Dissertation

Resource Bound Analysis of Imperative Programs

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Univ. Prof. Dr. Helmut Veith
E184/4
Institut für Informationssysteme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Florian Zuleger
0929415
Skodagasse 3/7
1080 Wien
Österreich

Wien, am 20.04.2011 Unterschrift des Verfassers

We must know - we will know!

David Hilbert,
Address to the Society of German
Scientists and Physicians, in
Königsberg (8 September 1930)

Acknowledgements

First and foremost, I would like to thank my advisor Helmut Veith, without whom this thesis would not have been possible. He suggested to me the topic of this thesis, introduced me to research, always believed in my abilities and gave me enough space to develop my own ideas. Throughout my studies his enthusiasm and dedication to research always were encouraging. Helmut strongly influenced the development of my logical viewpoint on computer science. I further would like to thank Helmut for his friendliness and reliability and for creating a nice group atmosphere by social events such as visiting conferences jointly, going to the ball together etc.

I would like to thank Sumit Gulwani for our joint work on bound analysis during my internship and thereafter, which had a substantial influence on this thesis. I learned from Sumit the importance of practical examples for theoretical research and how to focus on relevant problems. Sumit is an inspiring researcher whose passion sets an example for other researchers. I further would like to thank him for his continuing supervision and advice.

I would like to thank the members of my group for providing such a lively working environment and for all the fun inside and outside university. I especially thank Moritz Sinn for our joint work on bound analysis, his help with the figures and the proof-reading of this thesis, Andreas Holzer for letting me sleep on his couch before I had my own appartment, Johannes Kinder for storing my piano during my internship, and Michael Tautschnig for all his support on linux issues.

I would like to thank my parents for their continuing emotional and moral support, which gives me the safety to believe that things will always turn out well.

Abstract

For many practical applications it is important to bound the resources consumed by a program such as time, memory, network-traffic, power, and to estimate quantitative properties of data in programs, such as information leakage or uncertainty propagation. At the heart of many of these questions lies the problem of finding a symbolic worst-case bound on the number of visits to a given program location in terms of the inputs to that program; we call this the *reachability-bound problem*. The automatic computation of reachability-bounds is challenging because in general such bounds are complicated expressions that hinder the direct application of standard abstract domains and require disjunctive invariants.

We propose a two-step methodology for computing a reachability-bound of a given program location: First, we generate a transition system that disjunctively summarizes all pairs of states for which there is a program execution that visits the location once and again. Second, we compute a bound of the transition system to derive a reachability-bound. We present two approaches that implement this methodology.

Our first approach brings together two different techniques for reasoning about loops. We present an abstract-interpretation based iterative technique for computing disjunctive loop invariants, which we use for summarizing inner loops. We use a non-iterative proof-rules based technique for loop bound computation that takes over the role of doing inductive reasoning, while deriving its power from the use of SMT solvers to reason about abstract loop-free fragments. We evaluate the effectiveness of our approach on a .Net library and illustrate the precision of our algorithm for disjunctive invariant computation on a set of benchmark examples. Though effective our first approach lacks a unifying theory that allows to discuss properties like completeness, complexity, etc.

Our second approach is based on the so-called *size-change abstraction* (SCA). While SCA is an established abstract domain for termination analysis that has been successfully implemented in many tools for functional and declarative programs, we are the first to demonstrate its potential for the harder problem of bound analysis. SCA has the crucial property to be closed under the computation of transitive hulls, which we use for summarizing inner loops disjunctively. We show that SCA offers a separation of concerns for bound computation: we extract local progress measures from small program parts, and then compose these local progress measures to a global bound using only the size-change abstracted program. We state two program transformations that make imperative programs amenable to bound analysis with SCA. We evaluate the effectiveness of our approach on C benchmark programs.

Having demonstrated the practical relevance of SCA for bound analysis, we present results towards a theoretical characterization of the bounds expressible by SCA. In particular, we define complexity witnesses that establish lower bounds of abstract programs.

In a qualitative comparison we show that our solution to the reachability-bound problem captures the essential ideas of earlier termination and bound analyses in a simpler framework and outperforms these analyses in computing loop-bounds and proving termination.

Kurzfassung

In vielen praktischen Anwendungen benötigt man eine Beschränkung der von Computerprogrammen verbrauchten Resourcen (z.B. Ausführungszeit, Speicher, Netzwerkverkehr, Energie) und eine Abschätzung von quantitativen Programmeigenschaften (z.B. des Verlusts geheimer Informationen oder der Ausbreitung von Fehlern). Viele dieser Fragen lassen sich auf das Problem der Berechnung einer symbolischen Schranke zurückführen, wie oft ein bestimmter Programmpunkt in Abhängigkeit von den Programmeingaben besucht werden kann; wir nennen dies das *Erreichbarkeitsschrankenproblem*. Die automatische Berechnung von Erreichbarkeitsschranken ist anspruchsvoll, da solche Schranken im allgemeinen komplizierte Ausdrücke sind, die nicht direkt durch bewährte abstrakte Domänen berechnet werden können, und disjunktive Invarianten benötigen.

Die in dieser Dissertation vorgestellte Methode berechnet eine Erreichbarkeitsschranke eines gegebenen Programmpunktes in zwei Schritten: Im ersten Schritt erzeugen wir ein Transitionssystem, das disjunktiv alle Paare von Zuständen zusammenfasst, für die es eine Programmausführung gibt, die den Programmpunkt mindestens zweimal besucht. Im zweiten Schritt berechnen wir eine Schranke für das Transitionssystem und erhalten dadurch die gewünschte Erreichbarkeitsschranke. Wir präsentieren zwei Ansätze, die unsere Methode praktisch umsetzen.

Unser erster Ansatz benützt zwei verschiedene Techniken zur Analyse von Schleifen. Zum einen präsentieren wir eine auf abstrakter Interpretation basierende iterative Technik zur Berechnung disjunktiver Schleifeninvarianten. Zum anderen benutzen wir eine nicht-iterative auf Beweisregeln basierende Technik für die Berechnung von Schleifenschranken, welche ihre Leistungsfähigkeit aus dem Einsatz von SMT Beweisern bezieht; das ermöglicht präzise Schlüsse über lange schleifenfreie Codeabschnitte. Wir evaluieren die Effizienz unseres Ansatzes an einer .Net Bibliothek und an mehreren Beispielen für die Berechnung disjunktiver Invarianten aus der Literatur. Ungeachtet seiner praktischen Effizienz mangelt es dem ersten Ansatz an einer vereinheitlichenden Theorie, auf deren Grundlage wir Eigenschaften wie Vollständigkeit, Komplexität etc. diskutieren können.

Unser zweiter Ansatz basiert auf einer Abstraktion, die das monotone Verhalten numerischer Größen über den Programmzuständen beschreibt. Die sogenannte *size-change abstraction* (SCA) ist eine etablierte abstrakte Domäne für Terminationsanalyse, die für funktionale und deklarative Sprachen erfolgreich eingesetzt wird. In dieser Dissertation wird SCA erstmals für das schwierigere Problem der Schrankenanalyse eingesetzt. Für das disjunktiven Zusammenzufassen innerer Schleifen nutzen wir die Abschlusseigenschaft von SCA in Bezug auf transitive Hüllen. Mit Hilfe von SCA können wir Schranken stufenweise berechnen: Zunächst extrahieren wir lokale Fortschrittsmaße aus kleinen Programmabschnitten und setzen diese lokalen Fortschrittsmaße dann nur mit Hilfe des abstrahierten Programmes zu einer global gültigen Schranke zusammen. Durch zwei geeignete Programmtransformationen ermöglichen wir dann die Schrankenanalyse imperativer Programme. Wir evaluieren die Effizienz unserer Ansatzes anhand zweier Benchmarks für die Sprache C.

In Ergänzung des praktischen Relevanznachweises für SCA in der Schrankenanalyse präsentieren wir theoretische Beiträge, die auf eine mathematische Charakterisierung jener Schranken, die mit SCA ausgedrückt werden können, hinarbeiten. Insbesondere definieren wir Komplexitätszeugen für den Nachweis unterer Schranken abstrakter Programme.

Zusammenfassend diskutieren wir anhand eines qualitativen Vergleiches

ausführlich, wie unser Lösungsansatz des Erreichbarkeitsschrankenproblems auf den wesentlichen Ideen früherer Terminations- und Schleifenanalysen aufbaut und diese früheren Ansätze bei der Berechnung von Schleifenschranken und Terminationsbeweisen verbessert.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In formal verification, proving termination is an important step in proving the correctness of programs. A general method for constructing a termination proof of a program involves associating a measure with each step of the program [Turing, 1936]. The measure is taken from the domain of a well-founded relation, e.g. the ordinal numbers, and is commonly called a *ranking function*. If the ranking function decreases according to the well-founded relation for every step of the algorithm, the program must terminate, because there are no infinite descending chains with respect to a well-founded relation.

Termination analysis attempts to automatically detect whether a given program terminates on all inputs. Because the halting problem is undecidable [Turing, 1936], such an analysis is necessarily incomplete. However the goal is to find the answer "program does terminate" (or "program does not terminate") for as many programs occurring in practice as possible.

Recent years have seen a rapid progress in the field of termination analysis for imperative programs. This development started with automatic methods for constructing ranking functions. These methods built on the insight that many ranking functions have simple shapes that can be captured by ranking function templates; various constraint solving techniques have been proposed for finding the coefficients of such ranking function templates. The effectiveness of these techniques has been limited to small programs because larger programs in general have complex ranking functions whose shapes cannot be captured by templates. The key for handling larger programs was the advent of compositional techniques for termination analysis. In [Lee et al., 2001], Lee, Jones and Ben-Amram describe a size-change abstraction approach, which allows to prove termination of functional programs by constructing a global termination argument from local size-changes of data-structures. In [Podelski and Rybalchenko, 2004b] Podelski and Rybalchenko generalize this approach and describe a proof rule, which allows to prove termination of imperative programs by constructing a global termination argument from local size-changes of data-structures. Cook, Podelski and Rybalchenko describe in [Cook et al., 2006] how such termination arguments can be constructed iteratively with an abstraction refinement algorithm. Their approach has the advantage that ranking functions only need to be constructed for small program parts, which are simple enough to be handled by automatic techniques, and that it shifts the difficulty from finding to checking the termination argument, which can be done by techniques for safety checking. Note that this reduction to safety checking is a key enabling factor for termination analysis: The progress of software model checking and abstract interpretation techniques in establishing safety properties of imperative programs is leveraged to termination analysis. For the first time in this field of research [Cook et al., 2006] crossed the border between pure research and industrial development, being able to prove termination of Windows device drivers with several thousands lines of code.

However, as useful as it is to automatically reason about the termination of programs, it is not quite sufficient for many purposes. Programs make use of *physical resources* such as *time*, *memory*, *power*, *bandwidth*, etc. Many important applications require establishing worst-case bounds on usage of such resources. We list several examples:

- In memory-constrained environments such as embedded systems, it is important to bound the amount of *memory* required to run certain applications.
- In real-time systems, it is important to bound the worst-case *execution-time* of the program.
- Similarly, applications running on low-power devices or low-bandwidth environments must use up little *power* or *bandwidth* respectively.
- With the advent of cloud computing, where users will be charged per program execution, predicting resource usage characteristics would be a crucial component of *accurate bid placement* by cloud providers.
- For deciding whether to *parallelize* a loop, it is important to accurately estimate the number of times the loop is executed and the cost of executing the body of the loop one time.

One of the fundamental questions that lies at the heart of computing resource bounds for imperative programs is: *How many times is a given control location inside the program executed?*

Executing a program affects certain quantitative properties of the data that it operates on. In such cases even the correctness of programs relies on the presence of bounds to the effects of their executions on the quantitative data properties. We list two examples:

- For example, how much *secret* information is leaked by a program depends on the number of times a certain operation that leaks the data, either by direct or indirect information flow, is executed [Malacaria, 2007].
- Similarly, the amount of *perturbation* in the output data values resulting from a small perturbation or uncertainty in the input values depends on the number of times additive error propagation operators are applied, e.g., in numerical algorithms machine numbers are used instead of the mathematical real numbers.

Estimating such quantitative properties can be addressed by a similar question as above: *How many times is a given control location inside the program that performs certain operations executed?*

Note that computing a bound on how often a certain control location of a program is visited is a harder problem than proving termination, and that therefore the undecidability of the halting problem extends to resource bound analysis. Further note that resource bounds are quantitative properties of programs, as opposed to termination, which is a Boolean property. In this thesis we show how the successful techniques for termination analysis can be extended to resource-bound analysis. In this way we contribute to the *quantitative agenda* set forth recently [Henzinger, 2009] (as opposed to the Boolean agenda).

1.1 The Reachability-bound Problem

We define the *reachability-bound problem* to be the problem of computing a symbolic worst-case bound $b(s)$ on the number of visits to a given control location l of a given program P. The bound $b(s)$ is parameterized by the initial state s of P. [1] We illustrate that this definition allows to address resource-bound problems by two examples:

- Memory consumption (under the assumption of fixed-byte memory allocation):

$$\sum_{\text{locations } l \text{ that allocate memory}} b_l(s) \cdot \texttt{BytesAllocated}(l)$$

- Program complexity:

  The *loop-bound problem* is the problem of computing a bound on the number of iterations of a given loop. It is an instance of the reachability-bound problem – the control location under consideration is the immediate successor of the header of the loop. By computing such loop-bounds we can bound the complexity of a given program P:

$$\sum_{\text{location } l \text{ is the immediate successor of a loop header of } P} b_l(s)$$

1.1.1 Challenges in the Reachability-bound Problem

Computing reachability bounds faces the following four technical challenges:

(A) Bounds are often *complex non-linear arithmetic expressions* built from $+, *, \max$ etc. Therefore, abstract domains based on linear invariants (e.g. intervals, octagons [Miné, 2006], polyhedra [Cousot and Halbwachs, 1978]) are not directly applicable for bound computation.

(B) The proof of a given bound often requires *disjunctive invariants* that can express loop exit conditions, phases, and flags which affect program

[1]The formal definition will be given in Section 2.3.

behavior. Although recent research made progress on computing disjunctive invariants [Gulwani et al., 2009a; Podelski and Rybalchenko, 2004b; Cook et al., 2006; Berdine et al., 2007; Popeea and Chin, 2006; Gopan and Reps, 2006], this is still a research challenge. (The domains mentioned in (A) are conjunctive.)

(C) It is *difficult to predict a bound in terms of a template* with parameters because (1) the search space for suitable bounds is huge, and (2) the bound is a global property of the program and therefore a local analysis is not possible.

(D) It is not clear how to *exploit the loop structure of imperative programs* to achieve *compositionality* in bound analysis. This is in contrast to automatic termination analysis, where the *cut-point method* (e.g. [Cook et al., 2006; Berdine et al., 2007]) is used standardly to exploit the loop structure in order to achieve compositionality.

Next we illustrate these challenges on several hard instances of the reachability-bound problem.

1.1.2 Methodological Examples

Inner Loop Affecting the Iterations of Outer Loop. Let us consider Example 1.1. Computing a bound for the header of the outer loop l_1 exhibits the following difficulties: The inner loop cannot be excluded in the analysis of the outer loop (e.g. by the standard technique called *slicing* [Muchnick, 1997]) as it modifies the counter of the outer loop; this demonstrates the need for global reasoning in bound analysis (D). Further one needs to distinguish whether the inner loop has been skipped or executed at least one time as this determines whether $j = 0$ or $j > 0$. This exemplifies why we need disjunctive

```
     void main (int n){
           int i = 0;
           int j;
l1:        while(i < n) {
              i++;
              j := 0;
l2:           while((i < n) && nondet()){
                i++;
                j++;
              }
              if (j > 0)
                i--;
            }
     }
```

Program 1.1: The inner loop affects the iterations of outer loop.

invariants for loops (B). Moreover, the counter i may decrease, but this can only happen when i has been increased by at least 2 before. This presents a difficulty to an automatic analysis since it needs to be disjunctive and precise enough to capture arithmetic reasoning (A).

Loop Phases. Bound computation is difficult for loops that contain finite state machines that controls their dynamics. Program 1.2, found during our experiments on the `cBench` benchmark [CBenchWebPage, 2010], presents such a loop. The loop has three different phases: in its first iteration it assigns 1 or 2 to d, then it either increases or decreases s until it sets f to true; then it divides c by 2 until the loop is exited. In order to distinguish these loop phases, disjunctive reasoning is crucial (B). The loop has the bound $\max(255, s) + 3$, which is difficult to guess by a template (C). The bound cannot be obtained directly from classical abstract domains because of the exponential decrease of variable c and because of the maximum operator (A).

```
// cBench/consumer_lame/src/quantize-pvt.c
int bin_search_StepSize2 (int r, int s) {
  static int c = 4;
  int n;
  int f = 0;
  int d = 0;
  do {
    n = nondet();
    if (c == 1 ) break;
    if (f)
      c /= 2;
    if (n > r) {
      if (d == 1 && !f) {
        f = 1;
        c /= 2;
      }
      d = 2;
      s += c;
      if (s > 255)
        break;
    }
    else if (n < r) {
      if (d == 2 && !f) {
        f = 1;
        c /= 2;
      }
      d = 1;
      s -= c;
      if (s < 0)
        break;
    }
    else break;
  }
  while (1);
}
```

Program 1.2: The loop contains a finite state machines that its dynamics. This results into different loop phases.

```
       void main(int n, int[] A) {
$l_1$:   int i = 0;
$l_2$:   while (i<n) {
           int j = i+1;
$l_4$:     while (j<n) {
$l_5$:       if  (A[j]) {
$l_6$:         ConsumeResource();
               j--;
               n--;
             }
$l_9$:       j++;
           }
           i++;
         }
       }
```

Program 1.3: Demonstrates the difference between loop- and reachability-bounds

1.1.3 Difference between the Loop-bound and Reachability-bound Problem

We discuss the difference between the loop-bound problem and the reachability-bound problem on Program 1.3, which presents a loop skeleton from a .Net base-class library. We consider the problem of computing a symbolic bound on the number of times the procedure `ConsumeResource`() is called at location l_6. Using techniques for loop-bound computation (e.g. [Gulwani et al., 2009c,a]) one could approximate the number of calls at location l_4 by the number of iterations of the closest enclosing loop at location l_4. However, this approximation yields imprecise results since the number of iterations of the loop at location l_4 is bounded by n^2, while the number of executions of l_6 is bounded by n. We see that the reachability-bound problem is more general and requires more precise techniques than the loop-bound problem.

```
      void main (int n) {
        int i = 0;
l1:     while(i < n) {
          i++;
l2:       while((i < n) &&
                nondet())
            i++;
        }
      }
```

Program 1.4

```
      void main (int n) {
        int i = 0; int j = 0;
l1:     while(i < n) {
          i++;
l2:       while((j < n) &&
                 nondet())
            j++;
        }
      }
```

Program 1.5

Figure 1.1: Two programs for which the local method yields imprecise results.

1.1.4 Amortized Analysis

Consider the two programs shown in Figure 1.1, which have both inner loops. For both programs the outer loop has bound n and the inner loop can be iterated maximally n times between two iterations of the outer loop. A straightforward idea for obtaining a bound on the total number of iterations of the inner loop is to multiply these two bounds. We call this way of inferring bounds the *local method* and further discuss it in Subsection 1.1.5 below. This yields the quadratic bound n^2 for the total number of iterations of the inner loop. Clearly this bound is imprecise as both programs have linear bounds: In Example 1.4, the total number of visits to both cut-points is bounded by n because both loops iterate over the same counter variable i. In Example 1.5, the total number of visits to both cut-points is bounded by $n + n = 2n$ because the outer and inner loop have the different counter variables i and j and the counter variable of the inner loop j is not reset during an iteration of the outer loop.

The precise reasoning how often certain costly program operations – such as inner loops, calls of expensive functions, etc. – are executed is called *amortized analysis*. The challenge in amortized analysis is that these costly operations do not need to occur as often as the syntactic structure of the program might imply. We have given an example for the amortized analysis of inner loops in the above discussion on Examples 1.4 and 1.5. We have given an example for the amortized analysis of a call to an expensive function in Subsection 1.1.3, where we discussed that `ConsumeResource`() is not executed as often as its enclosing loop.

Note that the notion of a reachability-bound addresses the problem of computing amortized analysis. For both programs n is a reachability-bound of l_1 (the header of the outer loop) and l_2 (header of the inner loop). This implies the linear bound $2n$ on the total number of visits to l_1 and l_2 for both programs (which is not the most precise bound for Example 1.4, but at least a linear bound).

1.1.5 Cut-points and the Local Method

In this subsection we introduce the cut-point method for proving termination, which uses the syntactic structure of programs. After that we show that the local method – which we informally introduced in Subsection 1.1.4 above – is an extension of the cut-point method to bound analysis. We have already seen in Subsection 1.1.4 that the local method is not adequate for computing reachability-bounds because it fails to compute amortized bounds. This shows that using only the syntactic structure of programs is not sufficient for bound analysis and thus substantiates the claim stated in challenge (D).

We now explain the cut-point method for proving termination (a more detailed description can be found in [Berdine et al., 2007]). A set of cut-points C is a set of control locations of a given program P such that by deleting the locations C from the control flow graph of P the graph becomes acyclic. A standard choice for the set of cut-points C in structured (reducible) programs is the set of loop headers. A set of cut-points C allows the following procedure for proving termination: Choose an unmarked cut-point $c \in C$. Show that c cannot be visited infinitely often by executions that only visit unmarked cut-points. Mark c. Repeat these three steps until all cut-points are marked.

The cut-point method has a natural extension for loop bound computation, which we call the *local method* (a more detailed description can be found in [Gulwani et al., 2009c,a]): Choose an unmarked cut-point $c \in C$. Compute the bound b_c on the number of visits to c by executions that only visit unmarked cut-points. Mark c. Recursively compute a bound b_{rest} on the total number of visits to all remaining unmarked cut-points. Return $(b_c + 1) \cdot b_{rest}$ as bound on the total number of visits to the cut-points that were unmarked when $c \in C$ was chosen.

We apply the above approach to Examples 1.4 and 1.5: We choose $\{l_1, l_2\}$ as set of cut-points, compute the bound n for the header of the outer loop l_1, isolate the inner loop by forbidding visits to the header of the outer loop l_1, compute the bound n for the header l_2 of the inner loop, and then conclude the bound $(n + 1) \cdot n$ for the total number of visits to both cut-points. Note that this description matches the application of the local method in Subsection 1.1.4.

1.2 Summary of our Approach

Our method starts from the observation that progress in most software depends on the *linear change* of integer-valued functions on the program state (e.g., counter variables, size of lists, height of trees, etc.), which we call *norms.* The vast majority of non-linear bounds in real-life programs stems from two sources – nested loops and loop phases – and not from inherent non-linear behavior as in numeric algorithms. For most bounds, we have therefore the potential to exploit the nesting structure of the loops, and compose global bounds from bounds on norms. Upper bounds on norms typically consist of easily established facts such as size comparisons between variables and can be computed by classical conjunctive domains.

Two-step Analysis. To determine a reachability-bound of a location l of program P, we propose two steps:

- *First, compute a transition system $\mathcal{T}$ for l from P.* A transition system $\mathcal{T}$ for l is a set of transition relations such that for every execution of P and for every two consecutive visits of this execution to l the states at these visits are contained in one of the transition relations (see formal definitions in Section 2.2).
- *Second, compute a bound of $\mathcal{T}$.* This bound then gives a bound on the number of visits to l.

We implement these two steps as follows:

- *Transition System Computation.* We recursively compute transition systems for the inner loops of P and summarize them disjunctively by transitive hulls. Using these summaries we derive a disjunctive transition system $\mathcal{T}$ for l by the program transformation *pathwise analysis*. Pathwise analysis is based on two ideas: First, to abstract not only single program statements or blocks but complete program paths. Second, to exploit the looping structure of programs by choosing paths from loop header back to header; these paths are the program entities on which we expect the linear change of individual norms to take place.

Pathwise analysis enumerates all cycle-free paths from location l to l, inserts the transitive hulls of the inner loops on all of these paths at the header of the respective loops, and contracts the transition relations on the resulting paths in order to derive a disjunctive transition system. We describe our algorithm for computing transition systems (Algorithm 1) and discuss pathwise analysis in Section 2.5. Algorithm 1 is parameterized by an algorithm for computing transitive hulls.

- *Bound Computation.* We first compute bounds for single transition relations of $\mathcal{T}$ and then compose these bounds to an overall bound of $\mathcal{T}$.

In the above methodology we have left open how to compute transitive hulls and we have not given any details on how to compute bounds. We describe our first approach in the next subsection, then discuss the limitations of the first approach and after that describe our second approach, which overcomes these limitations.

1.2.1 Proof-rule Based Approach (Chapter 3)

Our first approach uses two different techniques, namely an iterative technique for computing transitive hulls based on abstract interpretation, and a non-iterative proof-rule based technique for computing bounds of transition systems.

Transitive Hull Computation. Our transitive hull algorithm is parameterized by an abstract domain and iteratively computes disjunctive invariants for transition systems over the powerset extension of the abstract domain. Algorithms over powerset set domains face the problem of when to merge elements of the base domain. The main idea of our algorithm is to use a fixed syntactic merging criterion for this. This is motivated by an assumption resembling convex theories that we found to be satisfied for the examples encountered in practice. We describe this transitive hull algorithm in Section 3.1. We also evaluated this algorithm on benchmark examples taken from recent work on computing disjunctive invariants. Our algorithm can discover required invariants in all examples, suggesting its potential for effective use in other applications requiring disjunctive invariants besides bound analysis.

Bound Computation. We compute bounds of transition systems by a non-iterative proof-rules based technique that requires discharging queries using an off-the-shelf SMT solver. We have collected patterns that describe the typical iterations of the individual loops of most software programs, e.g., increasing a counter variable, going through the elements of a list, etc. In Section 3.2 we list proof rules that check these iteration patterns on individual transitions of small program parts in order to obtain local ranking functions. In Section 3.3 we list proof rules that describe conditions that are sufficient for combining the local ranking functions for individual transitions into an overall bound of the transition system using three different mathematical operators, namely max, sum, and product. This methodology represents an interesting design choice for reasoning about loops, because SMT solvers are used to perform precise reasoning about transitions (loop-free code-fragments), whereas a simple proof-rules based technique takes over the role of performing inductive reasoning effectively.

We have implemented our solution to the reachability-bound problem in a tool that computes symbolic computational complexity bounds for procedures in .Net codebases. This involves computing amortized bounds for nested loops by solving the reachability-bound problem for nested loops.

Our experiments (described in Section 3.4) demonstrate the effectiveness of our approach.

Limitations of the Proof-rule based Approach:

Transitive Hull Computation. We argued that one challenge in bound analysis is the computation of disjunctive invariants. We have addressed this challenge by a fairly general algorithm that is capable of even handling the benchmark examples from recent work on computing disjunctive invariants (as demonstrated in Section 3.4). The algorithm makes use of a syntactic criterion for merging elements in the powerset abstract domain. However, the syntactic merging criterion itself is based on a merging function that is *a priori* unknown. In our implementation we select this merging function heuristically. While this approach for computing disjunctive invariants is fairly general and can be easily adjusted to different application domains, it is worthwhile to investigate algorithms that are more robust and specifically targeted towards bound computation.

Bound Computation. In our first approach to the reachability-bound problem we have given proof rules, which we found out to be effective throughout our experiments. In general such proof-rules based approaches are versatile and often give rise to efficient analyses. However, they are also ad hoc and do not establish a systematic theory of the investigated problem. This is unsatisfactory because such a theory allows to study the completeness and complexity of the analysis and to investigate the applicability to related problems and other programming paradigms.

.Net vs. C language. The .Net language for which we computed bounds in our experiments contains a lot of structure that can be exploited by the proof-rule based approach: rich type information, well-designed interfaces

and object-oriented standard libraries. In contrast, the C language is lacking these features and for this reason it is not clear how our implementation for .Net programs can be extended to C programs. We believe that instead of devising new proof rules for C it is more rewarding to investigate a more abstract model for bound analysis.

1.2.2 Size-change Abstraction Based Approach (Chapter 4)

Our second approach builds on the size-change abstraction (SCA) by Lee et al. [Lee et al., 2001; Ben-Amram, 2011]. SCA is a predicate abstract domain that consists of (in)equality constraints between integer-valued variables and boolean combinations thereof in disjunctive normal form (DNF). The inequality constraints consist of *control predicates* which describe control invariants and *transition predicates* which describe transitions of the program state.

SCA is well-known to be an attractive abstract domain: First, SCA is rich enough to capture the progress of many real-life programs. It has been successfully employed for automatic termination proofs of recursive functions in functional and declarative languages, and is implemented in widely used systems such as ACL2, Isabelle etc. [Manolios and Vroon, 2006; Krauss, 2007; Codish et al., 2005, 2010]. Second, SCA is simple enough to achieve a good trade-off between expressiveness and complexity. For example, SCA termination is decidable and ranking functions can be extracted on terminating instances in PSPACE [Ben-Amram, 2011]. The simplicity of SCA sets it apart from other disjunctive abstract domains used for termination/bounds such as transition predicate abstraction [Podelski and Rybalchenko, 2005] and powerset abstract domains used in [Berdine et al., 2007] and in our first approach.

SCA is the key for obtaining efficient algorithms for transitive hull and bound computation that at the same time overcome the above discussed limitations:

Transitive Hull Computation. Due to its built-in disjunctiveness and the transitivity of order relations, SCA is closed under taking transitive hulls, and transitive hulls can be efficiently computed. These properties of SCA give rise to a simple and robust summarization algorithm for inner loops, which we describe in Section 4.2.

Bound Computation. SCA is our key technique for composing global bounds from norms. The literature already knows how to compose global ranking functions from norms [Ben-Amram, 2011]. Therefore our approach is the natural next step. Like in the proof-rule base approach we compute norms locally, i.e., we extract small parts of the program under consideration. However, after the extraction we consider only the size-change-abstracted program for bound computation. Note that in this way we obtain the abstract model for bound analysis that we have been looking for: Instead of searching for better ad hoc proof rules we can systematically study SCA for obtaining better bound algorithms. We give a first bound algorithm built on SCA in Section 4.3, which gives good results in practice and is strictly more general than the proof-rules stated in Section 3.3.

Furthermore SCA is the natural abstract domain to be used in connection with two program transformations that increase the precision of bound analysis of imperative programs. Both transformations make use of the progress in SMT solver technology to reason about the long pieces of straight-line code given by program paths:

Pathwise analysis uses an SMT solver for reasoning over complete program paths. As described earlier pathwise analysis is part of our transition system generation algorithm given in Section 2.5. We want to point out that pathwise analysis is especially effective when combined with specialized abstract domains such as SCA (which is targeted at termination / bound analysis). This is because the specialized abstract domain does not need to be precise enough to reason about complete program paths itself. In this way a separation of concerns is achieved. We explain in Section 4.2.2 why the classical blockwise use of SCA is less precise than our pathwise use of SCA.

Contextualization enriches the state space by adding the information which transition is executed next to every control location. This allows to detect transitions that cannot be executed subsequently. Such transitions are then deleted from the control-flow graph (CFG) of the program. Our bound analysis (described in Section 4.3) uses contextualization (described in Section 4.3.1) as preprocessing step. Contextualization benefits the actual bound computation (described in 4.3.2), which exploits the SCC component graph of the CFG. Since pathwise analysis contracts large paths from l to l into single transitions, contextualization is particularly important after pathwise analysis.

We have implemented the above described approach in our tool `Loopus` that computes bounds for C programs and evaluated it on the compiler optimization cBench [CBenchWebPage, 2010] benchmark. `Loopus` automatically computes bounds for a large percentage of the benchmark programs (details can be found in Section 4.4).

Our Solution addresses the Identified Challenges. We now come back to challenges (A)-(D) that we identified in the beginning of the introduction and explain how our approach addresses them:

For **(A)**, we compute global invariants with standard linear abstract domains. Upper bounds on norms can usually be established by these invariants.

For **(B)**, we use SCA to compute disjunctive summaries of inner loops. Furthermore we compute disjunctive transition systems by enumerating cycle-free paths and contracting the transition relations of these paths. Additionally the program transformation contextualization entails disjunctive information by refining the CFG of the program.

For **(C)**, we extract norms from paths that start from loop headers and go back to the headers. Our experiments (and also those of [Cook et al., 2006]) confirm that these paths have the right granularity to extract norms. Our bound algorithm on size-change abstracted programs then composes bounds on the norms to complex bounds using the operators $+, *, \max$.

For **(D)**, our pathwise-analysis exploits the looping structure of imperative programs. In contrast to the cut-points method it retains enough information to compute precise bounds.

1.2.3 Fundamental Properties of the Size-change Abstraction (Chapter 5)

Finding the right abstraction for a problem is an essential step in computer science, especially for problems that are undecidable. The usefulness of an abstraction of a problem depends on whether many relevant practical instances can be handled by the abstraction (1) and whether the abstraction has decidable properties, gives rise to efficient algorithms, etc. (2).

In Chapter 4 we identify SCA as suitable abstraction for bound analysis and give evidence for (1) by describing practical algorithms for bound analysis based on SCA and by demonstrating the effectiveness of our approach through experiments on benchmark programs. In Chapter 5 we focus on (2) and give theoretical results on SCA towards a characterization of the bounds that can be expressed by SCA.

For our theoretical investigation we follow earlier work on SCA and study size-change systems (SCSs) as an abstract program model. An SCS consists of a CFG whose nodes are labeled by invariants and whose transitions are labeled by size-change relations (SCRs). Invariants and SCRs are sets of inequalities. We define the semantics of SCSs over well-ordered domains, i.e., well-founded linear orders. This is a natural choice: Linear orders ensure that we can compare two elements; thus, we can give semantics to inequalities. Well-foundedness ensures that there are no infinitely decreasing sequences; thus, we can define termination of SCSs. Note that we define the semantics of SCSs directly instead of first defining a concrete program model and then obtaining SCSs as an abstraction of this program model (as we do in Chapter 4).

In Section 5.1 we introduce inequalities and linear orders and discuss their properties. In contrast to earlier work on SCA we strictly separate syntactic and semantic properties in our investigation. These properties lay the foundations for our further investigations.

We introduce SCSs, their semantics and different termination notions in Section 5.2. Every well-ordered domain is equivalent to an ordinal. Therefore we define the semantics of SCSs over ordinals. This is in contrast to earlier work on SCA, which defines the semantics of SCSs over integers. We define a semantic notion of termination of SCSs in the standard way using the well-foundedness of ordinals. We also give a syntactic notion of termination of SCSs that can be used for detecting the semantic termination of SCSs. We state that for SCSs interpreted over sufficiently large ordinals the semantic and syntactic notions of termination coincide. This confirms that our semantics of SCSs is natural and robust.

We discuss asymptotic bounds of SCCs in Section 5.3. We introduce an adequate notion of for-loops for SCCs. Our main result is that these for-loops give rise to polynomial lower bounds of SCCs, which have the form $\Omega(1), \Omega(N), \ldots, \Omega(N^n)$ for an SCS that is interpreted over the natural numbers that are smaller than N and that has n variables. The importance of this result is that a for-loop provides a *complexity witness* for an SCS. We conjecture for-loop provide a complete characterization of the complexity of SCSs: Every SCS either does not terminate or has complexity $\Theta(N^k)$, which is witnessed by some for-loop. Thus we believe that our result is the first step towards a full characterization of the bounds expressible by SCA.

1.3 Related Work

The loop-bound problem (computing bounds on the number of loop iterations) is a special case of the reachability-bound problem – the control location under consideration is the location immediately after the loop header. Several recent papers on the loop-bound problem have been published [Gulwani et al., 2009c,a; Gulavani and Gulwani, 2008; Albert et al., 2008]. However, none of these papers directly addresses the more general reachability-bound problem that we introduce in this thesis.

We give a detailed comparison with earlier work on termination and bound analysis in Chapter 6, most notably the Terminator tool [Cook et al., 2006], and the more recent tools SPEED [Gulwani et al., 2009c,a] and Loopfrog [Kroening et al., 2010, 2011]. We show that our bound analysis captures the essential ideas of these approaches in a simpler framework

```
void main(int n,
          int x, int z)
  while (x < n)
      if (z > x) x++;
      else z++;
```

Program 1.6

```
void main(uint n, uint m)
  Assume(0 < n < m);
  j := n + 1;
  while (j < n ∨ j > n)
      if (j > m) j := 0;
      else j++;
```

Program 1.7

Figure 1.2: Loops from recent work on proving termination [Cook et al., 2006] (left) and loop bound computation [Gulwani et al., 2009a] (right).

and that our technique outperforms these recent approaches on loop-bound computation and termination analysis:

On the one hand, our technique is able to compute bounds for loops whose iterations are affected by inner loops for which existing bound techniques [Gulwani et al., 2009c,a; Gulavani and Gulwani, 2008; Albert et al., 2008] mostly fail; we give examples on which these techniques fail and explain the reasons for failure in Sections 6.1 and 6.9.

On the other hand even in case of loops without inner loops, our technique is able to compute bounds for loops using a much simpler uniform algorithm compared to existing termination techniques or specialized bound computation techniques. Figure 1.2 shows two such examples that have been used as motivating examples by previous techniques. The computation of the transition systems for these examples is almost trivial, and the bounds of the resulting transition systems are easily computed by our techniques. Since bound analysis generalizes termination analysis, many of our methods are relevant for termination.

1.4 Contributions

We summarize the above outline and list the contributions of this dissertation per chapter.

In Chapter 2, we give definitions and state the main steps of our approach:

- We define the reachability-bound problem and the notion of a precise solution to that problem (Section 2.3). This contributes to the problem of defining an entire quantitative program logic, which is part of the *quantitative agenda* set forth recently [Henzinger, 2009].
- We reduce the problem of computing the reachability-bound to the problem of computing the bound of a transition system of a location (Section 2.4).
- We give an algorithm for the generation of a transition system of a location. This algorithm uses the program transformation pathwise analysis, which exploits the loop structure of imperative programs (Section 2.5).

In Chapter 3, we describe our proof-rule based approach:

- We describe an abstract interpretation based iterative algorithm for computing the transitive closure of a transition system (Section 3.1). This algorithm handles the benchmark examples from state-of-the-art papers on computing disjunctive invariants as we demonstrate in our experiments.
- We describe pattern matching techniques that allow to obtain ranking functions of individual transition relations (Section 3.2). We describe

non-iterative proof rules (Section 3.3) that allow to compose the ranking functions of individual transition relations to symbolic bounds of transition systems (Section 3.2).

- We present experimental results evaluating the effectiveness of the proof-rule based approach (Section 3.4).

In Chapter 4, we describe our SCA approach:

- We exploit SCA for bound analysis: We describe how to compute transitive hulls for summarizing inner loops with SCA (Section 4.2). We describe how to compute bounds of size-change abstracted programs (Section 4.3).
- We describe how to apply SCA on imperative programs: We combine SCA with the program transformation pathwise analysis for the generation of precise transition systems (Section 4.2.2). We use the program transformation contextualization as preprocessing step for the computation of precise bounds (Section 4.3.1).
- Our experiments show that we can automatically compute bounds for a large percentage of benchmark programs (Section 4.4).

In Chapter 5, we make the following contributions to the theory of SCA:

- We systematically investigate properties of order relations and linear orders (Section 5.1), strictly separating syntactic and semantic properties in contrast to earlier work on SCA.
- We define non-standard semantics for SCS. We give evidence for the adequacy of our semantics by showing that semantic and syntactic notions of termination of SCSs coincide (Section 5.2).

- We define the notion of for-loops of SCSs. We show that for-loops give rise to lower bounds of SCSs. We state the conjecture that SCSs can express precisely polynomial bounds (Section 5.3).

In Chapter 6 we give a detailed comparison with related work. We show that our bound analysis captures the essential ideas of earlier termination and bound analyses in a simpler framework and that our technique outperforms these approaches on loop-bound computation and termination analysis.

Chapter 2

Problem Definition and Main Steps of the Analysis

2.1 Notation for Sets and Relations

Let A be a set. The concatenation of two relations $B_1, B_2 \in 2^{A \times A}$ is the relation $B_1 \circ B_2 = \{(e_1, e_3) \mid \exists e_2.(e_1, e_2) \in B_1 \wedge (e_2, e_3) \in B_2\}$. $Id(A) = \{(e, e) \mid e \in A\}$ is the *identity relation* over A. Let $B \in 2^{A \times A}$ be a relation. We inductively define the *k-fold exponentiation* of B by $B^k = B^{k-1} \circ B$ and $B^0 = Id(A)$. $B^+ = \bigcup_{k \geq 1} B^k$ resp. $B^* = \bigcup_{k \geq 0} B^k$ is the *transitive*- resp. *reflexive transitive closure* of B. We lift the concatenation operator $\circ$ to sets of relations by defining $\mathcal{C}_1 \circ \mathcal{C}_2 = \{B_1 \circ B_2 \mid B_1 \in \mathcal{C}_1, B_2 \in \mathcal{C}_2\}$ for sets of relations $\mathcal{C}_1, \mathcal{C}_2 \subseteq 2^{A \times A}$. We set $\mathcal{C}^0 = \{Id(A)\}$; $\mathcal{C}^k, \mathcal{C}^+$ etc. are defined analogously.

2.2 Program Model

We introduce a simple program model for sequential imperative programs without procedures. Our definition models explicitly the essential features of imperative programs, namely branching and looping. In Section 2.5 we will explain how to exploit the graph structure of programs in our analysis algorithm. We leave the extension to concurrent and recursive programs for future work.

Definition 1 (Transition Relations / Systems / Invariants)**.** *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* transition set $\mathcal{T} \subseteq \Gamma$ *is a finite set of transition relations. Let* ρ *be a transition relation.* $\mathcal{T}$ *is a* transition system *for* ρ*, if* $\rho \subseteq \bigcup \mathcal{T}$*.* $\mathcal{T}$ *is a* transition invariant *for* ρ*, if* $\rho^* \subseteq \bigcup \mathcal{T}$*.*

Definition 2 (Program, Path, Trace, Termination)**.** *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)*.*

A path *of* P *is a sequence* $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots$ *with* $l_i \xrightarrow{\rho_i} l_{i+1} \in E$ *for all* i*. Let* $\pi = l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} l_2 \cdots l_k \xrightarrow{\rho_k} l_{k+1}$ *be a finite path.* π *is* cycle-free*, if* π *does not visit a location twice except for the end location, i.e.,* $l_i \neq l_j$ *for all* $0 \leq i < j \leq k$*. The* contraction *of* π *is the transition relation* $\texttt{rel}(\pi) = \rho_0 \circ \rho_1 \circ \cdots \circ \rho_k$ *obtained from concatenating all transition relations along* π*. Given a location* l*,* $\texttt{paths}(P, l)$ *is the set of all paths* $l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} l_2 \cdots l_k \xrightarrow{\rho_k} l$ *of* P *with the start and end location* l*. A path* $\pi \in \texttt{paths}(P, l)$ *is* simple*, if all locations, except for the start and end location, are different from* l*.*

A trace *of* P *is a sequence* $(l_0, s_0) \xrightarrow{\rho_0} (l_1, s_1) \xrightarrow{\rho_1} \cdots$ *such that* $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots$ *is a path of* P*,* $s_i \in \Sigma$ *and* $(s_i, s_{i+1}) \in \rho_i$ *for all* i*.* P *is* terminating*, if there is no infinite trace of* P*.*

Note that a cycle-free path $\pi \in \mathtt{paths}(P, l)$ is always simple. Further note that our definition of programs allows to model branching and looping precisely and naturally: imperative programs can usually be represented as CFGs whose edges are labeled with assign and assume statements.

Definition 3 (Transition Relation of a Location)**.** *Let $P = (L, E)$ be a program and $l \in L$ a location. The* transition relation of l *is the set* $P|_l = \bigcup_{simple\ \pi \in \mathtt{paths}(P,l)} \mathtt{rel}(\pi)$.

2.3 The Reachability-Bound Problem

Let $P = (L, E)$ be a program and let $l \in L$ be a location. There are two classical problems associated with the reachability of l inside P:

- **Safety Problem:** Is the control location l never reached/visited for all executions of program P?
- Termination Problem: Is the control location l visited at most a finite number of times for all executions of program P?

In this thesis, we have motivated the following bound problem, which is a generalization of both the safety and termination problem.

- Reachability-bound Problem: Compute a worst-case symbolic bound $b(s)$ on the number of visits to control location l.

The notion of a worst-case symbolic bound is defined below.

Definition 4 (Worst-case Symbolic Bound)**.** *A function $b(s) : \Sigma \to \mathbb{N}$ from the program states to the natural numbers is a worst-case symbolic bound on the number of visits to control location l, if for any trace $(l_0, s_0) \xrightarrow{\rho_0} (l_1, s_1) \xrightarrow{\rho_1} \cdots$ of P there are at most $b(s_0)$ locations l_i with $l_i = l$.*

There may be multiple worst-case symbolic bounds for a given location. It is desirable to produce a bound that is *precise* in the sense that there exists a family $\phi(s)$ of worst-case inputs that exhibit the worst-case bound (up to some constant factor, as motivated by the definition of asymptotic complexity), formally defined as follows:

Definition 5 (Precision of Worst-case Symbolic Bounds)**.** *A worst-case symbolic bound $b(s)$ for l is* precise *(up to multiplicative constant factors), if there exist positive integers c_1, c_2, and a formula $\phi(s)$ such that:*

P1. For any state $s_0 \in \Sigma$ such that $\phi(s_0)$ holds, there is a trace $(l_0, s_0) \xrightarrow{\rho_0} (l_1, s_1) \xrightarrow{\rho_1} \cdots$ of P with at least $\frac{b(s_0)}{c_1} - c_2$ locations l_i with $l_i = l$.

P2. For any integer k, there exists a state s_0 with $\phi(s_0)$ and $b(s_0) > k$. In other words, the formula $\exists s_0 \in \Sigma.(b(s_0) \geq k \wedge \phi(s_0))$ has a satisfying assignment.

We refer to the triple (ϕ, c_1, c_2) as precision-witness *for bound b.*

Note that we have relaxed the definition of precision of bounds to up to multiplicative constants (as motivated by the definition of asymptotic complexity) since it would be almost impossible to find exact closed-form bounds in practice.

The following example explains and motivates the requirements P1 and P2 in the above definition.

Example 6. *A precision-witness for the bound n on the number of times location l_6 is visited in Program 1.3 can be $\phi = \forall m(0 \leq m < n.\ A[m])$, $c_1 = 1$ and $c_2 = 1$ since it can be shown that under the precondition ϕ, location l_6 is visited at least $n - 1$ times.*

A precision-witness for the bound n^2 on the number of times the inner loop (location l_5) is executed can be $\phi = \forall m(0 \leq m < n.\ \neg A[m])$, $c_1 = 4$ and $c_2 = 1$ since it can be shown that under the precondition ϕ, location l_5 is visited at least $n^2/4$ times. This is because, for example, i takes all values between 0 to $n/2 - 1$ at location l_4 (hence the number of visits to location l_2 is at least $n/2$), and for each of those visits, j takes all values between $n/2$ to $n - 1$ at location l_4 (i.e., the number of visits to location l_4 is at least $n/2$). Note that if we did not relax the requirement P1 to allow for constants c_1 and c_2, then computation of a precise bound would have required us to compute the exact bound $\frac{(n-1)(n-2)}{2}$. It would be impractical to find such exact closed-form solutions.

A bound of, say, 100, on the number of times location l_6 is visited is not precise. It may appear that $\phi = (\forall m(0 \leq m < 100.\ A[m]) \wedge n \leq 100)$, $c_1 = 1$ and $c_2 = 1$ is a precision-witness. Note however that it violates requirement P2 since for any $n > 100$, there does not exist a satisfying assignment for the formula $\phi \wedge n > 100$.

In this thesis, we describe an algorithm for computing a worst-case symbolic bound. Automatically establishing the precision of a bound b returned by our algorithm is an orthogonal problem that we leave for future work.

The duality between computing a symbolic bound and finding a witness to show the precision of the bound is similar to the duality between proving a given safety property and finding a concrete counterexample/witness to the violation of a safety property. However, the additional challenge in computing precision-witnesses is that such witness are symbolic and not concrete.

2.4 Main Steps of our Analysis

We now state the main steps of our approach to the reachability-bound problem:

```
Input: program $P = (L, E)$, location $l \in L$
Output: reachability-bound $b$ for $l$
1. Compute global invariants by standard abstract domains
2. Compute $\mathcal{T} = \texttt{TransSys}(P, l)$
3. Compute $b = 1 + \texttt{Bound}(\mathcal{T})$
```

We explain the steps of the above algorithm in the following. Let $P = (L, E)$ be a program and $l \in L$ be a location for which we want to compute a reachability-bound.

- In Step 1 we compute global invariants by standard abstract domains such as interval, octagon or polyhedra. As this step is standard, we do not discuss it in this thesis.

- In Step 2 we compute a transition system $\mathcal{T} = \texttt{TransSys}(P, l)$ for $P|_l$ by Algorithm 1, which is parameterized by a function `TransHull` that computes transition invariants for the summarization of inner loops.

- In Step 3 we compute a bound $b = \texttt{Bound}(\mathcal{T})$ for the transition system $\mathcal{T}$. The reachability-bound b of l is then obtained by adding 1 to the bound of transition system $\mathcal{T}$ to account for the first visit to l.

We give algorithms that implement the functions `TransHull` and `Bound` in Sections 3.1 and 3.3 of the proof-rule based approach, and in Sections 4.2 and 4.3 of the size-change abstraction based approach.

```
**Procedure**: TransSys$(P, l)$
**Input**: a program $P = (L, E)$, a location $l \in L$
**Output**: a transition system for $P|_l$
**Global**: array summary for storing transition invariants
**foreach** $(\mathcal{L}, \mathit{header}) \in$ NestedLoops$(P, l)$ **do**
    $\mathcal{T} :=$ TransSys$(\mathcal{L}, \mathit{header})$;
    summary$[\mathit{header}] :=$ TransHull$(\mathcal{T})$;
**foreach** *cycle-free path* $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} l_2 \cdots l_k \xrightarrow{\rho_k} l \in$ paths$(P, l)$ **do**
    $\mathcal{T}_\pi := \{\rho_0\} \circ$ ITE(IsHeader$(l_1)$, summary$[l_1]$, $\{Id\}) \circ \{\rho_1\} \circ$
                    ITE(IsHeader$(l_2)$, summary$[l_2]$, $\{Id\}) \circ \{\rho_2\} \circ \cdots \circ$
                    ITE(IsHeader$(l_k)$, summary$[l_k]$, $\{Id\}) \circ \{\rho_k\}$;
**return** $\bigcup_{\text{cycle-free path } \pi \in \texttt{paths}(P,l)} \mathcal{T}_\pi$;
```

Algorithm 1: `TransSys`(P, l) computes a transition system for $P|_l$

2.5 Computing Transition Systems

In this section we describe our algorithm for computing transition systems. We first present the actual algorithm, and then discuss specific characteristics. The function `TransSys` in Algorithm 1 takes as input a program $P = (L, E)$ and a location $l \in L$ and computes a transition system for $P|_l$, cf. Theorem 7 below. The key ideas of Algorithm 1 are (1) to summarize inner loops disjunctively by transition invariants, and (2) to enumerate all cycle-free paths for pathwise analysis. Note that for loop summarization the function `TransSys` is recursively invoked. Further note that Algorithm 1 is parameterized by the function `TransHull`. Algorithm 1 relies on the following property of `TransHull`: Given a transition relation ρ and a transition system $\mathcal{T}$ for ρ, then `TransHull`$(\mathcal{T})$ is a transition invariant for ρ.

Loop Summarization. In the first `foreach`-loop, Algorithm 1 iterates over all nested loops of P w.r.t. l. A loop $\mathcal{L}$ of P is a nested loop w.r.t. l, if it is strongly connected to l but does not contain l, and if there is no loop with the same properties that strictly contains $\mathcal{L}$. Let $\mathcal{L}$ be a nested loop of P w.r.t. l and let *header* be its header. (We assume that the program is reducible, see discussion below.) `TransSys` calls itself recursively to compute a transition system $\mathcal{T}$ for $\mathcal{L}|_{header}$. Next Algorithm 1 computes $\texttt{TransHull}(\mathcal{T})$. Note that $\texttt{TransHull}(\mathcal{T})$ is a transition invariant for $\mathcal{L}|_{header}$ because of the above stated assumption on the function `TransHull`. Finally Algorithm 1 stores $\texttt{TransHull}(\mathcal{T})$ in the array summary at location *header*.

After the first `foreach`-loop, Algorithm 1 has summarized all inner loops, not only the nested loops, because the recursive calls reach all nesting levels. For each inner loop $\mathcal{L}$ with header *header* a transition invariant for $\mathcal{L}|_{header}$ has been stored at $\mathsf{summary}[header]$. Summaries of inner loops are visible to all outer loops, because the array summary is a global variable.

Pathwise Analysis. In the second `foreach`-loop, Algorithm 1 iterates over all cycle-free paths of P with start and end location l. Let $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots l_k \xrightarrow{\rho_k} l$ be such a cycle-free path. For each location l_i the expression $\texttt{ITE}(\texttt{IsHeader}(l_i), \mathsf{summary}[l_i], \{Id\})$ evaluates to $\mathsf{summary}[l_i]$, if l_i is the header of an inner loop $\mathcal{L}_i$, and evaluates to the transition set $\{Id\}$, which contains only the identity relation over the program states, else. Algorithm 1 computes the transition set

$$\begin{aligned}\mathcal{T}_\pi = \{\rho_0\} \circ \texttt{ITE}(\texttt{IsHeader}(l_1), \mathsf{summary}[l_1], \{Id\}) \circ \{\rho_1\} \\ \circ \texttt{ITE}(\texttt{IsHeader}(l_2), \mathsf{summary}[l_2], \{Id\}) \circ \{\rho_2\} \circ \cdots \\ \circ \texttt{ITE}(\texttt{IsHeader}(l_k), \mathsf{summary}[l_k], \{Id\}) \circ \{\rho_k\},\end{aligned}$$

which represents the contraction of π, where the summaries of the inner loops $\mathcal{L}_i$ are inserted at their headers l_i. The transition set $\mathcal{T}_\pi$ overapproximates all paths starting and ending in l that iterate arbitrarily often through inner loops along π, because for every loop $\mathcal{L}_i$ the transition set $\mathsf{summary}[l_i]$ overapproximates all paths starting and ending in l_i that iterate arbitrarily often through $\mathcal{L}_i$ (as $\mathsf{summary}[l_i]$ is a transition invariant for $\mathcal{L}_i|_{l_i}$). Algorithm 1 returns the union $\bigcup_{\text{cycle-free path } \pi \in \mathtt{paths}(P,l)} \mathcal{T}_\pi$ of all those transition sets $\mathcal{T}_\pi$.

Theorem 7. *Algorithm 1 computes a transition system* $\mathtt{TransSys}(P, l)$ *for* $P|_l$.

Proof sketch, for a detailed proof see Section 2.6 below. Let $\pi' \in \mathtt{paths}(P, l)$ be a simple path. We obtain a cycle-free path $\pi \in \mathtt{paths}(P, l)$ from π' by deleting all iterations through inner loops of (P, l) from π'. The transition set $\mathcal{T}_\pi$ overapproximates all paths starting and ending in l that iterate arbitrarily often through inner loops of (P, l) along π. As π' iterates through inner loops of (P, l) along π we have $\mathtt{rel}(\pi) \subseteq \bigcup \mathcal{T}_\pi$. □

Implementation. We use conjunctions of formulae to represent individual transitions. This allows us to implement the concatenation of transition relations by conjoining their formulae and introducing existential quantifiers for the intermediate variables. We detect empty transition relations by asking an SMT solver whether their corresponding formulae are satisfiable. We further use SMT solver queries to check if one transition relation is contained in another transition relation. We use these checks at several points during the analysis to reduce the number of transition relations.

Irreducible Programs. Algorithm 1 refers to loop headers, and thus implicitly assumes that loops are reducible. (Recall that in a reducible program each SCC has a unique entry point called the header.) We have formulated Algorithm 1 in this way to make clear how it exploits the loop structure of imperative programs. However, Algorithm 1 can be easily extended to irreducible loops by a case distinction on the (potentially multiple) entry points of the SCCs.

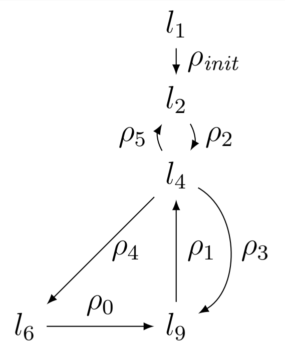
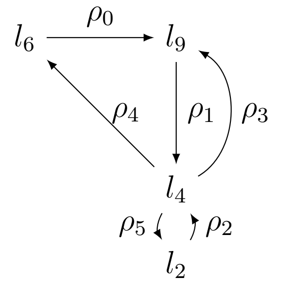
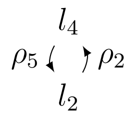
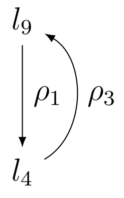
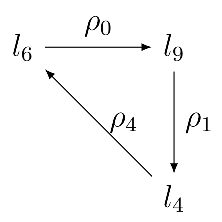
	$\rho_{init} \equiv i' = 0$ $\rho_0 \equiv j' = j - 1 \wedge n' = n - 1$ $\rho_1 \equiv j' = j + 1$ $\rho_2 \equiv i < n \wedge j' = i + 1$ $\rho_5 \equiv j \geq n \wedge i' = i + 1$ $\rho_3 \equiv j < n \wedge \neg A[j]$ $\rho_4 \equiv j < n \wedge A[j]$
	Inner loops of P with regard to l_6: $\mathcal{L} = (\{l_9, l_4\}, \{l_9 \xrightarrow{\rho_1} l_4, l_4 \xrightarrow{\rho_2} l_4, l_4 \xrightarrow{\rho_3} l_9\})$ with header l_9 $\mathcal{L}' = (\{l_4\}, \{l_4 \xrightarrow{\rho_5} l_2, l_2 \xrightarrow{\rho_2} l_4\})$ with header l_4
	$\texttt{TransSys}(\mathcal{L}', l_4) \equiv \{j \geq n \wedge i + 1 < n \wedge i' = i + 1 \wedge j' = i + 2\}$
	$\mathsf{summary}[l_4] \equiv \{i' = i \wedge j' = j \wedge n' = n,$ $j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$ $\texttt{TransSys}(\mathcal{L}, l_9) \equiv$ $\{j + 1 < n \wedge i' = i \wedge j' = j + 1 \wedge n' = n,$ $j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$
	$\mathsf{summary}[l_4] \equiv \{i' = i \wedge j' = j \wedge n' = n,$ $j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$ $\mathsf{summary}[l_9] \equiv \{i' = i \wedge j' \geq j \wedge n' = n,$ $j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$ $\texttt{TransSys}(P, l_6) \equiv \{i' = i \wedge j' \geq j \wedge n' = n - 1 \wedge j' < n,$ $j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2 \wedge n' = n - 1 \wedge j' < n\}$

Figure 2.1: Different stages of the computation of $\texttt{TransSys}(P, l_6)$ by Algorithm 1, where the program P is given by Program 1.3: The first row shows the CFG of P Program 1.3 and its transition relations. The second row shows the CFG of P with regard to location l_6 and its inner loops. The third, fourth and fifth row respectively belong to the nested recursive, recursive and actual call of `TransSys`; they show the part of the CFG which is needed for the enumeration of the cycle-free paths, the used summaries of the inner loops and the computed transition systems.

Example 8. *Let P be Program 1.3 on page 9. In the following we describe how Algorithm 1 computes* $\texttt{TransSys}(P, l_6)$. *We depict the different stages of this computation in Figure 2.1. In the first* `foreach`*-loop Algorithm 1 calls itself recursively on the nested loop* $\mathcal{L} = (\{l_9, l_4\}, \{l_9 \xrightarrow{\rho_1} l_4, l_4 \xrightarrow{\rho_2} l_4, l_4 \xrightarrow{\rho_3} l_9\})$ *with header l_9. In the first* `foreach`*-loop of the recursive call Algorithm 1 calls itself recursively on the nested loop* $\mathcal{L}' = (\{l_4\}, \{l_4 \xrightarrow{\rho_5} l_2, l_2 \xrightarrow{\rho_2} l_4\})$ *with header l_4. In the nested recursive call Algorithm 1 skips the first* `foreach`*-loop because $(\mathcal{L}', l_4)$ does not have nested loops. The second* `foreach`*-loop iterates over all cycle-free paths of* $\texttt{paths}(\mathcal{L}', l_4)$. *There is only one such a path $\pi = l_4 \xrightarrow{\rho_5} l_2 \xrightarrow{\rho_2} l_4$. Algorithm 1 computes*

$$\mathcal{T}_\pi = \{j \geq n \wedge i + 1 < n \wedge i' = i + 1 \wedge j' = i + 2\}$$

and returns $\mathcal{T}_\pi$ as the transition system $\texttt{TransSys}(\mathcal{L}', l_4)$. *After the return of the nested recursive call* $\mathcal{T} = \texttt{TransSys}(\mathcal{L}', l_4)$ *Algorithm 1 computes* $\texttt{TransHull}(\mathcal{T})$. *Let us assume*

$$\texttt{TransHull}(\mathcal{T}) = \{i' = i \wedge j' = j \wedge n' = n,$$
$$j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}.$$

Algorithm 1 then stores $\texttt{TransHull}(\mathcal{T})$ *in* $\mathsf{summary}[l_4]$. *As there is no other nested loop the first* `foreach`*-loop of the recursive call is finished. The second* `foreach`*-loop iterates over all cycle-free paths of* $\texttt{paths}(\mathcal{L}, l_9)$. *There is only one such a path $\pi = l_9 \xrightarrow{\rho_1} l_4 \xrightarrow{\rho_3} l_9$. Algorithm 1 computes*

$$\mathcal{T}_\pi = \{\rho_2\} \circ \mathsf{summary}[l_4] \circ \{\rho_4\} = \{j + 1 < n \wedge i' = i \wedge j' = j + 1 \wedge n' = n,$$
$$j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$$

and returns $\mathcal{T}_\pi$ as the transition system $\texttt{TransSys}(\mathcal{L}, l_9)$. *After the return of the recursive call* $\mathcal{T} = \texttt{TransSys}(\mathcal{L}, l_9)$ *Algorithm 1 computes* $\texttt{TransHull}(\mathcal{T})$.

Let us assume

$$\texttt{TransHull}(\mathcal{T}) = \{i' = i \wedge j' \geq j \wedge n' = n,$$
$$j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2\}$$

Algorithm 1 then stores $\texttt{TransHull}(\mathcal{T})$ *in* $\mathsf{summary}[l_9]$. *As there is no other nested loop the first* `foreach`*-loop is finished. The second* `foreach`*-loop iterates over all cycle-free paths of* $\texttt{paths}(P, l_6)$. *There is only one such path* $\pi = l_6 \xrightarrow{\rho_0} l_9 \xrightarrow{\rho_1} l_4 \xrightarrow{\rho_4} l_6$. *Algorithm 1 computes*

$$\begin{aligned}
\mathcal{T}_\pi &= \{\rho_0\} \circ \mathsf{summary}[l_9] \circ \{\rho_1\} \circ \mathsf{summary}[l_4] \circ \{\rho_3\} \\
&= \{i' = i \wedge j' \geq j \wedge n' = n - 1 \wedge j' < n, \\
&\qquad j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2 \wedge n' = n - 1 \wedge j' < n, \\
&\qquad j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 3 \wedge n' = n - 1 \wedge j' < n, \\
&\qquad j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2 \wedge n' = n - 1 \wedge j' < n\} \\
&= \{i' = i \wedge j' \geq j \wedge n' = n - 1 \wedge j' < n, \\
&\qquad j \geq n \wedge i + 1 < n \wedge i' \geq i + 1 \wedge j' \geq i + 2 \wedge n' = n - 1 \wedge j' < n\}
\end{aligned}$$

Note that $\{\rho_0\} \circ \mathsf{summary}[l_9] \circ \{\rho_1\} \circ \mathsf{summary}[l_4] \circ \{\rho_3\}$ *resulted into four transition relations because* $\mathsf{summary}[l_9]$ *and* $\mathsf{summary}[l_4]$ *contain two transition relations each. Further note that these four transition relations were contracted into two transition relations by checking for pairwise set containment. This check can be easily implemented by an SMT solver query and reduces the number of transition relations. Algorithm 1 then returns* $\mathcal{T}_\pi$ *as transition system* $\texttt{TransSys}(P, l_6)$ *for* $P|_{l_6}$.

We give another example application of Algorithm 1 in Section 4.2. The example application in Section 4.2 computes a transition system for a loop with multiple cycle-free paths and includes an instantiation of `TransHull` for computing transition invariants.

2.5.1 Disjunctiveness in Algorithm 1

Disjunctiveness is crucial for bound analysis as discussed in the introduction. Our transition system computation algorithm (Algorithm 1) incorporates disjunctive reasoning in two ways:

(1) *We summarize inner loops disjunctively.* Given a transition system $\mathcal{T}$ for some inner loop $\mathcal{L}$, we want to summarize $\mathcal{L}$ by a transition invariant. The most precise transition invariant $\mathcal{T}^* = \{Id\} \cup \mathcal{T} \cup \mathcal{T}^2 \cup \mathcal{T}^3 \cup \cdots$ introduces infinitely many disjunctions and is not computable in general. However, by overapproximating the infinite disjunction $\mathcal{T}^*$ by the finite disjunction $\texttt{TransHull}(\mathcal{T})$ we retain disjunctive information in loop summaries.

(2) *We summarize local transition relations disjunctively.* Given a program $P = (L, E)$ and location $l \in L$, we want to compute a transition system for $P|_l$. For a cycle-free path $\pi \in \texttt{paths}(P, l)$ the transition set $\mathcal{T}_\pi$ computed in Algorithm 1 overapproximates all simple paths in $\texttt{paths}(P, l)$ that iterate through inner loops along π. Because all $\mathcal{T}_\pi$ are sets, the set union $\bigcup_{\text{cycle-free path } \pi \in \texttt{paths}(P,l)} \mathcal{T}_\pi$ is a disjunctive summarization of all $\mathcal{T}_\pi$ that keeps the information from different paths separated. This is important for our analysis which relies on the observation that monotonic changes of norms can be observed along single paths from loop header back to the header.

2.5.2 Pathwise Analysis in Algorithm 1

It is well-known that analyzing large program parts jointly improves the precision of static analyses, e.g. [Colby and Lee, 1996]. Owing to the progress in SMT solvers this idea has recently seen renewed interest by static analyses such as abstract interpretation [Monniaux, 2009] and software model checking [Beyer et al., 2009], which use SMT solvers for abstracting large blocks of straight-line code jointly to increase the precision of the analysis.

We call the analyses of [Monniaux, 2009; Beyer et al., 2009] *blockwise*, because they do joint abstraction only for loop-free program parts. In contrast, our *pathwise analysis* abstracts complete paths at once: Algorithm 1 enumerates all cycle-free paths from loop header to loop header and inserts summaries for inner loops on these paths. These paths are then abstracted jointly in a subsequent loop summarization or bound computation. In this way our pathwise analysis is more precise than blockwise analysis. We illustrate this difference in precision in Section 4.2.2 on SCA.

Implementation. Pathwise analysis may lead to an exponential blow up in size because of the enumeration of all cycle-free paths and the insertion of the disjunctive summaries of inner loops on these paths. We observed in our experiments that by first extracting norms from the program under scrutiny and then slicing the program w.r.t. all statements on which these norms are control dependent [Muchnick, 1997] before continuing with the analysis normally results into programs small enough for making our analysis feasible.

2.6 Proof of Theorem 7

We prove a stronger statement than the one stated in Theorem 7: for every program $P = (L, E)$ and location $l \in L$ it holds that $\texttt{TransSys}(P, l)$ is a transition system for $P|_l$ and that for every inner loop $\mathcal{L}$ of P w.r.t. l with header *header* the transition set $\mathsf{summary}[\mathit{header}]$ is a transition invariant for $\mathcal{L}|_{\mathit{header}}$ (*). The stronger statement has the advantage to be inductive, whereas the statement of Theorem 7 is not. Our proof of (*) proceeds by induction on the loop nesting structure of programs.

Let $P = (L, E)$ be a program and $l \in L$ be some location. In the base case, there are no inner loops of P w.r.t. l. Let $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots l_k \xrightarrow{\rho_k} l \in$ $\mathtt{paths}(P, l)$ be a path with start and end location l. Because P does not have inner loops w.r.t. l, π is cycle free. Thus, no location l_i is the header of an inner loop. Therefore we have $\mathtt{ITE}(\mathtt{IsHeader}(l_i), \mathsf{summary}[l_i], \{Id\}) = \{Id\}$ for all i. Hence,

$$\begin{aligned}
\{\mathtt{rel}(\pi)\} &= \{\rho_0 \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_k\} \\
&= \{\rho_0\} \circ \{Id\} \circ \{\rho_1\} \circ \{Id\} \circ \{\rho_2\} \circ \cdots \circ \{Id\} \circ \{\rho_k\} \\
&= \{\rho_0\} \circ \mathtt{ITE}(\mathtt{IsHeader}(l_1), \mathsf{summary}[l_1], \{Id\}) \circ \\
&\qquad \{\rho_1\} \circ \mathtt{ITE}(\mathtt{IsHeader}(l_2), \mathsf{summary}[l_2], \{Id\}) \circ \{\rho_2\} \circ \cdots \circ \\
&\qquad\qquad \mathtt{ITE}(\mathtt{IsHeader}(l_k), \mathsf{summary}[l_k], \{Id\}) \circ \{\rho_k\}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\bigcup \mathtt{TransSys}(P, l) &= \bigcup_{\text{cycle-free path } \pi \in \mathtt{paths}(P,l)} \mathtt{rel}(\pi) \\
&= \bigcup_{\pi \in \mathtt{paths}(P,l)} \mathtt{rel}(\pi) = P|_l
\end{aligned}$$

Thus $\mathtt{TransSys}(P, l)$ is a transition system for $P|_l$.

.

In the inductive case, there are nested loops of P w.r.t. l. Let $\mathcal{L}$ be a nested loop of P w.r.t. l and let *header* be its header.

We show that $\mathsf{summary}[\mathit{header}]$ is a transition invariant for $\mathcal{L}|_{\mathit{header}}$. By the induction hypothesis we have that $\mathcal{T} = \mathtt{TransSys}(\mathcal{L}, \mathit{header})$ is a transition system for $\mathcal{L}|_{\mathit{header}}$. Because our assumption on the function $\mathtt{TransHull}$, we have that $\mathtt{TransHull}(\mathcal{T})$ is a transition invariant for $\mathcal{L}|_{\mathit{header}}$. Thus $\mathsf{summary}[\mathit{header}]$ is a transition invariant for $\mathcal{L}|_{\mathit{header}}$.

By the induction hypothesis we further have that for all inner loops $\mathcal{L}'$ of $\mathcal{L}$ w.r.t *header* with header *header'* the transition set $\mathsf{summary}[\mathit{header}']$ is a transition invariant for $\mathcal{L}'|_{\mathit{header}'}$.

Thus we have that for every inner loop $\mathcal{L}$ of P w.r.t. l with header *header* the transition set $\mathsf{summary}[header]$ is a transition invariant for $\mathcal{L}|_{header}$.

It remains to show that $\mathtt{TransSys}(P,l)$ is a transition system for $P|_l$. If suffices to show that for every path $\pi \in \mathtt{paths}(P,l)$ we have $\mathtt{rel}(\pi) \subseteq \bigcup \mathtt{TransSys}(P,l)$. Let $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots l_k \xrightarrow{\rho_k} l \in \mathtt{paths}(P,l)$ be a path of P with start and end location l. In the following we iteratively remove iterations through inner loops from π to obtain a cycle-free path. Let i_1 be the first index such that l_{i_1} appears multiple times in π. Let $\mathcal{L}_1$ be the innermost loop of P w.r.t. l that contains l_{i_1}. Because P w.r.t. l is reducible, there is a unique loop header *header* of $\mathcal{L}_1$. Because *header* is a dominator for l_{i_1} every path from l to l_{i_1} must visit *header* before. Because $\mathcal{L}_1$ is the innermost loop that contains l_{i_1}, every path of P that starts and ends in l_{i_1} must visit *header*. Therefore π also visits *header* multiple times. As l_{i_1} is the first location visited multiple times we must have $l_{i_1} = header$. Let j_1 be the last index such that $l_{j_1} = l_{i_1}$. We denote by $q_1 = \pi[i_1, j_1] = l_{i_1} \xrightarrow{\rho_{i_1}} l_{i_1+1} \xrightarrow{\rho_{i_1}+1} \cdots l_{j_1-1} \xrightarrow{\rho_{j_1}} l_{j_1}$ the subpath of π from index i_1 to index j_1. We have that $q_j \in \mathtt{paths}(\mathcal{L}_1, l_{i_1})$ is some iteration through the inner loop $\mathcal{L}_1$ with header l_{i_1}. Let π_1 be the result of deleting the subpath from index $i_1 + 1$ to index j_1 of π. π_1 is a path because $l_{i_1} = l_{j_1}$. Note that π_1 does not contain l_{i_1} multiple times any more, but does contain l_{i_1} exactly once. We iterate this approach to derive indices $i_2, j_2, i_3, j_3, \ldots, i_m, j_m$ and paths $q_2, \pi_2, q_3, \pi_3, \ldots, q_m, \pi_m$ until π_m does not contain a location that appears multiple times. By induction assumption we have that $\mathsf{summary}[l_{i_j}]$ is a transition invariant for $\mathcal{L}_j|_{l_{i_j}}$ for all $1 \leq j \leq m$.

Thus $\mathtt{rel}(q_j) \subseteq \mathcal{L}_j|^*_{l_{i_j}} \subseteq \bigcup \mathsf{summary}[l_{i_j}]$ for all $1 \leq j \leq m$. This gives us

$$\begin{aligned}
\mathtt{rel}(\pi) &= \rho_0 \circ \rho_1 \circ \cdots \circ \rho_k \\
&= \rho_0 \circ \rho_1 \circ \cdots \circ \rho_{i_1-1} \circ \rho_{i_1} \circ \cdots \circ \rho_{j_1} \circ \rho_{j_1+1} \circ \cdots \\
&\quad \circ \rho_{i_m-1} \circ \rho_{i_m} \circ \cdots \circ \rho_{j_m} \circ \rho_{j_m+1} \circ \cdots \circ \rho_k \\
&= \rho_0 \circ \rho_1 \circ \cdots \circ \rho_{i_1-1} \circ \mathtt{rel}(q_1) \circ \rho_{j_1+1} \circ \cdots \\
&\quad \circ \rho_{i_m-1} \circ \mathtt{rel}(q_m) \circ \rho_{j_m+1} \circ \cdots \circ \rho_k \\
&\subseteq \bigcup (\{\rho_0\} \circ \{\rho_1\} \circ \cdots \circ \{\rho_{i_1-1}\} \circ \mathsf{summary}[l_{i_1}] \circ \{\rho_{j_1+1}\} \circ \cdots \\
&\qquad \circ \{\rho_{i_m-1}\} \circ \mathsf{summary}[l_{i_m}] \circ \{\rho_{j_m+1}\} \circ \cdots \circ \{\rho_k\}) \\
&= \bigcup (\{\rho_0\} \circ \mathtt{ITE}(\mathtt{IsHeader}(l_1), \mathsf{summary}[l_1], \{Id\}) \circ \{\rho_1\} \\
&\qquad \circ \mathtt{ITE}(\mathtt{IsHeader}(l_2), \mathsf{summary}[l_2], \{Id\}) \circ \{\rho_2\} \circ \cdots \\
&\qquad \circ \mathtt{ITE}(\mathtt{IsHeader}(l_k), \mathsf{summary}[l_k], \{Id\}) \circ \{\rho_k\}) \\
&= \bigcup \mathtt{TransSys}(P, l).
\end{aligned}$$

This concludes the proof of (*).

Chapter 3

Proof-rule based Approach

In this chapter we present our first approach to the reachability-bound problem. We describe our abstract interpretation based algorithm for computing transitive hulls in Section 3.1. We present proof rules for computing ranking functions of individual transition relations in Section 3.2. We give proof rules for composing the ranking functions of individual transition relations to global bounds in Section 3.3. We present experimental results in Section 3.4.

3.1 Transitive Closure Computation

In this section, we give an algorithm for the function `TransHull` required by Algorithm 1 for computing transition invariants, which is based on abstract interpretation and computes transition invariants for transition sets.

Definition 9 (Transition Invariant for a Transition Set). *Let $\mathcal{T} = \{\rho_1, \dots, \rho_n\}$ be a transition set. A transition set $\mathcal{T}' = \{\rho'_1, \dots, \rho'_m\}$ is a* transition invariant *for $\mathcal{T}$, if*

- *$Id \subseteq \bigcup\{\rho'_1, \dots, \rho'_m\}$, and*
- *$\rho'_j \circ \rho_i \subseteq \bigcup\{\rho'_1, \dots, \rho'_m\}$ for all $i \in \{1, .., n\}$ and $j \in \{1, .., m\}$*

Transition invariants of transition systems overapproximate the transitive closures of the respective transition relations:

Proposition 10. *If $\mathcal{T}$ is a transition system for a transition relation ρ, and if $\mathcal{T}'$ is a transition invariant for $\mathcal{T}$, then $\mathcal{T}'$ is a transition invariant for ρ.*

Example 11. *In Example 8 on page 37 we have computed the transition system $\mathcal{T} = \texttt{TransSys}(\mathcal{L}', l_5)$ for $\mathcal{L}'|_{l_5}$. We have assumed that the function* `TransHull` *returns the transition set $\mathcal{T}' = \{i' = i \wedge j' = j \wedge n' = n, j \geq n \wedge i+1 < n \wedge i' \geq i+1 \wedge j' \geq i+2\}$. Note that $\mathcal{T}'$ is a transition invariant for $\mathcal{T}$. By Proposition 10 $\mathcal{T}'$ is a transition invariant for $\mathcal{L}'|_{l_5}$. $\mathcal{T}'' = \{i' \geq i \wedge n' = n\}$ would be another choice for a transition invariant for $\mathcal{T}$. However, $\mathcal{T}''$ is not as precise as $\mathcal{T}'$ and would lead to a transition system for $P|_{l_6}$ for which no bound exists.*

Computing transition invariants for transition systems is equivalent to computing invariants on loops whose transitions are the transition relations of a transition system. We have seen in Example 11 that such invariants need to be precise, in particular disjunctive. Several papers have been published on discovering disjunctive invariants [Handjieva and Tzolovski, 1998; Popeea and Chin, 2006; Gopan and Reps, 2006, 2007; Beyer et al., 2007; Rival and Mauborgne, 2007; Gulwani et al., 2009a]. The algorithm we present below takes advantage of its particular application to bound analysis. (We also remark that our technique can be used in general for proving safety properties of programs. In Section 3.4.2, we present preliminary results that demonstrate the effectiveness of our technique on a set of benchmark examples taken from a variety of recent literature on generating disjunctive invariants.)

Our algorithm for the computation of precise transitive closures is inspired by a *convexity-like* assumption that we found to hold true for all examples we have come across in practice. (This includes the transitive hulls of the transition systems of inner loops, as well as the benchmarks considered by previous work on computing disjunctive invariants.)

Recall that a theory is said to be *convex* iff for every quantifier-free formula ϕ in that theory, if ϕ implies a disjunction of equalities, then it implies one of those equalities, i.e.,

$$\left(\phi \Rightarrow \left(\bigvee_i (x_i = y_i)\right)\right) \implies \left(\bigvee_i (\phi \Rightarrow (x_i = y_i))\right).$$

By similarly distributing implication (i.e. $\subseteq$) over disjunctions (i.e. $\bigcup$) in the equations of Definition 9, we obtain the *convexity-like* assumption, which is defined formally below.

Definition 12 (Convexity-like Assumption)**.** *Let $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$ be a transition set and let $\mathcal{T}' = \{\rho'_1, \ldots, \rho'_m\}$ be a transitive closure of $\mathcal{T}$. We say that $\mathcal{T}'$ satisfies the* convexity-like assumption*, if there exists an integer $\delta \in \{1, .., m\}$ and a map $\sigma : \{1, .., m\} \times \{1, .., n\} \mapsto \{1, .., m\}$ such that*

- *$Id \subseteq \rho'_\delta$, and*
- *$\rho'_j \circ \rho_i \subseteq \rho'_{\sigma(j,i)}$ for all $i \in \{1, .., n\}$ and $j \in \{1, .., m\}$.*

We refer to such a tuple (δ, σ) as convexity-witness *of $\mathcal{T}'$.*

Note that the convexity-like assumption essentially implies that no case-split reasoning is needed to prove the inductiveness of transitive closures. For example, given a transition set $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$ and a transition set $\mathcal{T}' = \{\rho'_1, \ldots, \rho'_m\}$ we want to show that $\mathcal{T}'$ is a transition invariant for $\mathcal{T}$. Given two transitions $\rho'_j \in \mathcal{T}'$ and $\rho'_i \in \mathcal{T}$ we have to show $\rho'_j \circ \rho_i \subseteq \bigcup\{\rho'_1, \ldots, \rho'_m\}$ according to Definition 9. This may require case splits because $\rho'_j \circ \rho_i$ does not need to be included in only one of the ρ'_k. By the convexity-like assumption there is an index $k = \sigma(j, i)$ such that $\rho'_j \circ \rho_i \subseteq \rho'_k$. This releases us from doing a case split on $\rho'_j \circ \rho_i$ and justifies the naming of the map σ as convexity-witness.

Example 13. *In Figure 3.1 we show some examples that we found in a .Net base-class library. All the transitive hulls of the transition systems given in Figure 3.1 satisfy the convexity-like assumption*

The transitive hulls of Example 8 satisfy the convexity-like assumption. A convexity-witness for the transitive hull

$$\begin{aligned}\mathcal{T}' = \texttt{TransHull}(\mathcal{T}) = \{&i' = i \wedge j' = j \wedge n' = n,\\ &j \geq n \wedge i+1 < n \wedge i' \geq i+1 \wedge j' \geq i+2\}\end{aligned}$$

of the transition system

$$\mathcal{T} = \texttt{TransSys}(\mathcal{L}', l_5) = \{j \geq n \wedge i+1 < n \wedge i' = i+1 \wedge j' = i+2\}$$

is $\delta = 1$ *and* $\sigma = \{(1,1) \mapsto 1, (2,1) \mapsto 2\}$.

A convexity-witness for the transitive hull

$$\begin{aligned}\mathcal{T}' = \texttt{TransHull}(\mathcal{T}) = \{&i' = i \wedge j' \geq j \wedge n' = n,\\ &j \geq n \wedge i+1 < n \wedge i' \geq i+1 \wedge j' \geq i+2\}\end{aligned}$$

of the transition system

$$\begin{aligned}\mathcal{T} = \texttt{TransSys}(\mathcal{L}, l_9) = \{&j+1 < n \wedge i' = i \wedge j' = j+1 \wedge n' = n,\\ &j \geq n \wedge i+1 < n \wedge i' \geq i+1 \wedge j' \geq i+2\}\end{aligned}$$

is $\delta = 1$ *and* $\sigma = \{(1,1) \mapsto 1, (2,1) \mapsto 2, (1,2) \mapsto 2, (2,2) \mapsto 2\}$.

Algorithm 2 performs abstract interpretation over the powerset extension [Cousot and Cousot, 1979] of an underlying abstract domain (such as polyhedra [Cousot and Halbwachs, 1978], octagons [Miné, 2006], conjunctions

Program 3.1

```
void main(uint $n$, uint $m$){
  while($n > 0 \wedge m > 0$){
    $n$--; $m$--;
    while(nondet()){
      $n$--; $m$++;
    }
  }
}
```

$\{n' \leq n \ \wedge \ m' \geq m\}$

$\{n > 0 \wedge m > 0 \wedge n' \leq n - 1\}$

n

Program 3.2

```
void main(uint $n$, int[] $A$){
  while($n > 0$){
    int $t := A[n]$;
    while($n > 0 \wedge t = A[n]$)
      $n$--;
  }
}
```

$\{n' \leq n$

$\{n > 0 \wedge n' \leq n \wedge A[n] \neq A[n'],$
$n > 0 \wedge n' \leq 0\}$

n

Program 3.3

```
void main(uint $n$){
  bool flag := true;
  while(flag){
    flag := false;
    while($n > 0 \wedge$ nondet()){
        $n$--;
        flag := true;
    }
  }
}
```

$\{n' \leq n - 1 \wedge \texttt{flag}',$
$n' = n \wedge \texttt{flag}' = \texttt{flag}\}$

$\{\texttt{flag} \wedge \texttt{flag}' \wedge n > 0 \wedge n' \leq n - 1,$
$\texttt{flag} \wedge \neg\texttt{flag}' \wedge n' = n\}$

$n + 1$

Program 3.4

```
  int $i := 0$;
  while ($i < n$){
    flag := false;
    while(nondet()){
      if(nondet()){
        flag:=true; $n$--; }
    }
    if($\neg$flag) $i$++;
  }
}
```

$\{n' \leq n - 1 \wedge \texttt{flag}' \wedge i' = i$
$i' = i \wedge n' = n \wedge \texttt{flag}' = \texttt{flag}\}$

$\{i < n \wedge \texttt{flag}' \wedge n' \leq n - 1 \wedge i' = i,$
$i < n \wedge \neg\texttt{flag}' \wedge i' \geq i + 1 \wedge n' = n\}$

n

Figure 3.1: Programs from .Net class libraries that have outer loops whose iterators are modified by inner loops. For each loop, the second row shows the transitive closure of the inner loop computed by Algorithm 2 that is precise enough for enabling bound computation of the outer loop; the third row shows the transition-system generated by Algorithm 1 using the transitive closure for summarizing the inner loop; the fourth row shows the bound computed from the transition-system by Algorithm 3.

```
Procedure: TransHull($\mathcal{T}$)
Input: a transition set $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$
Output: transitive hull $\mathcal{T}' = \{\rho'_1, \ldots, \rho'_m\}$ of $\mathcal{T}$
$\rho'_\delta := Id(\Sigma)$;
for $j \in \{1, .., m\} \setminus \{\delta\}$ do $\rho'_j := \emptyset$;
repeat
    for $i \in \{1, .., n\}, j \in \{1, .., m\}$ do
        $\rho'_{\sigma(j,i)} :=$ Join($\rho'_{\sigma(j,i)}, \rho'_j \circ \rho_i$);
until $\mathcal{T}'$ has not changed;
return $\mathcal{T}'$;
```

Algorithm 2: `TransHull`($\mathcal{T}$) computes a transition invariant for $\mathcal{T}$.

of a given set of predicates), where the elements are restricted to at most m disjuncts. We assume that the underlying abstract domain is equipped with a `Join` operator, which takes two elements and returns the least upper bound of both elements. The algorithm uses the map σ to determine how to merge the $n \times m$ different disjuncts (into m disjuncts) that are obtained after the propagation of m disjuncts across n transitions using the `Join` operator.

The key distinguishing feature of Algorithm 2 from earlier work on computing disjunctive invariants is that our algorithm uses solely a syntactic criterion to merge disjuncts, which is given by the map σ. This is contrast to most of the earlier work on computing disjunctive invariants, which uses a semantic criterion based on the notion of differences between disjuncts. The following theorem states the remarkable result that on transition invariants that satisfy the convexity-like assumption no semantic merging criterion can be more powerful than a static syntactic criterion for merging data-flow facts.

Theorem 14 (Precision of Algorithm 2)**.** *Let $\mathcal{T}'' = \{\rho''_1, \ldots, \rho''_m\}$ be a transition invariant for a given transition set $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$ that satisfies the convexity-like assumption. Given the number of disjuncts m and a convexity-witness (δ, σ), Algorithm 2 outputs a transition invariant that is at least as* *precise as* $\mathcal{T}'' = \{\rho''_1, \ldots, \rho''_m\}$.

Proof. We prove $\rho'_k \subseteq \rho''_k$ for all $1 \leq k \leq m$ by induction on the number of steps of Algorithm 2.

Base Case: Algorithm 2 initializes $\rho'_j = \emptyset$ for $j \neq \delta$, and $\rho'_j = Id$ for $j = \delta$. By the definition of the convexity-like assumption we have $Id \subseteq \rho''_\delta$. Thus the claim holds for the base case.

Induction case: By the induction assumption we have $\rho'_k \subseteq \rho''_k$ for all $1 \leq k \leq m$. Assume that Algorithm 2 computes $\rho'_{\sigma(j,i)} := \texttt{Join}(\rho'_{\sigma(j,i)}, \rho'_j \circ \rho_i)$ for some $1 \leq j \leq m$ and $1 \leq i \leq m$. By the definition of the convexity-like assumption we have $\rho''_j \circ \rho_i \subseteq \rho''_{\sigma(j,i)}$. This gives us $\texttt{Join}(\rho'_{\sigma(j,i)}, \rho'_j \circ \rho_i) \subseteq \texttt{Join}(\rho''_{\sigma(j,i)}, \rho''_j \circ \rho_i) \subseteq \texttt{Join}(\rho''_{\sigma(j,i)}, \rho''_{\sigma(j,i)}) = \rho''_{\sigma(j,i)}$ using the induction assumption and the monotonicity and idempotence of the join operator. Thus the claim holds for the induction case. $\square$

There are two issues with Algorithm 2 that we discuss below.

Abstract Domains with Infinite Height. Algorithm 2 may not terminate on domains with infinite height. The standard solution would be to the apply a `Widen` operator (as defined in [Cousot and Cousot, 1977]) in place of the `Join` operator, in order to enforce termination.

Since the use of widening may overapproximate the least fixed point in general, it is no longer possible to formally prove precision results as in Theorem 14. However, we show experimentally (in Section 3.4.2) that our algorithm is able to compute sufficiently precise invariants with the use of standard widening techniques when applied on benchmarks taken from recent work on computing disjunctive invariants. It would be interesting to theoretically investigate conditions under which widening preserves enough precision.

Choice of m and a convexity-witness (δ, σ). Since we do not know the number of disjuncts m and the convexity-witness (δ, σ) of the desired transitive closure upfront, we have two options:

- We enumerate all possible (δ, σ) for a specifically chosen m: There are m^{mn} such possible maps since without loss of generality, we can assume that δ is 1. If m and n are small constants, say 2 (which is quite often an important special case), then there are 16 possibilities. Each choice for σ and δ results in some transitive closure computation by the algorithm. One can then select the strongest transitive closure among the various transitive closures thus obtained (or heuristically select between incomparable transitive closures). However, if m or n is large, then this approach quickly becomes prohibitive.

- We use heuristics for choosing m, δ, σ: The following heuristic turns out to be effective for our application of bound computation. We set m and δ to $n+1$, and select the map σ from the DAG of dependencies between the transition relations of $\mathcal{T}$. We generate such a DAG from a successful bound computation of $\mathcal{T}$ by Algorithm 3 described in Section 3.3. In particular, we set $\sigma(n+1, i) := i$ and $\sigma(i, i) := i$ for all $i \in \{1, .., n\}$, and set $\sigma(i, j) := i$ for all $i, j \in \{1, .., n\}$ except when $\neg\texttt{NI}(\rho_j, \rho_i, r)$, where $r \in \texttt{RankC}(\rho_i)$ is the ranking function that contributed to the bound computation of $\mathcal{T}$, in which case we set $\sigma(i, j) := j$. It can be proved that such a choice of the map δ and σ generates a transitive closure $\texttt{TransHull}(\mathcal{T})$ such that Algorithm 3 can compute a bound of the transition system $(\texttt{TransHull}(\mathcal{T}) \circ \mathcal{T})$, provided Algorithm 3 was able to generate a bound of the transition system $\mathcal{T}$.

  Transitive closures computed by this heuristic preserve important relationships between the program variables. Therefore, when used as summaries of inner loops, the transitive closures keep enough information such that bounds can be computed. For example, the required transitive-closures of the transition systems in Example 8 and Figure 3.1 can be computed by Algorithm 2 using the above heuristic for the construction of a convexity-witnesses.

3.2 Ranking Function for a Transition

In this section, we show how to compute *ranking functions* for transition relations. These ranking functions are made use of by the bound computation algorithm described in Section 3.3.

Definition 15 (Ranking Function for a Transition Relation)**.** *Let ρ be a transition relation. A function $r : \Sigma \to \mathbb{Z}$ from the program states to the integers is a* ranking function *for ρ, if it holds for all pairs of states $(s_1, s_2) \in \rho$, that*

- *$r(s_1) \geq 0$, and*
- *$r(s_1) > r(s_2)$.*

*We denote this by **`Rank`**(ρ, r).*

A ranking function r_1 is more precise *than a ranking function r_2, if $r_1(s_1) \leq r_2(s_1)$ for all $(s_1, s_2) \in \rho$.*

`Rank`(ρ, r) has the following intuitive meaning: r is bounded from below by 0 whenever ρ is enabled, and r decreases by at least 1 for every execution of ρ.

We discuss below the design of a functionality `RankC` that takes a transition relation ρ as input and outputs a set of ranking functions r for that transition. We use a pattern-matching based technique that relies on asking queries that can be discharged by an SMT solver. We found this technique to be effective (fast and precise) for almost all of examples we encountered throughout our experiments. However, other techniques, such as constraint-based techniques [Podelski and Rybalchenko, 2004a] or counter instrumentation enabled iterative fixed-point computation based techniques [Gulavani and Gulwani, 2008; Gulwani et al., 2009c] can also be used for generating ranking functions. Clearly, there are examples where the constraint-based or iterative techniques that perform precise arithmetic reasoning are more precise, but nothing beats the versatility of simple pattern matching that can handle non-arithmetic patterns with equal ease.

We list below some patterns that we found to be most effective. In these patterns we assume that the transition relation ρ is given as a conjunctive formula over the variables X and X'.

3.2.1 Arithmetic Iteration Patterns

One standard way to iterate over loops is to use an arithmetic counter. Ranking functions for such an iteration pattern can be computed using the following pattern.

$$\boxed{\text{If } \rho \Rightarrow (e > 0 \ \wedge \ e[X'/X] < e), \text{ then } e \in \texttt{RankC}(\rho)}$$

The candidates for expression e while applying the above pattern are restricted to those expressions that only involve variables from X and that occur syntactically in ρ as an operator of conditionals when normalized to the form $(e > 0)$, after rewriting a conditional of the form $(e_1 > e_2)$ to $(e_1 - e_2 > 0)$. In the following we give example transitions whose ranking functions can be computed using an application of this pattern:

- $\texttt{RankC}(i'{=}i{+}1 \wedge i{<}n \wedge i{<}m \wedge n'{=}n \wedge m'{\leq}m){=}\{n{-}i{,}m{-}i\}$
- $\texttt{RankC}(n > 0 \ \wedge \ n' \leq n \ \wedge \ A[n] \neq A[n']) = \{n\}$

The second example transition above (obtained from the transition system generated for the loop in Program 3.2 in Figure 3.1) is a good illustration of how simple pattern matching is used to guess a ranking function. An SMT solver (that can reason about combination of theory of linear arithmetic and theory of arrays) can be used to perform the relatively complicated reasoning of verifying the ranking function over a loop-free code fragment.

Another common arithmetic pattern is the use of a multiplicative counter whose value doubles or halves in each iteration (as in case of binary search). A more precise ranking function for such a transition can be computed by using the pattern below.

$$\boxed{\text{If } \rho \Rightarrow (e \geq 1 \ \wedge \ e[X'/X] \leq e/2), \text{ then } \log e \in \texttt{RankC}(\rho)}$$

The candidates for expression e while applying the above pattern are restricted to those expressions that only involve variables from X and those that occur syntactically in ρ as an operator of conditionals when normalized to the form $(e > 1)$, after rewriting a conditional of the form $(e_1 > e_2)$ that occurs in ρ to $(\frac{e_1}{e_2} > 1)$, provided e_2 is known to be positive. In the following we give example transitions whose ranking functions can be computed using an application of this pattern:

- $\texttt{RankC}(i' \leq i/2 \ \wedge \ i > 1) = \{\log i\}$
- $\texttt{RankC}(i' = 2 \times i \ \wedge \ i > 0 \ \wedge \ n > i \ \wedge \ n' = n) = \{\log{(n/i)}\}$

The above two patterns are good enough to compute ranking functions for most loops that iterate using arithmetic counters. However, for the purpose of completeness, we describe below two examples (taken from some recent work on proving termination) that cannot be matched using the above two patterns, and hence illustrate the limitations of pattern-matching. However, we can find ranking functions or bounds for these examples using the counter instrumentation and invariant generation techniques described in [Gulavani and Gulwani, 2008].

- Consider the terminating transition system $\{x' = x+y \wedge y' = y+1 \wedge x < n \wedge n' = n\}$ from [Bradley et al., 2005], which uses the principle of polyranking lexicographic functions for proving its termination. Note that the reason why the transition system terminates is because even though y is not known to be always positive, it will eventually become positive by virtue of the assignment $y' = y + 1$.
- Consider the terminating transition system $\{x' = y \wedge y' = x-1 \wedge x > 0\}$. This transition system can be proven terminating by monotonicity constraints as introduced in [Ben-Amram, 2009b]. Note, that the reason why the transition system terminates is because in every two iterations the value of x decreases by 1.

3.2.2 Boolean Iteration Patterns

Often loops contain a path/transition that is meant to execute just once. The purpose of such a transition is to switch between different phases of a loop, or to perform the cleanup action immediately before loop termination. Such an iteration pattern can be captured by the following rule, where the operator `Bool2Int`(e) maps boolean values `true` and `false` to 1 and 0 respectively.

If $\rho \Rightarrow (e \ \wedge \ \neg(e[X'/X]))$, then $\texttt{Bool2Int}(e) \in \texttt{RankC}(\rho)$

The candidates for boolean expression e while applying the above pattern are restricted to those expressions that only involve variables from X and that occur syntactically in the transition ρ. In the following we give example transitions whose ranking functions can be computed using an application of this pattern:

- $\texttt{RankC}(\texttt{flag}' = \texttt{false} \ \wedge \ \texttt{flag}) = \{\texttt{Bool2Int}(\texttt{flag})\}$
- $\texttt{RankC}(x' = 100 \ \wedge \ x < 100) = \{\texttt{Bool2Int}(x < 100)\}$

3.2.3 Bit-vector Iteration Patterns

One standard way to iterate over a bit-vector is to change the position of the lsb, i.e., the least significant one bit (or msb, i.e., most significant one bit). Such an iteration pattern can be captured by the following rule/lemma, where the function $\texttt{LSB}(x)$ denotes the position of the least significant 1-bit, counting from 1, and starting from the most significant bit-position. $\texttt{LSB}(x)$ is defined to be 0 if there is no 1-bit in x. Note that $\texttt{LSB}(x)$ is bounded above by the total number of bits in bit-vector x.

$$\boxed{\text{If } \rho \Rightarrow (\texttt{LSB}(x') < \texttt{LSB}(x) \ \wedge \ x \neq 0), \text{ then } \texttt{LSB}(x) \in \texttt{RankC}(\rho)}$$

The candidates for the variable x while applying the above pattern are all the bit-vector variables that occur in the transition ρ. The query in the above pattern can be discharged using an SMT solver that provides support for bit-vector reasoning, and, in particular, the `LSB` operator. (If the SMT solver does not provide first-class support for the `LSB` operator, then one can encode the `LSB` operator using bit-level manipulation as described in [Warren, 2002].) In the following we give example transitions whose bound can be computed using the above rule:

- $\texttt{RankC}(x' = x << 1 \ \wedge \ x \neq 0) = \{\texttt{LSB}(x)\}$
- $\texttt{RankC}(x' = x \& (x - 1) \ \wedge \ x \neq 0) = \{\texttt{LSB}(x)\}$

3.2.4 Data-structure Iteration Patterns

Iteration over data-structures or collections is quite common, and one standard way to iterate over a data-structure is to follow field dereferences until some designated object is reached. Such an iteration pattern can be captured by the following rule/lemma, where the function $\texttt{Dist}(x, z, f)$ denotes the number of field dereferences along field f required to reach z from x.

If $\rho \Rightarrow (x \neq z \ \wedge \ (\texttt{Dist}(x', z, f) < \texttt{Dist}(x, z, f)))$,
then $\texttt{Dist}(x, z, f) \in \texttt{RankC}(\rho)$.

The candidates for variables x, z and field f, while applying the above pattern are all variables X and field names that occur in ρ. The query in the above pattern can be discharged using an SMT solver that implements a decision procedure for the theory of reachability and can reason about its cardinalities (e.g., [Gulwani et al., 2009b]). Note, that $\texttt{Dist}(x, z, f)$ denotes the cardinality of the set of all nodes that are reachable from x before reaching z along field f. In the following we give example transitions whose ranking functions can be computed using an application of this pattern:

- $\texttt{RankC}(x \neq \texttt{Null} \wedge x' = x.\texttt{next}) \ = \ \{\texttt{Dist}(x, \texttt{Null}, \texttt{next})\}$
- $\texttt{RankC}(\texttt{Mem}'{=}\texttt{Update}(\texttt{Mem}, x.\texttt{next}, x.\texttt{next}.\texttt{next}) \wedge$
  $x \neq \texttt{Null} \ \wedge \ x.\texttt{next} \neq \texttt{Null}) \ = \ \{\texttt{Dist}(x, \texttt{Null}, \texttt{next})\}$

3.3 Bound Computation

In this section, we show how to compute a bound $\texttt{Bound}(\mathcal{T})$ of a transition system $\mathcal{T}$.

If a transition system consists of a single transition ρ, then a bound of the transition system can be obtained simply from any ranking function r of the transition ρ:

Theorem 16. *Let $\mathcal{T} = \{\rho\}$ and let $r \in$ **Rank**(ρ). Then,*

$$\texttt{Bound}(\mathcal{T}) = \max(0, r)$$

is a bound of $\mathcal{T}$, where the max *operator returns the maximum of its arguments.*

Proof. If the transition ρ is ever taken, then r denotes an upper bound on the number of iterations of ρ (since, by our definition of a ranking function, transition ρ implies that r is bounded from below by 0 and decreases by at least 1 in each iteration). The other case is that ρ is never executed (i.e., the number of iterations of ρ is 0). Combining these two cases, we obtain the result. □

The significance of sanitizing the bound by applying the max operator in Theorem 16 is illustrated in Example 22 below.

Obtaining a bound of a transition system consisting of multiple transitions is not as straightforward. We cannot simply add the ranking functions of all individual transitions to obtain the bound of the transition system, since the interleaving of these transitions with each other can invalidate the decreasing measure of the ranking function. An alternative can be to define the notion of lexicographic ranking functions [Bradley et al., 2005] or disjunctively well-founded ranking functions [Podelski and Rybalchenko, 2004b] for transition systems consisting of multiple transitions. However, such an approach might be sufficient for proving termination, but would usually not be precise for yielding bounds.

For the purpose of precise bound computation, we distinguish between the different ways in which two transitions of a transition system can interact with each other. These cases (described in Sections 3.3.1, 3.3.2, and 3.3.3) allow for composing the ranking functions of the two transitions using one of three operators max, sum, and product. These cases can be efficiently identified by asking queries to SMT solvers.

3.3.1 Max Composition of Ranking Functions

The bound of a transition system $\mathcal{T} = \{\rho_1, \rho_2\}$ consisting of two transition relations can be obtained by applying the max operator to ranking functions for the individual transitions, if the transition relations decrease each other's ranking functions. We make this precise in Theorem 18, which makes use of the following definition.

Definition 17 (Cooperative-interference). *We say there is* cooperative interference *between transitions ρ_1 and ρ_2 through their ranking functions r_1 and r_2, if the following condition holds:*

$$\forall (s_1, s_2) \in \rho_1.\ r_2(s_2) \leq \max(r_1(s_1), r_2(s_1)) - 1$$

We denote such a cooperative-interference by $\texttt{CI}(\rho_1, r_1, \rho_2, r_2)$.

Theorem 18 (Proof Rule for Max-Composition). *Let $\mathcal{T} = \{\rho_1, \rho_2\}$ and let $r_1 \in \texttt{RankC}(\rho_1)$ and $r_2 \in \texttt{RankC}(\rho_2)$. If $\texttt{CI}(\rho_1, r_1, \rho_2, r_2)$ and $\texttt{CI}(\rho_2, r_2, \rho_1, r_1)$ hold, then*

$$\texttt{Bound}(\mathcal{T}) = \max(0, r_1, r_2)$$

is a bound of $\mathcal{T}$.

Proof. We show below that $\max(r_1, r_2)$ is a ranking function for $\mathcal{T} = \{\rho_1, \rho_2\}$, i.e., that for all pairs of states $(s_1, s_2) \in \rho_i$ and $i \in \{1, 2\}$ we have

- $\max(r_1, r_2)(s_1) \geq 0$, and

- $\max(r_1(s_1), r_2(s_1)) > \max(r_1(s_2), r_2(s_2))$.

This is sufficient to prove the claim: If some transition ρ_i is ever taken, then $\max(r_1, r_2)$ denotes an upper bound on the number of iterations of $\mathcal{T}$ (since, by our definition of a ranking function, every transition relation $\rho_i \in \mathcal{T}$ implies that $\max(r_1, r_2)$ is bounded from below by 0 and decreases by at least 1 in each iteration). The other case is that no ρ_i is ever executed (i.e., the number of iterations of ρ is 0).

We now show that $\max(r_1, r_2)$ is a ranking function: Because of $r_1 \in \texttt{RankC}(\rho_1)$ we have $r_1(s_1) \geq 0$ for all pairs of states $(s_1, s_2) \in \rho_1$. Thus we have $\max(r_1, r_2)(s_1) \geq 0$ for all pairs of states $(s_1, s_2) \in \rho_1$. Similarly, we have $\max(r_1, r_2)(s_1) \geq 0$ for all pairs of states $(s_1, s_2) \in \rho_2$. This establishes the first condition on $\max(r_1, r_2)$. By assumption we have $\texttt{CI}(\rho_1, r_1, \rho_2, r_2)$. Thus $r_2(s_2) \leq \max(r_1(s_1), r_2(s_1)) - 1$ for all pairs of states $(s_1, s_2) \in \rho_1$. Because of $r_1 \in \texttt{RankC}(\rho_1)$ we have $r_1(s_1) > r_1(s_2)$ for all pairs of states $(s_1, s_2) \in \rho_1$. This gives us $\max(r_1(s_1), r_2(s_1)) > \max(r_1(s_2), r_2(s_2))$ for all pairs of states $(s_1, s_2) \in \rho_1$. Similarly, we get $\max(r_1(s_1), r_2(s_1)) > \max(r_1(s_2), r_2(s_2))$ for all pairs of states $(s_1, s_2) \in \rho_2$. This establishes the second condition on $\max(r_1, r_2)$. □

Example 19. *We obtained the transition system* $\texttt{TransSys}(P, l_6) = \{\rho_1, \rho_2\}$ *for* $P|_{l_6}$ *in Example 8, where:*

$$\begin{aligned}\rho_1 &\equiv (n' = n - 1 \ \wedge \ j < n \ \wedge \ j' \geq j \ \wedge \ i' = i)\\ \rho_2 &\equiv (n' = n - 1 \wedge i < n - 2 \ \wedge \ i' \geq i + 1 \ \wedge \ j' \geq i + 2)\end{aligned}$$

We can compute $\textit{\texttt{RankC}}(\rho_1) = \{n - j\}$ *and* $\textit{\texttt{RankC}}(\rho_2) = \{n - i - 2\}$. *We can prove* $\textit{\texttt{CI}}(\rho_1, n - j, \rho_2, n - i - 2)$ *and* $\textit{\texttt{CI}}(\rho_1, n - i - 2, \rho_2, n - j)$. *An application of the max-composition theorem (Theorem 18) yields the bound*

$\max(0, n-i-2, n-j)$ *for the transition system* $\{\rho_1, \rho_2\}$. *Using the invariants* $i \geq 0 \wedge j \geq 1$ *that hold during the first visit to* l_6 *(which can be obtained by generating invariants at control location* l_6*) we obtain the bound* $n-1$ *on the transition system. This implies the bound* n *on the number of visits to control location* l_6. *This concludes the bound computation for Example 1.3.*

3.3.2 Additive Composition of Ranking Functions

The bound of a transition system $\mathcal{T} = \{\rho_1, \rho_2\}$ consisting of two transition relations can be obtained by adding together the ranking functions for the two transitions when the transition relations do not increase each other's ranking functions. To state this formally (Theorem 21), we first define the notion of *non-interference* of a transition with respect to the ranking function of another transition relation.

Definition 20 (Non-interference)**.** *We say that a transition relation* ρ_1 *does not interfere with the ranking function* r_2 *of another transition relation* ρ_2*, if the following condition holds:*

$$\forall (s_1, s_2) \in \rho_1.\ r_2(s_2) \leq r_2(s_1)$$

We denote such a non-interference by $\texttt{NI}(\rho_1, \rho_2, r_2)$.

Theorem 21 (Proof Rule for Additive-Composition)**.** *Let* $\mathcal{T} = \{\rho_1, \rho_2\}$ *and let* $r_1 \in \texttt{RankC}(\rho_1)$ *and* $r_2 \in \texttt{RankC}(\rho_2)$. *If* $\texttt{NI}(\rho_1, \rho_2, r_2)$ *and* $\texttt{NI}(\rho_2, \rho_1, r_1)$ *hold, then is a bound of* $\mathcal{T}$.

$$\begin{aligned} \texttt{Bound}(\mathcal{T}) &= \texttt{Iter}(\rho_1) + \texttt{Iter}(\rho_2),\ \textit{where} \\ \texttt{Iter}(\rho_1) &= \max(0, r_1) \\ \texttt{Iter}(\rho_2) &= \max(0, r_2) \end{aligned}$$

Proof. The non-interference condition $\texttt{NI}(\rho_2, \rho_1, r_1)$ ensures that the value of the ranking function r_1 for transition ρ_1 is not increased by any interleaving of transition ρ_2. Hence, the total number of iterations of the transition ρ_1 is given by $\max(0, r_1)$ (based on an argument similar to that in proof of Theorem 16). Similarly, the total number of iterations of the transition ρ_2 is given by $\max(0, r_2)$. Hence, the result. $\square$

Example 22. *Consider the transition system $\mathcal{T} = \{\rho_1, \rho_2\}$ (obtained from the loop in Program 1.6 in Fig. 1.2) with the following 2 transitions:*

$$\begin{aligned} \rho_1 &\equiv z > x \ \wedge \ x < n \ \wedge \ x' = x + 1 \ \wedge \ z' = z \ \wedge \ n' = n \\ \rho_2 &\equiv z \leq x \ \wedge \ x < n \ \wedge \ z' = z + 1 \ \wedge \ x' = x \ \wedge \ n' = n \end{aligned}$$

We can compute $\texttt{RankC}(\rho_1) = \{n - x\}$ and $\texttt{RankC}(\rho_2) = \{n - z\}$. We can prove $\texttt{NI}(\rho_1, \rho_2, n - z)$ and $\texttt{NI}(\rho_2, \rho_1, n - x)$. An application of the additive-composition theorem (Theorem 21) yields the bound $\max(0, n-x)+\max(0, n-z)$ *for $\mathcal{T}$.*

We now explain the importance of using the max *operators in the statement of Theorem 16 and Theorem 21. If we defined $\texttt{Iter}(\rho)$ to simply r instead of* $\max(0, r)$*, then we would incorrectly conclude the bound on $\mathcal{T}$ to be $2n - x - z$. This is incorrect because, for example, suppose that the transition system was executed in the initial state $n = 100, x = 0, z = 200$, then the expression $n - x - z$ evaluates to 0, while the transition system $\rho_1 \vee \rho_2$ executes for 100 iterations.*

This example is also a good illustration of how our technique differs significantly from (and, in fact, provides a simpler alternative to) recently proposed techniques for proving termination [Cook et al., 2006] and loop bound *analysis [Gulwani et al., 2009a]. The control-flow refinement technique used in [Gulwani et al., 2009a] unravels the exact interleaving pattern between the two transitions to conclude that ρ_1 and ρ_2 interleave in lock-steps, only after which it is able to derive the bound. In contrast, our proof rule stated in Theorem 21 only requires to establish the non-interference property between the two transitions. The principle of disjunctively well-founded ranking functions used in [Cook et al., 2006] requires computing the transitive closure of the transition system only to conclude a quadratic bound. In contrast, our proof rule stated in Theorem 21 does not require computing any transitive-closure, and is even able to obtain a precise linear bound. (The transitive-closure is required in our technique only to summarize any inner nested loops, which, however, are not present in the loop in Program 1.6).*

Observe that the additive-composition theorem (Theorem 21) and max-composition theorem (Theorem 18) provide orthogonal proof-rules. The bound of the transition system in Example 19 can be computed using the max-composition theorem, but not using the additive-composition theorem. Similarly, the bound of the transition system in Example 22 can be computed using the additive-composition theorem, but not using the max-composition theorem.

3.3.3 Multiplicative Composition of Ranking Functions

If mutual cooperative-interference or mutual non-interference properties of two transitions cannot be established, then it is still possible to compute bounds provided one of the transition satisfies the non-interference property. The bound in such a case is obtained by multiplying together the ranking functions for the two transitions, as made precise in the following theorem. This is a common case for bounding iterations of an inner loop when its iterators are re-initialized inside the outer loop.

Theorem 23 (Proof Rule for Multiplicative-Composition)**.** *Let $\mathcal{T} = \{\rho_1, \rho_2\}$ and let $r_1 \in \mathtt{RankC}(\rho_1)$, and $r_2 \in \mathtt{RankC}(\rho_2)$. If $\mathtt{NI}(\rho_2, \rho_1, r_1)$, then*

$$\begin{aligned}
\mathtt{Bound}(\mathcal{T}) &= \mathtt{Iter}(\rho_1) + \mathtt{Iter}(\rho_2), \text{ where} \\
\mathtt{Iter}(\rho_1) &= \max(0, r_1) \\
\mathtt{Iter}(\rho_2) &= \max(0, r_2) + \max(0, u) \times \max(0, r_1)
\end{aligned}$$

is a bound of $\mathcal{T}$, where u denotes an invariant of $\mathcal{T}$ that is an upper bound on expression r_2.

Proof. From the non-interference condition $\mathtt{NI}(\rho_2, \rho_1, r_1)$, we can conclude that $\mathtt{Iter}(\rho_1) \leq \max(0, r_1)$ (the same argument as in the proof of Theorem 21). However, the same reasoning does not apply to ρ_2. Instead we observe that the maximum number of iterations of ρ_2 *in between any two interleavings of* ρ_1 is bounded above by $\max(0, u)$ (since the starting value of the ranking function r_2 may be reset to u by any execution of ρ_1). However, the number of iterations of ρ_2 before any interleaving of ρ_1 is still bounded by $\max(0, r)$. Hence, the total number of iterations of ρ_2 is bounded by $\max(0, r_2) + \max(0, u) \times \max(0, r_1)$. □

For a given transition system, application of both additive-composition theorem (Theorem 21) and multiplicative-composition theorem (Theorem 23) may be possible and may yield incomparable bounds, as illustrated by the example below.

Example 24. *Consider the transition system $\mathcal{T} = \{\rho_1, \rho_2\}$ with the following*

```
Procedure: Bound($\mathcal{T}$)
Input: a transition set $\mathcal{T} = \{\rho_1, \dots, \rho_n\}$
Output: a bound $b$ of $\mathcal{T}$
foreach $i \in \{1, .., n\}$ do $\texttt{Iter}[\rho_i] := \bot$;
repeat
    foreach $i \in \{1, .., n\}$ and $r \in \textit{RankC}(\rho_i)$ do
        $J := \{j \mid \neg\texttt{NI}(\rho_j, \rho_i, r)\}$;
        if $\textit{Iter}[\rho_i] = \bot$ and for all $j \in J$: $\textit{Iter}[\rho_j] \neq \bot$ then
            $\texttt{factor} := \sum_{j \in J} \texttt{Iter}[\rho_j]$;
            Let $u$ be an invariant of $\mathcal{T}$ that is an upper bound on $r$;
            $\texttt{Iter}[\rho_i] := \max(0, r) + \max(0, u) \times \texttt{factor}$;
until array $\textit{Iter}$ has not been changed;
if for all $i \in \{1, .., n\}$: $\textit{Iter}[\rho_i] \neq \bot$ then return $\sum_{i \in \{1,..,n\}} \texttt{Iter}[\rho_i]$;
else return "Potentially Unbounded";
```

Algorithm 3: Bound($\mathcal{T}$) composes a bound of $\mathcal{T}$ from ranking functions of the individual transitions of $\mathcal{T}$.

two transitions:

$$\rho_1 \equiv i' = i - 1 \wedge i > 0 \wedge j' = j - 1 \wedge j > 0 \wedge k' = k \wedge m' = m$$

$$\rho_2 \equiv j' = m \wedge k' = k - 1 \wedge k > 0 \wedge \wedge i' = i \wedge m' = m$$

We can compute $\textit{RankC}(\rho_1) = \{i, j\}$ *and* $\textit{RankC}(\rho_2) = \{k\}$*. We can prove* $\textit{NI}(\rho_1, \rho_2, k)$ *and* $\textit{NI}(\rho_2, \rho_1, i)$*. An application of additive-composition theorem (Theorem 21) yields the bound* $\max(0, i) + \max(0, k)$ *for the transition system* $\mathcal{T}$*. An application of multiplicative-composition theorem (Theorem 23) yields the incomparable bound* $\max(0, j) + \max(0, m) \times \max(0, k)$*.*

3.3.4 Combining the Composition Rules

In this section, we discuss how to compute bounds of transition systems with more than 2 transition relations by putting together the proof rules stated in Theorem 18, 21, and 23. Since different order of applications of the proof rules may generate different bounds, we present a strategy that we found effective throughout our experiments.

First, observe that an optimal way of applying the additive-composition theorem (Theorem 21) and multiplicative-composition theorem (Theorem 23) is to compute the total number of iterations for each transition individually, and then sum them up together.Algorithm 3 implements such a strategy based on a simple generalization of Theorem 21 and Theorem 23 to the case when a transition system contains more than 2 transitions. Algorithm 3 iteratively computes an array `Iter` such that $\texttt{Iter}[\rho_i]$ denotes a bound on the total number of iterations of the transition ρ_i during any execution of the transition system $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$. Algorithm 3 initializes all elements of `Iter` to $\perp$. Then Algorithm 3 chooses a transition relation ρ_i and a ranking function $r \in \texttt{RankC}(\rho_i)$ and computes the set J of all indices of transition relations that interfere with the ranking function r. If a bound on the total number of iterations of all those transition relations ρ_j with $j \in J$ is known, i.e., $\texttt{Iter}[\rho_j] \neq \perp$, then a bound on the number of iterations of ρ_i is obtained using a generalization of Theorems 21 and 23. Algorithm 3 iterates the above steps until the array `Iter` does not change any more. If a transition relation ρ_i remains such that $\texttt{Iter}[\rho_i] = \perp$, Algorithm 3 failed to compute a bound and reports that there is possibly no bound. Otherwise a bound on the entire transition system $\mathcal{T}$ is obtained by summing up the bounds $\texttt{Iter}[\rho_i]$ on the total number of iterations of the individual transition relations ρ_i.

For simplicity, we have presented the algorithm to output only one bound, but the Algorithm 3 can be easily extended to output multiple bounds by associating a set of bounds (as opposed to a single bound) with $\texttt{Iter}[\rho_i]$ and appropriately relaxing the condition $\texttt{Iter}[\rho_i] = \perp$ in the main loop to multiple bounds.

We now discuss how one could extend Algorithm 3 to also take advantage of the max-composition proof rule in Theorem 18. Before computing a bound with Algorithm 3, one could add $\max(r, r')$ to $\texttt{RankC}(\rho)$ for any transition ρ, where $r \in \texttt{RankC}(\rho)$ and $r' \in \texttt{RankC}(\rho')$ for some other transition ρ', provided $\texttt{Rank}(\rho, \max(r, r'))$ holds. This enables a subsequent application of additive-composition proof rule (Theorem 21) to obtain an additive bound that may be a constant factor of 2 away from the bound that would have been obtainable from an application of the max-composition proof rule (Theorem 18), but that is much better than a multiplicative bound.

3.4 Experiments

We have implemented our proposed solution to the reachability-bound problem in C# using the Phoenix Compiler Infrastructure [PhoenixWebPage, 2009] and the SMT solver Z3 [Z3WebPage, 2009]. This implementation is part of a tool that computes symbolic complexity for procedures in .Net binaries. Below we present two different sets of experimental results that measure the effectiveness of various aspects of our solution.

3.4.1 Loop Bound Computation

We considered the problem of computing symbolic bounds on the number of loop iterations, which is an instance of the reachability-bound problem where the control location under consideration is the loop header. We chose mscorlib.dll (a .Net base-class library), which had 2185 loops, as our benchmark. Our tool analyzes these 2185 loops in less than 5 minutes and is able to compute bounds for 1677 loops. The problem of loop bound computation is especially challenging under the following two cases for which earlier techniques for bound computation do not perform as well.

<u>**Case 1:**</u> Iterations of outer loops depending on inner loops (examples of the kind described in Figure 3.1). There were 113 such loops out of the total 2185 loops. The key idea of this thesis is to address such challenges is to summarize inner loops by their transitive-closure that preserves required relationships between the inputs and outputs of the loop. The *effectiveness of our transitive closure computation algorithm* is illustrated by the fact that our success ratio for such cases (80 out of 113, i.e., 70%) is similar to our overall success ratio (1677 out of 2185, i.e., 76%).

<u>**Case 2:**</u> Loop bound computation for nested loops. The challenge here is to compute precise amortized bounds on the total number of iterations of those loops, as opposed to the number of iterations per iteration of the immediate outer loop (the latter is an easier problem than the former). This is the same issue as exemplified by Example 1.3. There were 250 such loops out of the total 2185 loops. Unfortunately, we cannot evaluate the precision of our bounds automatically. As described in Section 2.3, the problem of computing a precision-witness for a given symbolic bound is an orthogonal problem to the one in this thesis. Instead, we manually investigated the generated bounds for most of these loops and found all these bounds to be precise (according to Definition 5). This points out the *effectiveness of our bound-computation algorithm* based on the three proof rules presented in Section 3.3.

Another interesting statistic is the distribution of the number of transitions generated for each loop, as shown in Figure 3.2. The small number of transitions validates the design choice behind our *transition system gener-*

# Transitions	1	2	3	4	5	6	7	8	9	≥10
# Loops	1561	224	107	44	25	11	9	5	8	191

Figure 3.2: The table shows the number of loops for respective number of transition relations.

ation algorithm that enumerates all paths between two program points (in order not to lose any precision) after slicing has been performed.

Out of the 508 loops for which we failed to compute a bound, the failure for 503 loops is attributed to not being able to compute ranking functions for some transition in the transition system corresponding to the loop. There were two main causes. (i) Our implementation is intra-procedural, meaning that our transition system generation algorithm fails when the value of loop iterators gets modified because of procedure calls. This problem can be addressed by simply inlining the procedure, provided there are no recursive calls. (ii) Of the various proof rules described in Section 3.2, we only implemented those corresponding to arithmetic and boolean iteration patterns, while several transitions were iterating using field dereferences or bit-vector manipulation. A sound handling of field dereferences would require use of an alias analysis. On the positive side these results show the *effectiveness of the proof-rule based technique for finding ranking functions*: a handful of patterns are sufficient to compute ranking functions for transitions arising in 76% of the examples.

There were only 5 cases (out of 1682 cases) for which we were able to compute a ranking function for each transition, but were not able to compute a bound of the transition system. This points out the *effectiveness of our proof rules for composing bounds from of ranking functions of individual transitions.*

3.4.2 Disjunctive Invariant Computation

We also evaluated the effectiveness of the transitive closure algorithm (Algorithm 2) described in Section 3.1 on a variety of benchmark examples chosen by recent state-of-the-art papers on computing disjunctive invariants. Figure 3.3 describes these four examples that have been used as flagship examples to motivate new techniques for proving non-trivial safety assertions. Proving validity of the assertions in all these examples requires disjunctive loop invariants. It turns out that the required disjunctive invariant for each of these examples satisfies the convexity-like assumption, and hence can be discovered by Algorithm 2. We have adapted Algorithm 2 slightly to take advantage of the initial condition `Init` (as is done by all the other approaches) by initializing s'_1 to $\texttt{Init} \land Id$ at the loop headers instead of Id. This allows Algorithm 2 to establish the desired disjunctive invariant using fewer disjuncts. We give an example for the application of Algorithm 2 without initial condition at the end of this section.

Given that the number of disjuncts in the desired transitive closure is 2 for all examples, and that the number of transitions in the transition system represented by the loop is either 2 or 3, the total number of possibilities for the map σ is 16 or 64 respectively. Hence, by trying out all possible maps, Algorithm 2 can discover the desired disjunctive invariants.

Instead, we used a heuristic that constructs the map σ dynamically for our experiments on the programs shown in Figure 3.3. We choose $m = 1$ and initialize ρ'_1 to $\texttt{Init} \land Id$. We maintain a partial map σ that is completely undefined at start, and construct σ on the fly: During the main loop of Algorithm 2 we compute $\rho = \rho'_j \circ \rho_i$ for transitions $\rho'_j \in \mathcal{T}'$, $\rho_i \in \mathcal{T}$. If $\sigma(j, i)$ is undefined and $\rho \neq \emptyset$, we heuristically find an index k such that ρ is close to an existing disjunct ρ'_k and define $\sigma(j, i)$ to be k. If we do not find such a k, we increase m by 1 and set $\sigma(j,i)$ to the new value of m. We used the octagon domain [Miné, 2006] in our experiments, which contains equalities and inequalities, and consider octagons close, if the share an equality. This heuristic combines semantic- and syntactic-merging criteria in an interesting manner: the semantic-merging criterion is used the construct the syntactic-merging criterion σ, which is used to compute the transitive hull in the light of Theorem 14. With this heuristic our prototype implementation is able to validate the assertion in each of the examples in less than 0.2sec.

We now return to the discussion on how we can apply Algorithm 2 without making use of the initial condition `Init`. For example, for the first example, we would require the transition invariant $\mathcal{T} = \{\rho_1, \rho_2, \rho_3, \rho_4, \}$, where

$$\begin{array}{rcl}
\rho_1 & \equiv & Id, \\
\rho_2 & \equiv & x \leq 50 \wedge x' \leq 51 \wedge x' - x = y' - y, \\
\rho_3 & \equiv & x \geq 51 \wedge x' \geq 52 \wedge x' - x = y' - y, \\
\rho_4 & \equiv & x \leq 50 \wedge x' \geq 52 \wedge 102 - x' - x = y - y'.
\end{array}$$

Note that the transition invariant consists of more disjuncts, and involves elements from richer numerical domains than the octagon abstract domain such as the polyhedra abstract domain. However, the above invariant still satisfies the convexity-like assumption, where a convexity-witness σ is as follows: $\sigma = \{(1,1) \mapsto 2, (2,1) \mapsto 2, (3,1) \mapsto 3, (4,1) \mapsto 4, (1,2) \mapsto 3, (2,2) \mapsto 4, (3,2) \mapsto 3, (4,2) \mapsto 4\}$. Thus Algorithm 2 instantiated with the polyhedra abstract domain is precise enough to discover the desired invariant $\mathcal{T}$. Note that $\mathcal{T}$ is a relational invariant that summarizes the loop. In contrast, none of the techniques that were proposed for the examples shown in Figure 3.3 can compute such loop summaries. We discuss related work on loop summaries in Section 6.6.

Original Example	**Various Details**

Gopan and Reps 06. Page 3, Fig. 1
`x:=0, y:=0;`
`while (*)`
`    if (x` $\leq$ `50) y++;`
`    else y--;`
`    if (y<0) break;`
`    x++;`
`assert(x=102)`
	$\rho_1 \equiv x \leq 50 \land y+1 \geq 0 \land y' = y+1 \land x' = x+1$
	$\rho_2 \equiv x > 50 \land y-1 \geq 0 \land y' = y-1 \land x' = x+1$
	———
	$Init \equiv x = 0 \land y = 0$
	———
	$\rho_1' \equiv 0 \leq x' \leq 51 \land x' = y'$
	$\rho_2' \equiv 52 \leq x' \leq 102 \land x' + y' = 102$
	———
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 1\}$

Beyer *et al.* **07.** Page 306, Fig. 4.
`x:=0; y:=50;`
`while (x<100)`
`    if (x<50) x++;`
`    else x++; y++;`
`assert(y=100);`
	$\rho_1 \equiv x \leq 50 \land x' = x+1 \land y' = y$
	$\rho_2 \equiv 51 \leq x \leq 100 \land x' = x+1 \land y' = y+1$
	———
	$Init \equiv x = 0 \land y = 50$
	———
	$\rho_1' \equiv 0 \leq x' \leq 50 \land y' = 50)$
	$\rho_2' \equiv (51 \leq x' \leq 100 \land x' = y'$
	———
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 1\}$

Gulavani *et al.* **06.** Page 5, Fig. 3. **Henzinger** *et al.* **02.** Page 2, Fig. 1.
`lock:=0;assume(`$x \neq y$`)`
`while (`$x \neq y$`)`
`    lock := 1;`
`    x := y;`
`    if (*)`
`        lock := 0;`
`        y++;`
`assert(lock = 1);`
	$\rho_1 \equiv x \neq y \land lock' = 1 \land x' = y \land y' = y$
	$\rho_2 \equiv x \neq y \land lock' = 0 \land x' = y \land y' = y+1$
	———
	$Init \equiv x \neq y \land lock = 0$
	———
	$\rho_1' \equiv x' = y' \land lock' = 1$
	$\rho_1' \equiv x' + 1 = y' \land lock' = 0$
	———
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (2,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2\}$

Popeea and Chin 06. Page 2
`x := 0; upd := 0;`
`while (`$x < N$`)`
`    if (*)`
`        l := x;`
`        upd := 1;`
`    x++;`
`assert(`$upd = 1$
`        `$\Rightarrow 0 \leq l < N$`);`
	$\rho_1 \equiv x < N \land x' = x+1 \land l' = l \land upd' = upd$
	$\rho_2 \equiv (x < N \land x' = x+1 \land l' = x \land upd' = 1$
	———
	$Init \equiv x = 0 \land upd = 0$
	———
	$\rho_1' \equiv x' \geq 0 \land l' = l \land upd' = 0 \land N' = N$
	$\rho_2' \equiv x' \geq 1 \land upd' = 1 \land N' = N \land 0 \leq l' < N$
	———
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 2\}$

Figure 3.3: Prominent disjunctive invariant challenges from recent literature. Corresponding to each example the entries in 2nd column show the following details in that order: transition-system representation $\mathcal{T} = \{\rho_1, \rho_2\}$ of the loop, initial condition `Init`, transitive closure $\mathcal{T}' = \{\rho_1', \rho_2'\}$ of $\mathcal{T}$, and the convexity-witness (δ, σ).

Chapter 4

Size-change Abstraction Approach

In this chapter we present our second approach to the reachability-bound problem. We introduce the size-change abstraction (SCA) in Section 4.1. We describe how we use SCA for computing transitive hulls in Section 4.2. We present our algorithm for bound computation in Section 4.3. We present experimental results in Section 4.4.

4.1 Size-change Abstraction

In this section we introduce SCA and at the same time give the definitions and notations necessary for the presentation of our transitive hull and bound algorithms in Sections 4.2 and 4.3.

4.1.1 Order Constraints

Let X be a set of variables. We denote by $\rhd$ any element from $\{>, \geq\}$.

Definition 25 (Order Constraint)**.** *An* order constraint *over X is an inequality $x \rhd y$ with $x, y \in X$.*

Given a variable x we denote by x' its *primed* version. We denote by X' the set $\{x' \mid x \in X\}$ of the primed variables of X.

Definition 26 (Valuation)**.** *The set of all* valuations *of X is the set $Val_X = X \to \mathbb{Z}$ of all functions from X to the integers.*

Given a valuation $\sigma \in Val_X$ we define its primed valuation *as the function $\sigma' \in Val_{X'}$ with $\sigma'(x') = \sigma(x)$ for all $x \in X$.*

Given two valuations $\sigma_1 \in Val_{X_1}, \sigma_2 \in Val_{X_2}$ with $X_1 \cap X_2 = \emptyset$ we define their union *$\sigma_1 \cup \sigma_2 \in Val_{X_1 \cup X_2}$ by $(\sigma_1 \cup \sigma_2)(x) = \begin{cases} \sigma_1(x), & \text{for } x \in X_1, \\ \sigma_2(x), & \text{for } x \in X_2. \end{cases}$*

Definition 27 (Semantics)**.** *We define a* semantic relation *$\models$ as follows: Let $\sigma \in Val_X$ be a valuation. Given an order constraint $x_1 \rhd x_2$ over X, $\sigma \models x_1 \rhd x_2$ holds, if $\sigma(x_1) \rhd \sigma(x_2)$ holds in the structure of the integers $(\mathbb{Z}, \geq)$. Given a set OC of order constraints over X, $\sigma \models OC$ holds, if $\sigma \models o$ holds for all $o \in OC$.*

4.1.2 Size-change Abstraction (SCA)

We are using integer-valued functions on the program states to measure progress of a program. Such functions are called norms in the literature. Norms provide us sizes of states that we can compare. We will use norms for abstracting programs.

Definition 28 (Norm)**.** *A* norm *$n \in \Sigma \to \mathbb{Z}$ is a function that maps the states to the integers.*

We fix a finite set of norms N for the rest of this subsection, and describe in Section 4.1.3 how to extract norms from programs automatically. Given a state $s \in \Sigma$ we define a valuation $\sigma_s \in Val_N$ by setting $\sigma_s(n) = n(s)$.

We will now introduce SCA. Our terminology diverts from the seminal papers on SCA [Lee et al., 2001; Ben-Amram, 2011] because we focus on a logical rather than a graph-theoretic representation. The set of norms N corresponds to the SCA "variables" in [Lee et al., 2001; Ben-Amram, 2011].

Definition 29 (Monotonicity Constraint, Size-change Relation / Set, Concretization)**.** *The set of* monotonicity constraints *OC is the set of all order constraints over $N \cup N'$. The set of* size-change relations *(SCRs) $SCRs = 2^{OC}$ is the set of all subsets of OC. An* SCR set *$\mathcal{S} \subseteq SCRs$ is a set of SCRs. We use the* concretization function *$\gamma : SCRs \to \Gamma$ to map an SCR $T \in SCRs$ to a transition relation $\gamma(T)$ by defining $\gamma(T) = \{(s_1, s_2) \in \Sigma \times \Sigma \mid \sigma_{s_1} \cup \sigma'_{s_2} \models T\}$ as the set of all pairs of states such that the evaluation of the norms on these states satisfy all constraints of T. We lift the concretization function to SCR sets by setting $\gamma(\mathcal{S}) = \{\gamma(T) \mid T \in \mathcal{S}\}$ for an SCR set $\mathcal{S}$.*

Note that the abstract domain of SCRs has only finitely many elements, namely $3^{|N|^2}$. Further note that an SCR set corresponds to a formula in DNF.

Definition 30 (Abstraction Function)**.** *The* abstraction function *$\alpha : \Gamma \to SCRs$ takes a transition relation $\rho \in \Gamma$ and returns the greatest SCR containing it, namely $\alpha(\rho) = \bigcup\{T \in SCRs \mid \rho \subseteq \gamma(T)\} = \{o \in OC \mid \rho \models o\}$. We lift the abstraction function to transition sets by setting $\alpha(\mathcal{T}) = \{\alpha(\rho) \mid \rho \in \mathcal{T}\}$ for a transition set $\mathcal{T}$.*

Note that the equality $\bigcup\{T \in SCRs \mid \rho \subseteq \gamma(T)\} = \{o \in OC \mid \rho \models o\}$ is a simple observation from the definitions, and that α is a best abstraction function.

Implementation of the abstraction. α can be implemented by an SMT solver under the assumption that the norms are provided as expressions and that the transition relation is given as a formula such that the order constraints between these expressions and the formula fall into a logic that the SMT solver can decide.

Using abstraction and concretization we can define concatenation of SCRs:

Definition 31 (Concatenation of SCRs)**.** *Given two SCRs $T_1, T_2 \in SCRs$, we define $T_1 \circ T_2$ to be the SCR $\alpha(\gamma(T_1) \circ \gamma(T_2))$. We lift the concatenation operator $\circ$ to SCR sets by defining $\mathcal{S}_1 \circ \mathcal{S}_2 = \{T_1 \circ T_2 \mid T_1 \in T_1, B_2 \in \mathcal{S}_2\}$ for SCR sets $\mathcal{S}_1, \mathcal{S}_2 \subseteq 2^{A \times A}$. $\mathcal{S}^0 = \{Id\}, \mathcal{S}^k, \mathcal{S}^+, \mathcal{S}^*$ etc. are defined in the natural way.*

Concatenation of SCRs is conservative by definition, i.e., $\gamma(T_1 \circ T_2) \supseteq \gamma(T_1) \circ \gamma(T_2)$ and associative because of the transitivity of order relations. Concatenation of SCRs can be effectively computed by a modified all-pairs-shortest-path algorithm (taking order relations as weights). Because the number of SCRs is finite, the transitive hull is computable.

4.1.3 Heuristics for Extracting Norms

We now describe our heuristics for extracting norms from programs. Let $P = (L, E)$ be a program and $l \in L$ be a location. We compute all cycle-free paths from the loop header back to the loop header. From each of these paths we extract local ranking functions checking the patterns described in Section 3.2 by an SMT solver. The expressions of the local ranking functions are then included into the set of norms. Similarly to program slicing [Muchnick, 1997], we iteratively add all expressions of inner loops to the set of norms on which the already extracted norms are control dependent until a fixed point is reached. We heuristically also include the sum and the difference of two norms, when an inner loop affects several norms at the same time.

We mention two approaches for handling complex data structures whose norms are not captured by the patterns in Section 3.2:

- One could preanalyze common data structures (such as provided in standard libraries) and provide a set of norms for them. This approach achieves a separation of concerns between analyzing application code and library functions.
- If this is not possible, one could automatically abstract such data structures before the actual analysis and derive an integer program using techniques such as as [Gulwani et al., 2009b; Magill et al., 2010].

4.2 Transitive Closure Computation

In this section, we give an algorithm for the function `TransHull` required by Algorithm 1 in Section 2.5 for computing transition invariants based on SCA.

The following theorem can be directly shown from the definitions given in Section 4.1.

Theorem 32. *Let $\mathcal{T}$ be a transition set. Then $\gamma(\alpha(\mathcal{T})^*)$ is a transition invariant for $\mathcal{T}$.*

Theorem 32 can immediately be used for computing transitive hulls. Algorithm 4 presents such an implementation of Theorem 32.

In the following example we show how Algorithm 1 is used by Algorithm 4 for computing transition invariants that summarize inner loops disjunctively.

```
**Procedure**: TransHull($\mathcal{T}$)
**Input**: a transition set $\mathcal{T}$
**Output**: transitive hull $\mathcal{T}'$ of $\mathcal{T}$
$\mathcal{S} := \{Id\}$;
**repeat**
    $\mathcal{S}' = \mathcal{S}$;
    $\mathcal{S} = \mathcal{S} \circ \alpha(\mathcal{T})$;
**until** $\mathcal{S}' = \mathcal{S}$;
$\mathcal{T}' = \gamma(\mathcal{S})$;
**return** $\mathcal{T}'$;
```

Algorithm 4: `TransHull`($\mathcal{T}$) computes a transition invariant for $\mathcal{T}$.

$$\begin{tikzcd}
\mathit{begin} \arrow[d, "\rho_0"] & \\
l_1 \arrow[r, "\rho_5"] \arrow[d, bend right, "\rho_1"'] & \mathit{end} \\
l_2 \arrow[u, "\rho_3"] \arrow[u, bend right, "\rho_4"'] \arrow[loop below, "\rho_2"] &
\end{tikzcd}$$

$\rho_0 \equiv i = 0$
$\rho_1 \equiv i < n \wedge i' = i + 1 \wedge j' = 0$
$\rho_2 \equiv i < n \wedge i' = i + 1 \wedge j' = j + 1$
$\rho_3 \equiv j > 0 \wedge i' = i - 1$
$\rho_4 \equiv j \leq 0$
$\rho_5 \equiv i \geq n$

Figure 4.1: Simplified CFG of Program 1.1 with transition relations

Example 33. *Let P Program 1.1. The (simplified) CFG and the transition relations of P are shown in Figure 4.1. We explain in the following how Algorithm 1 computes the transition system* $\texttt{TransSys}(P, l_1)$ *for* $P|_{l_1}$ *using Algorithm 4 for computing* `TransHull`*. In the first* `foreach`*-loop Algorithm 1 calls itself recursively on the nested loop* $\mathcal{L} = (\{l_2\}, \{l_2 \xrightarrow{\rho_2} l_2\})$ *with header* l_2*. In the recursive call Algorithm 1 skips the first* `foreach`*-loop because* $(\mathcal{L}, l_2)$ *does not have nested loops. The second* `foreach`*-loop iterates over all cycle-free paths of* $\texttt{paths}(\mathcal{L}, l_2)$*. There is only one such a path* $\pi = l_2 \xrightarrow{\rho_2} l_2$*. Algorithm 1 computes* $\mathcal{T}_\pi = \{i < n \wedge i' = i + 1 \wedge j' = j + 1 \wedge n' = n\}$ *and returns* $\mathcal{T}_\pi$ *as the transition system* $\texttt{TransSys}(\mathcal{L}, l_2)$*. After the return of the recursive call* $\mathcal{T} = \texttt{TransSys}(\mathcal{L}, l_2)$ *Algorithm 1 calls Algorithm 4 to compute* $\texttt{TransHull}(\mathcal{T})$*. Algorithm 4 size-change abstracts* $\mathcal{T}$ *by* $\alpha(\mathcal{T}) = \{n - i > 0 \wedge n' - i' < n - i \wedge j < j'\}$*, computes the disjunctive transitive closure in the abstract* $\mathcal{S} = \alpha(\mathcal{T})^* = \{n - i > 0 \wedge n' - i' = n - i \wedge j = j', n - i > 0 \wedge n' - i' < n - i \wedge j < j'\}$ *and returns the concretization* $\gamma(\mathcal{S})$*. Algorithm 1 then stores* $\texttt{TransHull}(\mathcal{T})$ *in* $\mathsf{summary}[l_2]$*. As there is no other nested loop, the first* `foreach`*-loop is finished. The second* `foreach`*-loop iterates over all cycle-free paths of* $\texttt{paths}(P, l_1)$*. There are two such paths, namely* $\pi_1 = l_1 \xrightarrow{\rho_1} l_2 \xrightarrow{\rho_3} l_1$ *and* $\pi_2 = l_1 \xrightarrow{\rho_1} l_2 \xrightarrow{\rho_4} l_1$*. Algorithm 1 computes*

$$\begin{aligned}
\mathcal{T}_{\pi_1} &= \{\rho_1\} \circ \mathsf{summary}[l_2] \circ \{\rho_3\} \\
&= \{n - i > 0 \wedge i_1 = i + 1 \wedge j_1 \\
&= 0 \wedge n' - i' = n - i_1 \wedge j' = j_1 \wedge j' > 0, \\
&\qquad n - i > 0 \wedge i_1 = i + 1 \wedge j_1 = 0 \wedge n' - i' < n - i_1 \wedge j' > j_1 \wedge j' > 0\} \\
&= \{false, n - i > 0 \wedge i' > i \wedge j' > 0\}, \text{ and}
\end{aligned}$$

$$\begin{aligned}\mathcal{T}_{\pi_2} &= \{\rho_1\} \circ \mathsf{summary}[l_2] \circ \{\rho_4\} \\ &= \{n - i > 0 \wedge i_1 = i + 1 \wedge j_1 = 0 \wedge n' - i' = n - i_1 \wedge j' = j_1 \wedge j' = 0, \\ &\qquad n - i > 0 \wedge i_1 = i + 1 \wedge j_1 = 0 \wedge n' - i' = n - i_1 - 1 \wedge j' > j_1 \wedge j' = 0\} \\ &= \{n - i > 0 \wedge n' - i' = n - i - 1 \wedge j' = 0, false\},\end{aligned}$$

and returns $\mathcal{T}_{\pi_1} \cup \mathcal{T}_{\pi_2} = \{n - i > 0 \wedge i' > i \wedge j' > 0, n - i > 0 \wedge n' - i' = n - i - 1 \wedge j' = 0\}$ *as transition system* $\texttt{TransSys}(P, l_1)$ *for* $P|_{l_1}$*. Note that for each path,* $false$ *indicates that one transition relation was detected to be unsatisfiable, e.g.* $n - i > 0 \wedge j = 0 \wedge n' - i' < n - i - 1 \wedge j < j' \wedge j' = 0$ *in* π_2*, and that Algorithm 1 returned only the two satisfiable transitions.*

4.2.1 Disjunctive Summarization of Loops with SCA

We point out that the need for disjunctive summaries of loops in the bound analysis is a major motivation for SCA, because it allows us to compute disjunctive transitive closures naturally:

Given a transition system $\mathcal{T}$ for some loop $\mathcal{L}$, we want to summarize $\mathcal{L}$ by a transition invariant. The most precise transition invariant $\mathcal{T}^* = \{Id\} \cup \mathcal{T} \cup \mathcal{T}^2 \cup \mathcal{T}^3 \cup \cdots$ introduces infinitely many disjunctions and is not computable in general. In contrast to this the abstract transitive closure $\alpha(\mathcal{T})^* = \alpha(\{Id\}) \cup \alpha(\mathcal{T}) \cup \alpha(\mathcal{T})^2 \cup \alpha(\mathcal{T})^3 \cup \cdots$ has only finitely many disjunctions and is effectively computable. This allows us to overapproximate the infinite disjunction $\mathcal{T}^*$ by the finite disjunction $\gamma(\alpha(\mathcal{T})^*)$.

4.2.2 Comparison of Blockwise and Pathwise SCA Analysis

In Section 2.5.2 we discussed the difference between pathwise and blockwise analysis and explained why pathwise analysis is more precise. In this section we review classical SCA analyses [Lee et al., 2001; Ben-Amram, 2009a], observe that these analyses are blockwise analyses and then show that our pathwise SCA analysis is more precise.

Classical SCA analyses [Lee et al., 2001; Ben-Amram, 2009a] do not explicitly discuss how to obtain abstract programs. However, their abstract program model consists of CFGs whose edges are labeled by SCRs. This lets us conclude that classical SCA analyses are blockwise analyses. These analyses abstract a program under scrutiny in one step, and then analyze the abstracted program by a single transitive hull computation. In contrast, our pathwise analysis abstracts a program in multiple steps and computes transitive hulls at multiple times during the analysis. Our approach can be seen as a generalization of classical SCA and is strictly more precise as we show in the following.

Let P be Program 1.1. By our pathwise analysis we obtain the transition system

$$\begin{aligned}\texttt{TransSys}(P, l_1) = \{&n - i > 0 \wedge n' - i' < n - i \wedge j' > 0,\\ &\qquad n - i > 0 \wedge n' - i' = n - i - 1 \wedge j' = 0\}\end{aligned}$$

for $P|_{l_1}$ as described in Section 2.5. Note that $\texttt{TransSys}(P, l_1)$ establishes that the variable i increases at every loop iteration and that $i < n$ is an invariant at l_1. $\texttt{TransSys}(P, l_1)$ is precise enough so that our Algorithm 5 can further size-change abstract it and compute a bound from the abstraction.

The blockwise analysis in classical SCA begins with abstracting P. Because of the inner loop at location l_2, each transition $\rho_1, \rho_2, \rho_3, \rho_4$ constitutes a program block and needs to be abstracted separately.

Thus we get the SCRs

$$\begin{aligned}
\alpha(\rho_1) &= n - i > 0 \wedge n' - i' < n - i \wedge j' > 0,\\
\alpha(\rho_2) &= n - i > 0 \wedge n' - i' < n - i \wedge j < j',\\
\alpha(\rho_3) &= j > 0 \wedge n' - i' > n - i,\\
\alpha(\rho_4) &= j \leq 0 \wedge n' - i' = n - i.
\end{aligned}$$

The termination analysis of classical SCA computes a transitive hull of these SCRs following the CFG of P. In particular classical SCA computes the SCR $\alpha(\rho_1) \circ \alpha(\rho_2) \circ \alpha(\rho_3) = n - i > 0 \wedge j' > 0$ for the path $l_1 \xrightarrow{\rho_1} l_2 \xrightarrow{\rho_2} l_2 \xrightarrow{\rho_3} l_1$. Note that classical SCA cannot establish that $n - i$ increases every time this path is taken (the concatenation of $n' - i' < n - i$ and $n' - i' > n - i$ in $\alpha(\rho_2)$ and $\alpha(\rho_3)$ loses all information on $n - i$) and therefore cannot prove the termination of P.

Parsers are a natural class of programs which illustrate the need for pathwise use of SCA. In our experiments we observed that many parsers increase an index while scanning the input stream and use lookahead to detect which token comes next. Like in Example 1.1, lookaheads may temporarily decrease the index. Pathwise abstraction is crucial to reason about the progress of these parsers with SCA.

4.3 Bound Computation

In this section, we show how to compute a bound $\texttt{Bound}(\mathcal{T})$ for a transition system $\mathcal{T}$ with SCA in two steps:

- The first step $P = \texttt{Contextualize}(\mathcal{T})$ transforms the transition system $\mathcal{T}$ into a program P. This is done by the program transformation contextualization, which we describe in Section 4.3.1.

- The second step $b = \texttt{Bnd}(P)$ consists of the bound computation for program P. It is performed by Algorithm 5, which we describe in Section 4.3.2.

4.3.1 Contextualization

Contextualization is a program transformation by Manolios and Vroon [Manolios and Vroon, 2006], who report on an impressive precision of their SCA-based termination analysis of functional programs. Note that we do not use their terminology (e.g. "calling context graphs") in this thesis. Our contribution is in adopting contextualization to imperative programs and in recognizing its relevance for bound analysis.

Definition 34 (Contextualization). *Let $\mathcal{T}$ be a transition set. The* contextualization $\texttt{Contextualize}(\mathcal{T})$ *of $\mathcal{T}$ is the program $P = (\mathcal{T}, E)$, where $E = \{\rho \xrightarrow{\rho} \rho' \mid \rho, \rho' \in \mathcal{T}$ and $\rho \circ \rho' \neq \emptyset\}$.*

The contextualization of a transition system is a program in which every location stores the context of which transition is to be executed next; the program has an edge between two locations only if the transitions of the locations can be executed one after another.

Contextualization restricts the order in which the transitions of the transition system can be executed. Thus, contextualization encodes information, that could otherwise be deduced from the pre- and postconditions of transitions, directly into the CFG. Since pathwise analysis contracts whole loop paths into single transitions, contextualization is particularly important after pathwise analysis: our subsequent bound algorithm does not need to compute the pre- and postcondition of the contracted loop paths but only needs to exploit the structure of the CFGs for determining in which order the loop

```
void main (int x, int b) {
  while (0 < x < 255) {
    if (b) x = x + 1;
    else x = x - 1;
  }
}
```

$$\begin{tikzcd} l_1 \arrow[loop below, "\rho_1"] & l_2 \arrow[loop below, "\rho_2"] \end{tikzcd}$$

Program 4.1

Figure 4.2: Program 4.1 with its CFG obtained from contextualization.

$$\begin{tikzcd} l_1 \arrow[loop above, "\rho_1"] \arrow[r, "\rho_1"] & l_2 \arrow[r, "\rho_2"] & l_3 \arrow[loop above, "\rho_3"] \end{tikzcd}$$

Figure 4.3: Contextualization of Program 1.7 (see page 23)

paths can be executed.

We illustrate contextualization on Program 4.1. The program has two paths, and gives rise to the transition system $\mathcal{T} = \{\rho_1, \rho_2\}$, where

$\rho_1 \equiv 0 < x < 255 \wedge b \wedge x' = x + 1 \wedge b'$

$\rho_2 \equiv 0 < x < 255 \wedge \neg b \wedge x' = x - 1 \wedge \neg b'$

Keeping track of the boolean variable b is important for bound analysis of $\mathcal{T}$: Without reference to b not even the termination of $\mathcal{T}$ can be proven. In Figure 4.2 (right) we show the contextualization of $\mathcal{T}$. Note that contextualization has encoded information about the variable b into the CFG in such a way that we do not need to keep track of the variable b anymore. Thus, contextualization releases us from taking the precondition b resp. $\neg b$ and the postcondition b' resp. $\neg b'$ into account for bound analysis.

We next describe the application of contextualization on the flagship example of a recent publication [Gulwani et al., 2009a] on the bound problem.

Example 35. *The authors of [Gulwani et al., 2009a] motivate control-flow* *refinement for bound analysis on Program 1.7. Their algorithm relies on a sophisticated interplay between program transformation and abstract interpretation. We show that our much simpler technique can also handle Program 1.7.*

We assume an invariant analysis (e.g. octagon analysis) computes the invariant $0 \leq n \leq m \wedge j \geq 0$ *at the header of the* `while`*-loop.*

The `while`*-loop has four paths because we consider the inequality* `j != n` *as the disjunction* $j < n \vee j > n$. *Algorithm 1 gives us the transition system* $\mathcal{T} = \{\rho_1, \rho_2, \rho_3\}$ *(there is no fourth transition relation because* $0 \leq n \leq m \wedge j < n \wedge j > m$ *is detected as unsatisfiable), where*

$\rho_1 \equiv 0 \leq n \leq m \wedge n < j \leq m \wedge j' = j + 1,$

$\rho_2 \equiv 0 \leq n \leq m \wedge j > m \wedge j' = 0,$

$\rho_3 \equiv 0 \leq n \leq m \wedge 0 \leq j < n \wedge j' = j + 1.$

Contextualization gives us the CFG depicted in Figure 4.3, which reflects the different loop phases precisely. This CFG structure is exploited by Algorithm 5 to compute the bound $maxId + 1$ *as we describe in Example 36 (stated in the next subsection).*

Implementation. We implement contextualization by encoding the concatenation $\rho_1 \circ \rho_2$ of two transitions ρ_1, ρ_2 into a logical formula and querying an SMT solver whether this formula is satisfiable. This approach is very simple to implement in comparison to the explicit computation of pre- and postconditions.

4.3.2 Bound Algorithm

Our bound algorithm reduces the computation of bounds to components whose bounds are combined into an overall bound. To this end, we exploit

```
Procedure: Bound($P$)
Input: a program $P = (L, E)$
Output: a bound $b$ on the length of the traces of $P$
$SCCs := \texttt{computeSCCs}(P)$; $b := 0$;
while $SCCs \neq \emptyset$ do
    $SCCsOnLevel := \emptyset$;
    forall the $SCC \in SCCs$ s.t. no $SCC' \in SCCs$ can reach $SCC$ do
        $r := \texttt{BndSCC}(SCC)$;
        Let $r \leq b_{SCC}$ be a global invariant;
        $SCCsOnLevel := SCCsOnLevel \cup \{SCC\}$;
    $b := b + \max_{SCC \in SCCsOnLevel} b_{SCC}$;
    $SCCs := SCCs \setminus SCCsOnLevel$;
return $b$;
```

Algorithm 5: Bound(P) composes a bound of P from the individual bounds of the SCCs of P

the structure of the CFGs obtained from contextualization: We partition the CFG of programs into its strongly connected components (SCCs) (SCCs are maximal strongly connected subgraphs). For each SCC, we compute a bound by Algorithm 6, and then compose these bounds to an overall bound by Algorithm 5.

Algorithm 5 arranges the SCCs of the CFG into levels. The first level consists of the SCCs that do not have incoming edges, the second level consists of the SCCs that can be reached from the first level, etc. For each level, Algorithm 5 calls Algorithm 6 to compute bounds for the SCCs of this level. Let *SCC* be an SCC of some level and let $r :=$ BndSCC(SCC) be the bound returned by Algorithm 6 on *SCC*. r is a (local) bound of *SCC* that may contain variables of P that are changed during the execution of P. Algorithm 5 uses global invariants (e.g. interval, octagon or polyhedra) in order to obtain a bound b_{SCC} on r in terms of the initial values of P. The SCCs of one level are collected in the set *SCCsOnLevel*. For each level, Algorithm 5 composes the bounds b_{SCC} of all SCCs $SCC \in SCCsOnLevel$ to a maximum expression. Algorithm 5 sums up the bounds of all levels for obtaining an overall bound.

```
**Procedure**: BndSCC($P$)
**Input**: strongly-connected program $P = (L, E)$
**Output**: a bound $b$ on the length of the traces of $P$
**if** $E = \emptyset$ **then return** 1;
$\mathit{NonIncr} := \emptyset$; $\mathit{DecrBnded} := \emptyset$; $\mathit{BndedEdgs} := \emptyset$;
**foreach** $n \in N$ **do**
    **if** $\forall\ l_1 \xrightarrow{\rho} l_2 \in E\ n \geq n' \in \alpha(\rho)$ **then**
        $\mathit{NonIncr} := \mathit{NonIncr} \cup \{n\}$;
**foreach** $l_1 \xrightarrow{\rho} l_2 \in E,\ n \in \mathit{NonIncr}$ **do**
    **if** $n \geq 0, n > n' \in \alpha(\rho)$ **then**
        $\mathit{DecrBnded} := \mathit{DecrBnded} \cup \{\max(n, 0)\}$;
        $\mathit{BndedEdgs} := \mathit{BndedEdgs} \cup \{l_1 \xrightarrow{\rho} l_2\}$;
**if** $\mathit{BndedEdgs} = \emptyset$ **then fail with** "there is no bound for $P$";
$b = $ Bnd($(L, E \setminus \mathit{BndedEdgs})$);
**return** $((\sum \mathit{DecrBnded}) + 1) \cdot b$;
```

Algorithm 6: BndSCC computes bounds for individual SCCs

Algorithm 6 computes the bound of a strongly-connected program P. First Algorithm 6 checks if $P = (L, E)$ is non-trivial, i.e., $E \neq \emptyset$, and returns 1, if this is not the case. Next Algorithm 6 collects all norms in the set *NonIncr* that on all transitions either decrease or stay equal. Subsequently Algorithm 6 checks for every norm $n \in \mathit{NonIncr}$ and transition $l_1 \xrightarrow{\rho} l_2 \in E$, if n is bounded from below by zero and decreases on ρ. If this is the case, Algorithm 6 adds $\max(n, 0)$ to the set *DecrBnded* and $l_1 \xrightarrow{\rho} l_2$ to *BndedEdgs*. Note that the transitions included in the set *BndedEdgs* can only be executed as long as their associated norms are greater than zero. Every transition in *BndedEdgs* decreases an expression in *DecrBnded* when it is taken. As the expressions in *DecrBnded* are never increased, the sum of all expressions in *DecrBnded* is a bound on how often the transitions in *BndedEdgs* can be taken. If *DecrBnded* is empty, Algorithm 5 fails, because the absence of infinite cycles could not be proven. Otherwise we recursively call Algorithm 5 on $(L, E \setminus \mathit{BndedEdgs})$ for a bound b on this subgraph. The subgraph can be entered as often as the transitions in *BndedEdgs* can be taken plus one (when it is entered first). Thus, $((\sum \mathit{DecrBnded}) + 1) \cdot b$ is an upper bound of P.

Example 36. *The control flow graph given in Figure 4.3 has 3 SCCs:* (ρ_1), (ρ_2), (ρ_3). *Algorithm 6 computes the following bounds of these SCCs:* $b_{\rho_1} = \max(\mathit{maxId} - \mathit{tmp}, 0)$, $b_{\rho_2} = 1$ *and* $b_{\rho_3} = \max(\mathit{id} - \mathit{tmp}, 0)$. *Algorithm 5 composes these bounds as follows:* $u(b_{\rho_1}) + u(b_{\rho_2}) + u(b_{\rho_3})$, *where* u *denotes an upper bound on the value of the given expression. Assuming that the invariant analysis provides* $u(b_{\rho_1}) = \mathit{maxId} - \mathit{id}$, $u(b_{\rho_2}) = 1$, $u(b_{\rho_3}) = \mathit{id}$, *we obtain the precise bound* $\mathit{maxId} + 1$.

Role of SCA in our Bound Analysis. Our bound analysis uses the size-change abstractions of transitions to determine how a norm n changes according to $n \geq n'$, $n > n'$, $n \geq 0$ in Algorithm 6. We plan to incorporate inequalities between different norms (like $n \geq m'$) in future work to make our analysis more precise.

Termination analysis. If in Algorithm 5 the global invariant analysis cannot infer an upper bound on some norm, the algorithm fails to compute a bound, but we can still compute a lexicographic ranking function, which is sufficient to prove termination. The respective adjustment of our algorithm is straightforward.

$$\begin{tikzcd}
l_2 \arrow[r, "\rho_2"'] \arrow[loop below, "\rho_2"] & l_3 \arrow[d, "\rho_3"'] \arrow[dr, "\rho_3"] & l_4 \arrow[d, "\rho_4"] \arrow[dl, "\rho_4"'] & l_1 \arrow[l, "\rho_1"'] \arrow[loop below, "\rho_1"] \\
 & l_5 \arrow[r, bend left, "\rho_5"] \arrow[loop below, "\rho_5"] & l_6 \arrow[l, bend left, "\rho_6"] \arrow[loop below, "\rho_6"] &
\end{tikzcd}$$

The size-change abstractions of the transition relations:

$\alpha(\rho_1) = l \geq 0 \wedge s' > s \wedge s' \leq 255 \wedge l' = l$
$\alpha(\rho_2) = l \geq 0 \wedge s' < s \wedge s' \geq 0 \wedge l' = l$
$\alpha(\rho_3) = \alpha(\rho_5) = l \geq 0 \wedge l' < l \wedge s' > s \wedge s' \leq 255$
$\alpha(\rho_4) = \alpha(\rho_6) = l \geq 0 \wedge l' < l \wedge s' < s \wedge s' \geq 0$

Figure 4.4: The CFG obtained from contextualizing the transition system of Program 1.2 (left) and the size-change abstractions of the transition relations (right)

4.3.3 A Complete Example

Let P be Program 1.2 (see page 8) and let l be the header of the loop. We assume a standard invariant analysis (such as the octagon analysis) to compute the invariant $c \geq 1$ at l, which is valid throughout the execution of the loop. Algorithm 1 computes the transition system $\mathcal{T} = \{\rho_1, \ldots, \rho_6\}$ by collecting all paths from loop header back to the loop header (omitting transition relations that belong to infeasible paths), where

$$\begin{aligned}
\rho_1 &\equiv c \geq 1 \wedge \neg f \wedge d \neq 1 \wedge d' = 2 \wedge s' = s + c \leq 255 \wedge c' = c \wedge f' = f, \\
\rho_2 &\equiv c \geq 1 \wedge \neg f \wedge d \neq 2 \wedge d' = 1 \wedge s' = s - c \wedge s' \geq 0 \wedge c' = c \wedge f' = f, \\
\rho_3 &\equiv c \geq 1 \wedge \neg f \wedge d = 1 \wedge f' \wedge c' = c/2 \wedge d' = 2 \wedge s' = s + c' \wedge s' \leq 255, \\
\rho_4 &\equiv c \geq 1 \wedge \neg f \wedge d = 2 \wedge f' \wedge c' = c/2 \wedge d' = 1 \wedge s' = s - c' \wedge s' \geq 0, \\
\rho_5 &\equiv c \geq 1 \wedge f \wedge c' = c/2 \wedge d' = 2 \wedge s' = s + c' \wedge s' \leq 255 \wedge f' = f, \\
\rho_6 &\equiv c \geq 1 \wedge f \wedge c' = c/2 \wedge d' = 1 \wedge s' = s - c' \wedge s' \geq 0 \wedge f' = f.
\end{aligned}$$

By our heuristics (cf. Section 4.1.3) we consider s and the logarithm of c (which we abbreviate by l) as program norms.

Figure 4.4 shows the CFG obtained from contextualizing the transition system of Example 1.2 on the left. The CFG shows that the transitions cannot interleave in arbitrary order. Our bound Algorithm 5 exploits the SCC decomposition of the CFG:

The CFG in Figure 4.4 has 5 SCCs: (ρ_1), (ρ_2), (ρ_3), (ρ_4), (ρ_5, ρ_6). Algorithm 6 computes the following bounds of these SCCs: $b_{\rho_1} = \max(255 - s, 0)$, $b_{\rho_2} = \max(s, 0)$, $b_{\rho_3} = 1$, $b_{\rho_4} = 1$, $b_{\rho_5,\rho_6} = \log c$. Algorithm 5 composes these bounds as follows: $u(\max(b_{\rho_1}, b_{\rho_2})) + u(\max(b_{\rho_3}, b_{\rho_4})) + u(b_{\rho_5,\rho_6})$, where u denotes an upper bound on the value of the given expression computed by an invariant analysis. Assuming that the invariant analysis provides $u(\max(b_{\rho_1}, b_{\rho_2})) = \max(255, \mathit{start})$, $u(\max(b_{\rho_3}, b_{\rho_4})) = 1$, $u(b_{\rho_5,\rho_6}) = 2$, we obtain the precise bound $\max(255, \mathit{start}) + 3$.

We point out how the above described approach *enables* automatic bound analysis by SCA. The loop phases of Program 1.2 are encoded into the CFG: The variables d and f do not appear in the abstracted transitions. Our automatic bound analysis uses only the CFG and does not need to take d and f into account explicitly.

We further point out how the approach described above *separates* bound analysis for the components from the composition of bounds from the components. In order to obtain the bound in the example, two different norms as well as the operators max and log are needed. By first extracting the norms from program paths and then composing them by our bound algorithm, an automatic analysis becomes feasible.

4.4 Experiments

Our tool `Loopus` implements the SCA based approach on bound analysis presented in this thesis for computing loop bounds of C programs. LOOPUS employs the LLVM compiler framework and performs its analysis on the

LLVM intermediate representation [Lattner and Adve, 2004]. For logical reasoning LOOPUS uses the *yices* SMT solver [Dutertre and de Moura, 2006]. We evaluated LOOPUS on the Mälardalen WCET [WCETWebPage, 2010] and the cBench [CBenchWebPage, 2010] benchmarks on an Intel Xeon CPU (4 cores with 2.33 GHz) with 16 GB Ram.

The Mälardalen benchmark is used in the area of *worst case execution time analysis* for the comparison and evaluation of methods and tools. It contains 7497 lines of code and 262 loops. In less than 35 seconds total time, LOOPUS computed a bound for 93% of the loops. On the loops with more than one path (in the following called *non-trivial loops*) LOOPUS had a success ratio of 72% (42 of 58 loops). The failure cases had the following reasons: (1) unimplemented modeling of memory updates [2 loops] (2) arithmetic instructions that cannot be handled by *yices* [4 loops] (3) insufficient invariant analysis [4 loops] (4) quantified invariants on array contents needed [6 loops in 2 programs].

The cBench benchmark was collected for research on program and compiler optimization. After removing code-duplicates it contains 1027 C source code files, 211.892 lines of code and a total number of 4302 loops. For 4090 loops LOOPUS answered within a 1000 seconds timeout (3923 loops in less than 4 seconds). On 71 loops LOOPUS exceeded the 1000 seconds timeout and 141 loops could not be analyzed because our current tool does not handle irreducible CFGs.

LOOPUS successfully computed a bound for 75% of the 4090 analyzed loops in the cBench benchmark. On the non-trivial loops LOOPUS was successful in 65% of the cases (1181 of 1902 loops).

For the class of inner loops (e.g. program location l_2 in Program 1.1) LOOPUS computed a bound for 65% (830 of 1345) of the loops. This class of loops is especially interesting for evaluating the precision of the automatically computed bounds in the presence of outer loops. A manual sample of around 100 loops in this class showed that the bounds obtained from LOOPUS were precise.

We evaluated our transitive hull algorithm on the class of loops for which an inner loop had to be summarized in order to compute an iteration bound. LOOPUS was successful in 56% of these cases (578 of 1102). The relatively low success ratio of 56% is caused by limits of our implementation of the transitive hull algorithm which currently does not support invariants involving values of memory locations.

In 992 of the total 1017 failure LOOPUS failed to bound a transition. An analysis revealed that the reasons for failed attempts were: (1) missing implementation features like pointer calculations and memory updates (2) insufficient invariant analysis (3) some loops were not meant to terminate, e.g. input loops (4) complex invariants like quantified invariants on the content of arrays needed. None of these reasons reveals a general limitation of our analysis. All but reason (4) can be solved by systematic engineering work. In the 25 remaining cases our tool computed a bound for each transition but was not able to compose an overall bound.

Chapter 5

Fundamental Properties of the Size-change Abstraction

In this chapter we give theoretical results on SCA towards a characterization of the bounds that can be expressed by SCA. In order to do so, we first define order constraints and linear orders in Section 5.1. Based on these definitions we introduce size-change systems (SCSs) in Section 5.2, which provide us an abstract program model. We prove lower bounds for SCSs and conjecture that these lower bounds give rise to a complete algorithm that decides the complexity of SCSs in Section 5.2.

In order to make the presentation self-contained we will restate some of the definitions used in Chapter 4 throughout this chapter.

5.1 Order Constraints

We introduce some notation that is convenient for dealing with inequalities. By $\triangleright$ we denote any element from $\{>, \geq\}$. By $\triangleright^c$ we denote the opposite relational sign of the element denoted by $\triangleright$, i.e., $>^c = \leq$ and $\geq^c = <$.

Given a variable x we denote by x' its *primed* version and by $x^{(i)}$ its version with i-primes. Given a set of variables X we denote by $X' = \{x' \mid x \in X\}$ the set of primed variables and by $X^{(i)} = \{x^{(i)} \mid x \in X\}$ the set of variables with i-primes.

Definition 37 (Order Constraint)**.** *Let X be a set of variables. An* order constraint *over X is an inequality $x_1 \rhd x_2$ with $x_1, x_2 \in X$. We denote the set of all order constraints over X by $OC(X)$.*

Given a constraint $c \in OC(X)$ we denote by $c[X'/X]$ the constraint over X' which is obtained from c by substituting *every variable $x \in X$ by its primed version $x' \in X'$.*

Given a set of order constraints $S \subseteq OC(X)$ we denote by $S[X'/X] = \{c[X'/X] \mid c \in S\}$ the set of constraints which results from substituting each variable in X by its corresponding variable in X'.

Given a set of order constraints $S \subseteq OC(X)$ and a subset $Y \subseteq X$ we denote by $S\restriction_Y = \{x \rhd y \mid x \rhd y \in S \text{ and } x, y \in Y\}$ the restriction of S to variables in Y.

In a logical context a set of order constraints $S \subseteq OC(X)$ denotes the formula $\bigwedge_{c \in S} c$.

Definition 38 (Decrease Constraint, Size-change Relation (SCR))**.** *Let X be a set of variables. A* decrease constraint *over X is an order constraint $x_1 \rhd x_2' \in OC(X \cup X')$ with $x_1, x_2 \in X$. We denote the set of all decrease constraints over X by $DEC(X)$. We call a set of decrease constraints a* size-change relation (SCR)*.*

Definition 39 (Valuations)**.** *Let $\mathcal{M}$ be a set of values and X a set of variables. We define the set of* valuations *$Val^X_{\mathcal{M}}$ to be the set of mappings $X \to \mathcal{M}$. Given a valuation $\sigma \in Val^X_{\mathcal{M}}$ we define its* primed valuation *as the function $\sigma' \in Val_{\mathcal{M}}^{X'}$ with $\sigma'(x') = \sigma(x)$ for all $x \in X$. Given two valuations $\sigma_1 \in Val_{\mathcal{M}}^{X_1}, \sigma_2 \in Val_{\mathcal{M}}^{X_2}$ with $X_1 \cap X_2 = \emptyset$ we define their* union *$\sigma_1 \cup \sigma_2 \in Val_{\mathcal{M}}^{X_1 \cup X_2}$ by* $(\sigma_1 \cup \sigma_2)(x) = \begin{cases} \sigma_1(x), & \text{for } x \in X_1, \\ \sigma_2(x), & \text{for } x \in X_2. \end{cases}$

Definition 40 (Theory of Linear Order)**.** *The* theory of linear order $\mathcal{T}_{Lin}$ *consists of the three axioms*

- $\forall a, b.\ a \geq b \wedge b \geq a \rightarrow a = b$ *(antisymmetry),*
- $\forall a, b, c.\ a \geq b \wedge b \geq c \rightarrow a \geq c$ *(transitivity),*
- $\forall a.\ a \geq a$ *(reflexivity).*

Definition 41 (Linear Order)**.** *Given a set of values $\mathcal{M}$ and a binary relation $\geq\ \subseteq \mathcal{M} \times \mathcal{M}$ over $\mathcal{M}$, the pair $(\mathcal{M}, \geq)$ is a* linear order*, if the axioms of $\mathcal{T}_{Lin}$ are valid when interpreted over the domain $\mathcal{M}$ and relation $\geq$.*

We also say that a binary relation $\geq\ \subseteq \mathcal{M} \times \mathcal{M}$ is a linear order*, when $(\mathcal{M}, \geq)$ is a linear order.*

Definition 42 (Semantics)**.** *Let X be a set of variables.*

Given a linear order $(\mathcal{M}, \geq)$ and a valuation $\sigma \in Val_{\mathcal{M}}^{X}$ we define the semantic relation $\models$ *as follows: $\sigma \models x_1 \rhd x_2$ holds for an order constraint $x_1 \rhd x_2 \in OC(X)$ iff $\sigma(x_1) \rhd \sigma(x_2)$. $\sigma \models S$ holds for a set of order constraints $S \subseteq OC(X)$ iff $\sigma \models c$ holds for all $c \in S$. $\sigma \models \mathbf{S}$ holds for a set of sets of order constraints $\mathbf{S} \subseteq 2^{OC(X)}$ iff $\sigma \models S$ for all $S \in \mathbf{S}$.*

$\mathbf{S}_1 \Rightarrow \mathbf{S}_2$ holds for two sets of sets of order constraints $\mathbf{S}_1, \mathbf{S}_2 \subseteq 2^{OC(X)}$ iff for every linear order $(\mathcal{M}, \geq)$ and every valuation $\sigma \in Val_{\mathcal{M}}^{X}$ with $\sigma \models \mathbf{S}_1$ there is a set of order constraints $S \in \mathbf{S}_2$ such that $\sigma \models S$ holds.

A set of order constraints $S \subseteq OC(X)$ is unsatisfiable*, if there is no linear order $(\mathcal{M}, \geq)$ and no valuation $\sigma \in Val_{\mathcal{M}}^{X}$ with $\sigma \models S$.*

Proposition 43 (without proof)**.** *For two sets of sets of order constraints* $\mathbf{S}_1, \mathbf{S}_2 \subseteq 2^{OC(X)}$*,* $\mathbf{S}_1 \Rightarrow \mathbf{S}_2$ *holds iff*

$$\mathcal{T}_{Lin} \wedge \bigwedge_{S \in \mathbf{S}_1} S \rightarrow \bigvee_{S \in \mathbf{S}_2} S$$

is a tautology of first-order logic.

Remark 44. *We will use sequences of sets of order constraints over X to denote sets of sets of order constraints over X, i.e., the sequence $S_1, \ldots, S_l$ denotes the set $\{S_i \mid 1 \leq i \leq l\}$. Given an order constraint c we write S, c instead of $S, \{c\}$.*

Definition 45 (Order Semiring, Order Set)**.** *The* order semiring $(Ge, \sqsubseteq, \sqcup, \cdot)$ *is an ordered commutative semiring, where*

- $Ge = \{\perp, \geq, >\}$ *is the* order set*,*
- *the order relation* $\sqsubseteq$ *is defined by* $\perp \sqsubseteq \geq \sqsubseteq >$*,*
- *the plus operator* $\sqcup$ *is defined as the maximum of two elements w.r.t.* $\sqsubseteq$*, and*
- *the multiplication operator* $\cdot$ *is defined by*
  - $\perp \cdot \rhd = \rhd \cdot \perp = \perp$ *for* $\rhd \in Ge$*, and*
  - $\rhd_1 \cdot \rhd_2 = \rhd_1 \sqcup \rhd_2$ *for* $\rhd_1, \rhd_2 \in \{\geq, >\}$*.*

It is easy to verify that $(Ge, \sqsubseteq, \sqcup, \cdot)$ is indeed a commutative semiring. We present the tables of the addition and multiplication operators of the order semiring in Figure 5.1.

We explain our motivation for defining the order semiring in the following. Let X be a set of variables and let $S \subseteq OC(X)$ be a set of order constraints.

$\sqcup$	$\perp$	$\geq$	$>$
$\perp$	$\perp$	$\geq$	$>$
$\geq$	$\geq$	$\geq$	$>$
$>$	$>$	$>$	$>$

$\cdot$	$\perp$	$\geq$	$>$
$\perp$	$\perp$	$\perp$	$\perp$
$\geq$	$\perp$	$\geq$	$>$
$>$	$\perp$	$>$	$>$

Figure 5.1: Tables of the addition and multiplication operators of the order lattice.

We define $S \models x \perp y$ to hold for all variables $x, y \in X$. The statement $S \models x \perp y$ does not allow to infer any order relation between x and y. Using this definition, $\rhd_2 \sqsubseteq \rhd_1$ and $S \models x \rhd_1 y$ imply $S \models x \rhd_2 y$ for all variables $x, y \in X$. Thus $\sqsubseteq$ is monotonic w.r.t to semantic validity. Because $\sqsubseteq$ is a linear order $S \models x \rhd_1 y$ and $S \models x \rhd_2 y$ imply $S \models x \rhd_1 \sqcup \rhd_2 y$ for all variables $x, y \in X$. Thus the plus operator $\sqcup$ respects semantic validity. Using the above definition $S \models x \rhd_1 z$ and $S \models z \rhd_2 y$ imply $S \models x \rhd_1 \cdot \rhd_2 y$. Thus the multiplication operator $\cdot$ allows to compute valid order constraints that are consequences of the transitivity of order relations.

Definition 46 (Resolution)**.** *Let $S \subseteq OC(X)$ be a set of order constraints. A constraint $x \rhd y$ is* derivable by resolution *$S \vdash x \rhd y$,*

- *if $x \rhd' y \in S$ and $\rhd \sqsubseteq \rhd'$, or*
- *if there are constraints $x \rhd_1 z, z \rhd_2 y$ with $S \vdash x \rhd_1 z, S \vdash z \rhd_2 y$ and $\rhd \sqsubseteq \rhd_1 \sqcup \rhd_2$. In this case we call $x \rhd y$ a z*-resolvent.

The first rule of resolution allows to derive the constraints included in the set of order constraints. The second rule of resolution allows to use transitivity of order relations and derive new constraints from two already derived constraints. Both rules allow to derive $\geq$ constraints, when $>$ constraints can be inferred.

Because every linear order is transitive, derivation by resolution is sound:

Proposition 47. *Let $S \subseteq OC(X)$ be a set of order constraints. $S \vdash x \rhd y$ implies $S \models x \rhd y$.*

Next we show that resolution is also complete:

Lemma 48. *Let $S \subseteq OC(X)$ be a set of order constraints. S is satisfiable iff there is no variable $x \in X$ with $S \vdash x > x$.*

Proof. Assume there is a variable $x \in X$ with $S \vdash x > x$. By the soundness of resolution S is unsatisfiable.

Assume that there is no variable $x \in X$ with $S \vdash x > x$. We define a relation $\succ \subseteq X \times X$ by $x \succ y$ iff $S \vdash x \geq y$. By the definition of resolution we have that $S \vdash x \geq z$ and $S \vdash z \geq y$ imply $S \vdash x \geq y$. Therefore $\succ$ is transitive. We define $\succeq$ to be the reflexive closure of $\succ$. The relation $\succeq$ is transitive and reflexive and therefore a preorder on X. We define equivalence classes $[x]_\succeq = \{y \mid x \succeq y \wedge y \succeq x\}$ and the set $X/\succeq = \{[x]_\succeq \mid x \in X\}$. We define a relation $\succsim$ on $X/\succeq$ by $[x]_\succeq \succsim [y]_\succeq$ iff $x \succeq y$. Note that because $\succeq$ is a preorder on X, the induced relation $\succsim$ is a partial order on $X/\succeq$. Every partial order can be extended to a linear order by standard arguments: Let

$$\mathcal{R} = \{R \subseteq X/\succeq \times X/\succeq \mid R \text{ is a partial order } \wedge \succsim \subseteq R\}$$

be the set of all partial orders on $X/\succeq$ that extend $\succeq$. $\mathcal{R}$ is ordered by set inclusion $\subseteq$ and closed under union over increasing chains w.r.t. $\subseteq$. By Zorn's Lemma $\mathcal{R}$ contains maximal elements w.r.t. $\subseteq$. All maximal elements of $\mathcal{R}$ are linear orders because maximality ensures that every two elements can be compared. Let $\geq$ be such a linear order on $X/\succeq$ that extends $\succeq$. We define a valuation $\sigma \in Val^X_{X/\succeq}$ by $\sigma(x) = [x]_\succeq$ for all $x \in X$ and show $\sigma \models S$: Let $x \rhd y \in S$ be a constraint. We have $S \vdash x \geq y$ and thus $x \succ y$. This gives us $x \succeq y$. Thus $[x]_\succeq \succsim [y]_\succeq$. This implies $[x]_\succeq \geq [y]_\succeq$. It remains to show that $\rhd = >$ implies $[x]_{\succeq} \neq [y]_{\succeq}$. Assume that $\rhd = >$ and $[x]_{\succeq} = [y]_{\succeq}$. We have $y \succeq x$ and thus $y \succeq x$. This gives us $y \succ x$. Therefore $S \vdash y \geq x$. Because $\rhd = >$ we have $S \vdash x > y$. By resolution we get $S \vdash x > x$. This is a contradiction to the assumption there is no variable $x \in X$ with $S \vdash x > x$. Therefore we have $\sigma \models x \rhd y$ for all constraints $x \rhd y \in S$. □

Definition 49 (Saturation). *Let S be a set of order constraints over X. The* saturation $\overline{S}$ *of S is the set $\{x_1 \rhd x_2 \mid S \models x_1 \rhd x_2 \wedge x_1, x_2 \in X\}$. S is* saturated, *if $S = \overline{S}$.*

Lemma 50. *Let S_1, S_2 be two sets of order constraints over X. $S_1 \Rightarrow S_2$ holds iff $S_2 \subseteq \overline{S_1}$.*

Proof. Assume $S_2 \subseteq \overline{S_1}$: Clearly, $S_2 \subseteq \overline{S_1}$ implies $\overline{S_1} \Rightarrow S_2$. Further, we have $S_1 \Rightarrow \overline{S_1}$ and therefore $S_1 \Rightarrow S_2$.

Assume $S_2 \not\subseteq \overline{S_1}$: There is a constraint $x_1 \rhd x_2 \in S_2$ such that $x_1 \rhd x_2 \notin \overline{S_1}$. As $\overline{S_1}$ is saturated, there is a linear order $\mathcal{M}$ and a valuation $\sigma \in Val_{\mathcal{M}}^{X}$ such that $\sigma \models \overline{S_1}, \{x_1 \rhd^c x_2\}$. Clearly we have $\sigma \models S_1$, but not $\sigma \models S_2$. □

Lemma 51. *Let S be a set of order constraints over a set X. For every order constraint $x \rhd y$ with $x, y \in X$ we have $S \vdash x \rhd y$ iff there is a $l \geq 1$ and there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l$ such that $x_0 = x$, $x_l = y$ and $\bigsqcup_{0 \leq i < l} \rhd_i \sqsupseteq \rhd$.*

Proof. The if direction follows directly from the definition of resolution. It remains to show the only-if direction. Let $x \rhd y$ be an order constraint with $x, y \in X$ and $S \vdash x \rhd y$. We show that there is a $l \geq 1$ and there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l$ such that $x_0 = x$, $x_l = y$ and $\bigsqcup_{0 \leq i < l} \rhd_i \sqsupseteq \rhd$ by induction on the number of derivation steps of $\vdash$. Assume $S \vdash x \rhd y$ holds because of the first case in the definition of $\vdash$.

Thus there is a constraint $x \rhd' y \in S$ with $\rhd \sqsubseteq \rhd'$. Therefore the claim clearly holds. Assume that $S \vdash x \rhd y$ holds because of the second case in the definition of $\vdash$. There are two constraints $x \rhd_a z, z \rhd_b w \in y$ such that $S \vdash x \rhd_a z, S \vdash z \rhd_b y$ and $\rhd \sqsubseteq \rhd_a \sqcup \rhd_b$. By induction assumption there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l_1$, $x_0 = x$, $x_{l_1} = z$ and $\bigsqcup_{0 \leq i < l_1} \rhd_i \sqsupseteq \rhd_a$ for some $l_1 \in \omega$ and constraints $y_i \rhd_i y_{i+1} \in S$ with $0 \leq i < l_2$, $y_0 = z$, $y_{l_2} = y$ and $\bigsqcup_{0 \leq i < l_2} \rhd_i \sqsupseteq \rhd_b$ for some $l_2 \in \omega$. We set $l_3 = l_1 + l_2$ and $z_i = x_i$ for $0 \leq i \leq l_1$ and $z_i = y_{i-l_1}$ for $l_1 < i \leq l_1 + l_2 = l_3$. We have $z_0 = x$, $z_{l_3} = y$ and $\bigsqcup_{0 \leq i < l_3} \rhd_i \sqsupseteq \rhd_a \sqcup \rhd_b \sqsupseteq \rhd$. □

Lemma 52. *Let S be a satisfiable set of order constraints over a set X and let $x \rhd y$ be an order constraint with $x, y \in X$ such that $S \not\vdash x \rhd y$. Then $S \cup \{x \rhd^c y\}$ is satisfiable.*

Proof. Assume that $S \cup \{x \rhd^c y\}$ is unsatisfiable. By Lemma 48 $S \cup \{x \rhd^c y\}$ is satisfiable iff there is no variable $z \in X$ with $S \cup \{x \rhd^c y\} \vdash z > z$. Thus there is a $z \in X$ with $S \cup \{x \rhd^c y\} \vdash z > z$. By Lemma 51 there is a $l \geq 1$ and there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l$ such that $x_0 = x$, $x_l = y$ and $\bigsqcup_{0 \leq i < l} \rhd_i \sqsupseteq >$. Assume that $x_i \rhd_i x_{i+1} \in S$ for all $0 \leq i < l$. This implies $S \vdash z > z$. By Lemma 48 S is unsatisfiable, which contradicts our assumption about S . Therefore at least one of the constraints $x_i \rhd_i x_{i+1}$ is $x \rhd^c y$. W.l.o.g. there is exactly one such constraint. Let $x_j \rhd_j x_{j+1}$ be this constraint. We have $x_j = y$, $x_{j+1} = x$,$\rhd_0 = \lhd^c$ and $x_i \rhd_i x_{i+1} \in S$ for all $0 \leq i < l$ with $i \neq j$. Thus $S \vdash x \bigsqcup_{j<i<l} \rhd_i \sqcup \bigsqcup_{0 \leq i < j} y$ and $\bigsqcup_{j<i<l} \rhd_i \sqcup \bigsqcup_{0 \leq i < j} \sqsupseteq \rhd$. Therefore $S \vdash x \rhd y$. This is a contradiction to the assumption. □

Lemma 53. *Let S be a satisfiable set of order constraints over a set X. For every order constraint $x \rhd y$ with $x, y \in X$ we have $S \vdash x \rhd y$ iff $S \models x \rhd y$.*

Proof. The only-if direction holds because of the soundness of resolution. We show the if direction. Let $x \rhd y$ be an order constraint with $x, y \in X$ and $S \not\vdash x \rhd y$. By Lemma 52 $S \cup \{x \rhd^c y\}$ is satisfiable. Thus there is a linear order $(\mathcal{M}, \geq)$ and a valuation $\sigma \in Val^X_\mathcal{M}$ with $\sigma \models S \cup \{x \rhd^c y\}$. Thus $S \not\models x \rhd^c y$. □

Corollary 54. *Let S be a satisfiable set of order constraints over a set X. We have $\overline{S} = \{x \rhd y \mid$ there is a $l \geq 1$ and there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l$ such that $x_0 = x, x_l = y$ and $\bigsqcup_{0 \leq i < l} \rhd_i \sqsupseteq \rhd\}$.*

Proof. The claim is a direct consequence of Lemmas 51 and 53. □

Remark 55. *We mention the following interpretation of Corollary 54: Let S be a satisfiable set of order constraints over a set X. Given some $l \geq 1$ and constraints $i_s \rhd_s i_{s+1} \in S$ with $0 \leq s < l$, $i_0 = i$, $i_l = j$, the constraint $i \bigsqcup_{0 \leq s < l} \rhd_s j$ is the strongest constraint between i and j that follows from these constraints.*

5.1.1 Computing Saturations

We present an algorithm for computing saturations of sets of constraints, when the set of variables is finite. The algorithm is an instantiation of the Floyd-Warshall all-pairs-shortest-path algorithm [Floyd, 1962], where the weights and associated addition and multiplication operations are given by the order semiring.

Every finite set of variables X with $|X| = n$ is isomorphic to the set $X = \{1, 2, \ldots, n\}$. We fix $X = \{1, 2, \ldots, n\}$ for the rest of this subsection. This allows us to define a restricted resolution that uses the order of the elements of X. Our algorithm for computing saturations of sets of constraints makes use of this restricted resolution. We denote elements of X by i, j, k.

Definition 56 (h-Resolution)**.** *Let $S \subseteq OC(X)$ be a set of order constraints. A constraint $i \rhd j$ is* derivable by h-resolution *$S \vdash_h i \rhd j$,*

- *if $h \geq 0$, $i \rhd' j \in S$ and $\rhd \sqsubseteq \rhd'$, or*
- *if $h > 0$ and there are two constraints $i \rhd_1 k, k \rhd_2 j$ with $S \vdash_{h_1} i \rhd_1 k, S \vdash_{h_2} k \rhd_2 j$ and $\rhd \sqsubseteq \rhd_1 \sqcup \rhd_2$ and $h_1, h_2 \in \{0, \ldots, k-1\}$ and $k \leq h$. In this case we call $x \rhd y$ a* h-resolvent*.*

In the following proposition we state the obvious fact that n-resolution is a restriction of full resolution:

Proposition 57. *Let $S \subseteq OC(X)$ be a set of order constraints. $S \vdash_n i \rhd j$ implies $S \vdash i \rhd j$.*

We need the following lemma in order to show that restricted resolution is equally strong as full resolution:

Lemma 58. *Let $S \subseteq OC(X)$ be a set of order constraints. Let $l \geq 1$ be some number and let $i_s \rhd_s i_{s+1} \in S$ be constraints with $0 \leq s < l$, $i_0 = i$, $i_l = j$, $\bigsqcup_{0 \leq s < l} \rhd_s \sqsupseteq \rhd$ and $i_s \neq i_t$ for all $0 < s < t < l$. We have $S \vdash_h i \rhd j$ with $h = \max_{0 < s < l} i_s$ (setting $\max \emptyset = 0$).*

Proof. We prove the claim by induction on h.

$h = 0$: We have $l = 1$, $i_0 \rhd_0 i_1 \in S$, $\rhd_0 \sqsupseteq \rhd_1$ and $h = \max_{0<s<1} i_s = \max \emptyset = 0$. Therefore $S \vdash_0 i \rhd j$.

$h > 0$: Let $r \in \{1, \ldots, l-1\}$ be an index such that $i_r = h$. Note that r is unique because of $i_s \neq i_t$ for all $0 < s < t < l$. We set $h_1 = \max_{0<s<r} i_s$ and $h_2 = \max_{r<s<l} i_s$. Clearly we have $h_1, h_2 < h$. By induction assumption we have $S \vdash_{h_1} i \bigsqcup_{0 \leq s < r} \rhd_s h$ and $S \vdash_{h_2} h \bigsqcup_{r \leq s < l} \rhd_s j$. From this we get $S \vdash_h i \rhd j$. □

In the next lemma we show that restricted resolution is equally strong as full resolution:

Lemma 59. *Let $S \subseteq OC(X)$ be a set of order constraints. Let $i \rhd j$ be a constraint with $S \vdash i \rhd j$. We have $S \vdash_n i \rhd j$ or $S \vdash_n k > k$ for some $k \in \{1, \dots, n\}$.*

Proof. By Lemma 51 there is a $l \geq 1$ and there are constraints $x_i \rhd_i x_{i+1} \in S$ with $0 \leq i < l$ such that $x_0 = i$, $x_l = j$ and $\bigsqcup_{0 \leq i < l} \rhd_i \sqsupseteq \rhd$. In the following we remove constraints from the sequence $i_0 \rhd_0 i_1, i_1 \rhd_1 i_2, \dots, \dots, i_{l-1} \rhd_{l-1} i_l$ in order to obtain a sequence that contains only pairwise different variables i_s.

Assume there are indices $0 \leq s < t \leq l$ with $i_s = i_t$. We distinguish two cases:

- $\bigsqcup_{s \leq r < t} \rhd_r = \geq$: We continue with the constraints $i_0 \rhd_0 i_1, i_1 \rhd_1 i_2, \dots,$ $i_{s-1} \rhd_{s-1} i_s, i_t \rhd_t i_{t+1}, \dots, i_{l-1} \rhd_{l-1} i_l$.
- $\bigsqcup_{s \leq r < t} \rhd_r = >$: We continue with the constraints $i_s \rhd_s i_{s+1}, i_{s+1} \rhd_{s+1}$ $i_{s+2}, \dots, i_{t-1} \rhd_{t-1} i_t$.

We renumber the constraints to get continuous indices that start from 0 and go to some positive number l. We repeat the above step until all variables i_s are pairwise different.

Because the obtained sequence does only contain pairwise different variables we can apply Lemma 58. The result follows directly. $\square$

We define the function *constraint* $: X^3 \to Ge$ by

$$constraint(i, j, k) = \begin{cases} >, & S \vdash_k i > j, \\ \geq, & S \vdash_k i \geq j \wedge S \not\vdash_k i > j, \\ \bot, & \text{else.} \end{cases}$$

The function *constraint* satisfies a recursive equation, which we state in the following lemma.

Lemma 60. *The function constraint satisfies the recursive equation*

$$\begin{aligned}\mathit{constraint}(i,j,k) = \mathit{constraint}(i,j,k-1) \sqcup \\ \mathit{constraint}(i,k,k-1) \cdot \mathit{constraint}(k,j,k-1)\end{aligned}$$

with initial values

$$\mathit{constraint}(i,j,0) = \begin{cases} >, & i > j \in S, \\ \geq, & i \geq j \in S \wedge i > j \notin S, \\ \bot, & \mathit{else}. \end{cases}$$

Proof. We use $\rhd$ to denote elements of Ge. We prove the recursive equation by a case distinction on k.

$k = 0$: Clearly the recursive equation holds in the base case.

$k > 0$: We fix some $i, j \in X$ and show that the recursive equation holds for $\mathit{constraint}(i,j,k)$. Let $\rhd = \mathit{constraint}(i,j,k)$.

Let $\rhd_1 = \mathit{constraint}(i,j,k-1)$. If $\rhd_1 \neq \bot$, we have $S \vdash_{k-1} i \rhd_1 j$ and thus $S \vdash_k i \rhd_1 j$. Therefore $\rhd_1 \sqsubseteq \rhd$ (1).

Let $\rhd_a = \mathit{constraint}(i,k,k-1)$ and $\rhd_b = \mathit{constraint}(k,j,k-1)$. We set $\rhd_2 = \rhd_a \cdot \rhd_b$. If $\rhd_2 \neq \bot$, there is the k-resolvent $i \rhd_2 j$ and we have $S \vdash_k i \rhd_2 j$. Therefore $\rhd_2 \sqsubseteq \rhd$ (2).

Combining (1) and (2) we get $\rhd_1 \sqcup \rhd_2 \sqsubseteq \rhd$.

By the definition of $\vdash_k$ we have that $S \vdash_k i \rhd j$ is established by using a k-resolvent or by using a k'-resolvent with $k' < k$. In the first case we have $\rhd = \rhd_2$. In the second case we have $S \vdash_{k-1} i \rhd j$ and thus $\rhd = \rhd_1$. From both cases we get $\rhd_1 \sqcup \rhd_2 \sqsupseteq \rhd$. Thus $\rhd_1 \sqcup \rhd_2 = \rhd$. □

```
Input: a set of order constraints $S \subseteq OC(X)$
Output: the saturation $\overline{S}$ of $S$
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        $constraint[i][j] = \begin{cases} >, & i > j \in S \\ \geq, & i \geq j \in S \wedge i > j \notin S \\ \perp, & \text{else} \end{cases};$
for $k := 1$ to $n$ do
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $constraint[i][j] :=$
            $constraint[i][j] \sqcup (constraint[i][k] \cdot constraint[k][j])$;
for $i := 1$ to $n$ do
    if $constraint[i][i] = \perp$ then return "$S$ is unsatisfiable"
return $\{i \rhd j \mid \rhd \sqsubseteq constraint[i][j]$ and $\perp \neq \rhd\}$;
```

Algorithm 7: `Floyd-Warshall` computes the saturation of a set of constraints

We use the recursive equation stated in Lemma 60 in order to compute $constraint(i, j, k)$ iteratively for increasing values of k. Algorithm 7 implements this iteration in a particular way. First Algorithm 7 initializes the array `constraint` as stated in Lemma 60. Then Algorithm 7 computes $constraint(i, j, k)$ for increasing values of k in its main loop (the for-loop that iterates over k). Note that Algorithm 7 is an *in-situ* algorithm that uses the two-dimensional array `constraint` for representing the values of $constraint(i, j, k)$ for two subsequent iterations $k - 1$ and k. This has the effect that Algorithm 7 overwrites values from iteration $k - 1$ while calculating k. This does not effect correctness because the only difference to the recursive equation stated in Lemma 60 is that valid constraints can probably be inferred earlier, i.e., for smaller values of k. However, the *in-situ* implementation has the advantage to greatly reduce the memory needed.

Lemma 61. *Let S be a set of order constraints over $X = \{1, \dots, n\}$. Algorithm 7 computes the saturation $\overline{S}$ in time $O(n^3)$ and space $O(n^2)$.*

Proof. Note that throughout all iterations of the main loop (the for-loop that iterates over k) we have the following invariant: $\mathit{constraint}(i, j, k) \sqsubseteq \mathit{constraint}[i][j]$ for all $i, j \in X$. Therefore after the termination of the main loop we have $\mathit{constraint}(i, j, n) \sqsubseteq \mathit{constraint}[i][j]$ for all $i, j \in X$ (1).

Assume that there is an i with $S \vdash_n i > i$. With (1) we then get $> = \mathit{constraint}(i, i, n) \sqsubseteq \mathit{constraint}[i][i]$. Thus $\mathit{constraint}[i][i] = >$. Algorithm 7 correctly recognizes S as unsatisfiable.

Assume that there is no i with $S \vdash_n i > i$. By Lemma 59 $S \vdash i \rhd j$ implies $S \vdash_n i \rhd j$ for all $i, j \in X$. Furthermore $\mathit{constraint}[i][j] = \rhd \neq \bot$ implies $S \vdash i \rhd j$ for all $i, j \in X$. Combining both implications we get $\mathit{constraint}[i][j] \sqsubseteq \mathit{constraint}(i, j, n)$ for all $i, j \in X$. With (1) we get $\mathit{constraint}[i][j] = \mathit{constraint}(i, j, n)$ for all $i, j \in X$. Thus $S \vdash i \rhd j$ iff $\rhd \sqsubseteq \mathit{constraint}[i][j]$ and $\bot \neq \rhd$. By 53 we have $S \vdash i \rhd j$ iff $S \models i \rhd j$. Thus Algorithm 7 computes the saturation $\overline{S}$.

Clearly Algorithm 7 executes in time $O(n^3)$ and space $O(n^2)$. □

5.2 Size-change Abstraction

In this section we define size-change systems (SCSs), which provide an abstract program model. An SCS consists of program variables, control locations and transitions between the control locations. Each control location is labeled by invariant constraints over the program variables that describe the valid program states at this control location. Each transition has a start control location and an end control location and is labeled by transition constraints over the primed and unprimed program variables that describe the possible state transitions from the start control location to the end control location. The term *size-change* accounts for the fact that invariant- and transition constraints are inequalities between two variables. Our SCSs have control location invariants whereas the original definition of SCSs [Lee et al., 2001] does not have invariants. The addition of invariants is motivated by [Ben-Amram, 2009b], where Ben-Amram generalizes SCSs to monotonicity constraints systems (MCSs) which include control location invariants. For the ease of explanation we restrict ourself to SCSs which have invariants and leave the generalization to MCSs for future work.

5.2.1 Size-change Systems

Definition 62 (Size-change System (SCS)). *A* size-change system $\mathcal{A} = (Var, L, \rightarrow, Inv)$ *consists of*

- *a finite set of* program variables *Var,*
- *a finite set of* control locations *L,*
- *a finite set of* transitions $\rightarrow \subseteq L \times 2^{DEC(Var)} \times L$*, and*
- *an* invariant function $Inv : L \rightarrow 2^{OC(Var)}$ *that assigns an* invariant $Inv(l)$ *to every control location* $l \in L$*.*

Given an SCS $\mathcal{A}$ we refer by $L^{\mathcal{A}}, \rightarrow^{\mathcal{A}}, Inv^{\mathcal{A}}$ to its set of control locations, its set of transitions and its invariant function. We refer by $Var^{\mathcal{A}}$ to the set of program variables over which the transitions and invariants are defined. We denote $(l_1, T, l_2) \in \rightarrow^{\mathcal{A}}$ by the more intuitive $l_1 \xrightarrow{T} l_2 \in \mathcal{A}$.

Remark 63. *In the above definition of SCSs we have one set of program variables Var. An alternative for defining SCSs would be to associate a set* *of variables $Var(l)$ with every control location $l \in L$ and to require $Inv(l) \subseteq OC(Var(l))$ for every location $l \in L$ and $T \subseteq OC(Var(l_1) \cup Var(l_2))$ for every transition $l_1 \xrightarrow{T} l_2$. Clearly our definition is a special case of the alternative definition. However, the alternative definition is not more expressive. We can add to every control location the program variables of all other locations by setting $Var = \bigcup_{l \in L} Var(l)$. Because we do not change the set of transitions we obtain an essentially identical SCS that matches our above definition of SCSs. Thus both definitions are equivalent. We choose to have only one set of program variables as this simplifies the presentation.*

A transition $l_1 \xrightarrow{T} l_2$ intuitively denotes – as we make precise in Definition 68 below – that the variables Var capture the values of the variables at the start point l_1 of the transition, the variables Var' capture the values of the variables at the end point of the transition l_2, and the SCR T which labels the transition describes how these values are related.

Definition 64 (Run). *A run $\mathcal{R}$ of $\mathcal{A}$ is a (finite or infinite) sequence of transitions $l_i \xrightarrow{T_i} l_{i+1}$. We denote a run by $l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$.*

In Definition 68 below we define traces of SCSs, which add to every location of a run a valuation over some value domain, and thus provide a semantics for SCSs. As we are interested in program termination, we want to have value domains in which infinite descent is not possible. Such domains are called well-founded in mathematics.

Definition 65 (Well-founded Sets). *A set W is* well-founded *with regard to a relation $> \subseteq W \times W$, if there is no infinite sequence $w_1, w_2, \ldots \in W^\omega$ such that $w_i > w_{i+1}$ for all $i \in \omega$.*

Well-founded sets also have an alternative characterization.

Remark 66. *A set W is* well-founded *with regard to a relation $> \subseteq W \times W$ iff every non-empty subset of W has a minimal element with respect to $>$, formally:*

$$\forall V \subseteq W.\ V \neq \emptyset \to \exists m \in V.\ \forall v \in V. m \not> v$$

Furthermore, we want to interpret variables in a linearly ordered domain as we need to assign truth values to invariants. Sets that are well-founded and linearly ordered are called well-ordered in mathematics.

Definition 67 (Well-ordered Set)**.** *A set W is* well-ordered *with regard to a relation $> \subseteq W \times W$, if W is well-founded with regard to $>$ and if the reflexive closure $\geq$ of $>$ is a linear order over W.*

As every well-ordered set is isomorphic to an ordinal we will interpret SCSs over ordinals. We use the letter α to denote an arbitrary ordinal, and we use ω to denote the smallest ordinal that is isomorphic to the natural numbers.

Definition 68 (Trace)**.** *Given an ordinal α and an SCS $\mathcal{A}$, a* trace *over α of $\mathcal{A}$ is a (finite or infinite) sequence $(l_0, \sigma_0) \xrightarrow{T_0} (l_1, \sigma_1) \xrightarrow{T_1} \cdots$ with $\sigma_i \in Val_\alpha$ such that $l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$ is a run of $\mathcal{A}$ and we have $\sigma_i \models Inv(l_i)$ and $\sigma_i \cup \sigma'_{i+1} \models T_i$ for all i.*

Note that traces are parameterized by the ordinal α.

Definition 69 (Termination of a Run (Over an Ordinal))**.** *Let $\mathcal{A}$ be an SCS and let $\mathcal{R} = l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$ be a run of $\mathcal{A}$. $\mathcal{R}$ is* terminating over α*, if there is no infinite trace $(l_0, \sigma_0) \xrightarrow{T_0} (l_1, \sigma_1) \xrightarrow{T_1} \cdots$ with $\sigma_i \in Val_\alpha$. $\mathcal{R}$ is* terminating*, if it is terminating over every α.*

Note that there are two definitions of the termination of a run, one that is parameterized by α and one that quantifies over all ordinals α.

Definition 70 (Uniform Termination)**.** *An SCS $\mathcal{A}$ is* uniformly terminating*, if every run of $\mathcal{A}$ is terminating.*

As uniform termination refers to the termination of a run, it implicitly quantifies over all ordinals α.

Definition 71 (Consistency)**.** *A control location f is* consistent*, if its invariant $Inv(l)$ is satisfiable.*

An SCS is consistent*, if all its control locations are consistent.*

As no trace includes an inconsistent control location, we can remove all inconsistent locations and their incident transitions without changing the set of traces. As it can easily be determined if a set of order constraints is satisfiable, we assume in the following that we always deal with consistent SCSs.

In [Kupferman and Vardi, 2001], Kupferman and Vardi introduce a Büchi complementation algorithm that uses level rankings of run DAGs as witnesses for the absence of an accepting run. We adopt their notion of a run DAG for our purposes.

Definition 72 (Run DAG)**.** *Given a run $\mathcal{R} = l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$ of some SCS, we define the* run DAG *of $\mathcal{R}$ as the directed acyclic graph (DAG) $\mathcal{G}_\mathcal{R} = (V, E)$ with vertices $V = \mathbb{N} \times Var$ and labeled edges $E \subseteq (\mathbb{N} \times Var) \times \{>, \geq\} \times (\mathbb{N} \times Var)$, where $((d_1, x_1), \rhd, (d_2, x_2)) \in E$ if and only if*

- *$d_2 = d_1 + 1$ and $x_1 \rhd x_2' \in T_{d_1}$, or*
- *$d_2 = d_1$ and $x_1 \rhd x_2 \in Inv(l_{d_1})$.*

For every vertex $v = (d, x)$, we call d the level *of v.*

Definition 73 (Walk, Thread)**.** *Let $\mathcal{R}$ be a run of some SCS and let $\mathcal{G}_{\mathcal{R}} = (V, E)$ be its run DAG. A* walk *of $\mathcal{R}$ is an infinite labeled path in $\mathcal{G}_{\mathcal{R}}$ with vertices (d_i, x_i) and labels $\rhd_i$ such that $((d_i, x_i), \rhd_i, (d_{i+1}, x_{i+1})) \in E$ for all $i \in \mathbb{N}$. A walk is* decreasing *if at least one $\rhd_i$ is $>$. A walk is* infinitely decreasing *if infinitely many $\rhd_i$ are $>$. A* thread *of $\mathcal{R}$ is a walk with $d_{i+1} = d_i + 1$ for all $i \in \mathbb{N}$.*

Note, the level d_0 of the first vertex of a walk is not specified.

Definition 74 (Walk-termination resp. Thread-termination)**.** *A run $\mathcal{R}$ is* walk-terminating *resp.* thread-terminating*, if there is an infinitely decreasing walk resp. thread of $\mathcal{R}$.*

Definition 75 (Uniform Walk- resp. Thread-termination)**.** *An SCS $\mathcal{A}$ is* uniformly walk-terminating *resp.* uniformly thread-terminating*, if every run of $\mathcal{A}$ is walk- resp. thread-terminating.*

Lemma 76. *An infinite $\mathcal{R}$ is terminating over every ordinal α, if it is walk-terminating.*

Proof. Let $\mathcal{R}$ be a walk-terminating run. The run DAG of $\mathcal{R}$ contains an infinitely decreasing walk. Clearly, no ordinal α can have an infinitely decreasing chain of elements. Therefore walk-termination implies termination. □

5.2.2 Equivalence of Syntactic and Semantic Termination

In this subsection we prove the other direction of Lemma 76. Before we are able to do so we need some definitions and lemmata.

$$\begin{tikzcd}
(0,x_1) \arrow[r, ">"] & (1,x_1) \arrow[r, ">"] & (2,x_1) \arrow[r, ">"] & (3,x_1) \arrow[r, ">"] & (4,x_1) \arrow[r, ">"] & (5,x_1) \arrow[r, ">"] & (6,x_1) \arrow[r, ">"] & \cdots \\
\underset{\in M_\omega}{(0,x_2)} \arrow[r, "\geq"] \arrow[dr, ">"] & \underset{\in M_\omega}{(1,x_2)} \arrow[r, "\geq"] \arrow[dr, ">"] & \underset{\in M_\omega}{(2,x_2)} \arrow[r, "\geq"] & \underset{\in M_\omega}{(3,x_2)} \arrow[r, "\geq"] \arrow[dr, ">"] & \underset{\in M_\omega}{(4,x_2)} \arrow[r, "\geq"] & \underset{\in M_\omega}{(5,x_2)} \arrow[r, "\geq"] & \underset{\in M_\omega}{(6,x_2)} \arrow[r, "\geq"] \arrow[dr, ">"] & \cdots \\
 & \underset{\in M_0}{(1,x_3)} & \underset{\in M_1}{(2,x_3)} \arrow[r, ">"] & \underset{\in M_0}{(4,x_3)} & \underset{\in M_2}{(3,x_3)} \arrow[r, ">"] & \underset{\in M_1}{(5,x_3)} \arrow[r, ">"] & \underset{\in M_0}{(6,x_3)} & \cdots
\end{tikzcd}$$

Figure 5.2: A run DAG and the orbits of its vertices.

Definition 77 (Decrease Order). *Let $\mathcal{G}_\mathcal{R} = (V, E)$ be the run DAG of some $\mathcal{R}$. The* decrease order *is the relation $>_\mathcal{R} \subseteq V \times V$ defined by: $v_1 >_\mathcal{R} v_2$ if there is a decreasing walk from v_1 to v_2 in $\mathcal{G}_\mathcal{R}$.*

Proposition 78. *$>_\mathcal{R}$ is a partial order.*

Proof. Clearly $>_\mathcal{R}$ is transitive by the definition of walks. $>_\mathcal{R}$ is irreflexive because all invariants are consistent. $\square$

For every subset $U \subseteq V$ we denote the set of minimal elements with regard to $>_\mathcal{R}$ by $\min_{>_\mathcal{R}} U$.

Definition 79 (Orbits). *Let $\mathcal{G}_\mathcal{R} = (V, E)$ be the run DAG of some $\mathcal{R}$. The* orbits *$M^\mathcal{R}_\alpha \subseteq V$ of $\mathcal{R}$ are defined by transfinite induction: $M^\mathcal{R}_\alpha = \min_{>_\mathcal{R}}(V \setminus \bigcup_{\beta<\alpha} M^\mathcal{R}_\beta)$.*

We illustrate the notion of an orbit in the next example.

Example 80. *Figure 5.2 shows the run DAG of some run $\mathcal{R}$. Every vertex (d, x_3) is element of some orbit $M^\mathcal{R}_k$ with $k \in \omega$. This is because every (d, x_3) has only finitely many successors with regard to $>_\mathcal{R}$. All vertices (d, x_2) are element of the orbit $M^\mathcal{R}_\omega$ because they have a successor in every $M^\mathcal{R}_k$ for large enough k but do not start an infinitely decreasing walk. No vertex (d, x_1) is element of some orbit $M^\mathcal{R}_\alpha$, because every such vertex start an infinitely decreasing walk.*

Proposition 81. *Let $\mathcal{R}$ be some run. For all ordinals α_1, α_2 with $\alpha_1 \neq \alpha_2$ their orbits are disjoint, i.e., $M^{\mathcal{R}}_{\alpha_1} \cap M^{\mathcal{R}}_{\alpha_2} = \emptyset$.*

Lemma 82. *Let $\mathcal{R}$ be some run. For all ordinals α and vertices $v \in M^{\mathcal{R}}_{\alpha}$ there is no infinitely decreasing walk of $\mathcal{R}$ that starts with v.*

Proof. We proceed by transfinite induction on α. Let α be an ordinal such that for all $\beta < \alpha$ and vertices v it holds that $v \in M^{\mathcal{R}}_{\beta}$ implies that there is no infinitely decreasing walk of $\mathcal{R}$ that starts with v. Assume that there is an infinitely decreasing walk $v_1 >_{\mathcal{R}} v_2 >_{\mathcal{R}} v_3 >_{\mathcal{R}} \cdots$ of $\mathcal{R}$ that starts with a vertex $v_1 \in M^{\mathcal{R}}_{\alpha}$. From the definition of $M^{\mathcal{R}}_{\alpha}$ we have that $v_1 \in \min_{>_{\mathcal{R}}}(V \setminus \bigcup_{\beta<\alpha} M^{\mathcal{R}}_{\beta})$. Thus we know that $v_2 \in \bigcup_{\beta<\alpha} M^{\mathcal{R}}_{\beta}$. Therefore there must be a $\beta < \alpha$ such that $v_2 \in M_{\beta}$. As $v_2 >_{\mathcal{R}} v_3 >_{\mathcal{R}} \cdots$ is an infinitely decreasing walk of $\mathcal{R}$ this is a contradiction to the induction assumption. □

Lemma 83. *Let $\mathcal{R}$ be a run, let $\mathcal{G}_{\mathcal{R}} = (V, E)$ be the run DAG of $\mathcal{R}$ and let $\emptyset \neq U \subseteq V$ be a set such that $\min_{>_{\mathcal{R}}} U = \emptyset$. For every vertex $v \in U$ there is an infinitely decreasing walk of $\mathcal{R}$ that starts with v.*

Proof. For every $u \in U$ we define sets $S_u = \{v \in U \mid u >_{\mathcal{R}} v\}$. We have $S_u \neq \emptyset$ for every $u \in U$ as U does not have a minimal element. By the axiom of choice there exists a function $f : U \to U$ with $f(u) \in S_u$. The function l gives an infinitely decreasing walk $u >_{\mathcal{R}} f(u) >_{\mathcal{R}} f^2(u) >_{\mathcal{R}} \cdots$ for every $u \in U$. □

We recall an axiom of set theory.

Definition 84 (Axiom Schema of Replacement)**.** *In ZFC the* axiom schema of replacement *is given by*

$$\forall w_1, \ldots, w_k \forall A([\forall x \in A \exists! y \phi(x, y, w_1, \ldots, w_k, A)]$$
$$\to \exists B \forall y[y \in B \Leftrightarrow \exists x \in A \phi(x, y, w_1, \ldots, w_k, A)]),$$

where ϕ is a formula in the language of set theory with free variables among $x, w_1, \dots, w_n, A$ such that B is not free in ϕ.

The axiom schema of replacement can be stated informally as follows: Suppose ϕ is a binary relation given as a formula in the language of set theory such that for every set x there is a unique set y such that $\phi(x, y)$ holds. We represent ϕ by a function F_ϕ with $F_\phi(x) = y$ if and only if $\phi(x, y)$. For every class A we can define a class B defined such that for every set y we have $y \in B$ if and only if there is a set $x \in A$ with $F_\phi(x) = y$. B is called the image of A under F_ϕ, and denoted $F_\phi(A)$ or $\{F_\phi(x) \mid x \in A\}$. The axiom schema of replacement states that for every set A and every function F_ϕ, the image $F_\phi(A)$ is also a set.

Lemma 85. *For every run $\mathcal{R}$ there is a least ordinal α such that $M^{\mathcal{R}}_\alpha = \emptyset$.*

Proof. We set $U = \{v \in V \mid \exists \alpha . v \in M_\alpha\}$. We instantiate the axiom schema of replacement by setting $k = 0$, $A = U$, $x = v$, $y = \alpha$ and $\phi(v, \alpha) = \alpha \text{ is an ordinal } \wedge v \in M_\alpha$. For every $v \in U$ there is only one ordinal α such that $v \in M_\alpha$ by Proposition 81. By the axiom schema of replacement we get that the class $B = \{\alpha \mid \exists v \in U . \alpha \text{ is an ordinal } \wedge v \in M_\alpha\} = \{\alpha \mid \alpha \text{ is an ordinal } \wedge \exists v \in V . v \in M_\alpha\}$ is a set. As a set cannot contain all ordinals there is a least ordinal α such that $\alpha \notin B$. By the definition of orbits we have that $M_\alpha = \emptyset$. □

Definition 86 (Rank of a Run)**.** *We define the* rank *$rank(\mathcal{R})$ to be the least ordinal α such that $M^{\mathcal{R}}_\alpha = \emptyset$.*

The rank always exists by Lemma 85.

Lemma 87. *For every run $\mathcal{R}$ that is not walk-terminating we have $V = \bigcup_{\beta < rank(\mathcal{R})} M^{\mathcal{R}}_\beta$.*

Proof. Let $\alpha = rank(\mathcal{R})$ and let $U = V \setminus \bigcup_{\beta<\alpha} M_\beta^{\mathcal{R}}$. By the definition of orbits we have $\emptyset = M_\alpha^{\mathcal{R}} = \min_{>_{\mathcal{R}}} U$. There is no $v \in U$ because such a v would start an infinite thread by Lemma 83. Thus we must have $\emptyset = U = V \setminus \bigcup_{\beta<\alpha} M_\beta^{\mathcal{R}}$. □

If $\mathcal{R}$ is not walk-terminating, there is a $\beta < rank(\mathcal{R})$ with $v \in M_\beta$ for every $v \in V$ by the above lemma. Moreover there is only one such a β for every $v \in V$ by Proposition 81. We use this fact to define ranks of vertices.

Definition 88 (Rank of a Vertex)**.** *Let $\mathcal{R}$ be a run which is not walk-terminating. We define the* rank *$rank(v)$ of the vertex $v \in V$ to be the unique $\beta < rank(\mathcal{R})$ with $v \in M_\beta$.*

In the next lemma we state that the rank respects the edge labels of the run DAG for all vertices.

Lemma 89. *Let $\mathcal{G}_{\mathcal{R}} = (V, E)$ be the run DAG of $\mathcal{R}$ which is not walk-terminating. We have $rank(v_1) \rhd rank(v_2)$ for every edge $(v_1, \rhd, v_2) \in E$.*

Proof. Let $(v_1, \rhd, v_2) \in E$ be an edge. We have $v_1 \in M_{rank(v_1)}$ and $v_2 \in M_{rank(v_2)}$. In the case $\rhd = \geq$ we have $rank(v_1) \geq rank(v_2)$ by the definition of orbits. In the case $\rhd = >$ we have $rank(v_1) > rank(v_2)$ by the definition of orbits. From both cases we get $rank(v_1) \rhd rank(v_2)$. □

Ranks of vertices provide us a way of defining canonic valuations.

Definition 90 (Canonic Valuations)**.** *Let $\mathcal{R} = l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$ be a run which is not walk-terminating. The* canonic valuations *$\sigma_0, \sigma_1, \ldots \in Val_{rank(\mathcal{R})}$ are defined by $\sigma_i(x) = rank(i, x)$.*

The canonic valuations give an infinite trace as stated by the next lemma.

Lemma 91. *Let $\mathcal{R} = l_0 \xrightarrow{T_0} l_1 \xrightarrow{T_1} \cdots$ be a run which is not walk-terminating and let $\sigma_0, \sigma_1, \ldots$ be the canonic valuations. $(l_0, \sigma_0) \xrightarrow{T_0} (l_1, \sigma_1) \xrightarrow{T_1} \cdots$ is an infinite trace for run $\mathcal{R}$.*

Proof. We show that every size-decrease and invariant constraint of $\mathcal{R}$ is satisfied by the canonic valuations.

Let $x \rhd y' \in T_i$ be a size-decrease constraint with $x, y \in \mathit{Var}$. We set $v_1 = (i, x)$ and $v_2 = (i+1, y)$. By the definition of the run DAG we have $(v_1, \rhd, v_2) \in E$. We get $\sigma_i(x) = \mathit{rank}(v_1) \rhd \mathit{rank}(v_2) = \sigma_{i+1}(y)$ by Lemma 89. Thus we have $\sigma_i, \sigma'_{i+1} \models x \rhd y'$.

Let $x \rhd y \in \mathit{Inv}(l_i)$ be an invariant constraint with $x, y \in \mathit{Var}$. We set $v_1 = (i, x)$ and $v_2 = (i, y)$. By the definition of the run DAG we have $(v_1, \rhd, v_2) \in E$. We get $\sigma_i(x) = \mathit{rank}(v_1) \rhd \mathit{rank}(v_2) = \sigma_i(y)$ by Lemma 89. Thus we have $\sigma_i \models x \rhd y$. □

We obtain as a corollary the theorem stated below.

Theorem 92. *If a run $\mathcal{R}$ is not walk-terminating, then it does not terminate over $\mathit{rank}(\mathcal{R})$.*

Up to now we did not use any structure of run DAGs except the assumption that they are DAGs which do not have infinitely decreasing walks.

For the proof of the existence of a rank for every run in Lemma 85 we relied on the axiom schema of replacement and for the proof of the existence of infinitely decreasing walks for subsets which do not have minimal elements in Lemma 83 we relied on the axiom of choice.

However, by exploiting more structure of run DAGs we do not have to rely on these axioms and we are able to obtain a stronger result.

For the proof of Lemma 83 we do not need to refer to the axiom of choice because the vertex set of the run DAG is countable.

In the following lemma we formulate an analogy to the insight in the Kupferman-Vardi construction [Kupferman and Vardi, 2001] that an endangered vertex starts an infinite path of endangered vertices (for a definition of endangered vertices see [Kupferman and Vardi, 2001]).

Lemma 93. *Let γ be a limit ordinal with $M_\gamma^{\mathcal{R}} \neq \emptyset$. Every vertex $v \in M_\gamma^{\mathcal{R}}$ has an infinite walk $v \rhd_1 v_1 \rhd_2 v_2 \cdots$ with $v_i \in M_\gamma^{\mathcal{R}}$.*

Proof. Let $v \in M_\gamma^{\mathcal{R}}$ be a vertex. We define

$$Reach_\gamma^{\mathcal{R}}(v) = \{u \in M_\gamma^{\mathcal{R}} \mid \text{there is a walk of } \mathcal{R} \text{ from } v \text{ to } u\}.$$

Further, we define

$$Next_\gamma^{\mathcal{R}}(v) = \{w \mid w \notin M_\gamma^{\mathcal{R}} \wedge \exists u \in Reach_\gamma^{\mathcal{R}}(v).\ u \rhd w \in E_{\mathcal{R}}\}.$$

Because of $w \notin M_\gamma^{\mathcal{R}}$ we have $rank(w) < \gamma$ for every $w \in Next_\gamma^{\mathcal{R}}(v)$ (*). By the definition of orbits we have $\gamma = \bigcup_{w \in Next_\gamma^{\mathcal{R}}(v)} rank(w)$ (**).

Assume $|Reach_\gamma^{\mathcal{R}}(v)|$ is finite. As $\mathcal{G}_{\mathcal{R}}$ is finitely branching $|Next_\gamma^{\mathcal{R}}(v)|$ is finite. Thus $\bigcup_{w \in Next_\gamma^{\mathcal{R}}(v)} rank(w) < \gamma$ by (*) and the assumption that γ is a limit ordinal. This is contradiction to (**). Therefore $|Reach_\gamma^{\mathcal{R}}(v)|$ is infinite.

$Reach_\gamma^{\mathcal{R}}(v)$ is an infinite subgraph of $\mathcal{G}_{\mathcal{R}}$. $Reach_\gamma^{\mathcal{R}}(v)$ is a DAG because every vertex in $Reach_\gamma^{\mathcal{R}}(v)$ can be reached from v. $Reach_\gamma^{\mathcal{R}}(v)$ is finitely branching because $\mathcal{G}_{\mathcal{R}}$ is finitely branching. There is an infinite walk starting from v by König's Lemma because $Reach_\gamma^{\mathcal{R}}(v)$ is an infinite finitely branching DAG. $\square$

In the Kupfermann-Vardi construction almost all levels in a run DAG contain an endangered vertex, if there is an endangered vertex. At every step all endangered vertices are removed. As the number of different elements at each level is bounded the construction terminates because almost all levels loose an element every step. In the next lemma we adopt this reasoning to our setting. We obtain a stronger result than in Lemma 85.

Lemma 94. *For every run $\mathcal{R}$ of an SCS $\mathcal{A}$ with $|Var^{\mathcal{A}}| = n$ we have $rank(\mathcal{R}) \leq n\omega$.*

Proof. We assume that $rank(\mathcal{R}) > n\omega$. Thus $M_\alpha \neq \emptyset$ for all $\alpha \leq n\omega$.

By Lemma 93 we have that for every $1 \leq k \leq n$ there are infinite walks $v_1^k \rhd_1^k v_2^k \rhd_2^k \cdots$ with $v_i^k \in M_{k\omega}^{\mathcal{R}}$.

We define $m = \max\{d_k \mid 1 \leq k \leq n \wedge v_1^k = (d_k, x_k)\}$ and set $V_m = \{v \mid v = (d, x) \wedge d \geq m\}$. Because the $M_{k\omega}$'s are pairwise disjoint as stated by Proposition 81 we have that these walks partition V_m. Thus for every vertex $v \in V_m$ we have $rank(v) = k\omega$ for some $1 \leq k \leq n$.

For all $v_1, v_2 \in V_m$ the edge $(v_1, >, v_2) \in E_{\mathcal{R}}$ implies $rank(v_1) > rank(v_2)$. As we have $rank(v) = k\omega$ for some $1 \leq k \leq n$ for every vertex $v \in V_m$ every decreasing walk in V_m can contain at most $n - 1$ strict decreases $>$.

Choose some $v \in V_m$. Because of $rank(v) \geq \omega$ there is a walk for any number n such that the walk contains at least n strict decreases $>$. This is a contradiction. □

Combining the results of Lemma 76, Theorem 92 and Lemma 94 we obtain that the notions of walk-termination and termination coincide for SCSs which are interpreted over ordinals greater than $n\omega$.

Theorem 95. *Let $\mathcal{R}$ be a run of an SCS $\mathcal{A}$ with $|Var^{\mathcal{A}}| = n$ For all ordinals $\alpha \geq n\omega$ we have that $\mathcal{R}$ is walk-terminating iff $\mathcal{R}$ is terminating over α.*

In the next lemma we state that Theorem 95 cannot be strengthened.

Lemma 96. *There is an SCS $\mathcal{A}$ with $|Var^{\mathcal{A}}| = n$ that has a $\mathcal{R}$ that is not walk-terminating but that is terminating over every $\alpha < n\omega$.*

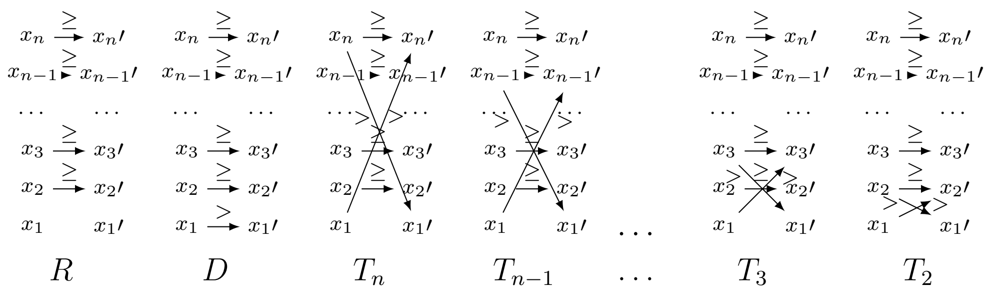


Figure 5.3: SCRs of the SCS used in the proof of Lemma 96.

Proof. Let $\mathcal{A}$ be the SCS $(\{x_1, \dots, x_n\}, \{l\}, \{l \xrightarrow{R} l, l \xrightarrow{D} l, l \xrightarrow{T_n} l, l \xrightarrow{T_{n-1}} l, \dots, l \xrightarrow{T_2} l\}, \{l \mapsto \emptyset\})$, whose SCRs are shown in Figure 5.3. Consider the run

$$\mathcal{R} = DT_nDT_{n-1} \cdots DT_2RD^2T_nD^2T_{n-1} \cdots D^2T_2RD^3T_nD^3T_{n-1} \cdots D^3T_2R \cdots .$$

Note that $\mathcal{R}$ is not walk-terminating but is terminating over every $\alpha < n\omega$. □

Assume that a run $\mathcal{R}$ of some SCS $\mathcal{A}$ is given. We want to determine if $\mathcal{R}$ is terminating. It is desirable to be able to decide this question based on the syntactic criterion of walk-termination. By Theorem 95 we know that walk-termination is sufficient if $\mathcal{A}$ is interpreted over $n\omega$. By Lemma 96 we know that we can not rely on walk-termination if $\mathcal{A}$ is interpreted over some ordinal less than $n\omega$. As $n\omega$ is the least ordinal such that the notions of walk-termination and termination coincide, we define $n\omega$ as the *canonic value domain* for SCSs.

5.2.3 Deciding the Termination of SCSs

In this subsection we state the criterion for termination of Lee et al. [Lee et al., 2001]. We first give the relevant definitions.

Walks have the undesirable property that they can contain edges which are not forward, i.e., a walk can contain an edge $v_1 \rhd v_2$ such that v_1 and v_2 are at the same level; in contrast, threads contain only forward edges. In the next definition we state a property of transitions which will allow us to state an equivalence between walk-termination and thread-termination shortly.

Definition 97 (Saturation)**.** *Let $l_1 \xrightarrow{T} l_2 \in \mathcal{A}$ be a transition of some SCS $\mathcal{A}$. We define the transition $l_1 \xrightarrow{\mathcal{H}} l_2$ to be the* saturation *of $l_1 \xrightarrow{T} l_2$, if $\mathcal{H} = \{x_1 \rhd x_2' \mid Inv(l_1), T, Inv(l_2)' \models x_1 \rhd x_2' \wedge x_1, x_2 \in Var^{\mathcal{A}}\}$, and call $l_1 \xrightarrow{T} l_2$* saturated*, if $\mathcal{H} = T$.*

Note that T' in the above definition does only include size-decrease constraints and therefore is an SCR.

Corollary 98. *Let $l_1 \xrightarrow{T} l_2 \in \mathcal{A}$ be a transition of some SCS $\mathcal{A}$. The saturation of $l_1 \xrightarrow{T} l_2$ can be computed in $O(n^3)$.*

Proof. Apply Lemma 61 and restrict the result to size-decrease constraints. □

Definition 99 (Saturation)**.** *An SCS $\mathcal{A}$ is* saturated*, if all its transitions are saturated.*

Lemma 100. *Let $\mathcal{A}$ be a saturated SCS $\mathcal{A}$. We have that $\mathcal{A}$ is thread-terminating, if it is walk-terminating.*

Proof. Let $\mathcal{R}$ be a run of $\mathcal{A}$. As $\mathcal{A}$ is walk-terminating it contains an infinitely decreasing walk $w = v_1 \rhd_1 v_2 \rhd_2 \cdots$. Let $(l_i, x_i) \rhd_i \ldots \rhd_{j-2} (l_{j-1}, x_{j-1}) \rhd_{j-1} (l_j, x_j)$ be a subsequence of w such that $l_i = l_{i+1} = \cdots = l_{j-1}$ and $l_j = l_i + 1$. Then the run DAG also contains the edge $(l_i, x_i) \rhd_i (l_j, x_j)$ because all transitions are saturated. In this way every subsequence of the above form can be replaced by a forward edge. We obtain an infinitely decreasing run. □

Corollary 101. *A saturated SCS $\mathcal{A}$ is terminating, iff it is thread-terminating.*

Definition 102 (Concatenation of SCRs)**.** *Given two SCRs T_1 and T_2, we define the concatenation $T_1 \circ T_2$ to be the SCR T, with $x_1 \rhd x_2' \in T$, if $T_1[\mathit{Var}''/\mathit{Var}'], T_2[\mathit{Var}''/\mathit{Var}] \models x_1 \rhd x_2'$.*

The next remark is a comment on the representation of SCRs in a computer.

Remark 103 (Representation)**.** *We can obtain a canonic representation of SCRs by dropping redundant constraints: Given an SCR T with $x > y' \in T$ and $x \geq y' \in T$, we remove the first constraint from T as it is implied by the second constraint.*

Given two SCRs T_1, T_2, the concatenation $T = T_1 \circ T_2$ can be obtained as follows:

- *$x_1 > x_2' \in T$, if there is a y, such that there are constraints $x_1 \rhd_1 y' \in T_1$ and $y \rhd_2 x_2' \in T_2$ and at least one of the constraints is strict, i.e., $\rhd_1 = >$ or $\rhd_2 = >$, and*
- *$x_1 \geq x_2' \in T$, if there is a y, such that there are constraints $x_1 \rhd_1 y' \in T_1$ and $y \rhd_1 x_2' \in T_2$, but not $x_1 > x_2' \in T$.*

Remark 104. *The set of size-change relations together with the operation $\circ$ is a semigroup.*

Definition 105 (Idempotent SCRs)**.** *We call a size-change relation T* idempotent, *if $T \circ T = T$.*

The number of SCRs is bounded by 3^{n^2} (n is the number of program variables), and is therefore finite. A finite non-empty semigroup of the form $\{T^k \mid k \in \mathbb{N}\}$ contains precisely one idempotent element. For every T we denote this idempotent element by T°.

```
**Procedure**: Closure($\mathcal{A}$)
**Input**: an SCS $\mathcal{A}$
**Output**: the transitive closure $\mathcal{A}^\star$
$\mathcal{A}^\star := \mathcal{A}$;
**while** $\exists l_1 \xrightarrow{T_1} l_2, l_2 \xrightarrow{T_2} l_3 \in \mathcal{A}^\star . l_1 \xrightarrow{T_1 \circ T_2} l_3 \notin \mathcal{A}^\star$ **do**
    $\mathcal{A}^\star := \mathcal{A}^\star \cup \{l_1 \xrightarrow{T_1 \circ T_2} l_3\}$;
**return** $\mathcal{A}^\star$
```

Algorithm 8: `Closure`($\mathcal{A}$) computes the transitive closure of $\mathcal{A}$

Definition 106 (Concatenation of Transitions). *Given two transitions $l_1 \xrightarrow{T_1} l_2$ and $l_2 \xrightarrow{T_2} l_3$ we define their concatenation to be the transition $l_1 \xrightarrow{T_1 \circ T_2} l_3$.*

Definition 107 (Closure). *Given an SCS $\mathcal{A}$ we define its* closure *$\mathcal{A}^\star$ to be the least set with*

- *$\mathcal{A} \subseteq \mathcal{A}^\star$, and*
- *for all transitions $l_1 \xrightarrow{T_1} l_2, l_2 \xrightarrow{T_2} l_3 \in \mathcal{A}^\star$ their concatenation $l_1 \xrightarrow{T_1 \circ T_2} l_3$ is in $\mathcal{A}^\star$.*

Lemma 108. *Algorithm 8 computes the closure $\mathcal{A}^\star$ of an SCS $\mathcal{A}$ in less than $m^2 \cdot 3^{n^2}$ iterations of the repeat-until loop, where $|Var^{\mathcal{A}}| = n$ is the number of program variables and m is the number of program locations.*

Proof. The number of SCRs is bounded by 3^{n^2}. Therefore the number of transitions is bounded by $m^2 \cdot 3^{n^2}$. In every iteration of the do-while loop, at least one transition is added. Therefore $\mathcal{A}^\star$ stabilizes within $m^2 \cdot 3^{n^2}$ steps. □

Theorem 109 ([Lee et al., 2001]). *A saturated SCS $\mathcal{A}$ is thread-terminating, if and only if for every transition $l \xrightarrow{T} l \in \mathcal{A}^\star$ with T idempotent, there is a size-decrease constraint $x > x' \in T$.*

5.3 Lower Bounds

In this section we state our results on the complexity of SCSs. The complexity of an SCS depends on the number of its variables. For the ease of explanation we fix the number of variables and assume for every SCS $\mathcal{A}$ in this section that $|\mathit{Var}^{\mathcal{A}}| = n$.

5.3.1 For-loops

Definition 110. *An SCR T* absorbs *an SCR $\mathcal{H}$, if $T \circ \mathcal{H} = T$.*

Note that absorbtion is a generalization of idempotence: an SCR T which absorbs itself is idempotent.

Definition 111 (For-loop)**.** *Given an SCS $\mathcal{A}$, a* for-loop *$\mathcal{L}$ of $\mathcal{A}$ is a pair $(l; L_1, \ldots, L_k)$ where $l \in L^{\mathcal{A}}$ is a location and $L_1, \ldots, L_k$ is a list of SCRs such that $l \xrightarrow{L_1} l, \ldots, l \xrightarrow{L_k} l \in \mathcal{A}^\star$ and L_i absorbs L_j for all $i \leq j$. We refer to l as the* header*, to $L_1, \ldots, L_k$ as the* loops *and to k as the* degree *of the* for-loop *$(l; L_1, \ldots, L_k)$.*

For the rest of this subsection we fix a for-loop $\mathcal{L} = (l; L_1, \ldots, L_k)$.

Definition 112 (Dependence Graph, Dependence Set)**.** *The* dependence graph *of $\mathcal{L}$ is the directed labeled graph $G_{\mathcal{L}} = (\mathit{Var}, E_{\mathcal{L}})$ with*

$$E_{\mathcal{L}} = \{(x, L_i, y) \mid 1 \leq i \leq k+1 \wedge x \rhd y' \in L_i\},$$

where the loop L_{k+1} is set to $\{x \rhd y' \mid \mathit{Inv}(l) \models x \rhd y\}$. An edge $(x, L_i, y) \in E_{\mathcal{L}}$ is decreasing*, if $x > y \in L_i$. A sequence of edges $(z_1, T_1, z_2)(z_2, T_z, z_3) \cdots (z_{l-1}, T_{l-1}, z_l)$ with $(z_i, T_i, z_{i+1}) \in E_{\mathcal{L}}$ for $1 \leq i < l$ is a* path *of $G_{\mathcal{L}}$. A path is* decreasing *if one of its edges is decreasing. A maximal SCC of $G_{\mathcal{L}}$ is a* dependence set *of $\mathcal{L}$. A dependence set S is* decreasing*, if one of its edges is decreasing, i.e., if there is a decreasing edge $(x, L_i, y) \in E_{\mathcal{L}}$ with $x, y \in S$.*

When we speak of runs and traces of a for-loop $\mathcal{L} = (l; L_1, \ldots, L_k)$, we refer to runs and traces of the SCS $\{l \xrightarrow{L_1} l, \ldots, l \xrightarrow{L_k} l\}$. As such runs and traces only involve the location l we will denote runs and traces differently in the rest of this section for ease of notation. A run $l \xrightarrow{T_1} l \xrightarrow{T_2} \cdots$ is denoted by $T_1 T_2 \cdots$. A trace $(l, \sigma_1) \xrightarrow{T_1} (l, \sigma_2) \xrightarrow{T_2} \cdots$ is denoted by $\sigma_1 \xrightarrow{T_1} \sigma_2 \xrightarrow{T_2} \cdots$.

By the definition of dependence sets we have the following lemma.

Lemma 113. *For every (decreasing) dependence set S and for all variables $x, y \in S$ there is a (decreasing) path $(z_1, T_1, z_2)(z_2, T_2, z_3) \cdots (z_{l-1}, T_{l-1}, z_l)$ with $x = z_1, y = z_l$ and $z_i \in S$.*

Definition 114 (Loop Indices, Activity Index, Active Variables)**.** *Let S be a dependence set. The* loop index set *$Loops(S)$ is the set $\{i \mid \exists x, y \in S.(x, L_i, y) \in E_{\mathcal{L}}\}$. The* activity index *$act(S) = \min Loops(S)$ is the minimum of the loop indices. The set of* active variables *of S is the set $Act(S) = \{x \in S \mid \exists y \in S.x \rhd y \in L_{act(S)}\}$.*

Note that always $Act(S) \neq \emptyset$.

Lemma 115. *For every dependence set S we have $x \rhd y \in L_{act(S)}$ for all variables $x \in Act(S), y \in S$, where $\rhd = >$, if S is decreasing, and where $\rhd = \geq$, if S is non-decreasing.*

Proof. Let S be a (decreasing) dependence set and let $x \in Act(S), y \in S$ be some variables. As $x \in Act(S)$ there is a $z \in S$ and an edge $(x, L_{act(S)}, z) \in E_{\mathcal{L}}$. By Lemma 113 there is a (decreasing) path $(z_1, T_1, z_2)(z_2, T_2, z_3) \cdots (z_{l-1}, T_{l-1}, z_l)$ with $z = z_1$,$y = z_l$ and $z_i \in S$. Thus we have $x \rhd y' \in L_{act(S)} \circ T_1 \circ \cdots \circ T_l$, where $\rhd = >$, if S is decreasing, and where $\rhd = \geq$, if S constant. For every T_i it holds that $i \in \mathit{Loops}(S)$ because we have $z_i, z_{i+1} \in S$. From $\mathit{act}(S) = \min \mathit{Loops}(S)$ and from the assumption that $L_{\mathit{act}(S)}$ absorbs all L_i with $i \geq \mathit{act}(S)$ we have $L_{\mathit{act}(S)} = L_{\mathit{act}(S)} \circ T_1 \circ \cdots \circ T_l$. Therefore we have $x \rhd y' \in L_{\mathit{act}(S)}$. □

Corollary 116. *For every decreasing dependence set S we have $\mathit{act}(S) \leq k$.*

Proof. Assume there is a decreasing dependence set S with $\mathit{act}(S) = k+1$. As every dependence set is non-empty there is a $x \in \mathit{Act}(S)$. By Lemma 115 we have $x > x' \in L_{k+1}$. By the definition of L_{k+1} we must have $\mathit{Inv}(l) \models x > x'$. This contradicts the assumption that all invariants are consistent. □

Definition 117 (Proper For-loop)**.** *A for-loop $(l; L_1, \dots, L_k)$ is* proper*, if it holds for all dependence sets S that $x \in S$ and $x > x' \in L_i$ imply $i = \mathit{act}(S)$.*

5.3.2 Value Intervals

Definition 118 (Partitioning)**.** *A sequence $t = S_0, \dots, S_h$ is an* ordered partitioning *of a set V, if all S_i are non-empty, pairwise disjoint and $V = \bigcup_{0 \leq i \leq h} S_i$. For every $v \in V$ we define the* index *$t(v) = i$ as the unique i such that $v \in S_i$.*

Definition 119 (Reverse-topological Ordering)**.** *Let $G = (V, E)$ be a directed graph. An ordered partitioning $t = S_0, \dots, S_h$ of V is a* reverse-topological ordering *of T, if*

- *every S_i is a maximal SCC of T,*
- *for every edge $(u, v) \in E$ we have $t(u) \geq t(v)$.*

Remark 120. *A reverse-topological ordering of a directed graph $G = (V, E)$ can be computed in linear time in the size of T, i.e., in $O(|V| + |E|)$ by Tarjan's algorithm.*

By the definition of reverse-topological orderings we have the following property of maximal SCCs:

Proposition 121. *Let $G = (V, E)$ be a directed graph, let t be a reverse-topological ordering of T and let S be a maximal SCC of T. There is a natural number t^S such that $t(v) = t^S$ for all $v \in S$.*

The above proposition motivates the following definition:

Definition 122. *Let $G = (V, E)$ be a directed graph, and let t be a reverse-topological ordering of T. For every maximal SCC S of T we define t^S to be the number such that $t(v) = t^S$ for all $v \in S$.*

The above definition associates a natural number with every SCC using a given reverse-topological ordering. By the definition of reverse-topological orderings these numbers have the following property:

Proposition 123. *Let $G = (V, E)$ be a directed graph, let t be a reverse-topological ordering of T and let S_1, S_2 be maximal SCCs of T with $S_1 \neq S_2$ such that there is an edge $(u, v) \in E$ with $u \in S_1, v \in S_2$. We have $t^{S_1} > t^{S_2}$.*

Definition 124 (Dependence Ordering)**.** *For every for-loop $\mathcal{L}$ we fix a reverse-topological ordering of the dependence graph $G_{\mathcal{L}}$. We denote this ordering by $t_{\mathcal{L}}$ and refer to it as the* dependence ordering *of $\mathcal{L}$.*

Definition 125 (Value Interval)**.** *Given a for-loop $\mathcal{L}$ and an ordinal β we define for every variable $x \in Var$ its* value interval *$I^\beta_{\mathcal{L}}(x) = [l^\beta_{\mathcal{L}}(x), r^\beta_{\mathcal{L}}(x)[= [t_{\mathcal{L}}(x) \cdot \beta \cdot n, (t_{\mathcal{L}}(x) + 1) \cdot \beta \cdot n[$.*

Lemma 126. *Let $\mathcal{L}$ be a for-loop with header l, and let σ be a valuation with $\sigma(x) \in I^\beta_{\mathcal{L}}(x)$ for every program variable $x \in Var$. For every constraint $x \triangleright y \in Inv(l)$ with $x \in S_1, y \in S_2$, where S_1, S_2 are dependence sets with $S_1 \neq S_2$, we have $\sigma \models x \triangleright y$.*

Proof. Let $x \rhd y \in Inv(l)$ be a constraint with $x \in S_1, y \in S_2$, where S_1, S_2 are dependence sets with $S_1 \neq S_2$. Because of $S_1 \neq S_2$ we have that S_1 and S_2 are different maximal SCCs of $G_\mathcal{L}$. Because of the edge $(x, L_{k+1}, y) \in E_\mathcal{L}$ we know that S_2 is reachable from S_1. Therefore we have $t_\mathcal{L}^{S_1} > t_\mathcal{L}^{S_2}$ and thus $t_\mathcal{L}(x) = t_\mathcal{L}^{S_1} > t_\mathcal{L}^{S_2} = t_\mathcal{L}(y)$. We have $\sigma(x) \geq t_\mathcal{L}(x) \cdot \beta \cdot n \geq (t_\mathcal{L}(y) + 1) \cdot \beta \cdot n > \sigma(y)$. □

Lemma 127. *Let $\mathcal{L}$ be a for-loop with loops $L_1, \dots, L_k$, and let σ_1, σ_2 be valuations with $\sigma_1(x), \sigma_2(x) \in I_\mathcal{L}^\beta(x)$ for every program variable $x \in Var$. For every constraint $x \rhd y' \in L_i$ with $x \in S_1, y \in S_2$, where S_1, S_2 are dependence sets with $S_1 \neq S_2$, we have $\sigma_1, \sigma_2' \models x \rhd y'$.*

Proof. Let $x \rhd y' \in L_i$ be a constraint with $x \in S_1, y \in S_2$, where S_1, S_2 are dependence sets with $S_1 \neq S_2$. Because of $S_1 \neq S_2$ we have that S_1 and S_2 are different maximal SCCs of $G_\mathcal{L}$. Because of the edge $(x, L_i, y) \in E_\mathcal{L}$ we know that S_2 is reachable from S_1. Therefore we have $t_\mathcal{L}^{S_1} > t_\mathcal{L}^{S_2}$ and thus $t_\mathcal{L}(x) = t_\mathcal{L}^{S_1} > t_\mathcal{L}^{S_1} = t_\mathcal{L}(y)$. We have $\sigma_1(x) \geq t_\mathcal{L}(x) \cdot \beta \cdot n \geq (t_\mathcal{L}(y) + 1) \cdot \beta \cdot n > \sigma_2(y)$. □

5.3.3 Offsets

Definition 128 (Offset Graph)**.** *Given a loop $\mathcal{L} = (l; L_1, \dots, L_k)$ and a dependence set S of $\mathcal{L}$, we define the* offset graph $G_\mathcal{L}^S$ *to be the dependence graph of the loop $(l; L_{act(S)+1}\restriction_S, \dots, L_k\restriction_S)$.*

Note that all offset graphs $G_\mathcal{L}^S$ are subgraphs of the dependence graph $G_\mathcal{L}$, induced by restricting the dependence graph $G_\mathcal{L}$ to vertices in S and to edges labeled by loops with an index greater than $act(S)$.

Definition 129 (Offset Ordering)**.** *For every loop $\mathcal{L} = (l; L_1, \dots, L_k)$ and every dependence set S of $\mathcal{L}$, we fix a reverse-topological ordering of the offset* *graph $G^S_{\mathcal{L}}$. We denote this ordering by $b^S_{\mathcal{L}}$ and refer to it as the* offset ordering *of S in $\mathcal{L}$.*

Lemma 130. *Let $\mathcal{L}$ be a proper for-loop. For every dependence set S of $G_{\mathcal{L}}$ the offset graph $G^S_{\mathcal{L}}$ does not have a decreasing dependence set.*

Proof. Assume $G^S_{\mathcal{L}}$ has a decreasing dependence set T. From $Act(T) \neq \emptyset$ we have that there is a $x \in Act(T)$. By Lemma 115 we have $x > x' \in L_{act(T)}$. By the definition of offset graphs we have $act(T) > act(S)$. This is a contradiction to the definition of proper for-loops. □

For the rest of this subsection we fix a proper for-loop $\mathcal{L} = (l; L_1, \dots, L_k)$ and a dependence set S of $\mathcal{L}$.

Corollary 131. *$Inv(l) \models x \rhd y$ with $x, y \in S$ implies $b^S_{\mathcal{L}}(x) \rhd b^S_{\mathcal{L}}(y)$.*

Corollary 132. *$x \rhd y' \in L_i$ with $x, y \in S$ and $i > act(S)$ implies $b^S_{\mathcal{L}}(x) \rhd b^S_{\mathcal{L}}(y)$.*

Lemma 133. *Let σ be a valuation and let c be a value such $\sigma(x) = c + b^S_{\mathcal{L}}(x)$ holds for every program variable $x \in S$. For every constraint $x \rhd y \in Inv(l)$ with $x, y \in S$ we have $\sigma \models x \rhd y$.*

Proof. Let $x \rhd y \in Inv(l)$ be a constraint with $x, y \in S$. We have $(x, L_{k+1}, y) \in E_S$. By Corollary 131 we have that $b^S_{\mathcal{L}}(x) \rhd b^S_{\mathcal{L}}(y)$. Therefore we have $\sigma(x) = c + b^S_{\mathcal{L}}(x) \rhd c + b^S_{\mathcal{L}}(y) = \sigma(y)$. □

Lemma 134. *Let σ_1, σ_2 be valuations and let c be a value such $\sigma_1(x) = \sigma_2(x) = c + b^S_{\mathcal{L}}(x)$ holds for every program variable $x \in S$. For every constraint $x \rhd y' \in L_i$ with $x, y \in S$ and $i > act(S)$ we have $\sigma_1, \sigma_2' \models x \rhd y'$.*

Proof. Let $x \rhd y' \in L_i$ be a constraint with $x, y \in S$ and $i > act(S)$. By Corollary 132 we have that $b^S_{\mathcal{L}}(x) \rhd b^S_{\mathcal{L}}(y)$. Therefore we have $\sigma_1(x) = c + b^S_{\mathcal{L}}(x) \rhd c + b^S_{\mathcal{L}}(y) = \sigma_2(y)$. □

5.3.4 Counters

Definition 135 (Counters, Constants)**.** *Let $(l; L_1, \ldots, L_k)$ be a for-loop. The set $Counters(L_i) = \bigcup_{S\colon \textit{dependence set } S \textit{ is decreasing } \wedge act(S)=i} S$ is the set of* counters *of loop L_i. The set $Const = Var - \bigcup_{1 \le i \le k} Counters(L_i)$ is the set of* constants*.*

Proposition 136. *We have $Const = \bigcup_{S\colon \textit{dependence set } S \textit{ is non-decreasing}} S$.*

Proof. By Lemma 116 we have that $act(S) \le k$ for all decreasing dependence sets S. □

By definition the set of constants $Const$ is disjoint from each set of counters $Counters(L_i)$. However, the counters of two different loops do not need to be disjoint in general. The situation is different for proper for-loops which follows directly from their definition.

Proposition 137. *Given a proper for-loop $(l; L_1, \ldots, L_k)$, we have that for all $i \neq j$ the sets $Counters(L_i)$ and $Counters(L_j)$ are disjoint.*

For the rest of this subsection we fix a proper for-loop $\mathcal{L} = (l; L_1, \ldots, L_k)$ and an ordinal β.

We use the fact that $Counters(L_1), \ldots, Counters(L_k), Const$ is a partitioning of Var to define a valuations through values $c_1, \ldots, c_k$ as stated in the next definition.

Definition 138. *We denote by $\sigma := c_1, \ldots, c_k$ the definition of a valuation $\sigma \in Val^{Var}_{n \cdot \beta \cdot n}$, where the value $\sigma(x)$ of variable $x \in Var$ is defined as follows:*

- *For $x \in Counters(L_i)$: Let S be the decreasing dependence set with $x \in S$. We set $\sigma(x) = t_{\mathcal{L}}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_{\mathcal{L}}(x)$.*
- *For $x \in Const$: We set $\sigma(x) = t_{\mathcal{L}}(x) \cdot \beta \cdot n$.*

Lemma 139. *Given a valuation σ defined by $\sigma := c_1, \dots, c_k$ with $c_1, \dots, c_k \in [0, \beta[$ we have $\sigma(x) \in I^\beta_\mathcal{L}(x)$ for every program variable $x \in Var$.*

Proof. Let $x \in Counters(L_i)$ be a variable with $I^\beta_\mathcal{L}(x) = [l_\mathcal{L}(x), r_\mathcal{L}(x)[= [t_\mathcal{L}(x) \cdot \beta \cdot n, (t_\mathcal{L}(x) + 1) \cdot \beta \cdot n[$. We have $l = t_\mathcal{L}(x) \cdot \beta \cdot n \leq t_\mathcal{L}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_\mathcal{L}(x) \leq t_\mathcal{L}(x) \cdot \beta \cdot n + c_i \cdot n + (n-1) < t_\mathcal{L}(x) \cdot \beta \cdot n + (c_i + 1) \cdot n \leq t_\mathcal{L}(x) \cdot \beta \cdot n + \beta \cdot n = (t_\mathcal{L}(x) + 1) \cdot \beta \cdot n = r$ and therefore $\sigma(x) \in I^\beta_\mathcal{L}(x)$.

Let $x \in Const$ be a variable with $I^\beta_\mathcal{L}(x) = [l, r[= [t_\mathcal{L}(x) \cdot \beta \cdot n, (t_\mathcal{L}(x) + 1) \cdot \beta \cdot n[$. We have $\sigma(x) = t_\mathcal{L}(x) \cdot \beta \cdot n = l$ and therefore $\sigma(x) \in I^\beta_\mathcal{L}(x)$. □

Lemma 140. *Given a valuation σ defined by $\sigma := c_1, \dots, c_k$ with $c_1, \dots, c_k \in [0, \beta[$ we have $\sigma \models Inv(l)$.*

Proof. Let $x \rhd y \in Inv(l)$ be an order constraint. Either there is a dependence set S with $x, y \in S$ (a) or there are two dependence sets S_1, S_2 with $x \in S_1, y \in S_2$ and $S_1 \neq S_2$ (b).

(a) Assume there is a dependence set S with $x, y \in S$: As $x, y \in S$ we have $t_\mathcal{L}(x) = t^S_\mathcal{L} = t_\mathcal{L}(y)$. Either S is decreasing (aa) or S is non-decreasing (ab).

(aa) Assume S is decreasing. Thus we have $x \notin Const$. Let i be the index such that $x, y \in S \subseteq Counters(L_i)$. We set $c = t_\mathcal{L}(x) \cdot \beta \cdot n + c_i \cdot n$. We have $\sigma(x) = t_\mathcal{L}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_\mathcal{L}(x) = c + b^S_\mathcal{L}(x)$ and $\sigma(y) = t_\mathcal{L}(y) \cdot \beta \cdot n + c_i \cdot n + b^S_\mathcal{L}(y) = t_\mathcal{L}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_\mathcal{L}(y) = c + b^S_\mathcal{L}(y)$. Thus we can apply Lemma 133 and we get $\sigma \models x \rhd y$.

(ab) Assume S is non-decreasing: We have $x, y \in S \subseteq Const$. Thus $\sigma(x) = t_\mathcal{L}(x) \cdot \beta \cdot n = t_\mathcal{L}(y) \cdot \beta \cdot n = \sigma(y)$. Because S is constant we have $\rhd = \geq$. We have $\sigma \models x \rhd y$ because of $\sigma(x) \geq \sigma(y)$.

(b) Assume there are two dependence sets S_1, S_2 with $x \in S_1, y \in S_2$ and $S_1 \neq S_2$: By Lemma 139 we have that $\sigma(x) \in I^\beta_\mathcal{L}(x)$ for every program variable $x \in Var$. By Lemma 126 we have that $\sigma \models x \rhd y$. □

5.3.5 Lower Bounds from For-loops

In this subsection we introduce for-loop programs which will be constructed from the for-loops. For-loop programs are a tool for inferring runs and corresponding traces for SCSs from for-loops. These traces then enable us to establish lower bounds for SCSs. We obtain the result that if an SCS $\mathcal{A}$ has a for-loop $\mathcal{L}$ of degree k, then $\mathcal{A}$ is $\Omega(N^k)$.

The for-loop program has variables $c_1, \dots, c_k$ which are the iterators of the k nested loops of the for-loop program. Further the program has a variable z which counts the number of steps that the for-loop program executes. By a step, we understand the execution of one of the nested loops. At the beginning of the for-loop program z is initialized to 1.

At every step of the for-loop program we define an SCR T_z to be one of the loops L_i and we define a valuation σ_z with the help of the program variables, i.e., we set $\sigma_z := c_1, \dots, c_k$. We will later prove that $\sigma_1 \xrightarrow{T_1} \sigma_2 \xrightarrow{T_2} \cdots$ is a trace of $\mathcal{A}$.

Definition 141 (For-loop Program)**.** *The for-loop program of for-loop $\mathcal{L}$ is the program*

```
$z := 1,;$
for $(c_1 := decrease(\beta); c_1 > 0; save(L_1), c_1 := decrease(c_1))$
  for $(c_2 := decrease(\beta); c_2 > 0; save(L_2), c_2 := decrease(c_2))$
    $\vdots$
    for $(c_k := decrease(\beta); c_k > 0; save(L_k), c_k := decrease(c_k))$
```

where $save(L_i)$ is a macro for $\sigma_z := c_1, \dots, c_k, T_z := L_i, z{+}{+}$ and $decrease(\beta)$ is a nondeterministic function which takes an ordinal $\beta > 0$ and returns an arbitrary ordinal $\gamma < \beta$.

In the following proposition we state that the values of the variables c_i always stay in the interval $[0, \beta[$.

Proposition 142. *For every execution of the for-loop program we have the invariant $c_i \in [0, \beta[$ for all variables $1 \leq i \leq k$.*

From the above proposition we get the following lemma on the valuations.

Lemma 143. *For every SCR T_z and every two valuations σ_z, σ_{z+1} defined during an execution of the for-loop program we have $\sigma_z, \sigma'_{z+1} \models T_z$.*

Proof. Let T_z be an SCR and let σ_z, σ_{z+1} be two valuations defined during an execution of the for-loop program. Let L_i be the loop which is assigned to T_z. By the definition of the for-loop program we have $\sigma_z := c_1, \dots, c_k$ for some $c_1, \dots, c_k$ and $\sigma_{z+1} := c'_1, \dots, c'_k$ where $c'_1 := c_1, \dots, c'_{i-1} := c_{i-1}, c'_i := \mathit{decrease}(c_i), c_{i+1} := \mathit{decrease}(\beta), \dots, c_k := \mathit{decrease}(\beta)$ (*).

Let $x \rhd y' \in L_i$ be a size-decrease constraint. Either there is a dependence set S with $x, y \in S$ (a) or there are two dependence sets S_1, S_2 with $x \in S_1, y \in S_2$ and $S_1 \neq S_2$ (b).

(a) Assume that there is a dependence set S with $x, y \in S$. As $x, y \in S$ we have $t_{\mathcal{L}}(x) = t_{\mathcal{L}}(y)$. Either S is decreasing (aa) or S is non-decreasing (ab).

(aa) Assume S is decreasing. Thus we have $x \notin \mathit{Const}$. Let i be the index such that $x, y \in S \subseteq \mathit{Counters}(L_i)$. We have either $i = \mathit{act}(S)$ (aaa) or $i > \mathit{act}(S)$ (aab).

(aaa) Assume that we have $i = \mathit{act}(S)$. Therefore we have $c_i > c'_i$ by (*). $\sigma_z(x) = t_{\mathcal{L}}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_{\mathcal{L}}(x) \geq t_{\mathcal{L}}(x) \cdot \beta \cdot n + c_i \cdot n > t_{\mathcal{L}}(y) \cdot \beta \cdot n + c'_i \cdot n + (n-1) \geq t_{\mathcal{L}}(y) \cdot \beta \cdot n + c'_i \cdot n + b^S_{\mathcal{L}}(y) = \sigma_{z+1}(y)$.

(aab) Assume that we have $i > \mathit{act}(S)$. Therefore we have $c_i = c'_i$ by (*). We set $c = t_{\mathcal{L}}(x) \cdot \beta \cdot n + c_i \cdot n$. We have $\sigma(x) = t_{\mathcal{L}}(x) \cdot \beta \cdot n + c_i \cdot n + b^S_{\mathcal{L}}(x) = c +$

$b^S_\mathcal{L}(x)$ and $\sigma(y) = t_\mathcal{L}(y)\cdot\beta\cdot n + c'_i\cdot n + b^S_\mathcal{L}(y) = t_\mathcal{L}(x)\cdot\beta\cdot n + c_i\cdot n + b^S_\mathcal{L}(y) = c + b^S_\mathcal{L}(y)$. Thus we can apply Lemma 134 and we get $\sigma_z, \sigma'_{z+1} \models x \rhd y'$.

(ab) Assume S is non-decreasing: We have $x, y \in S \subseteq \mathit{Const}$. Thus $\sigma_z(x) = t_\mathcal{L}(x)\cdot\beta\cdot n = t_\mathcal{L}(y)\cdot\beta\cdot n = \sigma_{z+1}(y)$. Because S is constant we have $\rhd = \geq$. We have $\sigma_z, \sigma'_{z+1} \models x \rhd y'$ because of $\sigma_z(x) \geq \sigma'_{z+1}(y)$.

(b) Assume that there are two dependence sets S_1, S_2 with $x \in S_1, y \in S_2$ and $S_1 \neq S_2$: We have $\sigma_z, \sigma'_{z+1} \models x \rhd y'$ by Lemma 127. $\square$

Lemma 144. *Every for-loop program terminates.*

Proof. A for-loop program terminates by the standard argument for the termination of for-loops: The iterator c_i of nested loop i is not changed during the execution of the respective inner loops $i+1, \dots, k$ and loop i decreases its iterator c_i during its update. Loop i can only decrease its iterator finitely many times before it returns control to loop $i-1$. The for-loop terminates when the outermost loop returns the control.

A more formal view is that $(c_1, \dots, c_k)$ is a lexicographic ranking function for the for-loop program: The value of the ranking function decreases every time a loop does its update. Thus loop updates can happen only finitely often.

Therefore the for-loop program terminates on all executions, i.e., for all choices of *decrease*. $\square$

Theorem 145. *Let $\mathcal{A}$ be an SCS, let α, β be ordinals with $\alpha \geq n\cdot\beta\cdot n$, and let $\mathcal{L}$ be a proper for-loop with degree k. The length of the longest trace of $\mathcal{A}$ over α has the lower bound β^k.*

Proof. Let $(l; L_1, \dots, L_k)$ be the proper for-loop $\mathcal{L}$ of degree k. We fix some execution of the for-loop program. At program termination the variable z stores the number of steps the for-loop program has performed during the execution. Let T_i be the SCRs T_i defined during the execution of the for-loop program for $1 \leq l < z$ and let σ_i be the valuations defined during the execution of the for-loop program for $1 \leq i \leq z$. Let $\mathcal{R} = T_1 T_2 \cdots T_{z-1}$ be the run given by the SCRs T_i. We have $\sigma_z \models Inv(l)$ for all $1 \leq z \leq l$ by Proposition 142 and Lemma 140. We have $\sigma_i, \sigma'_{i+1} \models T_i$ for all $1 \leq i < z$ by Lemma 143. Thus $\sigma_1 \xrightarrow{T_1} \sigma_2 \xrightarrow{T_2} \cdots \xrightarrow{T_{l-1}} \sigma_l$ (*) is a trace of $\mathcal{A}$.

For every execution of the for-loop program we obtain a trace (*) of $\mathcal{A}$. This means that for all choices of *decrease* we obtain traces (*) of $\mathcal{A}$. Thus the length of the longest run of $\mathcal{A}$ has the lower bound β^k. □

5.3.6 Discussion of the Complexity of SCSs

Restricting α to the natural numbers in Theorem 145 gives us the following result:

Theorem 146. *Let $\mathcal{A}$ be an SCS, and let $\mathcal{L}$ be a proper for-loop of $\mathcal{A}$ with degree k. Then the length of the longest trace of $\mathcal{A}$ over $N \in \mathbb{N}$ is $\Omega(N^k)$.*

In [Ben-Amram, 2009b], Ben-Amram shows how to obtain lexicographic ranking functions for SCSs over the natural numbers. These lexicographic ranking functions have n components in the worst case. His development can be easily adjusted to obtain the following result:

Theorem 147 ([Ben-Amram, 2009b])**.** *Let $\mathcal{A}$ be a terminating SCS. Then the length of the longest trace of $\mathcal{A}$ over $N \in \mathbb{N}$ is $O(N^n)$.*

The importance of Theorem 146 is that for-loops provide convexity-witnesses for SCS. In light of Theorem 147, we conjecture that for-loops give rise to a complete characterization of the complexity of SCSs:

```
Procedure: Complexity($\mathcal{A}$)
Input: an SCS $\mathcal{A}$
Output: the asymptotic complexity of $\mathcal{A}$
forall the $l \in L^{\mathcal{A}}$ do
    forall the $l \xrightarrow{T} l \in \mathcal{A}^\star$ do
        if $T$ is idempotent and there is no $x \in Var$ with $x > x' \in T$
        then
            return "$\mathcal{A}$ does not terminate"
for $k$ from $n$ downto $0$ do
    forall the $l \in L^{\mathcal{A}}$ do
        forall the $l \xrightarrow{L_1} l, \ldots, l \xrightarrow{L_k} l \in \mathcal{A}^\star$ do
            if $(l; L_1, \ldots, L_k)$ is a proper for-loop then
                return "$\mathcal{A}$ has complexity $O(N^k)$"
```

Algorithm 9: `Complexity`($\mathcal{A}$) computes the asymptotic complexity of $\mathcal{A}$

Conjecture 148. *Let $\mathcal{A}$ be an SCS. Then $\mathcal{A}$ either does not terminate or the length of the longest trace of $\mathcal{A}$ over $N \in \mathbb{N}$ is $\Theta(N^n)$ and there is a proper for-loop of $\mathcal{A}$ with degree k.*

We mention that conjecture 148 gives rise to an algorithm for deciding the complexity of an SCS $\mathcal{A}$, which we state in Algorithm 9. Algorithm 9 first decides the termination of $\mathcal{A}$ by enumerating all SCRs in $\mathcal{A}^\star$ and checking the condition stated in Theorem 109. In case $\mathcal{A}$ is terminating, Algorithm 9 searches for proper for-loops starting with the highest degree n going down to 0. Algorithm 9 is in PSPACE because the elements of $\mathcal{A}^\star$ can be enumerated in PSPACE (for details see Ben-Amram [2011]).

Chapter 6

Related Work

In this section we give a detailed comparison with earlier termination and bound analyses. We show that our bound analysis captures the essential ideas of these approaches in a simpler framework and that our technique outperforms these recent approaches on loop-bound computation and termination analysis.

6.1 Bound Analysis by the `SPEED` project

[Gulwani et al., 2009c] would fail to compute bounds for Programs 1.3, 3.2, 3.3, 3.4, 1.7 because the invariants required for establishing bounds on the counters are disjunctive. (It can compute bounds for Programs 3.1 and 1.6.) The multiplicative counter instrumentation strategy of [Gulwani et al., 2009c] are meant to alleviate the problem of computing disjunctive invariants, but does not help for the listed examples because there the outer loop has only one back-edge and therefore only one counter can be instrumented.

[Gulwani et al., 2009a] likewise proposes to use program transformation before performing bound analysis. The program transformation of [Gulwani et al., 2009a] is parameterized by an abstract domain, which is used simultaneously with the actual transformation algorithm to detect the infeasibility of certain paths. However, [Gulwani et al., 2009a] is vague about what abstract domains should be used, and the actual transformation algorithm is quite involved. In contrast, we propose two simple program transformation that are easy to implement. Our program transformations only rely on SMT solvers and do not need specific abstract domains. [Gulwani et al., 2009a] would fail to compute bounds for Programs 1.3, 3.2, 3.3, 3.4 because the invariants required for establishing bounds on the counters are disjunctive. (It can compute bounds for Programs 3.1, 1.6 and 1.7.) The control-flow refinement strategy of [Gulwani et al., 2009a] is meant to alleviate the problem of computing disjunctive invariants, but does not help in any of these cases since the control-flow is already refined and cannot be refined any further.

[Gulavani and Gulwani, 2008] describes two operations for lifting conjunctive linear numerical abstract domains. On the one hand interesting non-linear expressions are identified by user annotations. These expressions and an approximation of their arithmetic properties are then added to the abstract domain. This enables the computation of non-linear invariants. On the other hand the abstract domain is extended using the maximum operator. The maximum operator is used to infer additional inequalities during the joins of the abstract domain that involve the maximum operator. In this way the maximum operator enables the computation of disjunctive bounds for transition systems with multiple transitions. However, the lifted abstract domain contains an extend set of expressions and operators but is still conjunctive. Therefore it fails to compute bounds for Programs 1.3, 3.2, 3.3, 3.4, 1.7 because the invariants required for establishing bounds on the counters are disjunctive. (It can compute bounds for Program 3.1 and 1.6.) However, the technique described in [Gulavani and Gulwani, 2008] can be used in a synergistic manner with our technique, in particular, as an extension to the pattern-matching based technique to compute ranking functions for single transitions.

We report on implementations of symbolic bound generation for .Net binaries and C programs, while [Gulwani et al., 2009c,a; Gulavani and Gulwani, 2008] implemented bound generation for C++ programs. Hence, we only provide analytical (not experimental) comparison with these techniques. Quite significantly, our implementation scales to large programs, while [Gulwani et al., 2009c,a; Gulavani and Gulwani, 2008] have been applied to only small benchmarks.

6.2 Termination Analysis by Ranking Functions and Transition Invariants

There is a large body of work on proving termination of programs. The standard approach consists of finding ranking functions [Turing, 1936]. Such ranking functions are generally complex, e.g. the standard lexicographic ranking functions. In many cases these ranking functions be seen as a composition of several local ranking functions, e.g. a lexicographic ranking function is composed of its components. Recently transition invariants [Podelski and Rybalchenko, 2004b] have been recognized as an alternative method for proving termination: Local ranking functions can be composed to a global termination argument even without constructing a global ranking function. One can obtain a bound from a lexicographic ranking function, if one can show that the domains of all its components are bounded, by multiplying the heights of the domains of the components. Likewise one can obtain a bound from a transition invariant, if one can show that the domains of all its local ranking functions are bounded, by multiplying the heights of the domains of the local ranking functions. Similarly to the above sketched methods we also use local ranking functions to compute bounds. However, whereas the above sketched methods for computing bounds only use the multiplication operator we compose local ranking functions using the plus or maximum operators if possible in order to yield precise symbolic bounds.

There is superficial similarity between transition invariants [Podelski and Rybalchenko, 2004b] and our transition system approach in that transition invariants also summarize relationships between two different visits to a control location and often require disjunctive invariants. However, there are two key technical differences: (a) Our technique requires computing relationships between two immediate visits to a control location, while transition invariants require computing relationships between *any* two visits to a control location. (b) Our technique requires use of disjunctive invariants only to summarize nested loops. In particular, for the example programs 1.6 and 1.7 with no nested loops, our technique would not require computing disjunctive invariants.

Our approach can be regarded as an alternative new technique for proving termination. For example, the recently proposed approaches [Berdine et al., 2007] and [Cook et al., 2006] (further discussed in Section 6.4) that both implement the transition invariant approach cannot prove termination of the loop in Program 1.7, while our technique can.

6.3 Comparison of transition predicate abstraction (TPA) and SCA by Heizmann et al.

In [Heizmann et al., 2010] Heizmann et al. state that SCA is an instance of the more general technique of TPA [Podelski and Rybalchenko, 2005]. In particular they formally show that when a tail-recursive functional program F is translated into an imperative program P, then an SCA-based termination analysis on F can be mimicked by a TPA-based termination analysis on P whose predicates are order relations.

However, [Heizmann et al., 2010] does not

- deal with general imperative programs, but with programs obtained as translations from functional programs. Since functional programs can be size-change abstracted more easily (as explained in our comparison with SCA), this problem setting is much simpler.
- show how to obtain transition predicates for the independent analysis of imperative programs by TPA. It only shows that (as a result of the translation) TPA is more general than SCA.
- deal with a concrete programming language, and does not deal with practical issues, or concrete analysis tools.
- make use of the recent progress of the SCA [Ben-Amram, 2011], where SCA is extended from natural numbers to integers, and deals only with natural numbers.

This thesis not only fills in all these gaps left open in [Heizmann et al., 2010], but also unifies much of the previous work, e.g., by Terminator in

Section 6.4, `SPEED` in Section 6.1 etc., and is drawing most of its motivation from these other papers.

While we find it quite intuitive that SCA as well as our more general approach are instances of TPA, we are concerned with a different issue in this thesis. We argue that precisely because of its limited expressiveness SCA is suitable for bound analysis: abstracted programs are simple enough such that we can compute bounds for them. We have shown that imperative programs are amenable to bound analysis by SCA using appropriate program transformations, whereas [Heizmann et al., 2010] is a theoretical paper.

6.4 Termination Analysis by Terminator

The Terminator tool [Cook et al., 2006] is an automatic termination analyzer of imperative programs, which uses TPA [Podelski and Rybalchenko, 2005] for constructing a Ramsey based termination argument [Podelski and Rybalchenko, 2004b].

Our approach and Terminator share the idea of extracting progress measures locally (norms resp. local ranking functions) and composing them for a global analysis (bound resp. termination proof). Because of its non-constructive nature, the Ramsey based termination argument underlying TPA cannot be used for extracting a global ranking function out of the termination proof. In contrast, we use SCA for the first time to compose global bounds from bounds on norms. Earlier work on SCA [Ben-Amram, 2011] already has shown how to compute global ranking functions from norms.

In order to apply the Ramsey based termination argument, Terminator needs to analyze the transitive hull of programs. This analysis is the most expensive step in the analysis of Terminator and it has to be repeated again and again. In contrast, we abstract programs first and then analyze only the transitive hull of the abstract program. This has huge benefits for the speed of the analysis (further discussed in Subsection 6.5).

TPA can lose precision in every step of the analysis. In contrast, our pathwise analysis follows the structure of programs and loses precision only at well-defined places. Our analysis handles paths precisely by conjoining the formulae of the statements along the path (using all theories that can be handled by SMT solvers) and loses precision when handling loops. However, we handle loops precisely w.r.t. their monotonic behavior of norms: SCA is closed under taking transitive hulls because of the built-in disjunctiveness of SCA and the transitivity of the order relations, and these transitive hulls can be computed effectively.

Terminator is built on top of a full-fledged software model checker, which implements a complicated CEGAR loop in order to extract predicates and local ranking functions from programs. In contrast, our simple and lightweight static analysis relies only on an SMT solver, our set of transition predicates is fixed in advance (the monotonicity predicates of SCA) and our set of norms is extracted from program at the beginning of the analysis. It is an interesting direction of future work to investigate how to combine these approaches, e.g., by using our approach for filtering the "easy cases" and using a CEGAR like approach for coping with the "hard cases".

6.5 Termination Analysis byLoopfrog

[Kroening et al., 2010, 2011] observe that TPA-based approaches such as Terminator [Cook et al., 2006] spend almost all time in analyzing the transitive hulls of programs, i.e., the expensive step is proving $P|_l^+ \subseteq \bigcup \mathcal{T}$

for transition sets $\mathcal{T}$. Therefore [Kroening et al., 2010, 2011] take a different approach and give algorithms that search for a transitive transition system $\mathcal{T}$ for $P|_l$. A transition set $\mathcal{T}$ is *transitive*, if $\bigcup \mathcal{T}^2 \subseteq \bigcup \mathcal{T}$. A transitive transition system $\mathcal{T}$ for $P|_l$ already implies $P|_l^+ \subseteq \bigcup \mathcal{T}^+ \subseteq \bigcup \mathcal{T}$ by the transitivity of $\mathcal{T}$. This has the advantage that the expensive direct proof of $P|_l^+ \subseteq \bigcup \mathcal{T}$ is avoided.

The first version of LOOPFROG [Kroening et al., 2010] implements an algorithm that constructs such transitive transition systems iteratively. In every step LOOPFROG adds transition relations to a candidate transition set $\mathcal{T}$. We argue that the effectiveness of such an iterative algorithm is limited, and that what the authors of [Kroening et al., 2010] really want is SCA!

Note that a transitive transition system $\mathcal{T}$ for $P|_l$ that is precise enough to prove the termination of P is an overapproximation of P that still terminates. Let us consider an example transition set $\mathcal{T} = \{\rho_1, \rho_2\}$ with $\rho_1 = x > 0 \wedge x > x'$ and $\rho_2 = y > 0 \wedge y > y'$. $\mathcal{T}$ does not terminate because ρ_1 can increase the value of y arbitrarily and ρ_2 can increase the value of x arbitrarily. Let us assume that the analyzed program P nevertheless terminates because the variable x can only be decreased when y stays constant. Let us further assume that LOOPFROG has added ρ_1 to $\mathcal{T}$ in the first step and ρ_2 in the second step of its iteration. LOOPFROG could have added the information about x in the first step by setting $\rho_1 = x > 0 \wedge x > x' \wedge y = y'$, but not in the second step. Note that once the candidate transition set $\mathcal{T}$ does not terminate, it cannot be repaired by adding transition relations. Note further that in later steps the loss of information of earlier steps cannot be repaired as we have seen on the above example. Thus, we conclude that a candidate transition set $\mathcal{T}$ has to be constructed in one single step. This is exactly what we do in our analysis. We first compute transition system for programs, and then size-change abstract the transition relations for our bound analysis. Alternatively, these abstracted transition relations could be analyzed for termination by a transitive hull computation using the termination criterion of SCA [Ben-Amram, 2011]. This transitive hull then provides exactly the transitive transition system $\mathcal{T}$ for $P|_l$. From this we conclude that what the authors of [Kroening et al., 2010] really want is SCA!

The second version of LOOPFROG [Kroening et al., 2011] uses relational loop summarization (see also the next subsection). For this summarization [Kroening et al., 2011] uses template invariants. Only one of these templates contains disjunction (two disjuncts). [Kroening et al., 2011] states that these templates are inspired by the more general size-change abstract domain. We show in this thesis how to employ the full SCA domain by using pathwise analysis for exploiting the loop structure of imperative programs. This allows us to use the full disjunctive power of SCA. [Kroening et al., 2011] is only concerned with termination analysis, whereas we show how to use SCA for the more difficult problem of bound analysis.

6.6 Loop Summarization

Loop summarization as in Algorithm 1 is being recognized as important tool in program analysis, for example [Kroening et al., 2008] summarizes loops by overapproximations of the reachable states for automatic proofs of safety properties. Relational summarizations of loops have for the first time been used in the bound analysis of [Gulwani and Zuleger, 2010]. The termination analysis [Kroening et al., 2011], which is an extension of [Kroening et al., 2008], also uses relational summaries of loops.

Loop summarization is closely related to procedure summarization, e.g. [Gulwani and Tiwari, 2007].

6.7 Disjunctive Invariant Generation

Classical domains in abstract interpretation are normally good at inferring conjunctive invariants, and various domain exists in the precision/cost spectrum (like intervals [Cousot and Cousot, 1976], octagons [Miné, 2006], and polyhedra [Cousot and Halbwachs, 1978]). Because program verification is in need of discovering disjunctive invariants methods it has been proposed to lift classical conjunctive abstract domains to powerset domains [Cousot and Cousot, 1979]. Lifting an abstract domain to the powerset domain requires an adequate lifting of its join operator. Set union would be the precise join operator of the powerset lattice, but every set union increases the number of the base elements of the powerset elements. However, the powerset lattices of many classical abstract domains (like intervals, octagons or polyhedra) have infinite width. Thus practical implementations needs to limit the number of base elements of a powerset element in order to ensure the finiteness of the analysis. [Handjieva and Tzolovski, 1998; Popeea and Chin, 2006; Gopan and Reps, 2006, 2007; Rival and Mauborgne, 2007] address this issue by proposing various semantic-merging heuristics. In contrast, we have presented in Section 3.1 a result that calls for working with a static syntactic merge criterion under the convexity-like assumption (which appears to be satisfied by the benchmark examples).

Some syntactic techniques based on program restriction [Beyer et al., 2007] or control-flow refinement [Gulwani et al., 2009a] have also been suggested for discovering disjunctive invariants. By choosing a suitable convexity-witness σ these techniques can be viewed as instantiations of our more general framework that we described in Section 3.1.

In [Monniaux, 2009] Monniaux proposes calculating most precise abstract transformers. He requires the provision of a template which can be an arbitrary boolean combination of linear inequalities. His method relies on quantifier elimination over the reals. As quantifier elimination has a prohibitive complexity our approach is more scalable.

In recent work on termination analysis [Berdine et al., 2007] it is proposed to use a powerset abstract domain for computing disjunctive transition invariants. In the implementation section of [Berdine et al., 2007] it is suggested to use a lifting of the octagon / polyhedra domain as this powerset abstract domain. However, [Berdine et al., 2007] remains vague on how this lifting is implemented: "For our present empirical evaluation we use an extraction method after the fixed-point analysis has been performed in order to find disjunctive invariance/variance assertions.".

In contract, SCA is a finite powerset domain that naturally handles disjunction: SCA has finite width, therefore we never have to merge abstract elements and can handle disjunction precisely. This releases us from relying on complicated merging algorithms as [Gulwani and Zuleger, 2010; Berdine et al., 2007]. SCA has finite height, therefore we do not need widening to compute fixed points (e.g. transitive hulls). This releases us from lifting the widening operator of conjunctive domains (e.g. octagon, polyhedra) to powerset domains.

6.8 Size-change Abstraction

Despite its success in functional/declarative languages, e.g. [Manolios and Vroon, 2006], [Krauss, 2007], SCA [Lee et al., 2001; Ben-Amram, 2011] has not yet been applied to imperative programs. We describe the two main obstacles in the application of SCA to imperative programs and how we solve them:

In functional / declarative languages, algorithms typically operate on algebraic data structures where constructs and destructs happen in single steps. Due to this succinctness, SCA achieves sufficient precision on small program blocks. In imperative programs loops can have many intermediate stages and oftentimes only the program state at the loop header can be considered as “clean”. Therefore the abstraction of small program blocks to size-change relations loses too much precision. (This issue is well-known in the field of invariant computation.) We solve this issue by our pathwise analysis, which has the effect that large pieces of code that lie between the “clean” program locations are abstracted jointly.

The intended use of the SCA variables is as *local progress measures of the program.* In functional/declarative languages there is a natural set of such local progress measures such as the size of a data type, the height of a tree, the length of a list, or any arithmetic expression built up from those. In imperative programs, it is less clear what the shape of this local progress measures is and how they can be automatically extracted from programs. We give a solution to this problem by extracting norms from the conditions of complete loop paths (as described in our heuristics).

6.9 Other Approaches

[Albert et al., 2008] computes bounds by generating recurrence relations and then deriving a closed form expression for the maximum size of the unfoldings of the (possibly nondeterministic) recurrence relations into trees. Since their method does not precisely summarize inner loops, they cannot handle loops where the inner loop changes the iterators of the outer loop as in Programs 3.1, 3.2, 3.3, 3.4. Also, they can't handle Programs 1.6, 1.7 and are unable to compute the amortized bounds as for Program 1.3. We report on implementations of symbolic bound generation for .Net binaries and C programs, while [Albert et al., 2008] implemented bound generation for Java programs respectively. Hence, we only provide analytical (not experimental) comparison with these techniques. Quite significantly, our implementation scales to large programs, while [Albert et al., 2008] has been applied to only small benchmarks.

A series of works describes a type-based potential-method of amortized analysis for the estimation of resource usage in first-order functional programs, which reduces the problem to linear constraint solving. Recent enhancement includes the extension to multivariate polynomial bounds [Hoffmann et al., 2011] and higher-order programs [Jost et al., 2010].

[Crary and Weirich, 2000] presents a type system for the certification of resource bounds (once they are provided by the programmer), but does not infer bounds.

The embedded and real-time systems community has taken considerable effort on worst case execution time (WCET) estimation [Wilhelm et al., 2008]. WCET research is largely orthogonal, focused on distinguishing between the complexity of different code-paths and low-level modeling of architectural features such as caches, branch prediction, instruction pipelines. For establishing loop bounds, WCET techniques either require user annotation, or use simple techniques based on pattern matching or numerical analysis. These WCET techniques cannot compute bounds for most of the examples considered in this thesis. We report on the Mälardalen WCET [WCETWeb-

Page, 2010] benchmark in our experimental section.

[Goldsmith et al., 2007] computes symbolic bounds by curve-fitting timing data obtained from profiling. Their technique has the advantage of measuring real time in seconds for a representative workload. However, their technique does not provide worst-case bounds and their results are not sound for all inputs.

Chapter 7

Conclusion

In this thesis we have defined and motivated the reachability-bound problem. We have proposed a two-step methodology for computing reachability-bounds and have presented two approaches that implement this methodology. We have described an algorithm, which is used by both approaches, that computes transition systems based on the program transformation pathwise analysis. Our first approach uses two different techniques for reasoning about loops, namely an iterative technique for computing transitive hulls based on abstract interpretation, and a non-iterative proof-rule based technique for computing bounds of transition systems. We have discussed the limitations of our first approach, and motivated our second approach. Our second approach is based on the size-change abstraction (SCA). We summarize loops using the property of SCA to be closed under the computation of transitive hulls. We compute bounds using a separation of concerns offered by SCA: we compose locally extracted norms to a global bound based only on the size-change abstracted program. We have stated two program transformations (pathwise analysis and contextualization) that make SCA amenable to the bound analysis of imperative programs. We have presented results towards a characterization of the bounds expressible by SCA. In particular, we have defined complexity witnesses (for-loops) that establish lower bounds of abstract programs. In a qualitative comparison we have showed that our solution to the reachability-bound problem captures the essential ideas of earlier termination and bound analyses in a simpler framework and outperforms these analyses in computing loop-bounds and proving termination.

We point out several directions for future work:

Recursive Procedures and Concurrent Programs. Currently, our analysis is limited to sequential programs without procedures. The next technical challenges are to address the reachability-bound problem in context of recursive procedures and concurrent execution.

Enhancing the Bound Analysis based on SCA. We believe that SCA is the right abstract domain for the bound analysis of imperative programs. Our solution presents a first choice in the solution space offered by this powerful abstraction. Further investigations are worthwhile, for example, our bound algorithm handles many programs occurring in practice, but does not yet exploit the full strength of SCA.

Applications. We plan to integrate the proposed solution to the reachability-bound problem with other specific techniques to provide an integrated solution for several applications:

- Computing worst-case bounds for embedded systems.
- Identifying performance bottlenecks in standard code that is caused by inefficient calls to library functions.
- Proving bounded liveness properties of protocols.

- Controlling program complexity by introducing appropriate type system into programming languages.

Lower- and Average-case Bounds. There are two interesting problem extensions that we leave for future work:

(a) Establishing precision of the generated bounds by identifying a symbolic precision-witness.

(b) Generating average-case bounds. For example, [Goldsmith et al., 2007] computes symbolic bounds by curve-fitting timing data obtained from profiling. Their technique has the advantage of measuring real amortized complexity; however the results are not sound for all inputs.

Separated Research Communities. Most existing research on the termination/bound problem has been divided into separate lines for the imperative and functional/declarative world.Our work for the first time has bridged this gap by using SCA – developed for functional languages and applied to functional/declarative programs – in an imperative setting. We hope that our work will foster the transfer of ideas and techniques between these areas. We have given a first example of such a cross-fertilization by using the program transformation contextualization, which was originally developed for functional programs [Manolios and Vroon, 2006].

Bibliography

Albert, E., Arenas, P., Genaim, S., and Puebla, G. (2008). Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*.

Ben-Amram, A. (Mar. 2011). Monotonicity constraints for termination in the integer domain. Technical report.

Ben-Amram, A. M. (2009a). A complexity tradeoff in ranking-function termination proofs. *Acta Inf.*, 46(1):57–72.

Ben-Amram, A. M. (2009b). Size-change termination, monotonicity constraints and ranking functions. In *CAV*, pages 109–123.

Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P. W. (2007). Variance analyses from invariance analyses. In *POPL*, pages 211–224.

Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., and Sebastiani, R. (2009). Software model checking via large-block encoding. In *FMCAD*, pages 25–32.

Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. (2007). Path invariants. In *PLDI*, pages 300–309.

Bradley, A., Manna, Z., and Sipma, H. (2005). Termination of polynomial programs. In *VMCAI*.

CBenchWebPage (2010). `http://ctuning.org/wiki/index.php/CTools:CBench`.

Codish, M., Fuhs, C., Giesl, J., and Schneider-Kamp, P. (2010). Lazy abstraction for size-change termination. In *LPAR (Yogyakarta)*, pages 217–232.

Codish, M., Lagoon, V., and Stuckey, P. J. (2005). Testing for termination with monotonicity constraints. In *ICLP*, pages 326–340.

Colby, C. and Lee, P. (1996). Trace-based program analysis. In *POPL*, pages 195–207.

Cook, B., Podelski, A., and Rybalchenko, A. (2006). Termination proofs for systems code. In *PLDI*, pages 415–426.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Second International Symposium on Programming*, pages 106–130.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252.

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *POPL*, pages 269–282.

Cousot, P. and Halbwachs, N. (1978). Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL*, pages 84–96.

Crary, K. and Weirich, S. (2000). Resource bound certification. In *POPL '00*.

Dutertre, B. and de Moura, L. (2006). The yices smt solver. Technical report.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345.

Goldsmith, S., Aiken, A., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, pages 395–404.

Gopan, D. and Reps, T. W. (2006). Lookahead widening. In *CAV*.

Gopan, D. and Reps, T. W. (2007). Guided static analysis. In *SAS*.

Gulavani, B. S. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384.

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

Gulwani, S., Lev-Ami, T., and Sagiv, M. (2009b). A combination framework for tracking partition sizes. In *POPL*, pages 239–251.

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

Gulwani, S. and Tiwari, A. (2007). Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267.

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

Handjieva, M. and Tzolovski, S. (1998). Refining static analyses by trace-based partitioning using control flow. In *SAS*, pages 200–214.

Heizmann, M., Jones, N. D., and Podelski, A. (2010). Size-change termination and transition invariants. In *SAS*, pages 22–50.

Henzinger, T. (2009). From boolean to quantitative system specifications, keynote. In *Ist Workshop on Quantitative Analysis of Software. http://research.microsoft.com/users/sumitg/qa09/keynote.pdf.*

Hoffmann, J., Aehlig, K., and Hofmann, M. (2011). Multivariate amortized resource analysis. In *POPL*, pages 357–370.

Jost, S., Hammond, K., Loidl, H.-W., and Hofmann, M. (2010). Static determination of quantitative resource usage for higher-order programs. In *POPL*, pages 223–236.

Krauss, A. (2007). Certified size-change termination. In *CADE*, pages 460–475.

Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. M. (2008). Loop summarization using abstract transformers. In *ATVA*, pages 111–125.

Kroening, D., Sharygina, N., Tsitovich, A., and Wintersteiger, C. M. (2010). Termination analysis with compositional transition invariants. In *CAV*, pages 89–103.

Kroening, D., Sharygina, N., Tsitovich, A., and Wintersteiger, C. M. (2011). Loop summarization and termination analysis. In *TACAS*, page unkown.

Kupferman, O. and Vardi, M. Y. (2001). Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA. IEEE Computer Society.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. (2001). The size-change principle for program termination. In *POPL*, pages 81–92.

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

Malacaria, P. (2007). Assessing security threats of looping constructs. In *POPL*, pages 225–235.

Manolios, P. and Vroon, D. (2006). Termination analysis with calling context graphs. In *CAV*, pages 401–414.

Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100.

Monniaux, D. (2009). Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

PhoenixWebPage (2009). Microsoft Phoenix Compiler. `http://research.microsoft.com/phoenix/`.

Podelski, A. and Rybalchenko, A. (2004a). A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251.

Podelski, A. and Rybalchenko, A. (2004b). Transition invariants. In *LICS*, pages 32–41.

Podelski, A. and Rybalchenko, A. (2005). Transition predicate abstraction and fair termination. In *POPL*, pages 132–144.

Popeea, C. and Chin, W.-N. (2006). Inferring disjunctive postconditions. In *ASIAN*, pages 331–345.

Rival, X. and Mauborgne, L. (2007). The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5).

Turing, A. M. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265.

Warren, H. S. (2002). *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

WCETWebPage (2010). `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D. B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P. P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3).

Z3WebPage (2009). Z3 Theorem Prover. `http://research.microsoft.com/projects/Z3/`.

Curriculum Vitae

Skodagasse 3/7
1080 Wien
Austria
zuleger@forsyte.at
`www.forsyte.at/~zuleger`

Florian Zuleger

EDUCATION

Technische Universität München Munich, Germany

October 2004 – October 2008

Diploma in mathematics (major) and computer science (minor) with high distinction. Thesis title: Partial Decision Methods for the Halting Problem. Thesis supervisor: Prof. Helmut Veith.

Gymnasium Ottobrunn Ottobrunn, Germany

September 1994 – June 2003

PUBLICATIONS

F. Zuleger, S. Gulwani, M. Sinn and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction *submitted*, 2011.

M. Sinn and F. Zuleger. LOOPUS - A Tool for Computing Loop Bounds for C Programs. In *Proc. WING*, *to appear*, 2010.

S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Proc. PLDI*, pages 292–304, 2010.

J. Kinder, F. Zuleger and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proc. VMCAI*, pages 214–228, 2009.

TALKS

LOOPUS - A Tool for Computing Loop Bounds for C Programs, 3rd Workshop on Invariant Generation (WING), 2010

The Reachability-Bound Problem, Programming Language Design and Implementation (PLDI), 2010

The Reachability-Bound Problem, 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS) / Colloquium on Programming Languages and Principles of Computation, 2009

Systematic Disassembly based on Abstract Interpretation, Disputation / Disputationprüfung TopMath, 2008

Partial Decision Methods for the Halting Problem, Colloquium / Kolloquium TopMath, 2007

Termination of Goodstein Sequences, Application / Bewerbungsvortrag TopMath, 2006

Interactive Protocols, Presentation JASS, 2006

Awards	*Microsoft Research PhD Scholarship*, since 2007 *Max-Weber Programm* of the state Bavaria, 2007–2009 *Erfahrene Wege in die Forschung* of TU München, since 2007 *TopMath* fast-track promotional program, since 2007 Several prizes in German mathematical competitions, 1998–2003
Professional Activities	**Journal Referee** JSC 2011; STTT 2010; IPL 2010. **Conference Referee** DATE 2011; VMCAI 2011; POPL 2011; APLAS 2010; LPAR 2010; RTSS 2010; CSR 2010; FMCAD 2010; ATVA 2010; WING 2010; ICTAC 2010; CAV 2010; ESOP 2010; POPL 2010; VMCAI 2010; FMCAD 2009; CAV 2009; WING 2009. **Supervision** of the diploma thesis of Moritz Sinn at TU Darmstadt. **Microsoft Research** Redmond, USA **March 2009 – May 2009** Supervised by Sumit Gulwani **Technische Universität Wien, FORSYTE** *Research assistant* Vienna, Austria **since February 2010** Ph.D. student in the group of Prof. Helmut Veith.

Technische Universität Darmstadt, FG FORSYTE

Research assistant

Darmstadt, Germany **April 2008 – January 2010**

Ph.D. student in the group of Prof. Helmut Veith.

Technische Universität München, I7 *Research assistant*

Munich, Germany **October 2007 – March 2008**

Ph.D. student in the group of Prof. Helmut Veith.