, please see the article [64]. The second novel concept is called progress invariants that allows to reason about the progress of one particular loop with respect to another loop. We employ such information as an extension to our upper bound algorithm (see the Restarting-Increment Rule in Section 3.4.2). It helps in cases where a variable is incremented in an inner loop but reset in the outer one such that the upper bound is not the increment multiplied with the total number of iterations of the inner loop but with the number of iterations the inner loop can be executed for each outer loop transition.

Alias et al [7] outlines a bound analysis for flowchart programs with transitions described by affine expressions ($\mathbf{x}' = A\mathbf{x} + \mathbf{a}$). Our abstract model is a special case of it if A is the identity matrix. So, their model is more expressible, but it requires an expensive abstract interpretation over polyhedron abstract domain to construct an abstract program. Our evaluation shows that our simpler abstraction suffices for many common programs. We can also use our abstraction for the example programs given in the article [7]. A nice point of their work is that they proved their ranking function algorithm to be complete even though it is greedy (like our algorithm), meaning that it always finds a ranking function if at least an affine one exists. A comparison of their tool RANK with our tool LOOPUS can be found in Chapter 5.

Variable bounds computed by loop bounds

Gulwani formulated the problem of translating arithmetic expressions back to the procedure header [35]. The solution follows the idea to use already computed loop bounds during the backward translation to estimate the computation of a loop. We reuse the idea, but we integrate it at the heart of the bound algorithm. In Gulwani's work, loop bounds are assumed to be already given at the header of the strongly connected component (SCC) containing the loop, and the goal is to bound the variables of the bound by backward translating them to the header. In contrast, our method also computes loop bounds within an SCC and uses for that also dynamically computed variable bounds.

Recently, Brockschmidt et al [15] worked out how to compute variable and loop bounds in an alternating fashion. Similarly to our method, they use loop bounds to compute variable bounds and vice versa. In contrast to us, they support polynomial instead of linear ranking functions. We argue that our approach offers a simpler representation. Our evaluation (see Chapter 5) shows that we can derive more bounds by a greater precision in less time.

Linear Programming based solutions

A type-based amortized analysis for the estimation of resource usage in first-order functional programs is given by Hofmann et al. [45]. They reduce the problem to linear constraint solving. Later they advanced their method to polynomial bounds [44], multivariate polynomial bounds [43] and higher-order programs [48].

Inspired by the work of Hofmann et al., Carbonneaux et al. published recently an amoritzed resource analysis framework for C programs [18]. They extend Hoare-logic by rules for potential functions, which enables reasoning about resource usage. The potential functions are of a fixed shape such that it becomes possible to use linear programming to compute a derivation in the logic. The main restriction of their method is that they can only deduce linear bounds. A comparison of their tool with LOOPUS on more than 30 examples is given in their paper [18].

Recurrence Solving based solutions

The COSTA project [4, 5] aims a cost analysis and studies for that goal the extraction of cost recurrence relations from source code and computer algebra methods to solve them. Our method can easily be extended for cost analysis if we annotate each transition by a cost measure. For a comparison of their tool RANK with our tool LOOPUS see Chapter 5.

The ABC system of Kovács et al. [10] aims to compute symbolic bounds for nested loops, but not for sequences of loops. They start by finding a variable for each loop that in- or decreases towards a bound. Then they create for such a variable a polynomial recurrence equation that models the assignments to that variable in all other loops. A closed form of this recurrence equation is computed by symbolic summation algorithms and constitutes a loop bound for the corresponding loop.

A similar approach is taken by the tool r-TuBound by Kovács et al. [49], which has the advantage against [10] to support sequences of loops and multi-path loops. Unlike [10], they are restricted to linear recurrences with constant coefficients, which have the advantage that they are always solvable. They point out that such linear recurrences are sufficient to describe the behavior of most loops in their benchmarks. In contrast to us, they do not support nested-loop structures.

WCET Analysis

The worst-case execution time (WCET) is a long-standing problem in the embedded and real-time system community. WCET analysis happens usually on two levels. A low-level analysis models the target architecture and estimates the execution time of a program instruction. A high-level analysis works platform-independent and aims for example to compute a parametric WCET for a program given in source code [51]. A parameter represents for example the concrete input value of a variable or the maximal iteration count for a loop. Automated methods for inferring loop bounds developed by the WCET community can be grouped into pattern based loop-counter detection [52] and solutions based on abstract-interpretation [28, 51, 55] or abstract execution [41]. All those methods work usually only for loops with relatively simple flow and arithmetic [46]. For complex examples, loop iteration counts are considered to be given by the user [63].

CHAPTER 2

Program Model and Main Definitions

In this chapter, we formally define *difference constraint programs* (DCPs), which we will abstract from concrete programs and on which our loop and variable bound algorithms are defined.

As we know by Alan Turing that the halting problem, which asks the question if a given program terminates on all inputs, is undecidable, every attempt for termination analysis and therefore also for bound analysis ends up in solving only a subproblem. We identify such a subproblem formally by defining difference constraint programs for which termination is decidable. Our approach stands in contrast to many other recent bound analyses [3, 15, 37, 38], which do not separate between the abstract and concrete program and formulate bound algorithms on a general for example C-like language.

A difference constraint program does not contain any programming-language specific feature. So, we can define loop and variable bound algorithms (as done in Chapter 3) independently from the programming language such that a new language is supported if a front-end abstracting a given program into a difference constraint program is provided. Such a front-end for C is described (on a high level) in Chapter 4.

Clearly, abstraction has an inherent loss of precision (the abstract program can arrive in a state not reachable by the concrete program) as not all features of a language fit into the abstract model. We claim that difference constraint programs retain enough information for an effective bound analysis of real-world code (as shown by our evaluation in Chapter 5).

This chapter is organized in the following way: we start by defining the concrete program model, our abstract program model (difference constraint programs) and the relation between them. Afterwards, we formally define loop and variable bounds. We complete this chapter with examples of difference constraint programs.

2.1 Program Model

We use a common program representation by a directed graph. The nodes of a program are called control locations. The edges of a program are labeled with relations over a set of states.

Definition 2.1 (Program)**.** *Let Σ be a set of* states. *The set of* transition relations $\Gamma = 2^{\Sigma\times\Sigma}$ *is the set of relations over Σ. A* program *is a tuple $P = (L, E)$, where L is a finite set of* locations, *and $E \subseteq L\times\Gamma\times L$ is a finite set of* transitions. *We write $l_1 \xrightarrow{\rho} l_2$ to denote a transition (l_1, ρ, l_2).*

Definition 2.2 (Program Path)**.** *A* path *of a program P is a sequence $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_{n-1}} l_n$ with $l_i \xrightarrow{\rho_i} l_{i+1} \in E$ for all $0 \le i < n$, where n denotes the length of the path. A path is* cyclic, *if it has the same start- and end-location. A path is* simple, *if it does not visit a location twice except for start- and end-location. We write $\pi = \pi_1 \cdot \pi_2$ for the* concatenation *of two paths π_1 and π_2, where the end-location of π_1 is the start-location of π_2. We say π' is a* subpath *of a path π, if there are (possibly empty) paths π_1 and π_2 with $\pi = \pi_1 \cdot \pi' \cdot \pi_2$.*

Definition 2.3 (Trace of a Program)**.** *A* trace *of a program P is a sequence $(l_0, \sigma_0) \xrightarrow{\rho_0} (l_1, \sigma_1) \xrightarrow{\rho_1} \cdots$ such that $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots$ is a path of P, and $(\sigma_i, \sigma_{i+1}) \in \rho_i$ for all $i \ge 0$.*

Definition 2.4 (Program Termination)**.** *A program P is* terminating, *if there is no infinite trace of P.*

2.2 Difference Constraint Program

In this thesis, we use difference constraint programs as an abstract program model. The main characteristic is the structure of the abstract transitions, which are specified by a conjunction of predicates of the form $x' \le y + c$ with variables $x, y \in \mathbb{N}$ and a constant $c \in \mathbb{Z}$, which are called *difference constraints* in standard computer science literature [21]. Such a formula describes how a variable x (we write x' to denote the value after taking the transition) can change its value by taking the transition in respect to y and c.
Although such constraints only allow increments, decrements and assignments, we claim that difference constraint programs are well-suited for the purpose of bound analysis. This is confirmed by our experiments on real-world code.

Definition 2.5 (Difference Constraint Program)**.** *A* Difference Constraint Program *(DCP) is a tuple $\Delta\mathcal{P} = (L, E)$, where L is a finite set of* locations, *$E \subseteq L \times (\mathit{Var} \times \mathbb{Z})^n \times L$ is a finite set of* transitions *with n variables in Var.*
We write $l_1 \xrightarrow{u} l_2$ to denote an edge (l_1, u, l_2) for some vector $u \in (\mathit{Var} \times \mathbb{Z})^n$. The ith element of u is $u(i) = (x_j, c)$ and refers to the predicate $x'_i \le x_j + c$ where the primed version x'_i belongs to the value of x_i after taking the transition.
The set of valuations *of Var is the set $\mathit{Val}_{\mathit{Var}} = \mathit{Var} \to \mathbb{N}$ of mappings from Var to the natural numbers, including 0.*

Examples of difference-constraint programs can be found in Section 2.5. Note that a DCP is a program by definition.

Definition 2.6 (Trace of a Difference Constraint Program)**.** *A* trace *of a difference constraint program $\Delta\mathcal{P}$ with variables Var is a sequence $(l_0, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ such that $l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \cdots$ is a path of $\Delta\mathcal{P}$, $\sigma_i \in \mathit{Val}_{\mathit{Var}}$ and $\sigma_{i+1}(x) \le \sigma_i(y) + c$ for all $i \ge 0$, $x, y \in \mathit{Var}$ and $c \in \mathbb{Z}$ s.t. $x' \le y + c \in u_i$.*

We define now some common terminologies.

Definition 2.7 (Resets and Increments)**.** *Let $\Delta\mathcal{P}(L,E)$ be a DCP over variables Var, and let $v \in Var$.*

1. $\mathcal{C}(v) = \{l_1 \xrightarrow{u} l_2 \in E \mid v' \leq v + c \in u\}$
2. $\mathcal{R}(v) = \{l_1 \xrightarrow{u} l_2 \in E \mid v' \leq v_1 + c \in u \wedge v \neq v_1\}$

Let $\tau = l_1 \xrightarrow{u} l_2 \in \mathcal{R}(v)$ be a transition, and let $t = (l_1,\sigma_1) \xrightarrow{u} (l_2,\sigma_2)$ be a trace of length 1*. If $v' \leq v_1 + c \in u$, then we call v_1 a reset of v and we say that τ resp. t resets v. We denote by $\gamma_\tau(v)$ resp. $\gamma_t(v)$ the variable v_1.*

Let $\tau = l_1 \xrightarrow{u} l_2 \in \mathcal{C}(v)$ be a transition, and let $t = (l_1,\sigma_1) \xrightarrow{u} (l_2,\sigma_2)$ be a trace of length 1*. If $v' \leq v + c \in u$, then we call c the increment of v on τ and we say that τ resp. t increments v by c. We denote by $\delta_\tau(v)$ resp. $\delta_t(v)$ the increment c.*

2.2.1 Decidability of Termination of DCPs

For the general class of difference constraint programs defined by Ben-Amram, termination is undecidable [8]. As our programs belong by definition to the restricted class of fan-in free programs, termination becomes decidable [8].

A program is fan-in free, if for every transition it holds that a variable of the target state has only one variable of the input state constraining it. Our definition describes only fan-in free programs, because a target variable x_i' is only part of one transition predicate $x_i' \leq x_j + c$ and x_j does not yield a transitive constraint to x_i' on another variable as x_j is not constrained by any variable (x_j is only implicitly constraint by the fact that x_j is a natural number).

2.2.2 Reducibility

We introduce the notion of a reducible program, which allows us to precisely identify loops within a program. This restriction does not limit our approach, because every irreducible program can be transformed into a reducible one (see for example the program transformation in [47]). Our current implementation fails if a program is irreducible.

A C program is always reducible, if no *goto*-statements are used. Irreducible control flow is very rare in practice due to the influential idea of structured programming, as recently studied in [61]. The authors analyzed open-source projects written in C containing 10427 functions and discovered that only five functions are irreducible and that no irreducible loop have been added in the last ten years. Also, our experiments with C code confirm that the vast majority of loops is reducible.

We define reducibility in the following solely as a graph theoretic property of the control flow graph like presented in common compiler literature [1]. Let $G = (V,E)$ be a directed graph with a unique entry point such that all nodes are reachable from the entry point.

Definition 2.8 (Dominator relation)**.** *Let $a, b \in V$. Node a* dominates *a node b, if every path from entry to b includes a.*

Definition 2.9 (Back edge)**.** *An edge $b \to a \in E$ is a* back edge*, if a dominates b.*

Definition 2.10 (Reducible graph)**.** *G is* reducible*, if G becomes acyclic after removing all back edges.*

Definition 2.11 (Loop header)**.** *A node is a* loop header*, if it is the target of a back edge.*

Definition 2.12 (Natural loop)**.** *The* natural loop *of a loop header h in a reducible graph is the maximal set of nodes L such that for all $x \in L$ (1) h dominates x and (2) there is a back edge from some node n to h such that there is path from x to node n that does not contain h.*

We list some consequence of the above definition: Every natural loop is uniquely defined by its loop header. Two natural loops A and B are either disjoint (i.e., $A \cap B = \emptyset$) or nested inside each other (i.e., $A \subseteq B$ or $B \subseteq A$). Further, in case of $A \subseteq B$, set containment is strict if and only if A and B have different loop headers.

Definition 2.13 (Loop-path)**.** *A* loop-path *π is a simple cyclic path, which starts and ends at some loop header l, and visits only locations inside the natural loop of l.*

In the following we define common terminologies used in the rest of this thesis.

Definition 2.14 (Loop-path sets)**.** *Let $P(L, E)$ be a program, and let $T \subseteq E$.*

1. *$\mathcal{L}(P)$ is the set of all loop-paths π of P.*
2. *$\mathcal{L}_\vee(P, T)$ is the set of all loop-paths π of P which contain at least one edge from T.*

We write $\mathcal{L}$ and $\mathcal{L}_\vee(T)$ if it is clear which program is used.

Definition 2.15 (Instance of a loop-path)**.** *Let $\pi = l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} \cdots l_{n-1} \xrightarrow{u_{n-1}} l_1$ be a loop-path. A path ν is an* instance *of π iff ν is of the form $l_1 \xrightarrow{u_1} l_2 * l_2 \xrightarrow{u_2} l_3 * l_3 \cdots l_{n-1} * l_{n-1} \xrightarrow{u_{n-1}} l_n = l_1$, where $l_i * l_i$ denotes any (possibly empty) path starting and ending at location l_i which does not contain l_1. A path p* contains *an instance ν of π iff ν is a subpath of p. Let ν be an instance of π contained in p; a transition t on p* belongs *to ν, if t is on ν and $t = l_i \xrightarrow{u_i} l_{i+1}$ for some i.*

Observation 2.16. *Every transition in a given path belongs to* at most one *instance of a loop-path. Every transition in a given cyclic path belongs to* exactly one *instance of a loop-path.*

2.3 Abstraction of a Program

We introduce the notion of a norm and define invariants of a program to relate the concrete program with the abstract one. We assume in the following that a program has a unique entry location l_{init}.

Definition 2.17 (Reachable States)**.** *A state σ is reachable at location l if and only if there is a trace $(l_0, \sigma_0) \xrightarrow{\rho_0} (l_1, \sigma_1) \xrightarrow{\rho_1} (l_2, \sigma_2) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_{n-1}} (l_n, \sigma_n)$ with $l_0 = l_{init}$ and $\sigma_n = \sigma$. We denote by $Reach(l)$ the set of all states that are reachable at location l.*

Definition 2.18 (Program Invariant). *Let $P = (L, E)$ be a program, and let Σ be a set of states. Let $e_1, e_2 \in \Sigma \to \mathbb{Z}$ be integer-valued expressions over the states, and let $c \in \mathbb{Z}$ be some integer. We say $e_1 \geq 0$ is* invariant for *l, if $e_1 \geq 0$ holds for all states $\sigma \in Reach(l)$.*
We say $e_2' \leq e_1 + c$ is invariant for $l_1 \xrightarrow{\rho} l_2$, if $e_2(\sigma_2) \leq e_1(\sigma_1) + c$ holds for all $(\sigma_1, \sigma_2) \in \rho$ with $\sigma_1 \in Reach(l_1)$.

Definition 2.19 (Norm). *A* norm *x is a function that maps the states to the natural numbers, i.e., $x \in \Sigma \to \mathbb{N}$, including 0.*

In the following, we define under which conditions a DCP is an abstraction of a program.

Definition 2.20 (Abstraction of a Program). *A difference constraint program $\Delta\mathcal{P} = (L, E')$ with variables Var is an* abstraction *of a program $P = (L, E)$ if and only if*

(1) every $x \in Var$ is a norm of P, and

(2) for each transition $l_1 \xrightarrow{\rho} l_2 \in E$ there is a transition $l_1 \xrightarrow{u} l_2 \in E'$ such that every $x' \leq y + c \in u$ is invariant for $l_1 \xrightarrow{\rho} l_2$.

2.4 Loop-path and Variable Bound Definitions

Definition 2.21 (Loop-path Bound). *Given a program P over variables Var with entry location l_0, an expression b over Var is a* loop-path bound *for a loop-path π of P if for every trace $(l_0, \sigma_0) \xrightarrow{\rho_0} (l_1, \sigma_1) \xrightarrow{\rho_1} (l_2, \sigma_2) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_{n-1}} (l_n, \sigma_n)$ of P the path $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_{n-1}} l_n$ contains at most $\sigma_0(b)$ instances of π.*

Similarly, Gulwani and Zuleger stated the problem of bounding the number of times a given control-location is visited during program execution and called it the *reachability-bound* problem [38]. Note that the sum of all bounds of those loop-paths on which the given control location lies, build a reachability-bound.

Further, loop-path bounds can be used to obtain the *computational complexity* of a program by summing up the bounds of all loop-paths of all loops. A loop bound can be established by adding the bounds of all loop-paths of a loop.

Definition 2.22 (Upper Bound of a Variable). *Given a program P over variables Var with entry location l_0, an expression b over Var is an* upper bound *for a variable $v \in Var$ if for every trace $(l_0, \sigma_0) \xrightarrow{\rho_0} (l_1, \sigma_1) \xrightarrow{\rho_1} (l_2, \sigma_2) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_{n-1}} (l_n, \sigma_n)$ of P it holds that $\sigma_n(v) \leq \sigma_0(b)$.*

2.5 Examples

In Figure 2.1, we see a DCP for example $Ex4$ depicted in Figure 1.1 in the introduction section, and Figure 2.2 shows a DCP representing example $Ex5$.

To obtain the precise variable bound 100 for variable x in example $Ex4$, we have to use the complex norm $z_{y \geq 50} := max(0, 50 - max(0, y - 50))$, which cannot be generated by our current implementation.

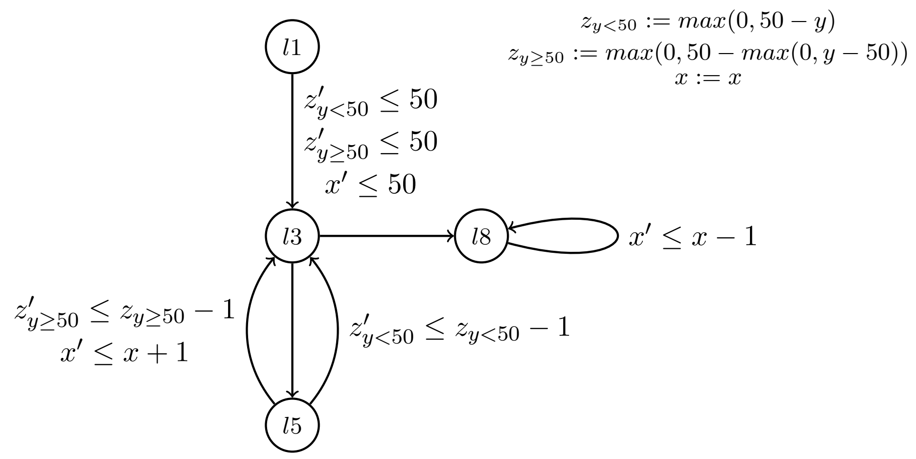

$z_{y<50} := max(0, 50 - y)$
$z_{y\geq 50} := max(0, 50 - max(0, y - 50))$
$x := x$
$l1$
$z'_{y<50} \leq 50$
$z'_{y\geq 50} \leq 50$
$x' \leq 50$
$l3$
$l8$
$x' \leq x - 1$
$z'_{y\geq 50} \leq z_{y\geq 50} - 1$
$x' \leq x + 1$
$z'_{y<50} \leq z_{y<50} - 1$
$l5$

Figure 2.1: A DCP representing example $Ex4$ given in the introduction 1.1. On the right top, norms for the original program are defined. We omit transition predicates of the form $x' \leq x$ for better readability.

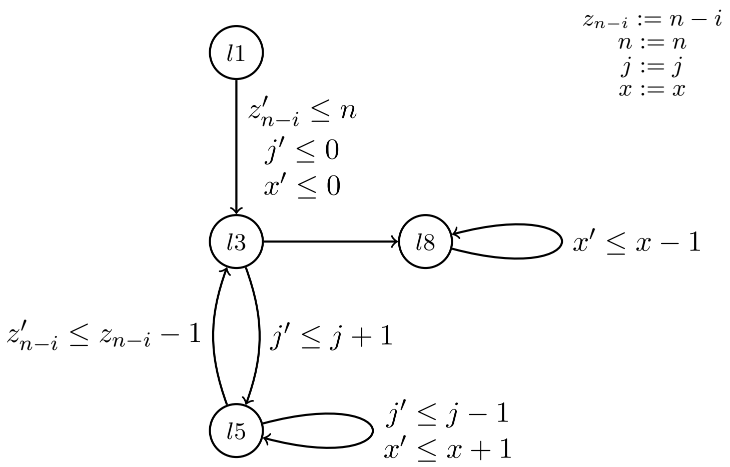


Figure 2.2: A DCP representing example $Ex5$ given in the introduction 1.1. On the right top, norms for the original program are defined. We omit transition predicates of the form $x' \leq x$ for better readability.

CHAPTER 3

Bound Computation

This chapter is organized in the following way: We state the loop-path and variable bound algorithms in Section 3.1. We give an example evaluation of the bound algorithms in Section 3.2. We show termination and soundness of the algorithms in Section 3.3. Further, we show some possible extensions improving the precision of the bounds computed by our bound algorithms in Section 3.4.

3.1 Loop-path and Variable Bound Algorithms

We assume to have given a difference constraint program $\Delta\mathcal{P}(L, E)$ over a set of variables Var with entry location l_0 and exit location l_e. Further, we have given for every variable $v \in \mathit{Var}$ the transition sets $\mathcal{R}(v)$ (i.e., all transitions resetting v) and $\mathcal{C}(v)$ (i.e., all transitions in- or decrementing v) as described in Definition 2.7.

Our bound algorithm is based on the assumption that for each loop-path π there exists a variable v which strictly decreases on π thereby limiting the number of consecutive executions of π. We therefore assume a mapping ζ to be given which assigns each loop-path a variable with the aforementioned property. We explain in Chapter 4 how to compute the mapping ζ.

Definition 3.1 (Local Ranking Mapping)**.** *Let $\zeta : \mathcal{L} \to \mathit{Var}$ be a mapping from the loop-paths of $\Delta\mathcal{P}$ to the variables. The mapping ζ is a* local ranking mapping *if and only if for all loop-paths π of $\Delta\mathcal{P}$ it holds that*

(i) $\forall \tau \in \pi : \tau \in \mathcal{C}(\zeta(\pi))$ *and*

(ii) $\sum\limits_{\tau \in \pi} \delta_\tau(\zeta(\pi)) < 0.$

Definition 3.2 (Variable Upper Bound VB and Loop-path Bound PB)**.** *Let ζ be a local ranking mapping. We define*
$\mathtt{VB}_\zeta : \mathit{Var} \mapsto \mathit{Expression}(\mathit{Var})$ *and* $\mathtt{PB}_\zeta : \mathcal{L} \mapsto \mathit{Expression}(\mathit{Var})$ *as:*

$$\texttt{VB}_\zeta(v) = \texttt{Increment}(v) + \max(v, \max_{\tau\in\mathcal{R}(v)} \texttt{Reset}(\tau, v))$$
$$\texttt{PB}_\zeta(\pi) = \texttt{Increment}(\zeta(\pi)) + \zeta(\pi) + \sum_{\tau\in\mathcal{R}(\zeta(\pi))} \texttt{TB}(\tau) \times \texttt{Reset}(\tau, \zeta(\pi))$$

where

- $\texttt{Increment}(v) = \sum_{\tau\in\mathcal{C}(v)} \texttt{TB}(\tau) \times \max(\delta_\tau(v), 0)$
- $\texttt{Reset}(\tau, v) = \texttt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)$
- $\texttt{TB}(\tau) = \sum_{\pi\in\mathcal{L}_\vee(\tau)} \texttt{PB}_\zeta(\pi) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$

3.2 Examples

3.2.1 Solving Example $Ex1$

As an example, we want to compute a loop-path bound for the loop at line 7 in $Ex1$ given in Figure 1.1 in the introduction section. Figure 3.1 depicts a *DCP* which represents example $Ex1$. By Definition of $\texttt{PB}_\zeta$ 3.2, if we assume that $\zeta(\pi_{l7}) = x1$, then we can start with:

$$\texttt{PB}_\zeta(\pi_{l7}) = \texttt{Increment}(x1) + x1 + \sum_{\tau\in\mathcal{R}(x1)} \texttt{TB}(\tau) \times \texttt{Reset}(\tau, x1).$$

We set in sub-definitions.

$$\texttt{PB}_\zeta(\pi_{l7}) = \sum_{\tau\in\mathcal{C}(x1)} \texttt{TB}(\tau) \times \max(\delta_\tau(x1), 0) + x1 + \sum_{\tau\in\mathcal{R}(x1)} \texttt{TB}(\tau) \times (\texttt{VB}_\zeta(\gamma_\tau(x1)) + \delta_\tau(x1))$$

We observe from the *DCP* seen in Figure 3.1, that $\mathcal{C}(x1) = \{l7 \to l7\}$ and $\mathcal{R}(x1) = \{l3 \to l7\}$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l7}) &= \texttt{TB}(l7 \to l7) \times \max(-1, 0) + x1 + \\ &\quad \texttt{TB}(l3 \to l7) \times (\texttt{VB}_\zeta(x) + 0) \\ &= x1 + \texttt{TB}(l3 \to l7) \times \texttt{VB}_\zeta(x)\end{aligned}$$

We see in Figure 3.1 that $l3 \to l7$ is not part of any loop-path and lies on a simple path $l1 \to l3 \to l7 \to le$ from the start to the end-location. We conclude $\texttt{TB}(l3 \to l7) = 1$.

$$\texttt{PB}_\zeta(\pi_{l7}) = x1 + \texttt{VB}_\zeta(x)$$

We start a sub-computation to compute a variable bound for x.

$$\begin{aligned}\texttt{VB}_\zeta(x) &= \texttt{Increment}(x) + \max(x, \max_{\tau\in\mathcal{R}(x)} \texttt{Reset}(\tau, x)) \\ &= \sum_{\tau\in\mathcal{C}(v)} \texttt{TB}(\tau) \times \max(\delta_\tau(x), 0) + \\ &\quad \max(x, \max_{\tau\in\mathcal{R}(x)} (\texttt{VB}_\zeta(\gamma_\tau(x)) + \delta_\tau(x)))\end{aligned}$$

We observe from the *DCP* seen in Figure 3.1, that $\mathcal{C}(x) = \{l4 \to l4\}$ and $\mathcal{R}(x) = \{l1 \to l3\}$. Further, we see that $l4 \to l4$ is not part of any simple path from the start to the end.

$$\begin{aligned}\texttt{VB}_\zeta(x) &= \texttt{TB}(l4 \to l4) \times \max(1, 0) + \max(x, \texttt{VB}_\zeta(0) + 0) \\ &= \texttt{TB}(l4 \to l4) + \max(x, 0) \\ &= \texttt{PB}_\zeta(l4 \to l4) + \max(x, 0)\end{aligned}$$

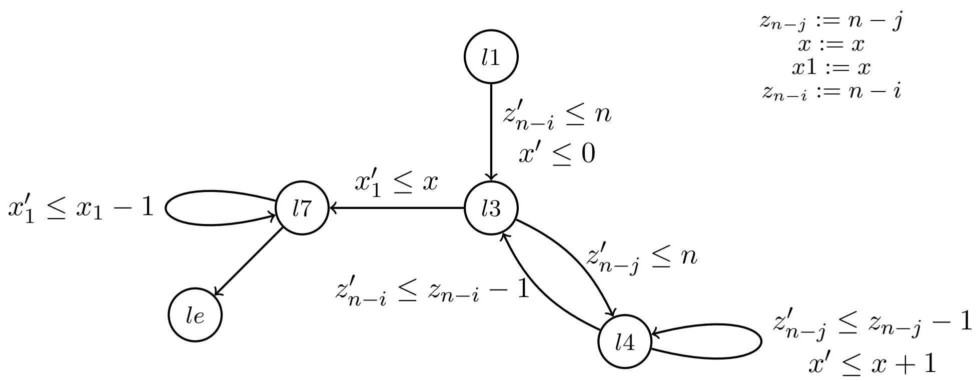


Figure 3.1: A DCP representing example $Ex1$ given in the introduction 1.1. We omit transition predicates like $x' \leq x$. On the right top, the used norms are defined.

We start a second sub-computation to compute a loop-path bound for the loop at line 4. We assume that $\zeta(\pi_{l4}) = z_{n-j}$. We observe from the DCP seen in Figure 3.1, that $\mathcal{C}(z_{n-j}) = \{l4 \to l4\}$ and $\mathcal{R}(z_{n-j}) = \{l3 \to l4\}$.

$\mathtt{PB}_\zeta(\pi_{l4}) = \mathtt{TB}(l4 \to l4) \times \max(-1, 0) + z_{n-j} + \mathtt{TB}(l3 \to l4) \times (\mathtt{VB}_\zeta(n) + 0)$

We see in Figure 3.1 that $l3 \to l4$ is part of the loop-path $l3 \to l4 \to l3$ and that $l3 \to l4$ is not part of any simple path from the start to the end. So, we have $\mathcal{L}_\vee(l3 \to l4) = \{l3 \to l4 \to l3\}$.

$\mathtt{PB}_\zeta(\pi_{l4}) = z_{n-j} + \mathtt{PB}_\zeta(l3 \to l4 \to l3) \times n$

We start a third sub-computation to compute the loop-bound for the loop π_{l3} at line 3. We assume that $\zeta(\pi_{l3}) = z_{n-i}$. We observe from the DCP seen in Figure 3.1, that $\mathcal{C}(z_{n-i}) = \{l4 \to l3\}$ and $\mathcal{R}(z_{n-i}) = \{l1 \to l3\}$.

$\mathtt{PB}_\zeta(\pi_{l3}) = \mathtt{TB}(l4 \to l3) \times \max(-1, 0) + z_{n-i} + \mathtt{TB}(l1 \to l3) \times (\mathtt{VB}_\zeta(n) + 0)$

We see in Figure 3.1 that $l1 \to l3$ is not part of any loop-path and lies on a simple path $l1 \to l3 \to l7 \to le$ from the start to the end-location. So, we have $\mathtt{TB}(l1 \to l3) = 1$.

$\mathtt{PB}_\zeta(\pi_{l3}) = z_{n-i} + n$
$\mathtt{PB}_\zeta(\pi_{l4}) = z_{n-j} + (z_{n-i} + n) \times n$
$\mathtt{VB}_\zeta(x) = z_{n-j} + (z_{n-i} + n) \times n + \max(x, 0)$
$\mathtt{PB}_\zeta(\pi_{l7}) = x1 + z_{n-j} + (z_{n-i} + n) \times n + \max(x, 0)$

We set $x, x1, z_{n-j}$, and z_{n-i} to 0 as they are not defined at the function header.

$\mathtt{PB}_\zeta(\pi_{l7}) = n^2$

3.2.2 Example motivating the definition of TB

The bound algorithms $\mathtt{VB}_\zeta$ and $\mathtt{PB}_\zeta$ use the function TB to get a transition bound for a given transition. Intuitively, a bound for a transition τ is the sum of all bounds of loop-paths containing τ. In addition, we have to add 1 to that sum if τ is part of a simple path from the entry to the exit location. A formal proof showing that $\texttt{TB}$ correctly computes a transition bound is given in Section 3.3.

The example depicted in Figure 3.2 shows why we have to add 1 to the sum of the corresponding loop-bounds to get a correct transition bound. We want to compute an upper bound for the variable j. Manual investigation observes that the bound for j is $n+1$, because the edge $l6 \to l4$, which contains the increment of j, can be executed as often as the back-edge from line 7 to line 4 can be taken and this can happen n times. Further, the edge $l6 \to l4$ is taken one additional time before i gets 1 at line 7. So, we get $n+1$ as the transition bound for $l6 \to l4$ and the variable bound for j.

We evaluate $\texttt{VB}_\zeta(j)$ in the following.

$$\begin{aligned}\texttt{VB}_\zeta(j) &= \texttt{Increment}(j) + \max(j, \max_{\tau \in \mathcal{R}(j)} \texttt{Reset}(\tau, j)) \\ &= \sum_{\tau \in \mathcal{C}(v)} \texttt{TB}(\tau) \times \max(\delta_\tau(j), 0) + \max(x, \max_{\tau \in \mathcal{R}(j)} (\texttt{VB}_\zeta(\gamma_\tau(j)) + \delta_\tau(j)))\end{aligned}$$

We observe from the DCP seen in Figure 3.2 on the right, that $\mathcal{C}(j) = \{l4 \to l6, l6 \to l4, l6 \to le\}$ and $\mathcal{R}(x) = \{l1 \to l4\}$.

$$\begin{aligned}\texttt{VB}_\zeta(j) &= \texttt{TB}(l4 \to l6) \times \max(1,0) + \texttt{TB}(l6 \to l4) \times \max(0,0) \\ &\quad + \texttt{TB}(l6 \to le) \times \max(0,0) + \max(j, \texttt{VB}_\zeta(0) + 0) \\ &= \texttt{TB}(l4 \to l6) + j\end{aligned}$$

The transition $l4 \to l6$ is part of the loop-path $l4 \to l6 \to l4$ and lies on the simple path $l1 \to l4 \to l6 \to le$ from the entry to the end location.

$$\texttt{VB}_\zeta(j) = \texttt{PB}_\zeta(l4 \to l6 \to l4) + 1 + j$$

We start a sub-computation to compute a loop-path bound for the loop $\pi_{l4} = l4 \to l6 \to l4$. We assume that $\zeta(\pi_{l4}) = i$. We observe from the DCP seen in Figure 3.2, that $\mathcal{C}(i) = \{l4 \to l6, l6 \to l4, l6 \to le\}$ and $\mathcal{R}(i) = \{l1 \to l4\}$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l4}) &= \texttt{TB}(l4 \to l6) \times \max(0,0) + \texttt{TB}(l6 \to l4) \times \max(-1,0) \\ &\quad + \texttt{TB}(l6 \to le) \times \max(0,0) + i + \texttt{TB}(l1 \to l4) \times (\texttt{VB}_\zeta(n) + 0) \\ &= i + \texttt{TB}(l1 \to l4) \times n\end{aligned}$$

The transition $l1 \to l4$ is not part of any loop, but lies on the simple path $l1 \to l4 \to l6 \to le$ from the entry to the end location.

$$\begin{aligned}&\texttt{PB}_\zeta(\pi_{l4}) = i + n \\ &\texttt{VB}_\zeta(j) = \texttt{PB}_\zeta(\pi_{l4}) + 1 + j = i + n + 1 + j\end{aligned}$$

We set i and j to 0 as they are not defined at the function header.

$$\texttt{VB}_\zeta(j) = n + 1$$

3.3 Termination and Soundness of the Bound Algorithms

3.3.1 Termination of the Bound Algorithms

The loop-path and variable bound algorithms given in Definition 3.2 obviously do not terminate if there exists a variable v such that during computation of $\texttt{VB}_\zeta(v)$ a recursive call to $\texttt{VB}_\zeta(v)$ is done or there exists a loop-path π such that during computation of $\texttt{PB}_\zeta(\pi)$ a recursive call to $\texttt{PB}_\zeta(\pi)$ is done. In the following, we will formulate the recursive dependencies between $\texttt{VB}_\zeta$ and

```
1 exTB(uint n) {
2   int i = n;
3   int j = 0;
4   do {
5     j++;
6   } while (i-- > 0);
7 }
```

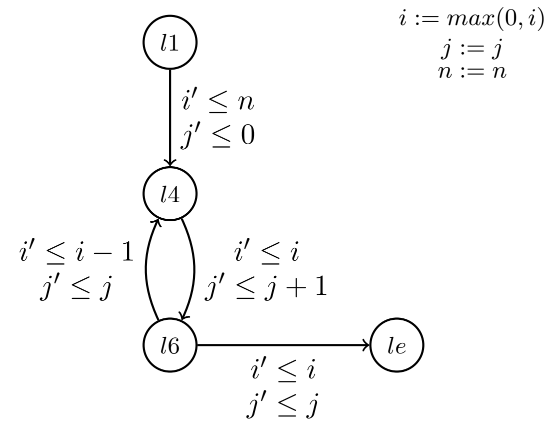


Figure 3.2: Example motivating the definition of the transition bound TB. On the right top, we list the used norms.

$\mathtt{PB}_\zeta$, and we will define the requirements a program has to fulfill such that a computation $\mathtt{VB}_\zeta$ or $\mathtt{PB}_\zeta$ cannot depend on itself (i.e. the computation must terminate).

Given a *DCP* $\Delta\mathcal{P}(L, E)$ over variables *Var*, the corresponding transition sets $\mathcal{R}$ and $\mathcal{C}$, and a mapping $\zeta : \mathcal{L} \mapsto Var$, we identify the following recursive dependencies between the bound computations $\mathtt{VB}_\zeta$ and $\mathtt{PB}_\zeta$ according to the definition of the bound algorithms.

(1) To compute an upper bound (i.e., $\mathtt{VB}_\zeta$) for a variable $v \in Var$ or a loop-path bound (i.e., $\mathtt{PB}_\zeta$) for a loop-path π where $\zeta(\pi) = v$, we have to compute an upper bound for each reset of v. So, $\mathtt{VB}_\zeta(v)$ and all $\mathtt{PB}_\zeta(\pi)$ where $\zeta(\pi) = v$ depend on the computation $\mathtt{VB}_\zeta(v_1)$ if there exists a transition $l_1 \xrightarrow{u} l_2 \in \mathcal{R}(v)$ such that $v' \leq v_1 + c \in u$.

(2) The computations $\mathtt{VB}_\zeta(v)$ and all $\mathtt{PB}_\zeta(\pi)$ where $\zeta(\pi) = v$ depend on the computation $\mathtt{PB}_\zeta(\pi_1)$ where $\zeta(\pi_1) = v_1$ if variable v is incremented on a transition $l_1 \xrightarrow{u} l_2 \in \pi_1$ (i.e., $v' \leq v + c \in u$ and $c > 0$).

(3) Given a variable $v \in Var$ such that there exists a loop-path π where $\zeta(\pi) = v$, then $\mathtt{PB}_\zeta(v)$ computes a loop-path bound for each loop-path containing a transition that resets v. So, $\mathtt{PB}_\zeta(v)$ depends on all $\mathtt{PB}_\zeta(\pi_1)$ where $\zeta(\pi_1) = v_1$ if there exists a transition $\tau \in \pi_1$ such that $\tau \in \mathcal{R}(v)$.

According to the recursive dependencies between the computations $\mathtt{VB}_\zeta$ and $\mathtt{PB}_\zeta$ described above, we define the binary relation $\mathfrak{R}$ in Definition 3.3, which reflects the points $(1 - 3)$ of the list above one to one.

Definition 3.3 (The Relation $\mathfrak{R}$). *We define the binary relation* $\mathfrak{R}_\zeta(\Delta\mathcal{P}) \subseteq Var \times Var$ *such that* $(v_1, v) \in \mathfrak{R}_\zeta(\Delta\mathcal{P})$ *if and only if*

(1) $\exists l_1 \xrightarrow{u} l_2 \in \mathcal{R}(v) : v' \le v_1 + c \in u$ *or*

(2) $\exists l_1 \xrightarrow{u} l_2 \in \mathcal{C}(v) : v' \le v + c \in u \wedge c > 0 \wedge \exists \pi \in \mathcal{L}_\vee(l_1 \xrightarrow{u} l_2) : \zeta(\pi) = v_1$ *or*

(3) $(\exists \pi \in \mathcal{L} : \zeta(\pi) = v) \wedge (\exists \pi \in \mathcal{L}_\vee(\mathcal{R}(v)) : \zeta(\pi) = v_1)$.

We denote by $\mathfrak{R}_\zeta(\Delta\mathcal{P})^*$ *the* transitive closure *of* $\mathfrak{R}_\zeta(\Delta\mathcal{P})$. *We often write* $\mathfrak{R}_\zeta$ *and* $\mathfrak{R}_\zeta^*$ *if it is clear which program is used.*

According to Definition 3.3, a pair of variables $(v_1, v) \in \mathfrak{R}$ exists if and only if either

(i) $\texttt{VB}_\zeta(v)$ recursively calls $\texttt{VB}_\zeta(v_1)$ or

(ii) $\texttt{VB}_\zeta(v)$ recursively calls $\texttt{PB}_\zeta(\pi_1)$ where $\zeta(\pi_1) = v_1$ or

(iii) $\texttt{PB}_\zeta(\pi)$ where $\zeta(\pi) = v$ recursively calls $\texttt{VB}_\zeta(v_1)$ or

(iv) $\texttt{PB}_\zeta(\pi)$ where $\zeta(\pi) = v$ recursively calls $\texttt{PB}_\zeta(\pi_1)$ where $\zeta(\pi_1) = v_1$.

Definition 3.4 (Strict Partial Order)**.** *A* strict partial order *is a binary relation R over a set S which satisfies for a, b, and $c \in S$:*

(i) $(a, a) \notin R$ *(irreflexive),*

(ii) $(a, b) \in R \wedge (b, c) \in R \implies (a, c) \in R$ *(transitive), and*

(iii) $(a, b) \in R \implies (b, a) \notin R$ *(asymmetric).*

Definition 3.5 (Cycle-free Variable Dependencies)**.** *We say that a program $\Delta\mathcal{P}(L, E)$ has* cycle-free variable dependencies *under mapping ζ if and only if $\mathfrak{R}_\zeta^*$ is a strict partial order.*

Theorem 3.6 (Termination)**.** *If $\mathfrak{R}_\zeta^*$ is a strict partial order, then* $\texttt{VB}_\zeta$ *and* $\texttt{PB}_\zeta$ *are total functions, i.e., the computations* $\texttt{VB}_\zeta$ *and* $\texttt{PB}_\zeta$ *terminate.*

3.3.2 Soundness of the Bound Algorithms

We structure the soundness proof of the bound algorithms in the following way. At first, we restrict the soundness to valid traces (see Definition 3.7) of amenable programs (see Definition 3.8). Next, we show the soundness of the transition bound function TB used by the bound algorithms. Finally, we prove the soundness of the loop and variable bound algorithms.

Valid Traces

A *DCP* allows traces that are infeasible for a 'normal' program. For example, a C function cannot stuck halfway through at a location different to the end-location (except for run-time errors). We give an example in Figure 3.3. We show the soundness of our bound algorithms only for traces that do not stuck at a location different to the end-location, which we call valid traces.

```
notgetstuck(int n) {
  int i = n;
  while (i > 0) {
    i--;
  }
}
```

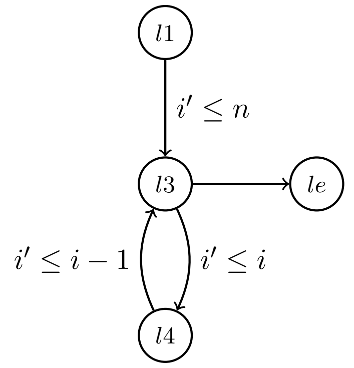


Figure 3.3: The DCP on the right may stuck at location $l4$ if i has value 0 at $l3$. In contrast, the original program never stucks at location $l4$.

Definition 3.7 (Valid Trace)**.** *We call a finite trace* $t = (l_0, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \cdots \xrightarrow{u_{n-1}} (l_n, \sigma_n)$ *of* $\Delta\mathcal{P}$ valid*, if and only if* $l_n = l_e$ *or there exists a trace* $t_2 = (l_n, \sigma_n) \xrightarrow{u_n} (l_{n+1}, \sigma_{n+1}) \xrightarrow{u_{n+1}} (l_{n+2}, \sigma_{n+2}) \xrightarrow{u_{n+2}} \cdots \xrightarrow{u_{n+k}} (l_e, \sigma_e)$*.*

Amenable Programs

We summarize the requirements that a program has to fulfill such that the loop and variable bound algorithms are sound in Definition 3.8. We say that a program is *amenable* if it is reducible and has a single, acyclic entry location and a single, acyclic exit location. We say a location is acyclic, if the location is not part of any loop.

Definition 3.8 (Amenable DCP)**.** *A DCP is* amenable *if and only if*

(i) has a single, acyclic entry location l_0 and a single, acyclic exit location l_e, and

(ii) is reducible (see Definition 2.10).

Soundness of the Transition Bound

In our bound algorithms, the function TB aims to return transition bounds for a given transition. Next, we formally define the notion of a transition bound and prove that TB(τ) is a transition bound for a transition τ.

Definition 3.9 (Transition Bound)**.** *Given a program $P(L, E)$ over variables Var with entry location l_0, an expression b over Var is a* transition bound *for a transition $\tau \in E$ such that on every trace* $(l_0, \sigma_0) \xrightarrow{\rho_0} (l_1, \sigma_1) \xrightarrow{\rho_1} (l_2, \sigma_2) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_{n-1}} (l_n, \sigma_n)$ *of P the transition τ appears at most $\sigma_0(b)$ times.*

Lemma 3.10. *Let $\Delta\mathcal{P}(L, E)$ be an amenable DCP over variables Var, and let $\tau \in E$ be a transition. If for all $\pi \in \mathcal{L}_\vee(\tau)$, the expression* PB$_\varsigma(\pi)$ *is a loop-path bound for π, then*

$\sum_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$ *is a transition bound for τ on all valid traces of $\Delta\mathcal{P}$.*

Proof. Let $t_v = (l_0, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \cdots \xrightarrow{u_{n-1}} (l_n, \sigma_n)$ be a valid trace of $\Delta\mathcal{P}$, and let $t = t_v \cdot t_e$ be a trace finishing t_p to the end-location l_e, i.e., $t_e = (l_n, \sigma_n) \xrightarrow{u_n} (l_{n+1}, \sigma_{n+1}) \xrightarrow{u_{n+1}} (l_{n+2}, \sigma_{n+2}) \xrightarrow{u_{n+2}} \cdots \xrightarrow{u_{n+k}} (l_e, \sigma_e)$.

Let $\sharp(\tau, t)$ denote the total number of occurrences of the transition τ on trace t. Clearly, we have $\sharp(\tau, t_v) \leq \sharp(\tau, t)$. We show the stronger proposition that

$\sharp(\tau, t) \leq \sigma_0(\sum_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi)) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$

assuming that $\sharp(\pi, t) \leq \sigma_0(\mathtt{PB}_\zeta(\pi))$ for all $\pi \in \mathcal{L}_\vee(\tau)$, where $\sharp(\pi, t)$ denotes the number of instances of loop-path π on trace t.

We can fully decompose the path $l_0 \xrightarrow{u_0} l_1 \cdots \xrightarrow{u_{n+k}} (l_e, \sigma_e)$ into segments $t_1 \cdot C_1 \cdot t_2 \cdots C_m \cdot t_m$ such that each C_i is a cyclic (non-empty) path, and the concatenation $t_i \cdot t_2 \cdots t_m$ is a simple path with start-location l_0 and end-location l_e.

Case 1: W.l.o.g. let τ be part of t_2, then τ cannot be part of any other t_i where $i \neq 2$, because otherwise $t_i \cdot t_2 \cdots t_m$ would not be simple, as the start- and end-location of τ would appear twice on it. We conclude that $\sharp_{case1}(\tau, t) = \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$.

Case 2: Let τ be part of a cyclic path C_i. Then τ is part of a loop-path, $\mathcal{L}_\vee(\tau) \neq \emptyset$. By the reducibility of $\Delta\mathcal{P}$ and observation 2.16, τ_i must belong to exactly one instance of a loop-path $\pi \in \mathcal{L}_\vee(\tau)$. So, we conclude $\sharp_{case2}(\tau, t) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \sharp(\pi, t)$.

Summing up both cases and applying our initial assumption results in

$$\sharp(\tau, t) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \sharp(\pi, t) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$$
$$\leq \sum_{\pi \in \mathcal{L}_\vee(\tau)} \sigma_0(\mathtt{PB}_\zeta(\pi)) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}.$$

□

Theorem 3.11 (Soundness)**.** *If $\mathfrak{R}^*_\zeta$ is a* strict partial order*, then we have for all valid traces of $\Delta\mathcal{P}$ that $\mathtt{PB}_\zeta(\pi)$ is a* loop-path bound *for any $\pi \in \mathcal{L}$ and $\mathtt{VB}_\zeta(v)$ is a* variable upper bound *for any $v \in Var$.*

Proof. Let π be a loop-path of $\Delta\mathcal{P}$, let $v \in Var$, and let $t = (l_0, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} (l_2, \sigma_2) \xrightarrow{u_2} \cdots \xrightarrow{u_{n-1}} (l_n, \sigma_n)$ be a valid trace of $\Delta\mathcal{P}$. We have to show $\sharp(\pi, t) \leq \sigma_0(\mathtt{PB}_\zeta(\pi))$ and $\sigma_n(v) \leq \sigma_0(\mathtt{VB}_\zeta(v))$, where $\sharp(\pi, t)$ denotes the number of instances of π on trace t.

We assume that $\mathfrak{R}^*_\zeta$ is a strict partial order. We proceed by induction over $(Var, \mathfrak{R}^*_\zeta)$.

Base Case 1: Let v be minimal in $(Var, \mathfrak{R}^*_\zeta)$, i.e. there is no $v_1 \in Var$ s.t. $(v_1, v) \in \mathfrak{R}^*_\zeta$. As v is minimal, we get by Definition 3.3 that there exists no $l_1 \xrightarrow{u} l_2 \in E$ s.t. it would hold that $v' \leq v + c \wedge c > 0$ or $v' \leq v_1 + c \wedge v \neq v_1$. Hence, we have by Definition 2.7 that $\mathcal{R}(v) = \emptyset$ and $\mathtt{Increment}(v) = 0$, and therefore $\mathtt{VB}_\zeta(v) = v$ by Definition 3.2. According to

Definition 3.3 of $\mathfrak{R}$, we can also say that there exists no $l_1 \xrightarrow{u} l_2 \in E$ s.t. $u \models v' > v$. So, we have $\sigma_n(v) \leq \sigma_0(v) = \sigma_0(\mathtt{VB}_\zeta(v))$.

Base Case 2: Let $v = \zeta(\pi)$ be minimal in $(\mathit{Var}, \mathfrak{R}^*_\zeta)$. Reasoning in the same way as for base case 1 leads to $\mathtt{PB}_\zeta(\pi) = v$. By the Definition 3.1 of ζ, variable v must be decremented by at least 1 during an iteration of π. As we have the invariant $v \geq 0$ at every location by Definition 2.5 of a DCP, the number of times π can be executed is bounded by the initial value of v, if v is not increased on any transition $\tau \in E$ not part of π. Like in base case 1, we conclude from Definition 3.3 that v is not increased on any $\tau \in E$. So, we have $\sharp(\pi, t) \leq \sigma_0(v) = \sigma_0(\mathtt{PB}_\zeta(\pi))$.

Step Case 1: Let v be not minimal in $(\mathit{Var}, \mathfrak{R}^*_\zeta)$, i.e. there exists some $v_1 \in \mathit{Var}$ s.t. $(v_1, v) \in \mathfrak{R}^*_\zeta$. The value of v by valuation σ_n is bounded by some base value increased by all increments along the trace since v have got assigned that base value. We make a case distinction if the base value is either the initial value of v or some reset of v.

If the base value of v is the initial value of v, then we can say that
$\sigma_n(v) = \sigma_0(v) + \sum\limits_{t_j \in t} \delta_{t_j}(v)$.

If $\tau_r = l_r \xrightarrow{u_r} l_{r+1}$ is the last transition that appears on trace t and resets v, i.e. we can partition $t = t_1 \cdot t_r \cdot t_2$ s.t. t_2 does not reset v and $t_r = (l_r, \sigma_r) \xrightarrow{u_r} (l_{r+1}, \sigma_{r+1})$. We can say that $\sigma_n(v) = \sigma_r(\gamma_{\tau_r}(v)) + \delta_{\tau_r}(v) + \sum\limits_{t_j \in t_2} \delta_{t_j}(v)$.

By the semantics of maxima and because t_2 is a subtrace of t, we can merge the two cases of different base values and we get that
$\sigma_n(v) \leq \max(\sigma_0(v), \sigma_r(\gamma_{\tau_r}(v)) + \delta_{\tau_r}(v)) + \sum\limits_{t_j \in t} \max(\delta_{t_j}(v), 0)$.

Let p_t be the underlying path of trace t. Then we get by logical reasoning:
$\sigma_n(v) \leq \max(\sigma_0(v), \sigma_r(\gamma_{\tau_r}(v)) + \delta_{\tau_r}(v)) + \sum\limits_{\tau \in p_{t_2}} \sharp(\tau, p_t) \times \max(\delta_\tau(v), 0)$
By Definition of $\mathcal{R}$ and $\mathcal{C}$, we have $\tau_r \in \mathcal{R}(v)$ and $\{\tau \in E \mid \tau \in p_t\} \subseteq \mathcal{C}(v)$. So, we get $\sigma_n(v)$
$\leq \max(\sigma_0(v), \max\limits_{\tau \in \mathcal{R}(v)} \sigma_r(\gamma_\tau(v)) + \delta_\tau(v)) + \sum\limits_{\tau \in \mathcal{C}(v)} \sharp(\tau, p_t) \times \max(\delta_\tau(v), 0)$
$\leq \sigma_0(\mathit{max}(v, \max\limits_{\tau \in \mathcal{R}(v)} \mathtt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)) + \sum\limits_{\tau \in \mathcal{C}(v)} \mathtt{TB}(\tau) \times \max(\delta_\tau(v), 0))$.
By Lemma 3.10 (Soundness of the Transition Bound):
$\sigma_n(v) \leq \sigma_0(\mathit{max}(v, \max\limits_{\tau \in \mathcal{R}(v)} \mathtt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)) + \sum\limits_{\tau \in \mathcal{C}(v)} \max(\delta_\tau(v), 0) \times$
$(\sum\limits_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \ldots \xrightarrow{u_n} l_e \\ 0 & \mathit{otherwise} \end{cases}))$
By Definition 3.2 of $\mathtt{VB}_\zeta$:
$= \sigma_0(\mathtt{VB}_\zeta(v))$.
As for every $\tau_r \in \mathcal{R}(v)$ and $\tau_c \in \mathcal{C}(v)$ s.t. $\delta_{\tau_c}(v) > 0$, we have $(\gamma_{\tau_r}(v), v) \in \mathfrak{R}^*_\zeta$ resp. $(\zeta(\pi), v) \in \mathfrak{R}^*_\zeta$ for all $\pi \in \mathcal{L}_\vee(\tau_c)$. Hence, the inequalities hold by assuming the induction hypothesis.

Step Case 2: Let $v = \zeta(\pi)$ be not minimal in $(\mathit{Var}, \mathfrak{R}^*_\zeta)$. By Definition 3.1 of ζ, variable v must be decremented by at least 1 during an iteration of π. As we have the invariant $v \geq 0$ at every location by Definition 2.5 of a DCP, the number of times π can be executed is bounded by the initial value of v plus all increasing assignments to v. Hence,

$\sharp(\pi, t) \leq \sigma_0(var) + \sum_{t_j \in t, t_j \text{not resets } v} \max(\delta_{t_j}(v), 0) + \sum_{t_j \in t, t_j \text{resets } v} \sigma_j(\gamma_{t_j}(v)) + \delta_{t_j}(v)$.
Let p_t be the underlying path of trace t. Then we get by logical reasoning:
$\sharp(\pi, t) \leq \sigma_0(var) + \sum_{\tau \in p_t, \tau \text{not resets } v} \sharp(\tau, t) \times \max(\delta_\tau(v), 0) +$
$\sum_{\tau \in p_t, \tau \text{resets } v} \sharp(\tau, t) \times (\max_{j=0}^{n} \sigma_j(\gamma_\tau(v)) + \delta_\tau(v))$.
By Definition of $\mathcal{R}$ and $\mathcal{C}$, we have that every $\tau \in p_t$ that resets v is in $\mathcal{R}(v)$ and that every $\tau \in p_t$ that not resets v is in $\mathcal{C}(v)$.
$\sharp(\pi, t) \leq \sigma_0(var) + \sum_{\tau \in \mathcal{C}(v)} \sharp(\tau, t) \times \max(\delta_\tau(v), 0) + \sum_{\tau \in \mathcal{R}(v)} \sharp(\tau, t) \times (\max_{j=0}^{n} \sigma_j(\gamma_\tau(v)) + \delta_\tau(v))$
$\leq \sigma_0(var + \sum_{\tau \in \mathcal{C}(v)} \mathtt{TB}(\tau) \times \max(\delta_\tau(v), 0) + \sum_{\tau \in \mathcal{R}(v)} \mathtt{TB}(\tau) \times (\mathtt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)))$
By Lemma 3.10 (Soundness of the Transition Bound):
$\sharp(\pi, t) \leq \sigma_0(var+$
$\sum_{\tau \in \mathcal{C}(v)} (\sum_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & otherwise \end{cases})$
$\times \max(\delta_\tau(v), 0) +$
$\sum_{\tau \in \mathcal{R}(v)} (\sum_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & otherwise \end{cases})$
$\times (\mathtt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)))$
By Definition 3.2 of $\mathtt{PB}_\zeta$:
$= \sigma_0(\mathtt{PB}_\zeta(\pi))$.
For every $\tau \in \mathcal{C}(v)$ s.t. $\delta_\tau(v) > 0$, we have $\forall \pi \in \mathcal{L}_\vee(\tau) : (\zeta(\pi), v) \in \mathfrak{R}^*_\zeta$. For every $\tau \in \mathcal{R}(v)$, we have $(\gamma_\tau(v), v) \in \mathfrak{R}^*_\zeta$ and $\forall \pi \in \mathcal{L}_\vee(\tau) : (\zeta(\pi), v) \in \mathfrak{R}^*_\zeta$. Hence, the inequalities hold by assuming the induction hypothesis. □

3.4 Extensions

In this section, we present three extensions for our bound algorithms improving the precision of the computed bounds.

3.4.1 Non-Monotonic In- or Decrease

The bound algorithms given in Definition 3.2 are imprecise (even asymptotical) if a variable is increased on a transition of a loop-path π but decreased on another transition of π. We give a formal definition of a variable with non-monotonic in- or decrease in Definition 3.12.

Definition 3.12 (Non-Monotonic In- or Decrease)**.** *Let $\Delta\mathcal{P}(L, E)$ be a DCP over variables Var. A variable $v \in Var$ is* non-monotonic in- or decreasing *if there exists a loop-path $\pi \in \mathcal{L}$ containing two transitions $\tau_1 = l_1 \xrightarrow{u_1} l_2$ and $\tau_2 = l_3 \xrightarrow{u_2} l_4$ such that $\tau_1 \neq \tau_2$, $(v' \leq v + c_1 \in u_1 \wedge c_1 > 0) \vee (v' \leq v_1 + c_1 \in u_1 \wedge v \neq v_1)$, and $(v' \leq v + c_2 \in u_2 \wedge c_2 < 0) \vee (v' \leq v_1 + c_1 \in u_1 \wedge v \neq v_1)$.*

In Figure 3.4, an example is depicted which contains variable i which is non-monotonic in- or decreasing, because i is increased by 100 on the edge $l3 \to l5$, decreased by 100 on the edge $l5 \rightarrow l3$, and the transition $l3 \rightarrow l5$ together with $l5 \rightarrow l3$ are part of the loop-path $l3 \rightarrow l5 \rightarrow l3$. Our original bound algorithms compute the loop-path bound $n * 100$ for the loop at line 5, because the increment by 100 of i on edge $l3 \rightarrow l5$ can happen n times and the decrement of 100 is not considered.

We give a modified version of our bound algorithms mitigating the problems due to non-monotonic in- or decrease in Definition 3.14. We require some additional sets of paths given in Definition 3.13. In the following, we assume to have given a difference constraint program $\Delta\mathcal{P}(L, E)$ over a set of variables Var with entry location l_0 and exit location l_e.

Definition 3.13 (Additional Path Sets)**.** *Let $T \subseteq E$, and $v \in Var$.*

1. *$\mathcal{L}_\wedge(\Delta\mathcal{P}, T)$ is the set of all loop-paths of $\Delta\mathcal{P}$ taking only edges in T*
2. *$\mathcal{A}(\Delta\mathcal{P}, l_1 \xrightarrow{u} l_2, T)$ is the set of all simple acyclic paths starting at l_2 and taking only edges in T*
3. *$\mathcal{L}^+(\Delta\mathcal{P}, v) = \{\pi \in \mathcal{L}_\wedge(\mathcal{C}(v)) \mid \sum_{\tau \in \pi} \delta_\tau(v) > 0\}$*
4. *$\mathcal{S}(\Delta\mathcal{P}, v) = \{\tau \in \mathcal{C}(v) \mid \tau$ is part of a simple path $l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e\}$*

We write $\mathcal{L}_\wedge(T)$, $\mathcal{A}(l_1 \xrightarrow{u} l_2, T)$, $\mathcal{L}^+(v)$ and $\mathcal{S}(v)$ if it is clear which program is used.

Definition 3.14 (Extension of VB and PB)**.** *Let $\zeta : \mathcal{L} \rightarrow Var$ be a local ranking mapping. We define*
$\mathtt{VB}_\zeta : Var \mapsto Expression(Var)$ *and* $\mathtt{PB}_\zeta : \mathcal{L} \mapsto Expression(Var)$ *as:*
$\mathtt{VB}_\zeta(v) = \mathtt{Increment}(v) + \max(v, \max_{\tau \in \mathcal{R}(v)} \mathtt{Reset}(\tau, v))$
$\mathtt{PB}_\zeta(\pi) = \mathtt{Increment}(\zeta(\pi)) + \zeta(\pi) + \sum_{\tau \in \mathcal{R}(\zeta(\pi))} \mathtt{TB}(\tau) \times \mathtt{Reset}(\tau, \zeta(\pi))$
where

- $\mathtt{Increment}(v) = \sum_{\pi \in \mathcal{L}^+(v)} \mathtt{PB}_\zeta(\pi) \times \sum_{\tau \in \pi} \delta_\tau(v) + \sum_{\tau \in \mathcal{S}(v)} \max(\delta_\tau(v), 0)$
- $\mathtt{Reset}(\tau, v) = \mathtt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v) + \max_{p \in \mathcal{A}(\tau, \mathcal{C}(v))} \sum_{\tau \in p} \delta_\tau(v)$
- $\mathtt{TB}(\tau) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \mathtt{PB}_\zeta(\pi) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$

Example

We want to compute a loop-path bound for the loop $\pi_{l5} = l5 \rightarrow l5$ at line 5. We assume that $\zeta(\pi_{l5}) = i_{+100}$.

$$\mathtt{PB}_\zeta(\pi_{l5}) = \mathtt{Increment}(i_{+100}) + i_{+100} + \sum_{\tau \in \mathcal{R}(i_{+100})} \mathtt{TB}(\tau) \times \mathtt{Reset}(\tau, i_{+100})$$

We observe from the *DCP* seen in Figure 3.4, that $\mathcal{C}(i_{+100}) = \{l3 \rightarrow l5, l5 \rightarrow l3, l3 \rightarrow le\}$ and $\mathcal{R}(i_{+100}) = \{l1 \rightarrow l3\}$ and that there exists no loop-path out of the edges $\mathcal{C}(i_{+100})$ such

```
exNonMono(uint n) {
  int i = 0;
  while (n > 0) {
    i += 100;
    while (i > 0 && ?)
      i--;
    i -= 100;
  }
}
```

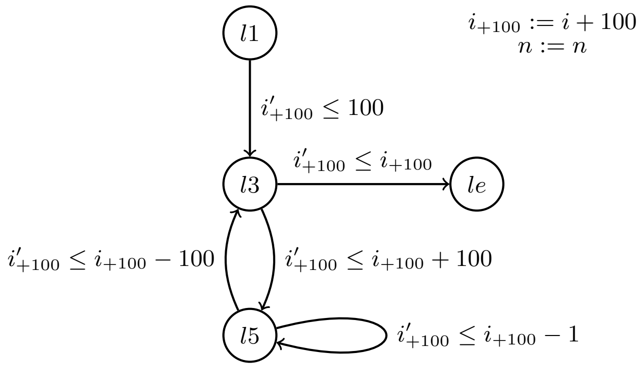

$i_{+100} := i + 100$
$n := n$
$l1$
$i'_{+100} \leq 100$
$l3$
$i'_{+100} \leq i_{+100}$
le
$i'_{+100} \leq i_{+100} - 100$
$i'_{+100} \leq i_{+100} + 100$
$l5$
$i'_{+100} \leq i_{+100} - 1$

Figure 3.4: Example showing variable i with a non-monotonic in- or decrease. On the right top, we list the used norms.

that i_{+100} is increased if taking the entire loop-path. Hence, $\mathcal{L}^+(i_{+100}) = \emptyset$. Further, we see that only the edge $l3 \to le$ out of $\mathcal{C}(i_{+100})$ is lying on a simple path from the start to the end location. Hence, $\mathcal{S}(v) = \{l3 \to le\}$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l5}) &= \max(\delta_{l3\to le}(i_{+100}), 0) + i_{+100} + \texttt{TB}(l1 \to l3) \times \texttt{Reset}(l1 \to l3, i_{+100})\\ &= i_{+100} + \texttt{TB}(l1 \to l3) \times \texttt{Reset}(l1 \to l3, i_{+100})\end{aligned}$$

As the transition $l1 \to l3$ is not part of any loop but lies on a simple path from the entry to the exit location, we have $\texttt{TB}(l1 \to l3) = 1$. Now, we set in the new definition for $\texttt{Reset}$.

$$\texttt{PB}_\zeta(\pi_{l5}) = i_{+100} + \texttt{VB}_\zeta(100) + 0 + \max_{p \in \mathcal{A}(l1 \to l3, \mathcal{C}(i_{+100}))} \sum_{\tau \in p} \delta_\tau(i_{+100})$$

We see in the DCP seen in Figure 3.4, that there are two simple acyclic paths starting from $l3$ and taking only edges of $\mathcal{C}(i_{+100})$: $l3 \to l5$ and $l3 \to le$. So, we have $\mathcal{A}(l1 \to l3, \mathcal{C}(i_{+100})) = \{l3 \to l5, l3 \to le\}$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l5}) &= i_{+100} + 100 + \max(\delta_{l3\to l5}(i_{+100}), \delta_{l3\to le}(i_{+100}))\\ &= i_{+100} + 100 + \max(100, 0)\\ &= i_{+100} + 200\end{aligned}$$

As i_{+100} is not defined at the function header, we assume it to be 0. So, we get finally the constant bound 200 instead of the linear bound $n * 100$.

$$\texttt{PB}_\zeta(\pi_{l5}) = 200.$$

In fact, the loop-path bound for π_{l5} in the original program is 100. This imprecision comes from our abstraction, as we have to use the norm $i_{+100} := i + 100$ such that i_{+100} is always positive. In our implementation we get the bound 100, as we allow norms to map to the integers (see Chapter 4).

3.4.2 Restarting-Increment Rule

Example $Ex2$ given in Figure 1.1 in the introduction section needs an extension for the bound algorithms given in Definition 3.2, such that we infer the linear bound n for the loop at line 8

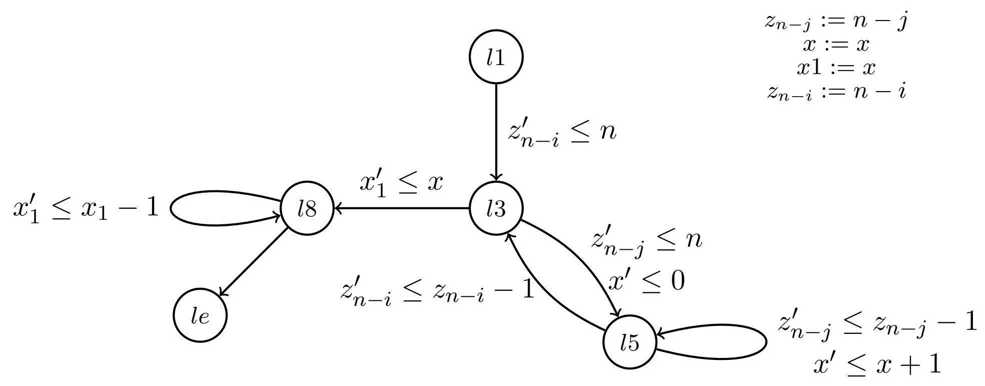


Figure 3.5: A DCP representing example $Ex2$ given in the introduction 1.1, which requires the Restarting-Increment Rule to obtain the linear bound n for the loop at line 8 instead of the quadratic bound n^2. We omit transition predicates like $x' \leq x$. On the right top, the used norms are defined.

instead of the quadratic bound n^2, which would be the result given by function PB.

In Figure 3.5, we show the abstract program model for example $Ex2$. The original function PB computes a quadratic bound, because the increment $x' \leq x + 1$ on edge $l5 \to l5$ can happen n^2 times. If we consider that x is reset to 0 on the edge from $l3 \to l5$, then we conclude that the upper bound for x is only n.

The Restarting-Increment Rule says that if there is an increment of a variable inside a loop and that variable is reset every time before reentering that loop (i.e. restarted), then we can multiply the increment by a modified loop-path bound such that the edge, which resets the variable, is considered to be taken only one time.

So, in our example we are allowed to multiply the increment 1 of variable x with a modified loop-path bound of loop at line 5 (i.e., the loop incrementing x by 1) such that the edge $l3 \to l5$ with reset $x' \leq 0$ is considered to be taken only one time. As a result, we get the modified loop-path bound n of loop at line 5 and further the loop-path bound n for loop at line 8.

We give a formal implementation of the Restarting-Increment Rule in Definition 3.15. We assume to have given a difference constraint program $\Delta\mathcal{P}(L, E)$ over a set of variables Var with entry location l_0 and exit location l_e.

Definition 3.15 (Restarting-Increment Rule extension to VB)**.** *Let $\zeta : \mathcal{L} \to Var$ be a local ranking mapping. We define*
$\mathtt{VB}_\zeta : Var \mapsto Expression(Var)$ *and* $\mathtt{PB}_\zeta : (\mathcal{L} \times (Var \cup \{\varepsilon\})) \mapsto Expression(Var)$ *as:*
$\mathtt{VB}_\zeta(v) = \mathtt{Increment}_{\mathtt{VB}}(v) + \max(v, \max\limits_{\tau \in \mathcal{R}(v)} \mathtt{Reset}(\tau, v))$
$\mathtt{PB}_\zeta(\pi, v) = \mathtt{Increment}_{\mathtt{PB}}(\zeta(\pi), v) + \zeta(\pi) + \sum\limits_{\tau \in \mathcal{R}(\zeta(\pi))} \mathtt{TB}_{\mathtt{PB}}(\tau, v) \times \mathtt{Reset}(\tau, \zeta(\pi))$
where

- $\mathtt{Increment}_{\mathtt{VB}}(v) = \sum\limits_{\tau \in \mathcal{C}(v)} \mathtt{TB}_{\mathtt{VB}}(\tau, v) \times \max(\delta_\tau(v), 0)$

- $\texttt{Increment}_{\texttt{PB}}(v, v_i) = \sum_{\tau \in \mathcal{C}(v)} \texttt{TB}_{\texttt{PB}}(\tau, v_i) \times \max(\delta_\tau(v), 0)$
- $\texttt{Reset}(\tau, v) = \texttt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)$
- $\mathcal{L}_{Restart}(\tau, v_i) = \{\pi \in \mathcal{L}_\vee(\tau) \mid v_i \neq \varepsilon \wedge \pi \textit{ resets } v_i\}$
- $\texttt{TB}_{\texttt{VB}}(\tau, v) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \texttt{PB}_\zeta(\pi, v) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & otherwise \end{cases}$
- $\texttt{TB}_{\texttt{PB}}(\tau, v_i) = \max(1, \sum_{\pi \in \mathcal{L}_\vee(\tau) \setminus \mathcal{L}_{Restart}(\tau, v_i)} \texttt{PB}_\zeta(\pi, \varepsilon)) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & otherwise \end{cases}$

Example

We apply the bound algorithms given in Definition 3.15 on the abstract program model for example $Ex2$ 1.1 given in Figure 3.5. We want to compute a loop-path bound for the loop $\pi_{l8} = l8 \to l8$ at line 8.

$$\texttt{PB}_\zeta(\pi_{l8}, \varepsilon) = \texttt{Increment}_{\texttt{PB}}(\zeta(\pi_{l8}), \varepsilon) + \zeta(\pi_{l8}) + \sum_{\tau \in \mathcal{R}(\zeta(\pi_{l8}))} \texttt{TB}_{\texttt{PB}}(\tau, \varepsilon) \times \texttt{Reset}(\tau, \zeta(\pi_{l8}))$$

We assume that $\zeta(\pi_{l8}) = x1$.

$$\texttt{PB}_\zeta(\pi_{l8}, \varepsilon) = \texttt{Increment}_{\texttt{PB}}(x1, \varepsilon) + x1 + \sum_{\tau \in \mathcal{R}(x1)} \texttt{TB}_{\texttt{PB}}(\tau, \varepsilon) \times \texttt{Reset}(\tau, x1)$$

We set in sub-definitions.

$$\texttt{PB}_\zeta(\pi_{l8}, \varepsilon) = \sum_{\tau \in \mathcal{C}(x1)} \texttt{TB}_{\texttt{PB}}(\tau, \varepsilon) \times \max(\delta_\tau(x1), 0) + x1 + \sum_{\tau \in \mathcal{R}(x1)} \texttt{TB}_{\texttt{PB}}(\tau, \varepsilon) \times (\texttt{VB}_\zeta(\gamma_\tau(x1)) + \delta_\tau(x1))$$

We observe from the DCP seen in Figure 3.5, that $\mathcal{C}(x1) = \{l8 \to l8\}$ and $\mathcal{R}(x1) = \{l3 \to l8\}$.

$$\texttt{PB}_\zeta(\pi_{l8}, \varepsilon) = \texttt{TB}_{\texttt{PB}}(l8 \to l8, \varepsilon) \times \max(-1, 0) + x1 + \texttt{TB}_{\texttt{PB}}(l3 \to l8, \varepsilon) \times (\texttt{VB}_\zeta(x) + 0)$$

As we see in Figure 3.5 that $l3 \to l8$ is not part of any loop, i.e., $\mathcal{L}_\vee(l3 \to l8) = \emptyset$, but lies on a simple path $l1 \to l3 \to l8 \to le$ from the start to the end-location. So, $\texttt{TB}_{\texttt{PB}}(l3 \to l8, \varepsilon) = 1$.

$$\texttt{PB}_\zeta(\pi_{l8}, \varepsilon) = x1 + \texttt{VB}_\zeta(x)$$

We start a sub-computation to compute an upper bound for x.

$$\texttt{VB}_\zeta(x) = \texttt{Increment}_{\texttt{VB}}(x) + \max(x, \max_{\tau \in \mathcal{R}(x)} \texttt{Reset}(\tau, x))$$

We set in sub-definitions.

$$\texttt{VB}_\zeta(x) = \sum_{\tau \in \mathcal{C}(x)} \texttt{TB}_{\texttt{VB}}(\tau, x) \times \max(\delta_\tau(x), 0) + \max(x, \max_{\tau \in \mathcal{R}(x)} (\texttt{VB}_\zeta(\gamma_\tau(x)) + \delta_\tau(x))$$

We observe from the DCP seen in Figure 3.5, that $\mathcal{C}(x) = \{l5 \to l5\}$ and $\mathcal{R}(x) = \{l3 \to l5\}$. Further, we note that $l5 \to l5$ is not part of a simple path form the start to the end.

$$\begin{aligned}\mathtt{VB}_\zeta(x) &= \mathtt{TB}_{\mathtt{VB}}(l5 \to l5, x) \times \max(1,0) + \max(x, \mathtt{VB}_\zeta(0)+0)\\ &= \mathtt{PB}_\zeta(l5 \to l5, x) + \max(x,0)\end{aligned}$$

We start a second sub-computation to compute a 'partial' loop bound for the loop $\pi_{l5} = l5 \to l5$ at line 5, such that loop-paths, which reset both $\zeta(\pi_{l5})$ and x, are considered to be taken only one time. We assume that $\zeta(\pi_{l5}) = z_{n-j}$.

$$\mathtt{PB}_\zeta(\pi_{l5}, x) = \mathtt{Increment}_{\mathtt{PB}}(z_{n-j}, x) + z_{n-j} + \sum_{\tau \in \mathcal{R}(z_{n-j})} \mathtt{TB}_{\mathtt{PB}}(\tau, x) \times \mathtt{Reset}(\tau, z_{n-j})$$

We set in sub-definitions.

$$\begin{aligned}\mathtt{PB}_\zeta(\pi_{l5}, x) = &\sum_{\tau \in \mathcal{C}(z_{n-j})} \mathtt{TB}_{\mathtt{PB}}(\tau, x) \times \max(\delta_\tau(z_{n-j}), 0) + z_{n-j} +\\ &\sum_{\tau \in \mathcal{R}(z_{n-j})} \mathtt{TB}_{\mathtt{PB}}(\tau, x) \times (\mathtt{VB}_\zeta(\gamma_\tau(z_{n-j})) + \delta_\tau(z_{n-j}))\end{aligned}$$

We observe from the *DCP* seen in Figure 3.5, that $\mathcal{C}(z_{n-j}) = \{l5 \to l5\}$ and $\mathcal{R}(z_{n-j}) = \{l3 \to l5\}$. Further, we note that $l3 \to l5$ is not part of a simple path from the start to the end.

$$\begin{aligned}\mathtt{PB}_\zeta(\pi_{l5}, x) &= \mathtt{TB}_{\mathtt{PB}}(l5 \to l5, x) \times \max(-1,0) + z_{n-j} + \mathtt{TB}_{\mathtt{PB}}(l3 \to l5, x) \times (\mathtt{VB}_\zeta(n)+0)\\ &= z_{n-j} + \max(1, \sum_{\pi \in \mathcal{L}_\vee(l3 \to l5) \setminus \mathcal{L}_{Restart}(l3 \to l5, x)} \mathtt{PB}_\zeta(\pi, \varepsilon)) \times n\end{aligned}$$

We observe from the *DCP* seen in Figure 3.5, that $l3 \to l5$ is part of the loop-path $l3 \to l5 \to l3$, on which also x is reset. Hence, we get: $\mathcal{L}_\vee(l3 \to l5) = \{l3 \to l5 \to l3\}$ and $\mathcal{L}_{Restart}(l3 \to l5, x) = \{l3 \to l5 \to l3\}$.

$$\begin{aligned}\mathtt{PB}_\zeta(\pi_{l5}, x) &= z_{n-j} + \max(1, \sum_{\pi \in \emptyset} \mathtt{PB}_\zeta(\pi, \varepsilon)) \times n\\ &= z_{n-j} + n\end{aligned}$$

We continue our variable bound computation for x.

$$\mathtt{VB}_\zeta(x) = z_{n-j} + n + \max(x,0)$$

We continue our loop-path bound computation for loop at line 8.

$$\mathtt{PB}_\zeta(\pi_{l8}, \varepsilon) = x1 + z_{n-j} + n + \max(x,0)$$

We eliminate $x, x1$, and z_{n-j} by setting them to 0, because they are not defined at the function header. We get finally the desired linear bound n for the loop at line 8.

$$\mathtt{PB}_\zeta(\pi_{l8}, \varepsilon) = n$$

3.4.3 Lost-Increment-Of-Reset Rule

In Figure 3.6, we see on the left an example that requires an extension of the bound algorithms 3.2. The loop at line 8 has a linear bound n, but as the original bound algorithm multiplies the reset of $r1$ to $r0$ on line 7 by the transition bound n of the edge $l8 \to l12$, we get the quadratic bound n^2 as n is the upper bound of $r0$. The key to a linear bound is to infer that $r0$ is reset to 0 on the same loop-path that resets $r1$ to $r0$. So, the increment of $r0$ at line 5 is lost after the loop at line 8, and we do not have to multiply it by the transition bound n of $l3 \to l8$ containing the reset $r0' \leq r1$.

In Definition 3.16, we give an extended PB function, that in contrast to the original version does not multiply an entire variable bound of a reset of a loop variable by the transition bound of the reset edge, but instead splits a reset into its reset and increment parts.

```
1  rrule(int n) {
2    int r0 = 0, r1;
3    while (n > 0) {
4      if (?)
5        r0++;
6      else {
7        r1 = r0;
8        while (r1 > 0 && ?)
9          r1--;
10       r0 = 0;
11     }
12     n--;
13   }
14 }
```

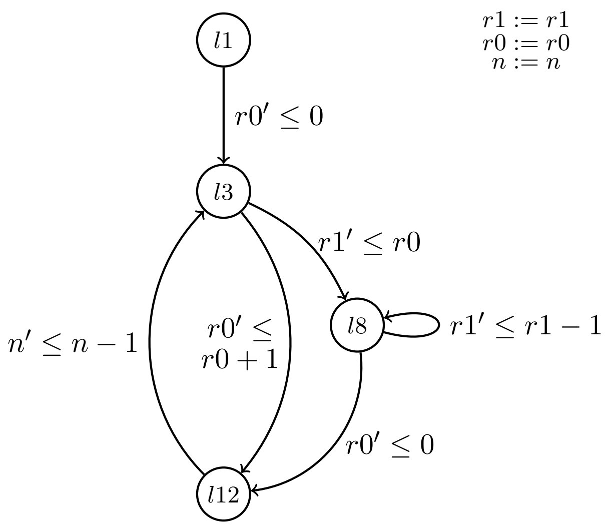


Figure 3.6: By the Lost-Increment-Of-Reset Rule, we get a linear bound for the loop at line 8. The increment of $r0$ at line 5 'reaches' $r1$ only one time, as $r0$ is reset to 0 after the loop at line 8. We omit transition predicates like $r0' \leq r0$. On the right top, the used norms are defined.

Consider the example in Figure 3.6, the loop variable $r1$ of the loop at line 8 is reset to $r0$ on the reset edge $\tau_r = l3 \rightarrow l8$. The upper bounds of the resets of $r0$ (i.e., 0 and the initial value of $r0$), are multiplied by the transition bound of the reset edge τ_r like in the original PB function. In contrast to the original PB, the increment of $r0$ by 1 on edge $l3 \rightarrow l12$ is only multiplied with the entire transition bound of the reset edge τ_r, if τ_r is not part of any loop resetting the reset $r0$. As the only loop-path $l3 \rightarrow l8 \rightarrow l12 \rightarrow l3$ going over τ_r resets $r0$ to 0 (i.e., on edge $l8 \rightarrow l12$), we only add the increment n (as $r0$ is incremented n times by 1 on edge $l3 \rightarrow l12$) to the loop-path bound of loop at line 8, without multiplying it by the transition bound of edge τ_r.

We assume to have given a difference constraint program $\Delta\mathcal{P}(L, E)$ over a set of variables *Var* with entry location l_0 and exit location l_e.

Definition 3.16 (Lost-Increment-Of-Reset Rule extension to PB)**.** *Let $\zeta : \mathcal{L} \rightarrow Var$ be a local ranking mapping. We define*
$\texttt{VB}_\zeta : Var \mapsto Expression(Var)$ *and* $\texttt{PB}_\zeta : \mathcal{L} \mapsto Expression(Var)$ *as:*

$$\texttt{VB}_\zeta(v) = \texttt{Increment}(v) + \max(v, \max_{\tau \in \mathcal{R}(v)} \texttt{Reset}(\tau, v))$$

$$\texttt{PB}_\zeta(\pi) = \texttt{Increment}(\zeta(\pi)) + \zeta(\pi) + \sum_{\tau \in \mathcal{R}(\zeta(\pi))} \texttt{TB}(\tau) \times \texttt{ResetsOfReset}(\tau, \zeta(\pi)) + \sum_{\tau \in \mathcal{R}(\zeta(\pi))} \texttt{TB}_{\texttt{Rule}}(\tau, \zeta(\pi)) \times \texttt{IncrementOfReset}(\tau, \zeta(\pi))$$

where

- $\texttt{Increment}(v) = \sum_{\tau \in \mathcal{C}(v)} \texttt{TB}(\tau) \times \max(\delta_\tau(v), 0)$

- $\texttt{Reset}(\tau, v) = \texttt{VB}_\zeta(\gamma_\tau(v)) + \delta_\tau(v)$
- $\texttt{TB}(\tau) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \texttt{PB}_\zeta(\pi) + \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$
- $\texttt{ResetsOfReset}(\tau, v) = \sum_{\tau_0 \in \mathcal{R}(\gamma_\tau(v))} \texttt{Reset}(\gamma_{\tau_0}(\gamma_\tau(v))) + \delta_\tau(v)$
- $\texttt{IncrementOfReset}(\tau, v) = \texttt{Increment}(\gamma_\tau(v))$
- $\mathcal{L}_{Rule}(\tau, v) = \{\pi \in \mathcal{L}_\vee(\tau) \mid \pi \textit{ resets } \gamma_\tau(v)\}$
- $\texttt{TB}_{\texttt{Rule}}(\tau, v) = \max(1, \sum_{\pi \in \mathcal{L}_\vee(\tau) \setminus \mathcal{L}_{Rule}(\tau,v)} \texttt{PB}_\zeta(\pi))$
  $+ \begin{cases} 1 & \text{if } \tau \text{ is part of a simple path } l_0 \xrightarrow{u_0} \dots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$

Example

We apply Definition 3.16 on the example in Figure 3.6 to compute a bound for the loop $\pi_{l8} = l8 \to l8$ at line 8.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l8}) = {} & \texttt{Increment}(\zeta(\pi_{l8})) + \zeta(\pi_{l8}) + \\ & \sum_{\tau \in \mathcal{R}(\zeta(\pi_{l8}))} \texttt{TB}(\tau) \times \texttt{ResetsOfReset}(\tau, \zeta(\pi_{l8})) + \\ & \sum_{\tau \in \mathcal{R}(\zeta(\pi_{l8}))} \texttt{TB}_{\texttt{Rule}}(\tau, \zeta(\pi_{l8})) \times \texttt{IncrementOfReset}(\tau, \zeta(\pi_{l8}))\end{aligned}$$

We assume that $\zeta(\pi_{l8}) = r1$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l8}) = {} & \texttt{Increment}(r1) + r1 + \\ & \sum_{\tau \in \mathcal{R}(r1)} \texttt{TB}(\tau) \times \texttt{ResetsOfReset}(\tau, r1) + \\ & \sum_{\tau \in \mathcal{R}(r1)} \texttt{TB}_{\texttt{Rule}}(\tau, r1) \times \texttt{IncrementOfReset}(\tau, r1)\end{aligned}$$

We set in sub-definitions.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l8}) = {} & \sum_{\tau \in \mathcal{C}(r1)} \texttt{TB}(\tau) \times \max(\delta_\tau(r1), 0) + r1 + \\ & \sum_{\tau \in \mathcal{R}(r1)} \texttt{TB}(\tau) \times (\sum_{\tau_0 \in \mathcal{R}(\gamma_\tau(r1))} \texttt{Reset}(\gamma_{\tau_0}(\gamma_\tau(r1))) + \delta_\tau(r1)) + \\ & \sum_{\tau \in \mathcal{R}(r1)} \texttt{TB}_{\texttt{Rule}}(\tau, r1) \times \texttt{Increment}(\gamma_\tau(r1))\end{aligned}$$

We observe from the *DCP* seen in Figure 3.6, that $\mathcal{C}(r1) = \{l8 \to l8\}$ and $\mathcal{R}(r1) = \{l3 \to l8\}$.

$$\begin{aligned}\texttt{PB}_\zeta(\pi_{l8}) = {} & \texttt{TB}(l8 \to l8) \times \max(-1, 0) + r1 + \\ & \texttt{TB}(l3 \to l8) \times (\sum_{\tau_0 \in \mathcal{R}(r0)} \texttt{Reset}(\gamma_{\tau_0}(r0)) + 0) + \\ & \texttt{TB}_{\texttt{Rule}}(l3 \to l8, r1) \times \texttt{Increment}(r0) \\ = {} & r1 + \\ & \texttt{TB}(l3 \to l8) \times \sum_{\tau_0 \in \mathcal{R}(r0)} \texttt{Reset}(\gamma_{\tau_0}(r0)) + \\ & \texttt{TB}_{\texttt{Rule}}(l3 \to l8, r1) \times \texttt{Increment}(r0)\end{aligned}$$

We set in sub-definitions for `Reset` and `Increment`.

$$
\begin{aligned}
&\mathtt{PB}_\zeta(\pi_{l8}) = r1+ \\
&\qquad \mathtt{TB}(l3 \to l8) \times \sum_{\tau_0 \in \mathcal{R}(r0)} (\mathtt{VB}_\zeta(\gamma_{\tau_0}(r0)) + \delta_{\tau_0}(r0)+ \\
&\qquad \mathtt{TB}_{\mathtt{Rule}}(l3 \to l8, r1) \times \sum_{\tau \in \mathcal{C}(r0)} \mathtt{TB}(\tau) \times \max(\delta_\tau(r0), 0)
\end{aligned}
$$

We observe from the DCP seen in Figure 3.6, that $\mathcal{C}(r0) = \{l3 \to l12\}$ and $\mathcal{R}(r0) = \{l1 \to l3, l8 \to l12\}$.

$$
\begin{aligned}
&\mathtt{PB}_\zeta(\pi_{l8}) = r1+ \\
&\qquad \mathtt{TB}(l3 \to l8) \times (\mathtt{VB}_\zeta(0) + 0 + \mathtt{VB}_\zeta(0) + 0)+ \\
&\qquad \mathtt{TB}_{\mathtt{Rule}}(l3 \to l8, r1) \times \mathtt{TB}(l3 \to l12) \times \max(1, 0) \\
&\qquad = r1 + \mathtt{TB}_{\mathtt{Rule}}(l3 \to l8, r1) \times \mathtt{TB}(l3 \to l12)
\end{aligned}
$$

We observe from the DCP seen in Figure 3.6, that $l3 \to l8$ is part of the loop-path $l3 \to l8 \to l12 \to l3$, which contains the transition $l8 \to l12$ that resets the reset of $r1$, i.e., $\gamma_{l8 \to l12}(r1) = r0$. Hence, we get: $\mathcal{L}_\vee(l3 \to l8) = \{l3 \to l8 \to l12 \to l3\}$ and $\mathcal{L}_{Rule}(l3 \to l8, r1) = \{l3 \to l8 \to l12 \to l3\}$.

Further, we note that $l3 \to l8$ is not part of a simple path from the start to the end-location. So, $\mathtt{TB}_{\mathtt{Rule}}(l3 \to l8, r1) = \max(1, \sum_{\pi \in \emptyset} \mathtt{PB}_\zeta(\pi)) + 0 = 1$.

$$\mathtt{PB}_\zeta(\pi_{l8}) = r1 + \mathtt{TB}(l3 \to l12)$$

Assume that we have computed $\mathtt{TB}(l3 \to l12) = n$, then we finally get

$$\mathtt{PB}_\zeta(\pi_{l8}) = r1 + n$$

As $r1$ is not defined at the function header, we set it to 0 and we obtain the linear bound n for the loop at line 8.

$$\mathtt{PB}_\zeta(\pi_{l8}) = n$$

CHAPTER 4

Implementation

We implemented our proposed algorithm as new bound procedure for the tool LOOPUS, which determines worst-case runtime bounds of loops written in C in terms of procedure inputs, where recursion is disregarded. A version of this tool already equipped with the new bound computation is available online[1].

C is a natural choice because it is still the most widely used language for system development. C is also standard for the development of safety-critical embedded and real-time systems, for which resource bounds are of special interest.

LOOPUS is originally developed by Sinn and Zuleger at TU Vienna and is based on their work on size-change abstraction [64] and lossy vector addition systems with states [60]. See Section 1.6 for more information about previous work.

LOOPUS is built on top of the modern compiler infrastructure LLVM[2] in version 2.9 originally authored by Lattner [50]. For logical reasoning we apply Z3[3], a high-performance theorem prover being developed at Microsoft [27].

To analyze a source file written in C, the first step is to compile it to the LLVM intermediate language (LLVM IR). This task can be done by tools like clang[4]. Starting LOOPUS on such a compiled file with a function name as parameter, the following main computation steps will be taken, which are explained in the next sections.

1. Program abstraction (Section 4.1)
2. Program transformations (Section 4.2)
3. Termination analysis (Section 4.3)
4. Bound computation (Section 4.4)

[1]`http://forsyte.at/software/loopus/`
[2]`http://llvm.org/`
[3]`http://z3.codeplex.com/`
[4]`http://clang.llvm.org/`

4.1 Abstracting programs to DCPs

In this section we describe how to abstract a program to a difference-constraint program (DCP). Note that we only support reducible programs, so we first make a reducibility check of the given procedure. If a program is reducible, we start with the abstraction process.

4.1.1 Extended DCP

Definition 2.5 of a DCP was defined with the aim to keep the loop and variable bound algorithms as simple as possible. However, in order to ease the abstraction of C programs given in three-address-code [1] into DCPs, we extend Definition 2.5.

For example, if we have to abstract an assignment of the form $x = y + z$ where x, y and z are variables, we need an upper bound for either y or z in advance in order to abstract the assignment into the form of a difference constraint (i.e., $x' \leq y + c$). As we cannot use our variable bound algorithm before we actually have constructed the abstract program, we extend Definition 2.5 of a DCP such that the increment c in a transition predicate $v_2' \leq v_1 + c$ can also be a variable. So, we proceed in a two-phase approach: First, we abstract the C program to a related notion of a DCP (i.e., extended DCP). We then define loop and variable bound algorithms such that the second abstraction step from extended DCPs to DCPs with constant increments is done on the fly during bound computation.

We extend DCPs to extended DCPs defined in Definition 4.1 such that

(i) the increment c in a transition predicate $v_2' \leq v_1 + c$ can be either a constant (i.e., $\mathbb{Z}$) or a variable (i.e., Var),

(ii) variables are allowed to have negative values, i.e. the set of *valuations* of variables Var is the set $Val_{Var} = Var \to \mathbb{Z}$ of mappings from Var to the integers,

(iii) each transition $l_1 \xrightarrow{u} l_2 \in E$ is annotated with a (possibly empty) set of conditions of the form $v > 0$ where $v \in Var$, and

(iv) the number of transition predicates given on a transition is arbitrary, but there is at most one predicate for each variable.

Definition 4.1 (Extended Difference Constraint Program)**.** *An* Extended Difference Constraint Program *(DCP) is a tuple $\Delta\mathcal{P} = (L, E)$, where L is a finite set of* locations*, $E \subseteq L \times 2^{Var} \times 2^{Var \times Var \times (Var \cup \mathbb{Z})} \times L$ is a finite set of* transitions *with variables in Var. A transition $(l_1, C, T, l_2) \in E$ consists of a start location, a set of conditions, a set of predicates and an end location. A transition contains at most one predicate for each variable, i.e., if $(l_1, C, T, l_2) \in E \wedge v' \leq v_1 + c_1 \in T \wedge v' \leq v_2 + c_2 \in T$, then $v_1 = v_2 \wedge c_1 = c_2$.*

The set of valuations *of Var is the set $Val_{Var} = Var \to \mathbb{Z}$ of mappings from Var to the integers. We write $(l_1, \sigma_1) \xrightarrow{C,u} (l_2, \sigma_2)$ as part of a trace, if there is a transition $l_1 \xrightarrow{C,u} l_2 \in E$, $\sigma_1, \sigma_2 \in Val_{Var}$, $\sigma_1 \models \bigwedge_{v \in C} v > 0$, and $\sigma_2(v_2) \leq \sigma_1(v_1) + \sigma_1(c)$ for all $(v_2, v_1, c) \in u$. We write also $v_2' \leq v_1 + c$ if $(v_2, v_1, c) \in u$.*

Loop and variable bound algorithms based on extended *DCP*s are stated in Section 4.4.

As the variables of an extended *DCP* are allowed to have negative values, we also extend the range of a norm to the integers. The domain of an extended norm now consists also the program locations, which helps us to show the soundness of Algorithm 4.2 given in Section 4.2.2.

Definition 4.2 (Extended Norm)**.** *An* extended norm x *is a function that maps a pair of a location and a state to the integers, i.e.,* $x \in (\Sigma \times L) \to \mathbb{Z}$*. If* x *is not defined for a pair* (σ, l) *of a location and a state, then we write* $x(\sigma, l) = \infty$*.*

4.1.2 Data structures

We abstract programs using *pointers, arrays and data structures* by approaches like presented in [36, 53]. We introduce appropriate norms such as the length of a list, the size of an array or the number of elements in a tree. We make the following optimistic assumptions, which are reported to the user but not checked for their validity. This check could be accomplished by standard tools for shape analysis.

- Pointers do not alias.
- A recursive data structure is acyclic if a loop iterates over it.
- An array of characters is finite if an inequality check on the string termination character `'\0'` is found.

4.1.3 Guessing norms

We set up an initial set of *norms* built up by conditions used inside a loop which limit the number of times a loop-path can be executed. We consider loop-conditions e of '**while**(e)'-statements, '**for**(...; e; ...)'-statements and of ' **if**(e)'-statements when they are used within a loop.

So, we compute the set of all loop-paths (see Definition 2.13) and we add an expression e to the set of *norms* N, if e is a local ranking function (see Definition 4.4) for some loop-path.

Definition 4.3 (Guard)**.** *Let* $p = l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_{n-1}} l_n$ *be a path. We call an expression* $e > 0$ *a guard of* p *if and only if there exists at least one* ρ_i *such that* $\rho_i \models e > 0$*.*

Definition 4.4 (Local ranking function)**.** *Let* $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_{n-1}} l$ *be a simple cyclic path, and let* $\mathtt{rel}(\pi) = \rho_0 \circ \rho_1 \circ \cdots \circ \rho_{n-1}$ *be the concatenation of all transition relations along* π*. An expression* r *is a* local ranking function *for* π *if and only if*

(i) $r > 0$ *is a guard (Definition 4.3) of* π *and*

(ii) $r - r' > 0$*, where* r' *denotes the expression* r *where every variable* x *is replaced by expression* e *according to the update* $x' = e$ *of* $\mathtt{rel}(\pi)$*.*

We make global assumptions during the process of guessing norms, if we have extracted a condition $e \neq 0$. We require that either $e > 0$ or $-e > 0$ holds. We assume that $e > 0$ holds, if all loop-paths that contain $e \neq 0$ imply that e decreases on that loop-paths. If this is not the

```
void neq(int a, int b) {
  while (a != b) {
    a--;
  }
}
```

Figure 4.1: Example where we have to make assumptions such that termination is guaranteed.

```
void nd(int a, int b) {
  while (a > 0) {
    a++;
    if (?) {
      for (int i=0; i < n; i++)
        ;
    }
  }
}
```

Figure 4.2: Variable i is not defined in the outer loop-path which skips the if-branch.

case, but e increases on the loop-paths containing $e \neq 0$, we assume $-e > 0$. Otherwise, the norm e is not used. The resulting set of assumptions is reported to the user.

Figure 4.1 depicts function neq, where our heuristics establish the condition $a - b \neq 0$. As $a - b$ decreases on the loop-path that contains $a - b \neq 0$, we generate the assumption $a - b > 0$.

4.1.4 Abstracting transition conditions

In contrast to the initial definition of a DCP, an extended DCP has no implicit condition how often a transition can be executed, because the variables are over the integers instead of the natural numbers. We have to add therefore conditions to a transition. Our conditions are of the form $v > 0$.

Let $l_1 \xrightarrow{\rho} l_2$ be a transition of the original program, and let $l_1 \xrightarrow{C,u} l_2$ be its abstraction. We add a transition condition $v > 0$ for a given norm $v \in N$ to the set of transition conditions C if $v > 0$ is a guard of $l_1 \xrightarrow{\rho} l_2$.

4.1.5 Abstracting transition predicates

Starting with an initial set of guessed norms N, we construct for each norm $x \in N$ transition predicates of the form $x' \leq y + z$, describing for a transition $\tau = l_1 \xrightarrow{\rho} l_2$, how $l_1 \xrightarrow{\rho} l_2$ may change the value of x. During that process, we have to add new norms to N, for which we also start our abstraction process.

We give concrete abstraction rules for simple norms, i.e., norms that contain only one variable, in Table 4.1. We assume that the program that we want to abstract is given in three-address code.

We list explanations for some rules stated in Table 4.1:

1) We derive the predicate $x' \leq x + y_-$ for an assignment $x = x - y$ where $y_- := -y$ is a norm. To check if this derivation is correct, we have to make sure that $x' \leq x + (-y)$ is invariant on the transition, which is obviously true as we have according to the assignment $x' = x - y$ and so we have that $x' \leq x + (-y) = x - y$ is true.

2) Currently, we do not support assignments of the form $x = y - x$ or $x = -x$. If we would abstract $x = y - x$ resp. $x = -x$ to $x' \leq z_{y-x}$ resp. $x' \leq x_-$ where $z_{y-x} := y - x$ and $x_- := -x$ are norms, then we would get a cyclic dependency between the upper bounds $\texttt{VB}_\zeta(x)$ and $\texttt{VB}_\zeta(z_{y-x})$ resp. $\texttt{VB}_\zeta(x_-)$, because we get the transition predicates $z'_{y-x} \leq x$ resp. $x'_- \leq x$ for the assignments $x = y - x$ resp. $x = -x$.

3) We support the assignments $x = x * c$ and $x = x * y$ on a transition τ only if τ is not part of a loop. If τ is not part of a loop, then we can add a new variable $x1$ representing x in the control flow before transition τ. We add $x1_{*c} := x1 * c$ and $x1_{*y} := x1 * y$ to the set of norms.

4) We abstract the assignments $x = y * c$ and $x = y * z$ to $x' \leq y_{*c}$ resp. $x' \leq y_{*z}$, where $y_{*c} := y * c$ resp. $y_{*z} := y * z$ are new norms.

5) We abstract the assignment $x = a[i]$ to $x' \leq a_U$, where a_U denotes an upper bound for all values in array a, i.e., $\forall i \geq 0 : a[i] \leq a_U$. We require that the variable a and the entire array (i.e., all $a[i]$ where $i \geq 0$) are constant.

6) We abstract the field dereference $x = y.f$ to $x' \leq y_f$, where $y_f := y.f$ is a new norm. We require that the variable y is constant.

In Table 4.2 we state abstraction rules for a complex norm of the form $n - i$, which is obtained commonly from loop conditions of the form $i < n$.

If there is a variable v within a complex norm x such that v is not defined by τ, i.e., $\forall(\sigma_i, \sigma_{i+1}) \in \rho : v(\sigma_{i+1}, l_2) = \infty$, then we derive no transition predicate. Consider function nd depicted in Figure 4.2. As variable i is not defined in the outer loop-path which skips the if-branch, we do not derive a transition predicate for the norm $n - i$ for that loop-path.

4.2 Program Transformations

The program that we abstract has to be amenable (see Definition 3.8) such that our loop and variable bound algorithms are sound. We check reducibility of the program, as already noted, before we actually start the program abstraction. We can also be sure that the program has a unique, not-cyclic entry-location, because we use the function-header as entry-location. However, a program can have several exit-locations. We give in Section 4.2.1 an algorithm transforming a program with several exit-locations to a program with one exit-location.

Assignment to x on transition τ	Abstract Predicate	Additional Norms	Restriction
$x = c$	$x' \leq c$	-	-
$x = y$	$x' \leq y$	y	-
$x = x + c$	$x' \leq x + c$	-	-
$x = y + c$	$x' \leq y + c$	y	-
$x = x + y$	$x' \leq x + y$	y	-
$x = y + z$	$x' \leq y + z$	y, z	-
$x = x - y$	$x' \leq x + y_-$ 1)	$y_- := -y$	-
$x = y - z$	$x' \leq y + z_-$	$y, z_- := -z$	-
$x = -x$	$x' \leq x_{\|\|}$ 2)	$x_{\|\|} := \|x\|$	-
$x = -y$	$x' \leq y_-$	$y_- := -y$	-
$x = y - x$	not implemented 2)		
$x = x * c$	$x' \leq x1_{*c}$ 3)	$x1_{*c} := x1 * c$	$\nexists \pi \in \mathcal{L} : \tau \in \pi$
$x = x * y$	$x' \leq x1_{*y}$ 3)	$x1_{*y} := x1 * y$	$\nexists \pi \in \mathcal{L} : \tau \in \pi$
$x = y * c$	$x' \leq y_{*c}$ 4)	$y_{*c} := y * c$	-
$x = y * z$	$x' \leq y_{*z}$ 4)	$y_{*z} := y * z$	-
$x = x/c$	$x' \leq x - 1$	-	$\tau \models x > 0 \vee c > 0$
$x = x/y$	$x' \leq x - 1$	-	$\tau \models x > 0 \vee y > 0$
$x = a[i]$	$x' \leq a_U$ 5)	$a_U := \max_{0 \leq i} a[i]$	a and all $a[i]$ are constant
$x = y.f$	$x' \leq y_f$ 6)	$y_f := y.f$	y is constant

Table 4.1: The table shows rules for abstracting transition predicates for a simple norm x. The possible updates of x are assumed to be given in three-address code. The symbol c denotes a constant. The variables x and y are assumed to be different. In column "Additional Norms", we list the norms that are added to the set N.

We also have to make sure that the loop and variable bound algorithms terminate. As discussed in Section 3.3, a variable bound computation $\texttt{VB}_\zeta(x)$ does not terminate if it depends on itself. This can be the case if $\texttt{VB}_\zeta(x)$ depends another variable bound computation $\texttt{VB}_\zeta(y)$ where $\texttt{VB}_\zeta(y)$ depends also on $\texttt{VB}_\zeta(x)$. In Section 4.2.2 we discuss a program transformation which dissolve such kinds of dependencies.

4.2.1 Unique Exit Location

Algorithm 4.1 translates a program into semantically equivalent program with a unique exit location. In line 1 of Algorithm 4.1, a set is constructed which contains all exit locations of the input program. Next, the algorithm creates a new location and adds an edge from all previous exit-locations to that location such that the new location is the new unique exit location.

Assignments on transition τ	Abstract Predicate	Additional Norms
$n = n, i = 0$	$x' \leq n$	n
$n = n, i = j$	$x' \leq z_{n-j}$	$z_{n-j} := n - j$
$n = n, i = i + c$	$x' \leq x - c$	-
$n = y, i = i$	$x' \leq z_{y-i}$	$z_{y-i} := y - i$
$n = n + c, i = i$	$x' \leq x + c$	-
$n = y, i = j$	$x' \leq z_{y-j}$	$z_{y-j} := y - j$
$n = y, i = i + c$	$x' \leq z_{y-i} - c$	$z_{y-i} := y - i$
$n = n + c, i = j$	$x' \leq z_{n-j} + c$	$z_{n-j} := n - j$
$n = n + c_1, i = i + c_2$	$x' \leq x + c_3$	$c_3 := c_1 - c_2$

Table 4.2: The table shows rules for abstracting transition predicates for norms x of the form $n - i$. The symbol c denotes a constant. The variables j and y are assumed to be different. In column “Additional Norms”, we list the variables that are added to the set of norms N.

```
Procedure: UniqueExit($P$)
Input: a program $P(L, E)$
Output: $P$ with a unique exit location
1 $exitLocations := \{l \in L \mid \nexists l \xrightarrow{u} l_1 \in E : l_1 \neq l\}$
2 create new location $l_e$, add $l_e$ to $L$
3 foreach $l \in exitLocations$ do
4     add $l \to l_e$ to $E$
5 end
6 return $P$
```

Algorithm 4.1: UniqueExit transforms a program into a program with a unique exit location.

4.2.2 Cycle-free Variable Dependencies

The program transformation that we explain in the following aims to dissolve dependencies between variable bound computations. We formally define dependencies between variable bound computations in Definition 4.5.

Definition 4.5 (Variable Bound Dependencies). *Let $\Delta\mathcal{P} = (L, E)$ be a DCP over variables Var. We define relation $\mathfrak{R}\mathtt{1} \subseteq$ Var $\times$ Var as: $(v_1, v_2) \in \mathfrak{R}\mathtt{1}$ iff there exists a transition $\tau = l_1 \xrightarrow{u} l_2 \in E$ such that $v_2' \leq v_1 + c \in u \wedge v_1 \neq v_2$. Let $\mathfrak{R}\mathtt{1}^*$ be the transitive closure of $\mathfrak{R}\mathtt{1}$.*

We say that a variable bound computation $\mathtt{VB}_\zeta(v_1)$ depends on $\mathtt{VB}_\zeta(v_2)$ where $v_1, v_2 \in$ Var, if and only if $(v_1, v_2) \in \mathfrak{R}\mathtt{1}^*$.
We say that two variable bound computations $\mathtt{VB}_\zeta(v_1), \mathtt{VB}_\zeta(v_2)$ where $v_1, v_2 \in$ Var have cyclic dependencies if and only if $(v_1, v_2) \in \mathfrak{R}\mathtt{1}^*$ and $(v_2, v_1) \in \mathfrak{R}\mathtt{1}^*$.

Algorithm 4.2 applies a program transformation such that the resulting program contains no two variables $v_1, v_2 \in$ *Var* such that $\mathtt{VB}_\zeta(v_1)$ and $\mathtt{VB}_\zeta(v_2)$ have cyclic dependencies with each other, i.e., they are recursively calling each other (see Definition 4.5). We say that the gained program is *cycle-free*.

To keep the Algorithm 4.2 as simple as possible, we assume that each increment c in a predicate $v_2' \leq v_1 + c$ is a constant (note that an extended DCP does not require this), However, the algorithm could be easily extended to support variables as increments.

We show that Algorithm 4.2 is sound in Section 27, which means that a variable bound for any variable or a loop-path bound for any loop of the transformed program is also a bound of the corresponding variable resp. loop-path, We assume that an input program for Algorithm 4.2 is stratifiable (see Section 27) and fully-defined (see Section 27).

Stratifiability

The term *stratifiability* was first introduced in the field of termination analysis by Ben-Amram and Lee [9]. They worked out common forms of size-change graphs where size-change termination can be decided in polynomial time. One such form is stratifiability. Size-change termination relies on the fact that a program must terminate if there is a continuous decrease of a variable taking values on a well-founded domain. Stratifiability eases the analysis, because it disallows variables (defined at the same location) to swap their values during computation, which would require a more complex termination criterion. For example in Figure 4.3, the variable x increases towards n only in each second iteration. The program is unstratifiable, because x and y are defined at the same location and are swapping their values.

We give a formal definition of stratifiability of a program in Definition 4.8, which requires the notion of a data-flow graph (see Definition 4.6) and of a strongly connected component (see Definition 4.7).

Definition 4.6 (Data-flow graph of a DCP). *Let $\Delta\mathcal{P} = (L, E)$ be a DCP over variables Var. The* data-flow graph (DFG) *of $\Delta\mathcal{P}$ is the graph $G = (VL, E_{dfg})$ where $VL = Var \times L$ and $E_{dfg} = \{(v_1, l_1) \xrightarrow{u,c} (v_2, l_2) \mid l_1 \xrightarrow{u} l_2 \in E, v_2' \leq v_1 + c \in u\}$.*

Definition 4.7 (Strongly connected component). *A* strongly connected component (SCC) *of a graph $G(V, E)$ is a maximal set of nodes $\{n_1, ...n_k\} \subseteq V$ such that for all $1 \leq i, j \leq k$ there is a path from n_i to n_j in G or $n_i = n_j$.*

Definition 4.8 (Stratifiable program). *A program $\Delta\mathcal{P}$ is* stratifiable *if and only if for each SCC of the data-flow graph of $\Delta\mathcal{P}$ it holds, that each location l_i of $\Delta\mathcal{P}$ appears at most one time.*

Examples. The most simple pattern of unstratifiability appears if a program swaps the value of two variables either on one loop-path (see example in Figure 4.3) or on two loop-paths with the same header (see example in Figure 4.4).

In Figure 4.3, we see that $(x, l3)$ and $(y, l3)$ build an SCC within the data-flow graph depicted on the right. Thus, the program is not stratifiable, because location $l3$ appears twice in one SCC.

In Figure 4.4, the variable x and y are interchanged over two paths and location $l4$ appears twice in one SCC. In our experiments, among the few cases of unstratifiability this pattern is the most frequent one, as it models a simple backtracking mechanism.

```
**Procedure**: CycleBreak($\Delta\mathcal{P}$)
**Input**: a stratifiable difference-constraint program $\Delta\mathcal{P}$
**Data**: a map $SubstitutedVariable : (Var_2 \times L) \mapsto (Var \cup \{\varepsilon\})$
**Output**: a cycle-free difference-constraint program $\Delta\mathcal{P}_2$
1  $\Delta\mathcal{P}_2(L_2, E_2)$ := "copy of $\Delta\mathcal{P}$ over variables $Var_2$"
2  $DFG(VL, E_{dfg})$ := "data-flow graph of $\Delta\mathcal{P}_2$"
3  $SCCs_{dfg}$ := "set of all SCCs of $DFG$"
4  foreach $scc \in SCCs_{dfg}$ do
5      create new variable $v_s$, add $v_s$ to $Var_2$
       // redirect flows from outside of the SCC to $v_s$
6      foreach $(v_0, l_0) \xrightarrow{u,c} (v, l) \in E_{dfg} : (v, l) \in scc \wedge (v_0, l_0) \notin scc$ do
7          add $v'_s \leq v_0 + c$ to $u$
8          remove update of $v$ in $u$
9      end
       // let flows from inside of the SCC starting from $v_s$
10     foreach $(v, l) \xrightarrow{u,c} (v_1, l_1) \in E_{dfg} : (v, l) \in scc \wedge (v_1, l_1) \notin scc$ do
11         set update of $v_1$ in $u$ to $v'_1 \leq v_s + c$
12     end
       // redirect all inner flows of the SCC to $v_s$
13     foreach $(v_1, l_1) \xrightarrow{u,c} (v_2, l_2) \in E_{dfg} : (v_1, l_1) \in scc \wedge (v_2, l_2) \in scc$ do
14         add $v'_s \leq v_s + c$ to $u$
15         remove update of $v_2$ in $u$
16     end
       // update conditions
17     foreach $(v, l) \in scc$ and $l \xrightarrow{C,u} l_1 \in E_2$ do
18         if $v > 0 \in C$ then
19             add $v_s > 0$ to $C$
20             remove $v > 0$ from $C$
21         end
22     end
       // remember substituted variables
23     foreach $(v, l) \in scc$ do
24         $SubstitutedVariable(v_s, l) \leftarrow v$
25     end
26 end
27 return $\Delta\mathcal{P}_2$
```

Algorithm 4.2: CycleBreak transforms a stratifiable difference-constraint program into a cycle-free one. We need the map $SubstitutedVariable$ only for the soundness proof, such that we can reconsider, which variable was substituted by which variable at which location.

```
1 void strat1(int n) {
2   int x = 0, y = 0, tmp;
3   while (x < n) {
4     tmp = y;
5     y = x + 1;
6     x = tmp;
7   }
8 }
```

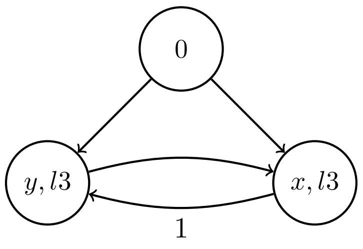


Figure 4.3: Function *strat1* shows the most simple pattern for unstratifiability: two variables are swapping their values. On the right we see the corresponding data-flow graph 4.6 with the SCC $\{(y, l3), (x, l3)\}$. For the sake of convenience, we do not annotate the edges with the transition predicates.

```
 1 void strat2(int n) {
 2   int i = n;
 3   int x = 0, y = 0, tmp;
 4   while (i > 0) {
 5     if (*)
 6       y = x + 1;
 7     else
 8       x = y;
 9   }
10 }
```

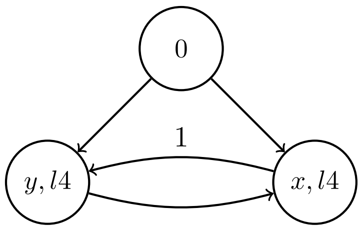


Figure 4.4: Function *strat2* depicts another simple pattern for unstratifiability: one variable stores the value of another variable, which is used later on for a possible backtracking. On the right we see the corresponding data-flow graph with the SCC $\{(y, l4), (x, l4)\}$. For the sake of convenience, we do not annotate the edges with the transition predicates.

Another pattern of unstratifiability is shown in Figure 4.5. Location $l5$ appears twice in one data-flow SCC. The problem is the non-deterministic choice allowing i either to be decremented or reset to j. As it is quite unintuitive how often the inner loop may be executed, a programmer rarely produces such a program. In our evaluation, we experienced only a few examples of this pattern.

Fully-Defined Program

To further ease the soundness proof of Algorithm 4.2 given in the next section, we exclude programs that are not considered to be analyzed. We define a *fully-defined* DCP (see Definition 4.9), such that we ensure that the computation of an extended DCP is based solely on the

```
void strat3(int n) {
  int i = n, j;
  while (i > 0) {
    j = i - 1;
    while (j > 0)
      j--;
    if (*)
      i = j;
    else
      i--;
  }
}
```

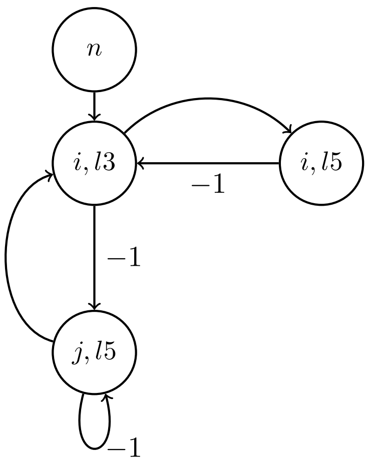


Figure 4.5: Function *strat3* shows us a more advanced pattern for unstratifiability: variable i of the outer loop can either hold its value (decremented by 1) or is reset to j, which already depends on i. On the right we see the corresponding data-flow graph 4.6 with the SCC $\{(i, l3), (i, l5), (j, l5)\}$. For the sake of convenience, we do not annotate the edges with the transition predicates.

```
void undefined() {
  int x;
  while(x > 0)
    x--;
}
```

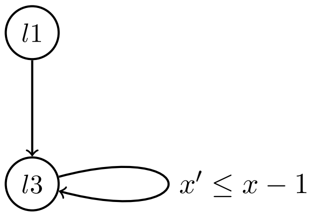


Figure 4.6: Variable x has no initial value. So, the *DCP* on the right is not fully-defined.

given start-values of the variables. A 'normal' *DCP* (see Definition 2.5) like we have defined in Section 2.2 is by definition fully-defined, as each transition must have one transition predicate per variable. As an extended *DCP* does not require one transition predicate for each variable, we exclude meaningless programs like in Figure 4.6 where the variable x has no initial value.

Definition 4.9 (Fully-defined *DCP*). *Let $\Delta\mathcal{P}(L, E)$ be a DCP with a unique start-location l_0 that is not part of any loop. We say that $\Delta\mathcal{P}$ is* fully-defined, *if and only if for each predicate $v_2' \leq v_1 + c_1$ of every transition $l_1 \xrightarrow{u} l_2 \in E$ we have that $\forall l_3 \xrightarrow{u_3} l_1 \in E : \exists v_3, c_3 : v_1' \leq v_3 + c_3 \in u_3$.*

Soundness of the Cycle-breaking Program Transformation

In the following, we assume to have given a stratifiable, fully-defined extended *DCP* $\Delta\mathcal{P}(L, E)$ over variables *Var* with a unique entry location l_0 that is not part of any loop. Further, let

CycleBreak($\Delta\mathcal{P}$) be the program returned by Algorithm 4.2, with variables Var_2, locations L, and edges E_2. Let $DFG_2(VL_2, E_{dfg2})$ be the data-flow graph of CycleBreak($\Delta\mathcal{P}$).

Lemma 4.10. *If $\Delta\mathcal{P}$ is stratifiable, then $SubstitutedVariable$ is only one time updated by algorithm 4.2 for a pair of a variable $v_s \in \mathit{Var}_2$ and a location $l \in L$.*

Proof. Let $v_s \in \mathit{Var}_2$. Let scc be the SCC of the data-flow graph of $\Delta\mathcal{P}$ for that Algorithm 4.2 created v_s. Assume that there are two nodes within scc s.t. $(x, l_1), (y, l_1)$. Then $SubstitutedVariable$ would be updated twice for the pair (v_s, l_1), but l_1 would appear twice within scc, which would mean that $\Delta\mathcal{P}$ is not stratifiable. $\square$

Observation 4.11. *Given an edge $l_1 \xrightarrow{u_2} l_2 \in E_2$, and a predicate $v_2' \leq v_1 + c \in u_2$, we observe that $SubstitutedVariable(v_2, l_2) \neq \varepsilon$ and $SubstitutedVariable(v_1, l_1) \neq \varepsilon$.*

Theorem 4.12. *CycleBreak$(\Delta\mathcal{P})$ is cycle-free.*

Proof. W.l.o.g. assume that there are two variables $v_1, v_2 \in \mathit{Var}$ such that $\mathtt{VB}_\zeta(v_1)$ and $\mathtt{VB}_\zeta(v_2)$ are cyclic with each other. This means according to the Definition 4.5, that $v_1 \neq v_2$, that there exists an edge $l_1 \xrightarrow{u} l_2 \in E_2$ s.t. $v_2' \leq v_1 + c \in u$, and that there exists an edge $l_3 \xrightarrow{u} l_4 \in E_2$ s.t. $v_1' \leq v_2 + c \in u$.

By Observation 4.11, we get that $SubstitutedVariable(v_2, l_2) \neq \varepsilon$, $SubstitutedVariable(v_2, l_3) \neq \varepsilon$, $SubstitutedVariable(v_1, l_1) \neq \varepsilon$, and $SubstitutedVariable(v_1, l_4) \neq \varepsilon$.
By Lemma 4.10, we get that $SubstitutedVariable(v_2, l_2)$, $SubstitutedVariable(v_2, l_3)$, $SubstitutedVariable(v_1, l_1)$ and $SubstitutedVariable(v_1, l_4)$ map to exactly one variable.
W.l.o.g. let $x_1 = SubstitutedVariable(v_2, l_2)$, $x_2 = SubstitutedVariable(v_2, l_3)$, $y_1 = SubstitutedVariable(v_1, l_1)$ and $y_2 = SubstitutedVariable(v_1, l_4)$.

(i) By definition of Algorithm 4.2, we easily see that $SubstitutedVariable$ is only updated for v_2 and locations l_2, l_3 if (x_1, l_2) and (x_2, l_3) are part of an SCC of the DFG of $\Delta\mathcal{P}$. So similarly, also (y_1, l_1) and (y_2, l_4) are part of an SCC of the DFG of $\Delta\mathcal{P}$.

(ii) By definition of Algorithm 4.2, we easily observe from $l_1 \xrightarrow{u} l_2 \in E_2$ s.t. $v_2' \leq v_1 + c \in u$ that there must be an edge $l_1 \xrightarrow{u} l_2 \in E$ s.t. $x_1' \leq y_1 + c \in u$. Similarly, we get by the fact that $l_3 \xrightarrow{u} l_4 \in E_2$ s.t. $v_1' \leq v_2 + c \in u$ that there must be an edge $l_3 \xrightarrow{u} l_4 \in E$ s.t. $y_2' \leq x_2 + c \in u$. So, we have the edges $(y_1, l_1) \to (x_1, l_2)$ and $(x_2, l_3) \to (y_2, l_4)$ within the DFG of $\Delta\mathcal{P}$.

By (i) and (ii) we have that (x_1, l_2), (x_2, l_3), (y_1, l_1) and (y_1, l_4) are part of the same SCC of the DFG of $\Delta\mathcal{P}$. But as Algorithm 4.2 creates only one variable per SCC, v_1 and v_2 cannot be different, which stands in contrast to our assumption that $v_1 \neq v_2$. $\square$

Similarly to our original Definition 2.20 under which conditions a *DCP* is an abstraction of a program, we define now the conditions under an extended *DCP* is an abstraction of a program.

Definition 4.13 (Extended Abstraction of a Program)**.** *An extended DCP $\Delta\mathcal{P} = (L, E')$ with variables Var is an* abstraction *of a program $P = (L, E)$ if and only if*

(1) every $x \in$ Var is an extended norm of P, and

(2) for each transition $l_1 \xrightarrow{\rho} l_2 \in E$ there is a transition $l_1 \xrightarrow{C,u} l_2 \in E'$ such that every $x' \leq y + c \in u$ and $x > 0 \in C$ is invariant for $l_1 \xrightarrow{\rho} l_2$.

Theorem 4.14 (Soundness)**.** *CycleBreak$(\Delta\mathcal{P})$ is an abstraction of $\Delta\mathcal{P}$ according to Definition 4.13, if we interpret each variable $v_i \in Var_2$ as an extended norm 4.2 of $\Delta\mathcal{P}$ such that*
$$v_i(\sigma, l) = \begin{cases} \sigma(v) & \text{if } SubstitutedVariable(v_i, l) = v \wedge v \neq \varepsilon \\ \infty & \text{otherwise} \end{cases}$$
where $\sigma \in \Sigma$ is a state of $\Delta\mathcal{P}$, $l \in L$, and $SubstitutedVariable$ is created by calling algorithm CycleBreak$(\Delta\mathcal{P})$.

Proof. Every variable $v_i \in Var_2$ is an extended norm of $\Delta\mathcal{P}$. Hence, condition 1 of Definition 4.13 is satisfied.

Let $l_1 \xrightarrow{C,u} l_2 \in E$ be an arbitrary transition of $\Delta\mathcal{P}$, and let $l_1 \xrightarrow{C_2,u_2} l_2 \in E_2$ be the corresponding transition of CycleBreak$(\Delta\mathcal{P})$, which must exist as the edges of $\Delta\mathcal{P}_2$ are a copy of $\Delta\mathcal{P}$ (see line 1 of Algorithm 4.2). We have to show that each $v_i \leq v_j + c \in u_2$ and each $v_i \in C_2$ is invariant for $l_1 \xrightarrow{C,u} l_2$. Let $(l_1, \sigma_1) \xrightarrow{C,u} (l_2, \sigma_2)$ be an arbitrary trace of $\Delta\mathcal{P}$.

Let $v_2' \leq v_1 + c \in u_2$. We have to make sure that $v_2(\sigma_2, l_2) \leq v_1(\sigma_1, l_1) + c$ holds. By Observation 4.11, we get that $SubstitutedVariable(v_2, l_2) \neq \varepsilon$ and $SubstitutedVariable(v_1, l_1) \neq \varepsilon$. By Lemma 4.10, we get that $SubstitutedVariable(v_2, l_2)$ and $SubstitutedVariable(v_1, l_1)$ map to exactly one variable. W.l.o.g. let $x = SubstitutedVariable(v_2, l_2)$ and $y = SubstitutedVariable(v_1, l_1)$. We get by definitions of the norms v_2 and v_1: $v_2(\sigma_2, l_2) = \sigma_2(x)$ and $v_1(\sigma_1, l_1) = \sigma_1(y)$. So, we have to show that $\sigma_2(x) \leq \sigma_1(y) + c$. By definition of Algorithm 4.2 there must be a predicate $x' \leq y + c \in u$, as otherwise $v_2' \leq v_1 + c$ would not have been created. Hence, $\sigma_2(x) \leq \sigma_1(y) + c$ and $v_2(\sigma_2, l_2) \leq v_1(\sigma_1, l_1) + c$ hold.

It remains to show that each $v_i \in C_2$ is invariant for $l_1 \xrightarrow{C,u} l_2$. Let $v_1 \in C_2$. We have to make sure that $v_1(\sigma_1, l_1) > 0$. By Observation 4.11, we get that $SubstitutedVariable(v_1, l_1) \neq \varepsilon$. By Lemma 4.10, we get that $SubstitutedVariable(v_1, l_1)$ maps to exactly one variable. W.l.o.g. let $x = SubstitutedVariable(v_1, l_1)$. By definition of the norm v_1, we have $v_1(\sigma_1, l_1) = \sigma_1(x)$. So, we have to show that $\sigma_1(x) > 0$. By definition of Algorithm 4.2 there must be a condition $x > 0 \in C$, as otherwise $v_1 > 0$ would not have been created. Hence, $\sigma_1(x) > 0$ and $v_1(\sigma_1, l_1) > 0$ hold. □

Corollary 4.15. *Following Theorem 4.14, a loop-path bound for a loop in CycleBreak$(\Delta\mathcal{P})$ is a loop-path bound for that loop in $\Delta\mathcal{P}$.*

Examples and Localization

In Figure 4.7, we show two possible abstract programs for example $Ex6$ given in Figure 1.1. We see on the top a version before running Algorithm 4.2 and below after running it. In the version below the variable bound computations $\texttt{VB}_\zeta(x)$ and $\texttt{VB}_\zeta(y)$ are not depending on each other.

In Figure 4.8, we show two examples, where Algorithm 4.2 does not have to resolve cyclic variable bound computations, but we get more precise bounds than on the original abstract program. We benefit from the property that in the transformed program a variable represents one original variable only at a specific set of locations and that an upper bound for such a *'localized'* variable may be lower than the upper bound for the variable in the entire program.

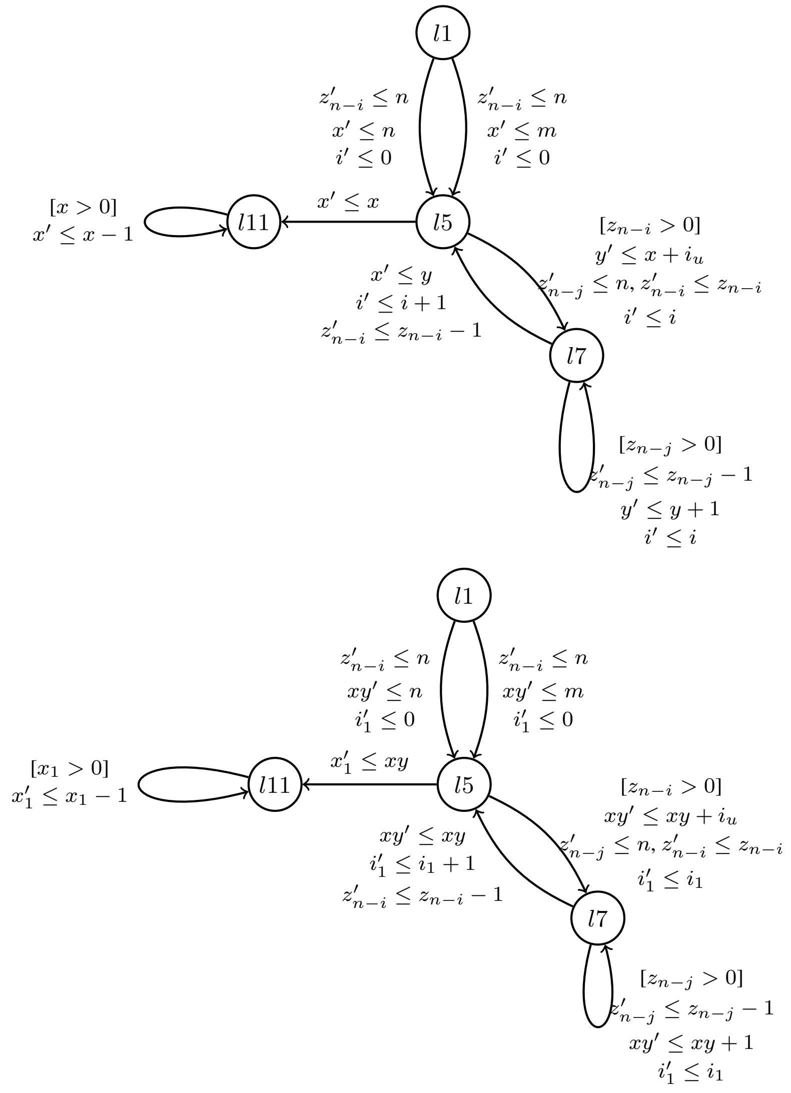
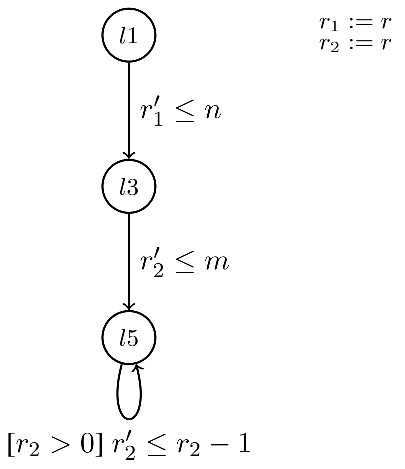


Figure 4.7: On the top, we see an abstract program for function Ex6 given in Figure 1.1 before running Algorithm 4.2 and below we see the abstract program after running Algorithm 4.2 on it. Note that in the program transformed by Algorithm 4.2 no two variables v_1 and v_2 exist such that the variable bound computations for v_1 and v_2 are depending on each other.

```
1  void mlocal(int n, m){
2    int r = n;
3    if (r > m) {
4      r = m;
5      while (r > 0)
6        r--;
7    }
8  }
```



```
1  void lin(int n, m) {
2    int x = 0;
3    while(n > 0 && ?) {
4       x = n;
5       n--;
6     }
7    while(x > 0)
8       x--;
9  }
```

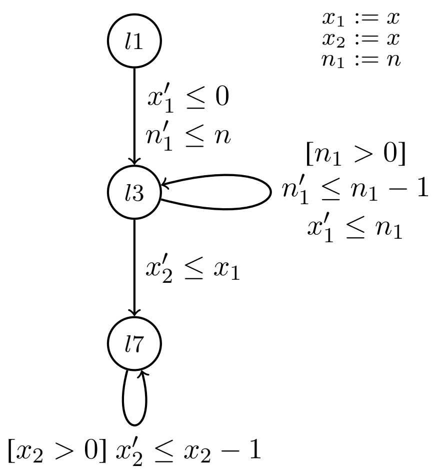


Figure 4.8: Examples which show that variables are 'localized' after applying Algorithm 4.2, and that bounds for localized variables are more precise. If we would consider in example $mlocal$ the assignment $r = n$ as a reset for r at location 5, then we would get the bound $max(n, m)$ instead of m. If we would consider in example lin the assignment $x = n$ as a direct reset of x at location $l2$, then we would get a quadratic bound n^2 instead of n, because the reset for x to n can happen n times. On the right top, the used norms are defined.

4.3 Termination Analysis

We use Algorithm 4.3 as an heuristic to establish a mapping ζ from the loop-paths of $\Delta\mathcal{P}$ to the variables Var such that ζ satisfies for every loop-path π that

(i) $\pi \models \zeta(\pi) > 0$ (Bounded) and

(ii) $\pi \models \zeta(\pi)' < \zeta(\pi)$ (Ranking).

I.e., π cannot be taken infinitely often without taking other loop-paths. However, there can still exist an infinite trace of the program, if there are two loop-paths π_1 and π_2 alternately increasing their loop variables $\zeta(\pi_1)$ and $\zeta(\pi_2)$, i.e., $\pi_1 \models \zeta(\pi_2)' > \zeta(\pi_2)$ and $\pi_2 \models \zeta(\pi_1)' > \zeta(\pi_1)$. So, the loop-path bound computations $\texttt{PB}_\zeta(\pi_1)$ and $\texttt{PB}_\zeta(\pi_2)$ are recursively calling each other and the bound computation does not terminate. In order to eliminate such kind of nonterminating loop-path bound computation, we establish a lexicographic order between the loop-paths such that

(iii) $\langle x_1, \ldots, x_n \rangle$ is an n-tuple of variables and for $j < i$ where $\zeta(\pi_i) = x_i$ we have that $\forall \tau \in \pi_i : \tau \models x'_j \leq x_j$ (Unaffecting).

In fact, Algorithm 4.3 constructs a lexicographic ranking function (see Definition 4.16). A lexicographic ranking function witnesses the termination of a program [12]. I.e., Algorithm 4.3 establishes a termination analysis.

Definition 4.16 (Lexicographic Ranking Function). *A* Lexicographic Ranking Function *for a difference-constraint program $\Delta\mathcal{P}$ over variables Var is an n-tuple of variables $\langle x_1, \ldots, x_n \rangle$ where $x_i \in Var$, such that for each loop-path π of $\Delta\mathcal{P}$ for some $i \in \{1, \ldots, n\}$,*

- *(Bounded) $\pi \models x_i > 0$;*
- *(Ranking) $\pi \models x'_i < x_i$;*
- *(Unaffecting) for $j < i$, $\forall \tau \in \pi : \tau \models x'_j \leq x_j$.*

4.4 Bound Computation

In our implementation, we use the loop and variable bound algorithms for extended *DCP*s given in Definition 4.18. We integrated the extensions (see Section 3.4) for non-monotonic in- or decrease, the “Restarting-Increment” rule and the “Lost-Increment-Of-Reset” rule.

As already noted in Section 4.3, we get a mapping ζ from the loop-paths of $\Delta\mathcal{P}$ to the variables by Algorithm 4.3. However, also if Algorithm 4.3 returns a lexicographic ranking function, our bound algorithm may not terminate.

Figure 4.9 shows a function that terminates on all inputs and we get the lexicographic ranking function $\langle a, x, y \rangle$ for the loop-paths $\pi_1 = l_1 \to l_2 \to l_3 \to l_1$, $\pi_2 = l_2 \to l_2$ and $\pi_3 = l_3 \to l_3$ in that order. Anyway, our bound algorithm does not terminate, because we have that loop-path bound $\texttt{PB}_\zeta(\pi_2)$ depends on $\texttt{VB}_\zeta(b)$, variable bound $\texttt{VB}_\zeta(b)$ depends on $\texttt{PB}_\zeta(\pi_3)$, and loop-path

```
Procedure: Ranking($\Delta\mathcal{P}$)
Input: a difference-constraint program $\Delta\mathcal{P}$
Output: a lexicographic ranking function $l$, which has one component for every
        loop-path $\pi$ of $\Delta\mathcal{P}$; a mapping $\zeta$ from the loop-paths to the variables such that
        $\zeta(\pi)$ is the component of $\pi$ in $l$
1  $\mathcal{L}$ := "set of all loop-paths of $\Delta\mathcal{P}$"
2  $l$ := "lexicographic ranking function with no components"
3  $\zeta$ := "mapping from the loop-paths to the variables"
4  while there is a $\pi \in \mathcal{L}$ and a variable $x$ such that $\pi \models x' < x \wedge x > 0$ and for all $\pi' \in \mathcal{L}$
   we have $\forall \tau \in \pi : \tau \models x' \leq x \vee x$ is not defined on $\tau$ do
5      $\mathcal{L} := \mathcal{L} \setminus \pi$
6      $l$ := $l$.append($x$)
7      $\zeta(\pi) := x$
8  end
9  if $\mathcal{L} = \emptyset$ then return $l, \zeta$
10 else return "$P$ maybe non-terminating"
```
Algorithm 4.3: Ranking computes a lexicographic ranking function for a given difference-constraint program.

bound $\texttt{PB}_\zeta(\pi_3)$ depends on $\texttt{PB}_\zeta(\pi_2)$. So, we have to maintain a stack registering calls to $\texttt{PB}_\zeta$ and $\texttt{VB}_\zeta$, and we fail if there was already a call to a computation $\texttt{PB}_\zeta$ or $\texttt{VB}_\zeta$.

4.4.1 Bounds for extended *DCP*s

In contrast to our original Definition 2.5 of a *DCP*, an extended *DCP* (see Definition 4.1) enables the following three pitfalls, which we have to handle in the bound algorithm (see Definition 4.18) for extended *DCP*s.

- In contrast to the variable and loop bounds that we defined for our original *DCP*s, we have to encapsulate recursive bounds within $max(0, \ldots)$-expressions, because variables (and so their upper bounds) can also have negative values in an extended *DCP*, and negative bounds can falsify loop-path bounds. Namely, the definition of $\texttt{PB}_\zeta$ requires to sum up the variable bounds for all resets multiplied by the corresponding transition bounds. If such a variable bound is negative, then it would falsely equalize that sum.

- An extended *DCP* does not require that every transition must constrain each variable. So, a variable v of an extended *DCP* may not have a symbolic upper bound. We require that a *DCP* is fully-defined (see Definition 4.9). So, the value of a variable v at some location l_1 where v is used as a constraint (i.e., there exists a predicate $v_1' \leq v + c$ on a transition starting at l_1), can be expressed by the input values of a set of variables (i.e., initial state).

  Let v be variable, and let $\texttt{VB}_\zeta(v)$ be its variable bound by Definition 4.18. We claim that $\texttt{VB}_\zeta(v)$ is a variable bound of v for traces that

  (i) are valid (see Definition 3.7), and

```
void terminateButNoBound(int a, int b) {
   int x, y;
l1 : while(a > 0) {
     a--;
     x = b;
     y = 0;
l2 :   while (x > 0) {
       x--;
       y++;
     }
l3 :   while (y > 0) {
       y--;
       b++;
     }
   }
}
```

Figure 4.9: The function 'terminateButNoBound' terminates on all inputs, but the bound algorithm does not terminate on the function.

(ii) end at a location l such that $\forall l_1 \xrightarrow{u} l \in E : \exists v_1 : v' \leq v_1 + c \in u$. So, variable v is constrained by a variable v_1 on all ingoing edges $l_1 \xrightarrow{u} l$ and as the *DCP* is fully-defined, v_1 must also be constrained on all ingoing edges to l_1.

Let π be a loop-path, and let $\texttt{PB}_\zeta(v)$ be its loop-path bound by Definition 4.18. We claim that $\texttt{PB}_\zeta(\pi)$ is a loop-path bound of π for traces that are valid.
We restrict our bounds to hold only on valid traces for the same argument as for our original *DCP*s (see Section 3.1): *DCP*s and also extended *DCP*s allow traces that are infeasible for 'normal' programs.

- Extended *DCP*s allow the increment c of a predicate $v_2' \leq v_1 + c$ to be a variable. So, we have to compute an upper bound for such a variable c.

In the following, we assume to have given an extended *DCP* $\Delta\mathcal{P}(L, E)$ over a set of variables *Var* with entry location l_0 and exit location l_e.

Definition 4.17 (Parameters of an extended *DCP*)**.** *A variable $v \in Var$ is a parameter of $\Delta\mathcal{P}$ if and only if there exists an edge $l_0 \xrightarrow{u} l_1 \in E$ and a variable v_1 such that $v_1' \leq v + c \in u$.*

Definition 4.18 (Variable Upper Bound VB and Loop-path Bound PB for extended *DCP*s)**.** *Let $\zeta : \mathcal{L} \to Var$ be a local ranking mapping (Definition 3.1). We define*
$\texttt{VB}_\zeta : Var \mapsto Expression(Var)$ *and* $\texttt{PB}_\zeta : \mathcal{L} \mapsto Expression(Var)$ *as:*
$\texttt{VB}_\zeta(v) = \texttt{Increment}(v) + \max(\texttt{Init}(v), \max_{\tau \in \mathcal{R}(v)} \texttt{Reset}(\tau, v))$

```
foo(int x, int y) {
  while (x < y)
    x++;
}
```

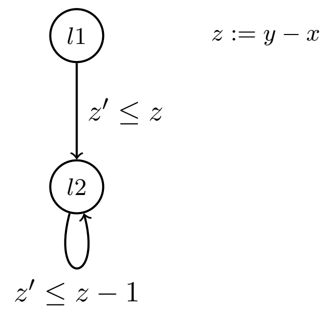


Figure 4.10: Concrete program has bound $y - x$. Abstract program has bound z.

$$\texttt{PB}_\zeta(\pi) = \texttt{Increment}(\zeta(\pi)) + \max(0, \texttt{Init}(\zeta(\pi))) + \sum_{\tau \in \mathcal{R}(\zeta(\pi))} \texttt{TB}(\tau) \times \max(0, \texttt{Reset}(\tau, \zeta(\pi)))$$

where

- $\texttt{Init}(v) = \begin{cases} v & \textit{if } v \textit{ is a parameter of } \Delta\mathcal{P} \\ \varnothing & \textit{otherwise} \end{cases}$
- $\texttt{Increment}(v) = \sum_{\tau \in \mathcal{C}(v)} \texttt{TB}(\tau) \times \max(\texttt{VB}_\zeta(\delta_\tau(v)), 0)$
- $\texttt{Reset}(\tau, v) = \texttt{VB}_\zeta(\gamma_\tau(v)) + \texttt{VB}_\zeta(\delta_\tau(v))$
- $\texttt{TB}(\tau) = \sum_{\pi \in \mathcal{L}_\vee(\tau)} \texttt{PB}_\zeta(\pi) + \begin{cases} 1 & \textit{if } \tau \textit{ is part of a simple path } l_0 \xrightarrow{u_0} \ldots \xrightarrow{u_n} l_e \\ 0 & \textit{otherwise} \end{cases}$

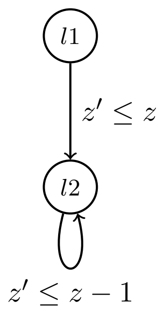
4.4.2 Bounds for Original Program

To get bounds for the original program, we have to adapt a bound given by Definition 4.18 in the following way.

We have to substitute each variable in the bound by the corresponding norm definition. For example, we get the abstract bound z for the loop in the abstract program shown on the right in Figure 4.10. To get a bound in terms of the original program inputs, we have to substitute z with its norm definition. Hence, we get a bound for the original program of $y - x$.

Also, if we abstracted operations on data structures (see Section 4.1.2), we have to substitute variables by its norm definitions. For example, we get the abstract bound z for the loop in the abstract program shown on the right in Figure 4.11. We obtain a bound for the original program by substituting z by its norm definition, i.e., $length(lst, next, 0)$. We denote by $length(lst, next, 0)$ the length of the list that (1) starts from pointer lst, (2) has elements that are connected together by the field $next$ (i.e., the next element of a pointer lst is lst->$next$), and (3) ends at the element lst where lst->$next = 0$.

```
traverse(List* lst) {
  while (lst != 0)
    lst = lst->next;
}
```



$z := length(lst, next, 0)$

Figure 4.11: Concrete program has bound $length(lst, next, 0)$. Abstract program has bound z.

CHAPTER 5

Evaluation

In this chapter, we present our experimental results obtained with our tool LOOPUS.

5.1 Comparison to Tools from the Literature

We compare LOOPUS against the tools KoAT [15], PUBS [2, 6] and RANK [7].

KoAT The KoAT tool was recently published and uses (similar to this work) runtime bounds to deduce variable bounds.

PUBS PUBS extracts cost recurrence relations out of the programs and tries to solve them by enhanced computer algebra systems.

RANK The tool RANK iteratively constructs loop bounds by solving linear constraint systems. Variable bounds are computed by abstract interpretation over the polyhedron abstract domain.

5.1.1 Benchmark Setup

We refer by the name KoAT benchmark to the set of programs that was assembled to evaluate the bound analysis tool KoAT [15]. The suite is available at their website [14] and contains examples from related literature about bound analysis [3, 6, 7, 10, 15, 32, 33, 37, 38, 64] and termination analysis [13, 30].

The benchmark is given originally in form of integer transition systems. As LOOPUS expects C programs, we translated the transition systems by script to C or by hand when the script was not sufficient. Our script failed in the case of recursive programs, but we translated them by hand if they were tail-recursive (which would be an easy extension to LOOPUS). Instead of translating the examples gathered from the T2-project, we used the goto-programs from [13], which were created from the transition systems used as input for the T2 termination prover [30] (the original sources from which the transition systems were created are not available).

The original KoAT benchmark consists of 682 examples. We excluded the 13 RAML programs from the benchmark, because we do not see a meaningful way to compare C translations to functional programs. The remaining 669 examples are left to be analyzed by the tools KoAT and PUBS. As 37 example programs are recursive and RANK does not support recursion, RANK can analyze 632 examples. We found for 27 out of the 37 recursive programs Java sources on the original web source [3]. 15 Java programs were actually not recursive and 5 tail-recursive. Hence, we translate 20 programs to non-recursive C code and we ended up with 652 example programs that can be analyzed by LOOPUS.

5.1.2 Results and Discussion

Table 5.1 states the overall results for each of the four tools. The results of our tool LOOPUS were obtained on a Linux desktop computer with 4 GB RAM and an Intel Pentium Dual-Core CPU clocked at 3.2 GHz. The results of the tools KoAT, PUBS and RANK were taken from [15] using an equally powerful machine (3.07 GHz Intel i7 CPU, 6 GB RAM). All four tools were executed with a limited time of 60 seconds for each example.

We see in the column "Bounded" of the Table 5.1 that our tool LOOPUS can bound more loops than the other tools on this benchmark. LOOPUS finishes each example, for which it computed a bound, in an average time of 0.5 seconds, which is less or equal the time needed by any other tool. As LOOPUS breaks the time-out limit of 60 seconds only for two examples, LOOPUS has also a very low average running-time in case of failed bound computations in contrast to the tools KoAT and PUBS.

Figure 5.1 shows how many examples, which are solved by LOOPUS, are also bounded by the tools KoAT, PUBS, and RANK. LOOPUS can deduce bounds for 79 examples (12 % of all 669 examples), for which no other tool was able to compute a bound. For more than 90 % of all examples bounded by any other tool than LOOPUS, LOOPUS inferred also a bound.

Table 5.2 separates the obtained bounds according to their corresponding asymptotic class. We see that the greater the asymptotic class the more loops can be bounded by LOOPUS in relation to the other tools. For example, LOOPUS computed 20 more quadratic bounds than KoAT but only 8 more constant bounds. We conclude that LOOPUS is able to handle more complex examples, because a greater asymptotic loop bound may indicate a more complex loop pattern.

In Table 5.2, we see that in contrast to PUBS our tool LOOPUS is not able to infer logarithmic or exponential bounds, but we could easily extend LOOPUS to support also such kind of bounds.

We compare the precision of the bounds generated by the four analysis tools on the asymptotical level in Table 5.3. The first sub-table shows for each asymptotic class of bounds that KoAT computed, how many fall into which class when the bound is computed by LOOPUS. We conclude that LOOPUS has inferred 21 bounds that are asymptotical more precise than the bounds from KoAT and 94 bounds for examples which KoAT was actually not able to solve. KoAT has computed 10 more precise bounds in comparison to LOOPUS and 18 bounds where LOOPUS failed. The next section will give a detailed look on those examples.

In the middle sub-table in Table 5.3 we compare LOOPUS with the tool PUBS. We conclude that LOOPUS has computed 9 bounds that are asymptotical more precise than the bounds from PUBS and 137 bounds which PUBS was actually not able to compute. PUBS has inferred 16

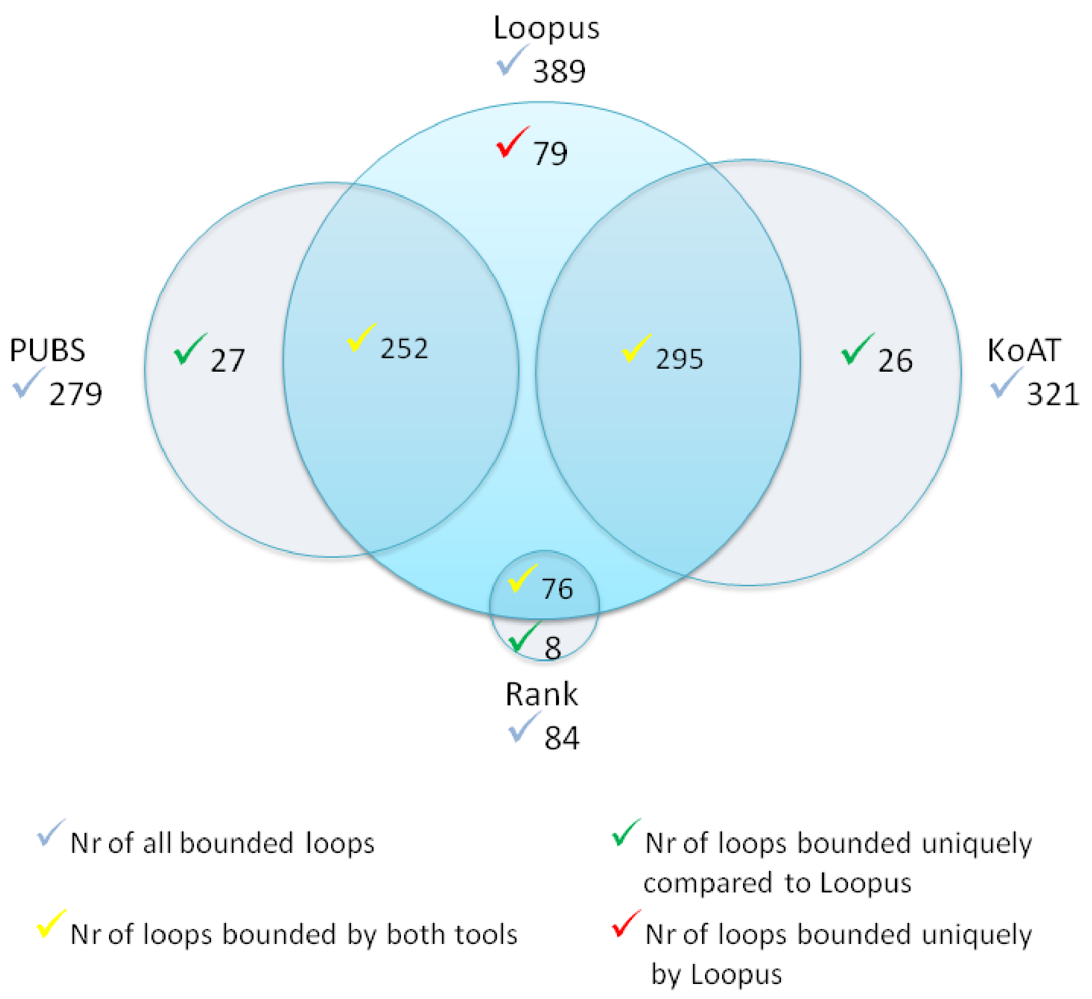

Loopus
✓ 389
✓ 79
PUBS
✓ 279
✓ 27
✓ 252
✓ 295
✓ 26
KoAT
✓ 321
✓ 76
✓ 8
Rank
✓ 84
✓ Nr of all bounded loops
✓ Nr of loops bounded uniquely compared to Loopus
✓ Nr of loops bounded by both tools
✓ Nr of loops bounded uniquely by Loopus

Figure 5.1: Diagram showing how many examples, which are solved by LOOPUS, are also bounded by the tools KOAT, PUBS, and RANK. Note that the graphic does not show the intersection between more than two tools.

	Analyzed	**Bounded**	Success	Avg-Time Success	Timeout	Failed w.o. Timeout	Avg-Time Failed
LOOPUS	652	389	60 %	0.5 s	2	261	1.3 s
KoAT	669	321	50 %	1.1 s	279	69	48.9 s
PUBS	669	279	40 %	0.8 s	58	332	10.9 s
RANK	632	84	10 %	0.5 s	6	542	0.9 s

Table 5.1: The table shows how many of the examples from the set assembled to evaluate [15] are analyzed and successfully bounded by each tool. In the column “Avg-Time Success” resp. “Avg-Time Failed”, we give the average time each tool was running on those example where the respective tool succeeded resp. failed. The number of timeouts (60 s) is given in the column “Timeout”. The number of examples, where the tool failed to compute a bound before reaching the timeout-limit is given in the column “Failed w.o. Timeout”.

	Bounded	1	$logn$	n	$nlogn$	n^2	n^3	n^4	$n^{>=5}$	EXP
LOOPUS	389	129	0	173	0	74	6	6	1	0
KoAT	321	121	0	143	0	54	0	1	2	0
PUBS	279	116	5	129	5	15	4	0	0	5
RANK	84	56	0	19	0	8	1	0	0	0

Table 5.2: The table shows how often each tool inferred a bound of a specific aymptotic class for the examples from [15]. The column headers 1, $logn$, n, $nlogn$, n^2, n^3, n^4 and $n^{>=5}$ represent the corresponding asymptotic complexity classes. EXP is the class of exponential functions.

bounds where LOOPUS failed and 37 bounds which are more precise in comparison to LOOPUS. We analyzed those 37 examples manually and explored that 10 out of them result from the fact that PUBS supports logarithmic expressions and that 27 bounds are actually not correct for the Java programs from which the transition systems are gained. Probably, the transition systems do not coincide with the Java programs. We are not able to verify if the transition systems are correct, as the transition systems are not human-readable without enormous effort.

In the third sub-table in Table 5.3 we compare LOOPUS with the tool RANK. We conclude that LOOPUS has inferred 294 bounds which RANK was actually not able to compute. RANK has computed 8 bounds where LOOPUS failed and 3 more precise bounds in comparison to LOOPUS. Two examples are from the T2 benchmark and one was actually not correct computed by RANK.

5.1.3 Detail comparison between LOOPUS and KoAT

As the results of the tool KoAT are closer to the results of LOOPUS than the results of PUBS and RANK, we choose KoAT for a more detail comparison against LOOPUS.

KoAT has computed 10 more precise bounds in comparison to LOOPUS, which can be inferred from the first sub-table of Table 5.3.

- We explored manually that 5 out of those 10 bounds were actually not correctly computed

		Loopus									
		1	$logn$	n	$nlogn$	n^2	n^3	n^4	$n^{>=5}$	EXP	No result
KoAT	1	111	0	3	0	0	0	0	0	0	5
	$logn$	0	0	0	0	0	0	0	0	0	0
	n	2	0	118	0	4	3	0	0	0	11
	$nlogn$	0	0	0	0	0	0	0	0	0	0
	n^2	0	0	16	0	35	0	0	0	0	2
	n^3	0	0	0	0	0	0	0	0	0	0
	n^4	0	0	0	0	0	1	0	0	0	0
	$n^{>=5}$	0	0	0	0	0	0	2	0	0	0
	EXP	0	0	0	0	0	0	0	0	0	0
	No result	16	0	36	0	35	2	4	1	0	245

		Loopus									
		1	$logn$	n	$nlogn$	n^2	n^3	n^4	$n^{>=5}$	EXP	No result
Pubs	1	108	0	1	0	2	0	0	0	0	4
	$logn$	0	0	3	0	0	0	0	0	0	1
	n	3	0	87	0	21	5	0	0	0	10
	$nlogn$	0	0	3	0	2	0	0	0	0	0
	n^2	0	0	3	0	10	0	0	0	0	1
	n^3	0	0	0	0	0	1	3	0	0	0
	n^4	0	0	0	0	0	0	0	0	0	0
	$n^{>=5}$	0	0	0	0	0	0	0	0	0	0
	EXP	0	0	0	0	0	0	0	0	0	0
	No result	18	0	76	0	39	0	3	1	0	247

		Loopus									
		1	$logn$	n	$nlogn$	n^2	n^3	n^4	$n^{>=5}$	EXP	No result
Rank	1	51	0	1	0	1	0	0	0	0	3
	$logn$	0	0	0	0	0	0	0	0	0	0
	n	0	0	14	0	1	0	0	0	0	4
	$nlogn$	0	0	0	0	0	0	0	0	0	0
	n^2	0	0	0	0	7	0	0	0	0	1
	n^3	0	0	0	0	0	1	0	0	0	0
	n^4	0	0	0	0	0	0	0	0	0	0
	$n^{>=5}$	0	0	0	0	0	0	0	0	0	0
	EXP	0	0	0	0	0	0	0	0	0	0
	No result	77	0	149	0	60	1	6	1	0	254

Table 5.3: In the main diagonal of each table, the number of bounds are shown that fall into the same asymptotic class. Upper the main diagonal, the number of bounds are counted where the tool noted on the left side of the table (KoAT, Pubs, and Rank) computed a more precise bound than Loopus. The same applies for Loopus under the main diagonal.

by KoAT. In 4 out of those 5 cases, the transition system analyzed by KoAT probably does not represent our C program, because we gained those 4 programs by translating the original source code (written in Java), whereas KoAT translated the transition systems used by PUBS. Interestingly, the outcome of this translation leaded to recursive programs whereas the original Java-programs are non-recursive.

- 4 out of the 10 examples where KoAT inferred a more precise bound than LOOPUS come from the T2 test suite. It is difficult to tell the reason why LOOPUS is not able to compute a more precise bound, because the goto-programs and the integer transition systems, from which the goto-programs were generated, are not human-readable.
- Another uncommon pattern can be found in the remaining example where KoAT performs better than LOOPUS. That example contains the loop

  “**while**(a>b) { tmp=a; a=b; b=tmp; }’,

  which body runs only one time. LOOPUS computes the loop bound $a - b$ and KoAT the bound 2. We think that this example is uncommon, because a programmer usually does not introduce a loop-construct that actually does not loop. However, by techniques like contextualization [64] or loop unrolling [1], we would be able to compute a constant bound.

We see in Table 5.3 that LOOPUS has inferred 21 bounds that are asymptotical more precise than the bounds from KoAT and 94 bounds which KoAT was actually not able to compute. Out of those 115 examples, 67 are from the T2 test suite, which contains as already mentioned curious goto-loops.

After looking for some simple syntactic patterns which describe the examples where LOOPUS performs better, we found out that all those examples have at least one loop with more than one path. KoAT timed out in most of the multi-path examples. We conclude that LOOPUS scales better than KoAT on the used benchmark.

5.2 Evaluation on Real-World Code

We evaluated our tool LOOPUS on three benchmarks from different areas consisting of more than 1.000.000 lines of code.

5.2.1 Benchmark Setup

cBench [26] The Collective Benchmark (cBench) is a collection of open-source sequential programs, assembled for research in program and architecture optimization.

SPEC CPU2006 [42] SPEC CPU is a set of compute-intensive benchmarks designed to test the CPU performance (we only consider those written in C).

Mälardalen WCET [40] The Mälardalen WCET benchmarks are collected by the WCET community for experimental evaluations.

5.2.2 Overall Results

Our results were obtained on a Linux desktop computer with 4 GB RAM and an Intel Pentium Dual-Core CPU clocked at 3.2 GHz. We set a timeout of 420 seconds for each example.

Table 5.4 shows how many of the analyzed loops could be bounded by LOOPUS. We distinguish between the task of computing a bound whose variables are defined at the SCC-header and whose variables are defined at the function header.

	Loops	**Bound at Function header**	Bound at SCC header
cBench	4301	3024 70 %	3174 74 %
SPEC	15392	11204 73 %	11481 75 %
WCET	262	238 91 %	243 93 %

Table 5.4: The table shows how many loops are contained within each benchmark and for how many loops our tool LOOPUS is able to compute a bound in terms function parameters. Additionally, we state how many bounds can be computed that are defined at the SCC header.

In Table 5.5 we see that more than 80 % of all loops are handled by LOOPUS in less than 1 second no matter if computing a bound for the function or the SCC header. Unsurprisingly, we see that reasoning about the entire function at once takes clearly more time. We think that there is still a great potential of a possible runtime optimization in future work.

	Avg-Time Success	Avg-Time Failed	**Avg-Time**	Time ≤ 1 s	Time > 1 s	Time > 1 min	Timeout
cBench Bound at SCC header	0.7 s	4.3 s	1.6 s	4002	285	23	0
cBench Bound at Function header	3.2 s	7.3 s	4.5 s	3568	733	83	0
SPEC Bound at SCC header	1.5 s	6.0 s	2.5 s	13674	1362	151	0
SPEC Bound at Function header	3.3 s	10.8 s	5.3 s	12070	3115	322	3
WCET Bound at SCC header	0.5 s	0.5 s	0.5 s	262	0	0	0
WCET Bound at Function header	0.8 s	0.6 s	0.7 s	262	0	0	0

Table 5.5: In the column “Avg-Time”, we give the average running time of LOOPUS on one example. The column “Avg-Time Success” resp. “Avg-Time Failed” shows the average running time of LOOPUS on the examples which LOOPUS solved resp. failed. The columns “Time ≤ 1 s”, “Time > 1 s” and “Time > 1 min” state the number of examples, where LOOPUS needed less or equal than one second, more than one second or more than one minute to give a result (i.e., a bound or a failure message). The number of timeouts (420 s) is given in the column “Timeout”.

Discussion about errors

We show in Table 5.6 reasons why LOOPUS failed to compute a bound for the examples found in our benchmarks.

The most common reason for failure is that LOOPUS cannot establish a local ranking function. We list some common shortcomings of our method in the following, which are illustrated by the examples shown in Figure 5.3.

1. Our modelling of C programs is incomplete for example in the case of bit vector arithmetic (e.g., function `bit_count` in Figure 5.3).

2. As described in the previous Chapter 4 about the implementation of our tool, we have to make assumptions about the variables and data structures of a program. For example, we have to know the sign of an integer or that a list is acyclic. Our current assumption generator is still work in progress and misses possible assumptions (e.g., functions `mainsort` and `main1` in Figure 5.3).

3. The termination of a loop may depend on side effects or return values of a function call. We already support such examples by inlining called functions. However, this feature is currently applied very restrictive (e.g., function `handle_compress` in Figure 5.3).

LOOPUS is also not able to compute a bound if a loop is irreducible. Further, LOOPUS exits on some examples anomalously due to segmentation faults, not enough memory or exceptions raised by Z3 or LLVM.

We investigated 90 examples of cBench where the bound computation of LOOPUS actually failed and found out two main reasons for failures, which are counted up in Table 5.7.

1. Bound computation fails if the original program cannot be abstracted into our abstract program model by our current implementation. Those examples include assignments inside of loops of the form $x = x \circ c$, where $\circ$ is a bit operation, multiplication or division. Examples can be found in Figure 5.4.

   Repetitive assignments of the form $x = x * y$ lead to an exponential grow of x. Hence, it is impossible to find a constant that overestimates the increase of x in each loop iteration. For binary operations like $x = x|y$ there exists also no constant c, such that we could abstract the assignment to a predicate like $x' \leq x + c$.

   Further, "self-depending" pointer arithmetic like "p += *p" fails to be abstracted.

2. As a knock-on effect, bound computations fail if the bound depends on another failed example.

Most of the remaining examples could be bounded by taking another local ranking function, by modelling of system calls or by introducing invariants about arrays and data structures. For example, to bound the loop at line 5 in the example depicted in Figure 5.2, we would have to generate the invariant that each element of the array `a` is not greater than 10 at line 5.

```
void array_local(int n) {
  int a[11];
  for (int i = 0; i < 11; ++i)
    a[i] = i;
  for (int j = 0; j < a[i]; ++j)
    ;
}
```

Figure 5.2: Example for array invariants.

	Loops	**Failed**	Failed Termination	Irreducible	Timeout	Error	Failed Bound
cBench Bound at Function header	4301	1277 30 %	962 22 %	144 3 %	0	81 2 %	90 2 %
SPEC Bound at Function header	15392	4188 27 %	3534 23 %	160 1 %	3	204 1 %	287 2 %
WCET Bound at Function header	262	24 9 %	17 6 %	0	0	0	7 3 %

Table 5.6: The table shows how many loops are contained within each benchmark and for how many loops LOOPUS is not able to compute a bound (column “Failed”). Column “Failed” sums up all reasons for failed computations listed on the right part of the table. The column “Failed Termination” shows for how many examples termination could not be proven by LOOPUS. In the next column, the number of irreducible control flow graphs are shown (which cannot be analyzed by LOOPUS). Note that we count all loops of a function as irreducible, if we compute a bound over the function parameters and there exists one irreducible loop in that function. The column “Error” notes the number of examples where LOOPUS terminates abruptly without result. In the last column, we show the amount of examples where bound computation actually failed.

	Failed Bound	Failed Abstraction	Depends on Failed
cBench Bound at Function header	90	32	37

Table 5.7: The table counts the number of examples falling into one of the two main reasons of failed bound computations. One reason for failure is that a bound computation depends on another failed bound. Such examples are counted up in the column “Depends on Failed”. The second main reason for failure are examples that could not be abstracted into our scheme.

5.3 Evaluation on Challenging Loop Classes

We specify in the following syntactic loop classes, which are of special interest to us as they exclude standard for-loops “**for**(i=0;i<n;i++)’ and include cases hard for bound analysis.

```
 1  //automotive_bitcount/
 2  //bitcnt_1.c
 3  int bit_count(long x) {
 4    int n = 0;
 5    if (x)
 6    do
 7      n++;
 8    while(0 !=
 9      (x = x & (x - 1)));
10    return (n);
11  }
```

Enhanced modelling of bit vectors

```
 1  //bzip2d/blocksort.c
 2  void mainsort() {
 3    int h = 1;
 4    do
 5      h = 3 * h + 1;
 6    while(h <= 256);
 7    do {
 8      h = h / 3;
 9    } while(h != 1);
10  }
```

Enhanced assumption generation
$h >= 3$ at line 8

```
 1  //automotive_qsort1/
 2  //qsort_large.c
 3  void main1() {
 4    int i, count = 0;
 5    while(nondet() &&
 6        count < 60000)
 7      count++;
 8    for(i=0;i<count;
 9        i+=count/100)
10      ;
11  }
```

Enhanced assumption generation
count $\geq$ 100 at line 9

```
 1  //bzip2d/blocksort.c
 2  typedef struct estate {
 3    int avail_in;
 4  } Estate;
 5  void copy_output_until_stop(
 6      Estate* s) {
 7    while(s->avail_in > 0
 8        && nondet())
 9      s->avail_in--;
10  }
11  void handle_compress(
12      Estate* s) {
13    while(1) {
14      copy_output_until_stop(s);
15      if (s->avail_in == 0)
16        break;
17    }
18  }
```

Enhanced function inlining at line 14

Figure 5.3: Examples illustrating the challenging task of computing local ranking functions. The examples are taken from cBench (in sliced version). We note below each function how our method could be enhanced to establish a ranking function.

```
 1  //office_ghostscript/iscannum.c
 2  void scan_number(int decode, int c, int d,
 3        int* sp, int* e) {
 4     int exp10 = c, iexp = decode;
 5     for (; ; iexp = 10 * iexp + d)
 6       if (sp++ >= e)
 7         break;
 8     exp10 += iexp;
 9     while (exp10 > 6)
10       exp10 -= 6;
11  }
```

Exponential grow of `iexp` at line 5

```
 1  //security_pgp_d/fileio.c
 2  int copyfile(FILE * f, FILE * g, int longcount) {
 3    int count;
 4    do {
 5      if (longcount < 4096) count = longcount;
 6      else count = 4096;
 7      count = fread(textbuf, 1, count, f);
 8      for(int i=0;i<count;i++)
 9        ;
10      longcount -= count;
11    } while (count == 4096);
12  }
```

(Partial) modelling of system call `fread` would allow to deduce that `count` is never increased at line 7 and would allow to introduce a norm representing the remaining bytes of stream f.

```
 1  //security_pgp_d/zinflate.c
 2  void inflate_stored(int k, int bb, int *inptr) {
 3    int n = k & 7, b = bb;
 4    while (k < 16) {
 5      b |= *inptr++;
 6      k += 8;
 7    }
 8    n = b & 0xffff;
 9    while (n-- > 0)
10      ;
11  }
```

The assignment at line 5 cannot be abstracted into the form of our abstract transition predicates $x' \leq x + c$.

Figure 5.4: Examples that cannot be abstracted into our abstract program model by the current version of LOOPUS. The examples are taken from cBench (in sliced version).

```
 1 exAmor(int m) {
 2   int i=m, n = 0;     //stack = emptyStack();
 3   while (i > 0) {
 4     i--;
 5     if (?)  //push
 6       n++;          //stack.push(element);
 7     else     //popMany
 8       while (n > 0 && ?)
 9         n--;      //element = stack.pop();
10   }
11 }
```

Figure 5.5: Example for amortized complexity analysis

5.3.1 Amortized Complexity Analysis

Bound analysis is challenging for loops that are *amortized* in the sense that they have an asymptotically lower bound than one would expect from the loop-nesting depth of the loop. In such cases, we would not get a precise bound if we just multiply the bound of the immediate parent loop with the number of iterations of the loop per iteration of the parent loop.

Amortized analysis was motivated by Robert Endre Tarjan [62] by using the example of a stack, which supports two operations: *push*, which adds a new item to the top of the stack, and *popMany*, which removes several items from the stack. Consider an initially empty stack and a sequence of m stack operations. If we assume that *push* has cost 1 and the cost of *popMany* is the number of removed elements, then a single operation can have a cost up to m, which happens if $m - 1$ *push* operations are followed by a call to *popMany* removing all items on the stack. However, as each removed element corresponds to one pushed element, we have an amortized running time of $2m$.

Tarjan gives a possible view of amortization by the use of a potential function. In our stack example, we have that each *push* increases the potential cost for a following *popMany* operation by one. Consider function 'exAmor' depicted Figure 5.5, which models a possible interaction with a stack. Our bound method achieves amortization by using variable n as a potential function for the loop in line 8. The potential number of iterations of that loop is increased by one for each *push* operation (if-branch). As the if-branch is bounded by m, the loop bound for the loop in line 8 is m and we have an overall computational complexity of $2m$.

5.3.2 Definition of challenging Loop Classes

We describe five loop classes which are challenging for bound analysis. We distinguish four different syntactic loop patterns (i.e., AmortA1-3 and InfluencesOuter) which are typical for loop constructs with an overall amortized complexity.

We use the helper function $LoopCounters(l)$ to denote the set of variables, which are maybe involved in the termination of the loop at location l. A formal definition for the set of $LoopCounters$ is given in Pani's thesis [57].

AmortA1 Given a nested loop at location l_1 with its parent loop at location l_0, we say that loop l_1 is part of loop class AmortA1, if and only if there is a variable $x \in LoopCounters(l_1)$ that is not modified on a simple path going from l_0 to l_1. For example, the inner loop of the amortization example depicted in Figure 5.5 and the inner loop of the following example fall into this category.

```
for(i=0; i<n; i++)
    for(;j<n && nondet(); j++) ;
```

AmortA2 Given a nested loop at location l_1 with its parent at location l_0, we say that loop l_1 is part of loop class AmortA2, if and only if there is a variable $x \in LoopCounters(l_1)$ that is not reset on a simple path going from l_0 to l_1. In contrast to class AmortA1, AmortA2 includes also the inner loop of the following example.

```
for(i=0;i<n;i++){
    j++;
    for(;j<n && nondet();j++)
        ;
}
```

AmortA3 Given a nested loop at location l_1 with its parent at location l_0, we say that loop l_1 is part of loop class AmortA3, if and only if there is a variable $x \in LoopCounters(l_1)$ that is not reset to a constant expression on a simple path going from l_0 to l_1. In contrast to class AmortA1 and AmortA2, AmortA3 includes also the inner loop of the following example.

```
for(i=0;i<n;i++){
    j = x;
    for(;j<n && nondet();j++)
        ;
    x = j;
}
```

InfluencesOuter Given a nested loop at location l_1 with its parent at location l_0, we say that loop l_1 is part of loop class InfluencesOuter if and only if there is a variable $x \in LoopCounters(l_0)$ that is modified within loop l_1. For example, the inner loop of the following example falls into this category, as the loop counter i of the outer loop is incremented during the inner loop.

```
for(i=0;i<n;i++)
    for(;i<n && nondet();i++)
        ;
```

	Total	Bounded	Terminate	Bounded / Terminate
AmortA1	136	69 51 %	72 53 %	96 %
AmortA2 w.o. AmortA1	99	42 42 %	50 51 %	84 %
AmortA3 w.o. AmortA2	159	92 59 %	96 60 %	96 %
InfluencesOuter	658	259 39 %	292 44 %	87 %
Paths>1	1276	719 56 %	752 59 %	96 %

Table 5.8: The table shows how many loops contained within the *cBench* are falling into which loop class and for how many loops LOOPUS could infer termination and bounds.

	Total	Bounded	Terminate	Bounded / Terminate
AmortA1	99	57 58 %	59 60 %	97 %
AmortA2 w.o. AmortA1	77	36 47 %	36 47 %	100 %
AmortA3 w.o. AmortA2	277	173 63 %	188 68 %	92 %
InfluencesOuter	847	439 52 %	468 55 %	94 %
Paths>1	4297	2735 64 %	2790 65 %	98 %

Table 5.9: The table shows how many loops contained within the *SPEC2006* benchmark are falling into which loop class and for how many loops LOOPUS could infer termination and bounds.

Paths>1 Given a loop at location l, we say that loop l is part of loop class Paths>1 if and only if the loop l_s, which is the sliced version of loop l such that it contains only variables of $LoopCounters(l)$, has more than one simple path going from location l_s to l_s.

Related Work on Syntactic Loop Classes

Thomas Pani systematically studies and describes syntactic loop patterns that are hard for program analysis in his master thesis [57]. He developed the tool SLOOPY, which automatically categorizes loops. We use a version of SLOOPY adapted to our described loop classes, which are inspired from Pani's work on classes of nested loops interesting for bound analysis, but our classes are more restrictive.

Our loop classes resemble the categories defined within the recent work to LOOPUS [60]. The loop class 'Inner Dependent' is equivalent to 'AmortA2' and the class 'Outer Dependent' is similar to our class 'InfluencesOuter', but includes the outer loop whereas we include the inner loop, because the bound of the inner loop is probably amortized.

5.3.3 Overall Results for Challenging Loop Classes

We give the results of LOOPUS on the examples of cBench, SPEC and WCET falling in one of the five loop classes in Table 5.8, Table 5.9 resp. Table 5.10. We denote by 'A w.o. B' all loops

	Total	Bounded	Terminate	Bounded / Terminate
AmortA1	0	0	0	-
AmortA2 w.o. AmortA1	7	4 57 %	4 57 %	100 %
AmortA3 w.o. AmortA2	4	1 25 %	2 50 %	50 %
InfluencesOuter	20	14 70 %	15 75 %	93 %
Paths>1	24	16 66 %	16 66 %	100 %

Table 5.10: The table shows how many loops contained within the *WCET* benchmark are falling into which loop class and for how many loops LOOPUS could infer termination and bounds.

	Bounded	Amortized
All (No Class)	3174	147 5 %
AmortA1	69	6 9 %
AmortA2 w.o. AmortA1	42	6 14 %
AmortA3 w.o. AmortA2	92	2 2 %
InfluencesOuter	259	26 10 %

Table 5.11: The table shows for how many loops of each loop class found in the *cBench* our tool LOOPUS could infer an amortized bound.

falling into class A but not in class B.

We see for example in Table 5.8 that the success rates on the cBench in all five loop classes are significantly reduced compared to the overall success rate of 74% seen in Table 5.4 for bounds defined at the header of the comprising SCC. We are interested only in bounds at SCC-level, because otherwise we cannot conclude from the asymptotic class of a bound if the bound is amortized.

The reduced success rates seen in the Tables 5.8, 5.9, and 5.10 show that we have discovered loop patterns that challenges bound analysis. However, success rates are high enough to say that LOOPUS is also able to manage complex loop patterns. Especially, since we observe that LOOPUS can compute a bound in most of the cases where termination was proven.

5.3.4 Results for Amortized Examples

Loops falling into one of the loop classes AmortA1, AmortA2, AmortA3 or InfluencesOuter are possibly amortized, which means that the degree of the asymptotical class is smaller than the nesting depth of the loop. Figure 5.6 shows some amortized examples.

In Table 5.11, Table 5.12, and Table 5.13 we give for each benchmark an overview for how many of the bounded loops, LOOPUS actually computed an amortized bound. We see that LOOPUS allows to compute amortized bounds and that those amortized bounds are more likely computed for loops described by one of our loop classes.

```
1  //consumer_jpeg_c/jchuff.c
2  void encode_one_block() {
3    r=0;
4    for(k=1;k<64;k++) {
5      if(nondet()) {
6        r++;
7      } else {
8        while(r>15) {
9          r-=16;
10       }
11       r=0;
12     }
13   }
14 }
```

Amortized loop of class AmortA1
at line 8

```
1  //office_ispell/tree.c
2  void treeoutput() {
3    struct dent *cent;
4    struct dent *lent;
5    ...
6    for(lent=cent; lent!=0;
7        lent=lent->next) {
8      while(lent->mask[0] &
9          (1 << 30))
10       lent=lent->next;
11   }
12 }
```

Amortized loop of class InfluencesOuter
at line 8

```
1  //consumer_tiff2bw/
2  //tif_luv.c
3  int LogL16Decode(
4      TIFF *tif) {
5    cc = tif->tif_rawcc;
6    for(i=0; i<npixels &&
7        cc>0;) {
8      cc -= 2;
9      while (--cc &&
10         nondet())
11       i++;
12   }
13 }
```

Amortized loop of class AmortA2
at line 9

```
1  //office_ispell/dump.c
2  void subsetdump() {
3    int cnum;
4    int rangestart;
5    for(cnum=0;cnum<128;
6        cnum++){
7      for(rangestart=cnum;
8          cnum<128;cnum++)
9          ;
10     while(rangestart<cnum)
11       rangestart++;
12   }
13 }
```

Amortized loop of class AmortA3
at line 10

Figure 5.6: Examples from the *cBench* for that LOOPUS computed an amortized bound.

	Bounded	Amortized
All (No Class)	11481	1140 10 %
AmortA1	57	19 33 %
AmortA2 w.o. AmortA1	36	5 14 %
AmortA3 w.o. AmortA2	173	41 24 %
InfluencesOuter	439	114 26 %

Table 5.12: The table shows for how many loops of each loop class found in the *SPEC2006* benchmark our tool LOOPUS could infer an amortized bound.

	Bounded	Amortized
All (No Class)	243	16 7 %
AmortA1	0	0
AmortA2 w.o. AmortA1	1	0
AmortA3 w.o. AmortA2	4	0
InfluencesOuter	14	4 29 %

Table 5.13: The table shows for how many loops of each loop class found in the *WCET* benchmark our tool LOOPUS could infer an amortized bound.

CHAPTER 6

Conclusion and Future Work

Bounds on the number of times loops may iterate have important applications in resource bound analysis. Defining a loop bound requires to establish variable bounds (upper or lower bounds on the value of a variable during execution of a program).

We motivated the problem of computing variable bounds, because state-of-the-art invariant generation tools are mostly insufficient for the purpose of bound analysis. They do not scale to non-linear or disjunctive invariants, which are required to shape bounds in cases of nested loops or loops with complex control flow.

Based on the observation that loop and variable bounds can be expressed in terms of each other, we elaborated an integrated bound analysis where the loop and variable bound algorithms are calling each other recursively. Our method does not involve expensive computations like abstract interpretation or computer algebra in contrast to related tools.

Our bound algorithms are defined for a simple abstract program model called difference constraint program (DCP), such that the formal definitions of the algorithms are short, clear and free from any programming-language specific constructs. The choice for DCPs is also justified by the fact that termination is decidable for DCPs.

We have shown methods to abstract a given C program into a DCP and implemented those methods and our bound algorithms into the tool LOOPUS. We think that C is a natural choice for resource bound analysis because many applications for embedded systems are written in that language.

We evaluated LOOPUS on three real-world benchmarks, where we were able to bound more than 70% of all $\sim 20,000$ loops. More than the half of all loops could be bounded in less than one second. We have also shown that LOOPUS outperforms existing bound tools in respect to the total number of bounded examples, precision of computed bounds and running time of the analysis. Further, we defined syntactical loop-patterns indicating amortized running time, and we have shown that LOOPUS really computed significantly more amortized bounds for those classes of loops.

6.1 Future Work

We point out several directions for future work:

Recursive Procedures and Functional Programs

A naive support for recursive programs would be the transformation to programs using imperative loop-constructs. For tail-recursive examples, such an approach may be sufficient. Transformations of other patterns of recursion must maintain a call stack. The question is how well DCPs can be extended by a stack. An inspiration could be pushdown automatons.

Concurrent programs

Common properties of models (e.g., petri nets) of concurrent systems are their nondeterminism and - clearly - their ability of concurrent executions. A difference-constraint program also has the nondeterminic choice which transition out of the enabled ones (i.e., transitions that predicates and conditions are satisfied) should be taken next. An interesting opportunity for future work would be the computation of loop and variable bounds for DCPs extended with concurrent executions.

Data Structures

LOOPUS is able to compute bounds for loops iterating over some data structure (e.g., linked list, array, etc.). However, we currently have to make assumptions like that a list is acyclic or that a specific element is contained in a list. Those assumptions could be proven by methods like shape analysis. Such methods could also be used to establish invariants of values stored in a data structure and to determine properties (e.g, length) of a data structure.

Software Verification and Invariant Generation

Variable bounds have a natural application in software verification and invariant generation. A typical example is the verification of the absence of array buffer overflows. Another application for invariant generation would be in LOOPUS itself: LOOPUS makes assumptions about the sign of variables (see Section 4.1). Checking the correctness of those assumptions constitutes a possible extension to LOOPUS.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, pages 221–237, 2008.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. PUBS: A practical upper bounds solver. `http://costa.ls.fi.upm.es/pubs/`, 2008.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.

[5] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[6] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Trans. Comput. Log.*, 14(3):22, 2013.

[7] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133, 2010.

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

[9] Amir M. Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.

[10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic bound computation for loops. In *LPAR*, pages 103–118, 2010.

[11] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Trans. Comput. Log.*, 2(1):1–11, 2001.

[12] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[13] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *CAV*, pages 413–429. 2013.

[14] Marc Brockschmidt, Fabian Emmes, Claßen Falke, Carsten Fuhs, and Jürgen Giesl. Empirical evaluation of: Alternating runtime and size complexity analysis of integer programs. `http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity/`.

[15] Marc Brockschmidt, Fabian Emmes, Claßen Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, pages 140–155, 2014.

[16] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, pages 33–52, 2013.

[17] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for c programs. In *PLDI*, page 30. ACM, 2014.

[18] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. Technical report, 2014. Yale University.

[19] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIXATC*, pages 21–21, 2010.

[20] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In *ISMM*, pages 151–160, 2008.

[21] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[22] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[23] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[24] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[25] Fábio Coutinho, Luís Alfredo V. de Carvalho, and Renato Santana. A workflow scheduling algorithm for optimizing energy-efficient grid resources usage. In *DASC*, pages 642–649, 2011.

[26] cTuning. Collective Benchmark (cBench). `http://cTuning.org/cbench`.

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[28] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *WCET*, 2007.

[29] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[30] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *RTA*, pages 32–47, 2009.

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

[32] Sumit Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV*, pages 51–62, 2009.

[33] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. *SIGPLAN Not.*, 44(6):375–385, 2009.

[34] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *POPL*, pages 277–289, 2007.

[35] Sumit Gulwani and Sudeep Juvekar. Bound analysis using backward symbolic execution. Technical report, 2009. Microsoft Research.

[36] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.

[37] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[38] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[39] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, pages 634–640, 2009.

[40] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present and Future. In *WCET*, pages 136–146, 2010.

[41] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, pages 57–66, 2006.

[42] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[43] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL*, pages 357–370, 2011.

[44] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306, 2010.

[45] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003. ACM SIGPLAN Notices 38(1), January 2003.

[46] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET 2008 – report from the tool challenge 2008. In *WCET*, pages 149–171, 2008.

[47] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997.

[48] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, pages 223–236, 2010.

[49] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *Ershov Memorial Conference*, pages 227–242, 2011.

[50] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, 2004.

[51] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET*, pages 77–80, 2003.

[52] Tidorum Ltd. Bound-T - a time and stack analyser. `http://www.bound-t.com/`.

[53] Stephen Magill, Ming-Hsien Tsai, Peter Lee 0001, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

[54] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL*, pages 225–235, 2007.

[55] Marianne De Michiel, Armelle Bonenfant, Hugues Cass, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA*, pages 161–166, 2008.

[56] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.

[57] Thomans Pani. Loop Patterns in C. Master's thesis, Vienna University of Technology, Austria, 2014.

[58] Enric Rodriguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*, 2004.

[59] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using grbner bases. In *POPL*, pages 318–329, 2004.

[60] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *CAV*, pages 743–759, 2014.

[61] James Stanier and Des Watson. A study of irreducibility in c programs. *Softw., Pract. Exper.*, 42(1):117–130, 2012.

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

[63] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenstrm. The Worst-Case Execution Time Problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[64] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.