FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automated Complexity Analysis for Imperative Programs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Moritz Sinn, Msc.
Registration Number 1128872

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ass.Prof. Dipl.-Math. Dr. Florian Zuleger
Second advisor: Univ.Prof. Dipl.-Ing. Dr. Helmut Veith

The dissertation has been reviewed by:

Florian Zuleger

Tomáš Vojnar

Vienna, 25th August, 2016

Moritz Sinn

Erklärung zur Verfassung der Arbeit

Moritz Sinn, Msc.
Arsenal 1/29b, 1030 Wien

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

Wien, 25. August 2016

Moritz Sinn

Danksagung

Zuvorderst möchte ich Florian Zuleger für die ausgezeichnete und angenehme Zusammenarbeit danken. Florian hat mir die Befassung mit dem spannenden Thema der vorliegenden Dissertation ermöglicht und meine wissenschaftliche Tätigkeit immer hervorragend betreut.

Und ich möchte an Helmut Veith erinnern. Helmut verstarb Anfang dieses Jahres plötzlich, unerwartet und viel zu früh. Selbst mit größter Leidenschaft als Wissenschaftler tätig, verstand Helmut es wie kaum ein anderer, Begeisterung für die Informatik zu vermitteln. Seine hervorragenden Vorlesungen in Darmstadt haben mich denn auch zur Mitarbeit in seiner Gruppe motiviert. Ihm habe ich es zu verdanken, dass mich mein Weg in die Wissenschaft und nach Wien führte. Helmut stand immer mit Rat und Tat zur Seite und wusste sowohl bei wissenschaftlichen als auch alltäglichen Problemen zu helfen.

Ich danke Fabian Souczek, Clemens Danninger, Bernhard Gleiss und Raphael Mader für die aktive Mitarbeit an der Entwicklung unserer Implementierung “loopus”. Ich danke Thomas Pani für seine Mithilfe bei der Durchführung der Experimente.

Ich danke meinem Kollegen Ivan Radicek für die nette Gesellschaft die er mir während unserer mehrjährigen Bürogemeinschaft bot.

Weiters danke ich allen Mitgliedern der Forsyte-Gruppe an der TU Wien für viele ergiebige und inspirierende Diskussionen sowie für die angenehme Arbeitsatmosphäre.

Kurzfassung

Unsere Arbeit widmet sich dem Problem der *automatisierten Komplexitätsanalyse*, welches alternativ auch als Problem der *automatisierten Abschätzung des Ressourcenverbrauchs* formuliert wird. Eine Lösung dieses Problems wird angestrebt durch die Entwicklung einer *Quellcodeanalyse* (im Folgenden Bound-Analyse genannt) zur Abschätzung der Ausführungskosten eines gegebenen Programms. Der Begriff *Kosten* wird, wie üblich, durch ein *Kostenmodell* definiert, welches jeder Programminstruktion eine Ausführungskost zuweist. Je nach Anwendungsgebiet können Ausführungskosten beispielsweise in Form der benötigten Zeit, der benötigten Energie oder der Anzahl auszuführender Operationen definiert werden. Unsere Bound-Analyse behandelt das gegebene Programm als ein mathematisches Objekt und schätzt die Ausführungskosten bzw. den Ressourcenverbrauch des Programms mittels automatisierter Anwendung mathematischer und logischer Methoden ab, ohne das Programm auszuführen. Eine gewonnene Abschätzung wird in Form eines symbolischen Ausdrucks über den Programmparametern angegeben.

Wir denken, dass Bound-Analysen in den verschiedensten Bereichen von großem Nutzen sein können. In unserer Arbeit diskutieren wir Anwendungsszenarien in den Bereichen *Software Profiling*, *Program Understanding*, *Program Verification*, *Software Security* und *Automatic Parallelization*.

In den letzten Jahren wurden großen Fortschritte im Bereich der Bound-Analysen erzielt. Gegenwärtige Bound-Analysen können die Ausführungskosten beeindruckend komplizierter Code-Beispiele vollautomatisch deduzieren. Dennoch sehen wir zwei maßgebliche Nachteile aktueller Ansätze im Bereich der Bound-Analysen: (1) Die existierenden Analysen skalieren nicht ausreichend, um den Quellcode ganzer Programme, welcher oft Tausende oder Hunderttausende Code-Zeilen umfasst, zu analysieren. (2) Obwohl gegenwärtige Ansätze hinreichend genaue Abschätzungen des Ressourcenverbrauchs vieler und vor allem diffiziler Code-Beispiele berechnen können, werden dennoch die Kosten mancher, durchaus natürlicher Schleifeniterationsschemata nur sehr unzuverlässig und ungenau ermittelt.

Mit unserer Arbeit wollen wir dazu beitragen, beide Probleme zu überwinden:

(1) Die Bound-Analyse, welche wir in vorliegender Arbeit präsentieren, geht das Problem mangelnder Skalierbarkeit mittels einfacher statischer Analyse an: Während existierende Bound-Analysen mächtige Werkzeuge wie z.B. Abstract Interpretation, Computeralgebra

oder lineare Optimierung verwenden, setzt unsere Analyse auf eine Reduktion des Problems mittels Programmabstraktion: In einem ersten Schritt abstrahieren wir das gegebene Programm in ein stark vereinfachtes Programmmodell. Anschließend wenden wir unseren Algorithmus für die Berechnung der Ausführungskosten auf dem vereinfachten Programm an.

(2) Unsere Analyse erweitert die Möglichkeiten der Bound-Analyse: Wir erhalten asymptotisch präzise Abschätzungen der Kosten bzw. des Ressourcenverbrauchs für Instanzen einer Klasse von Schleifeniterationsschemata, für welche existierende Analysen meist fehlschlagen oder nur grobe Abschätzungen deduzieren können. Instanzen dieser Iterationsschemata werden häufig in Parser-Implementierungen sowie in String-Matching-Routinen verwendet. Wir diskutieren mehrere Beispiele in unserer Arbeit. Darüber hinaus ist unsere Analyse in der Lage, den überwiegenden Teil der in der Literatur diskutierten Bound-Analyse-Probleme zu lösen.

Unsere Bound-Analyse basiert auf dem abstrakten Programmmodell der *Difference Constraints*. Difference Constraints wurden für die Terminationsanalyse eingeführt und bezeichnen relationale Ungleichungen der Form $x' \leq y + c$. Eine solche Ungleichung drückt aus, dass der Wert von x im gegenwärtigen Zustand höchstens so groß ist wie der Wert von y im vorherigen Zustand, erhöht um eine Konstante $c \in \mathbb{Z}$. Wir denken, dass Difference Constraints, neben ihrer Anwendung in der Terminationsanalyse, auch eine gute Wahl für die Komplexitätsanalyse imperativer Programme darstellen, da die Komplexität solcher Programme meist aus dem Zusammenspiel von Zählerinkrementen bzw. -dekrementen der Form $x := x + c$ und Zähler-Resets der Form $x := y$ (wobei $x \neq y$) resultiert, diese Operationen können mittels der Difference Constraints $x' \leq x + c$ und $x' \leq y$ modelliert werden.

Unsere Arbeit trägt zur Entwicklung von Bound-Analyse-Techniken wesentlich bei durch:
(1) Effiziente Abstraktionstechniken: Wir zeigen, das Difference Constraints ein adäquates abstraktes Programmmodell für die Bound-Analyse bilden.
(2) Einen neuen Bound-Analyse-Algorithmus: Wir definieren und diskutieren einen Algorithmus, welcher den Bereich der Bound-Analysen auf eine Klasse schwer zu analysierender, aber natürlich vorkommender Schleifeniterationsschemata ausweitet. Wir beweisen die Korrektheit unseres Algorithmus.
(3) Eine skalierende Bound-Analyse: Unsere Analyse skaliert wesentlich besser als existierende Bound-Analysen.

Unser Beitrag wird durch einen sorgfältigen experimentellen Vergleich auf Drei verschiedenen Benchmarks unterstützt: Wir vergleichen unsere Implementierung mit gegenwärtigen Bound-Analyse-Tools auf (a) einer großen Benchmark, bestehend aus C-Code, (b) einer Benchmark, welche Code-Beispiele aus der Literatur enthält, und (c) einer Benchmark, welche schwierige Schleifeniterationsschemata beinhaltet, die wir aus dem Quellcode echter Programme extrahiert haben.

Wie die meisten existierenden Ansätze im Bereich der Bound-Analysen zielt unsere Analyse auf eine *obere* Abschätzung der Ausführungskosten.

Abstract

Our work contributes to the field of *automated complexity and resource bound analysis* (bound analysis) that develops source code analyses for estimating the *cost* of running a given piece of software. The term *cost* is usually defined by a *cost model* which assigns an execution cost to each program instruction or operation. Depending on the application domain, the cost is estimated, e.g., in terms of consumed time, consumed power, or the number of executed statements. A bound analysis treats the program under scrutiny as a mathematical object, inferring a bound on the execution cost (with respect to the given cost model) by means of automated formal reasoning. The computed bound is usually expressed by a symbolic expression over the program's parameters.

We argue that bound analysis could be applied with great benefits, e.g., in the areas of *software profiling*, *program understanding*, *program verification*, *software security*, and *automatic parallelization*. We state several application scenarios in this thesis.

In recent years, the research on bound analysis has made great progress. State-of-the-art bound analysis techniques can automatically infer execution costs for impressively complicated code examples. Nevertheless, we see two main drawbacks of current bound analysis techniques: (1) Present approaches do not scale sufficiently for analyzing real code, which often consist of thousands or hundreds of thousands of lines of code with many nested conditionals. (2) Though existing approaches can infer *tight* bounds for many challenging examples, they nevertheless grossly over-approximate the cost of certain code patterns that are common in real source code.

With our work we aim towards overcoming both problems:

(1) The bound analysis we present in this work tackles the scalability problem by *simple static analysis*: In contrast to existing bound analysis techniques, which employ general purpose reasoners such as abstract interpreters, computer algebra tools or linear optimizers, we take an orthogonal approach based on the well-known *program abstraction* methodology: We first abstract a given program into an abstract program model by means of static analysis. We then apply our bound algorithm on the abstracted program, which is a simplified version of the original program.

(2) Our bound algorithm extends the range of bound analysis. It infers tight bounds for a class of loop iteration patterns on which existing approaches fail or infer bounds that are not tight. Instances of such loop iterations can often be found in parsing and

string matching routines. We state several examples in this work. At the same time our approach is general and can handle most of the bound analysis problems which are discussed in the literature.

Our bound analysis is based on the *abstract program model* of *difference constraints*. Difference constraints have been used for termination analysis in the literature, where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of x in the current state is at most the value of y in the previous state plus some constant $c \in \mathbb{Z}$. Intuitively, difference constraints are also a good choice for complexity and resource bound analysis because the complexity of imperative programs typically arises from counter increments resp. decrements of form $x := x + c$ and resets of form $x := y$ (where $x \neq y$), which can be modeled naturally by the difference constraints $x' \leq x + c$ resp. $x' \leq y + 0$.

Our work contributes to the field of automated complexity and resource bound analysis by:
(1) Providing efficient abstraction techniques and demonstrating that *difference constraints* are a suitable abstract program model for automatic complexity and resource bound analysis.
(2) Providing a new, soundness proven bound algorithm which extends the range of bound analysis to a class of challenging but natural loop iteration patterns that can be found in real source code.
(3) Presenting a bound analysis technique which is more scalable than existing approaches.

Our contributions are supported by a thorough experimental comparison on three benchmarks: We compare our implementation to other state-of-the-art bound analyses (a) on a large benchmark of real-world C code, (b) on a benchmark built of examples taken from the bound analysis literature and (c) on a benchmark of challenging iteration patterns which we found in real source code.

As most approaches to bound analysis, our analysis infers *upper bounds* on the execution cost of a program.

Contents

CHAPTER 1

Introduction

Complexity is a rigorous notion of *efficiency* on the level of *algorithms*. It is well known that the same functionality can be implemented in various ways, with some implementations consuming more time and/or memory space than others. A typical example is the task of sorting a list of n numbers: A naive implementation needs n^2 computation steps whereas efficient algorithms such as *heapsort* perform the same task in $n \log(n)$ steps at most.

An enormous body of literature has been written on the *manual* analysis of algorithms. The standard book [CLRS01] states that "*Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.*" Analysis of algorithms is an essential subject of any computer science curriculum because *resources* such as *computation power*, *time*, *memory*, *network bandwidth*, etc., are *limited* and often *short* (consider, e.g., embedded systems, sensor networks, or smart phones) and it is therefore important to find and implement *efficient* solutions.

However, modern software is implemented by hundreds of thousands of lines of code. A pen-and-paper reasoning as advocated for the analysis of algorithms is highly intractable for real implementations. The field of *automated complexity and resource bound analysis* (short: bound analysis) (e.g., [GZ10, ADFG10, BEF$^+$16, AAGP11]) is dedicated to the development of methods for inferring bounds on the resource consumption of software by an *automated* analysis of the program's source code. Such an analysis treats the program under scrutiny as a mathematical object, inferring a bound on the program's resource consumption by means of automated formal reasoning.

To this day, bound analysis remains an academic research discipline. As we elaborate in Section 1.6, and Section 6.4, current approaches to automated resource bound analysis lack applicability for several reasons, one important being their poor *performance* characteristics. While impressive academic examples can be solved, present approaches do not scale up to real-world code. Moreover, existing approaches fail on common iteration

```
void foo(uint n) {
  int i = n;
  while(i > 0) {
    i = i - 1;
  }
}
```

Cost: 4n + 3

(a)

```
void foo(uint n) {
  int i = n;
  while(i > 0) {
    ConsumeR(2);
    i = i - 1;
  }
}
```

Resource Bound: 2n

(b)

```
void foo(uint n) {
  int i = n;
  while(i > 0) {
    AllocateR(2);
    i = i - 1;
    FreeR(1);
  }
}
```

Resource Bound: n

(c)

Figure 1.1: Examples for Bound Analysis. (a) Cost Model: The cost of an *assignment* is 2, the cost of an operation (*comparison* or *decrement*) is 1, (b) Cost Model: `ConsumeR(2)` consumes 2 units of `R`, (c) `AllocateR(2)` allocates 2 units of `R`, `FreeR(1)` deallocates 1 unit of `R`

patterns which can, e.g., be found in parsing and string matching routines. With our work we aim towards overcoming these drawbacks.

1.1 Complexity, Resource Bounds and Cost

Whereas the *analysis of algorithms* is mainly concerned with the *asymptotic* complexity and resource consumption of algorithms, the field of *automated complexity and resource bound analysis* analyzes the *cost* of executing concrete implementations. Depending on the application domain, the unit of *cost* is a time unit, a power unit or other. The term *cost* is usually further defined by a *cost model* which assigns an execution cost to each program instruction or operation. A *bound* on the total execution cost of a program is then expressed by a symbolic expression over the program's parameters.

As an example, consider Figure 1.1 (a). If we assume that an assignment costs 2 units and that the operations *comparison* and *decrement* cost 1 unit each, a bound on the overall cost of executing the program is $4n + 3$: Each of the n iterations of the loop costs 4 (1 comparison, 1 decrement and 1 assignment), initially the assignment of n to i costs 2, the final comparison evaluating the loop condition to *false* costs 1.

In particular, a bound on the consumption of a *resource* `R` is obtained by assigning cost to those instructions or operations which consume `R`. The cost that is assigned to an instruction `I` must be a bound on the amount of `R` that may be consumed when executing `I`. E.g., assume that we add the statement '`ConsumeR(2)`' to the body of the loop in Figure 1.1 (a) as shown in Figure 1.1 (b). Let us assume that this statement consumes 2 units of `R`. Using a cost model which assigns the cost 2 to '`ConsumeR(2)`' and the

cost 0 to all other instructions, we obtain the bound $2n$ on the consumption of `R` for Figure 1.1 by a straightforward manual reasoning.

Note, however, that resources such as *memory* can also be *freed*. This can be modeled by assigning *negative costs*. E.g., consider Figure 1.1 (c) where the resource `R` is allocated and *deallocated*. For Figure 1.1 (c), we expect a resource bound analysis to infer the bound n on the total consumption of `R`.

Uniform Cost Model. In our example we have assigned *constant costs* to single instructions or operations. Such a cost model is called a *uniform* cost model. Depending on the application area, a uniform cost model might not always be accurate. For an example, see our discussion on *worst case execution time analysis* in Section 1.6. On the other hand, a uniform cost model simplifies the bound analysis problem and is sufficient for many application areas, of which we sketch some in Section 1.2. The approach we present in this thesis assumes a uniform cost model.

Program Complexity and the Back-Edge Metric. The examples in Figure 1.1 contain an instruction which we did not consider so far, namely the *back jump* to the *while* statement at the end of a loop iteration. [CHS15] calls the cost model which assigns the cost 1 to each *back jump instruction* and the cost 0 to all other instructions or operations the *back-edge metric*. Using the back-edge metric we obtain the cost n for the examples in Figure 1.1 because in total n back jumps are executed during program run. The back-edge metric is interesting because it reflects the *asymptotic time complexity* of a program: independent from the concrete definition of what is to be considered as *computation step*, the time complexity of an algorithm or program is asymptotically determined by the number of times that instructions can be *repeated* during program execution. In our upcoming discussions we will suppose the back-edge metric to exemplify the bound analysis problem on the respective examples. Throughout this work we refer to the back-edge metric when we speak of the *complexity* of a program. E.g., we say that the complexity of the examples in Figure 1.1 is n because n is the cost of executing the examples when the back-edge metric is assumed. Note, however, that our analysis can be easily instrumented by other (uniform) cost models. We discuss such instrumentations in Section 3.10 and Section 3.10.1.

Loop Bound. We use the back-edge metric also to define the term *loop bound*: Consider the loop in Figure 1.1. We call the `while`-instruction the *header* of the loop. A *loop bound* for the loop in Figure 1.1 is a bound on the number of times that any back jump to the loop's header (the `while`-instruction) can be executed. We thus have that n is a loop bound for the single loop in Figure 1.1. Generally: Let `l` denote the header of a loop `L`, i.e., `l` is the location from which `L` is entered. A *bound* for `L` is a bound on the number of times that a back jump to `l` is executed.

Worst-Case Bounds. Depending on the application area, different kinds of bounds on the cost of a program execution may be required. For example, a bound can be a

lower-, an average-, or a worst-case bound. Like most approaches to bound analysis, our approach aims at inferring worst-case bounds. More precisely, we consider any *upper* bound on the worst-case cost to be *sound*. Obviously, an upper bound on the worst-case cost is only of interest if it is *tight*, i.e., if it is close to the worst-case cost of the program. We therefore aim at developing methods which obtain *tight upper bounds on the worst-case cost*.

1.2 Application Domains

Software Profiling. In industry *profiling*, i.e., monitoring resource consumption during program run, is the state-of-the-art for analyzing the performance behaviour of software products. The corresponding analysis tools, called *Profilers*, link the resource consumption that they observe to the statements by which these resources are allocated. However, profiling can only experiment with a limited number of execution scenarios and will thus always miss other scenarios which may later cause performance problems in operation mode. Because profiling can only be applied when the product is already in an executable stage, a performance bottleneck is detected late in the development cycle. In fact, *performance bugs*, i.e., usability restrictions due to poor program performance, are omnipresent in modern software and, e.g., familiar to every smart phone user. [ZAH11] states that "*performance bugs take more time to fix, need to be fixed by more experienced developers and require changes to more code than non-performance bugs*". Discovering and fixing performance bugs is subject to an intensive research in software engineering (e.g., [ZAH11, JSS+12, NJT13, KNP+10]). However, *avoiding* performance bugs is obviously preferable over *fixing* performance bugs. Moreover, the late detection of performance bugs can cause high costs. The basic requirement for avoiding performance bugs is that the programmer must be *aware* of badly performing implementation parts already *early* during software development. A tool that informs about the worst-case amount of resources which may be consumed when executing a given piece of code, would obviously be of great help. We propose *bound analysis* as a means to avoid performance bugs.

With respect to the detection and analysis of performance bugs, bound analysis has three important advantages over the currently common *profiling* approach:

- It covers *all* possible execution scenarios. I.e., if a bound on the resource consumption is inferred, this bound is guaranteed to hold for all executions of the software.
- Isolated parts of the software, e.g., functions, methods, sub-routines, etc., can be analyzed independently, even if the software as a whole is not (yet) in an executable stage. This allows early detection of (potential) performance problems.
- Bound analysis expresses resource consumption in terms of the input parameters of the program (or program part) under scrutiny. This helps understanding the cause of performance problems.

We conclude that bound analysis can complement classical profiling in industrial software development.

We now sketch further potential application domains of bound analysis:

Program Understanding. Complexity and resource bound analysis can be used to explore unfamiliar code or to annotate library functions by their performance characteristics; [JSS+12] states that a substantial number of performance bugs can be attributed to a "*wrong understanding of API performance features*".

Program Verification. In many applications such as *embedded systems* there is a hard constraint on the availability of resources like CPU time, memory, bandwidth, etc. Limits on the resource consumption must not be exceeded during program run, otherwise this may cause the program to behave unexpectedly. For example, the program may crash. Or, in the case of real-time systems, an intermediary result that is not available in time may cause the computation of overall wrong results (see our discussion on worst case execution time analysis in Section 1.6). Exceeding resource limits during program run is thus to be considered a *safety error* and a violation of *functional correctness*. Bound analysis can serve as a means to prove the absence of such bugs: By computing an upper bound on the resource consumption we can show that exceedance of the given limits can never happen.

Software Security. In the field of *security*, bound analysis could be applied for deriving a bound on how much *secret information* is *leaked* in order to decide whether this leakage is acceptable [Smi09]. For example, consider a loop where up to x bits of information are leaked in each iteration. A bound on the total number of loop iterations allows to conclude whether or not the *overall* leakage is sufficient to guess the *secret*. Our analysis could be a useful tool in the process of proving that the amount of leaked information is *not* sufficient for obtaining the secret.

Automatic Parallelization. A research problem still unsolved is how we can take advantage of the modern multi-core processors. An attractive idea is the automatic parallelization of sequential code. Besides understanding the data dependencies in the code, two fundamental questions need to be answered in order to decide whether to parallelize a certain loop under consideration: How often is the loop repeated? What is the cost of a single loop iteration? Both questions can be answered by automated resource bound analysis.

1.3 Aim of the Work and Methodological Approach

Our goal is to develop an automatic complexity and resource bound analysis for imperative programs. Given that the underlying *halting problem* is *undecidable* [Tur36] we nevertheless aim at a *practical* solution. By "practical solution" we refer to the

following two essential properties which our solution shall comply with: 1) We want to automatically infer the program complexity implemented by typical iteration patterns used in imperative code. 2) Our solution shall be *efficient*, meaning that 2a) bounds are inferred in reasonable time, 2b) resources of a standard desktop computer are sufficient to run our analysis on real world software, 2c) our analysis fails fast if no bound can be inferred.

As our demands are high, we restrict ourselves to the following essential program paradigms in order to make a solution technically feasible within the limits of a PhD thesis: Our analysis handles *integer programs* with *loops* (or *tail-recursion*). Note that this setting allows the handling of non-recursive function calls by *function inlining*. We leave the extension of our approach to the handling of data structures, pointers and recursion for future work. We note that *recursion* is not often used in C programs (only 91 out of 1751 functions in our benchmark, see Section 5.2.1) and that recursive programs often turn out to be tail-recursive after deleting all instructions which are irrelevant for the program's complexity (program slicing). Furthermore, programs using data structures can, in principle, be modeled by pure integer programs using techniques such as [MTLT10, GLAS09]. The examples which we discuss next (Section 1.4) demonstrate that inferring the complexity of integer programs with loops is very challenging for iteration patterns as they can be found in real-world source code.

Methodologically we follow an approach that is well-established in computer science, in particular in the area of *computer aided verification*. Given that *every non-trivial program property is undecidable* (Rice's theorem), many advances in program verification are due to the development of suitable *program abstractions* and abstraction mechanisms. A program abstraction is suitable, if inferring the desired property becomes feasible in the abstraction and crucial information about the program can be maintained by appropriate abstraction mechanisms. This methodology underlies many approaches to *software model checking* and *program analysis*.

In order to identify suitable program abstractions, abstraction mechanisms, and algorithms, we adopt an interplay between theory and practice: We develop a theoretical framework for bound analysis and, at the same time, implement our ideas into a prototype tool. Applying our implementation on real-world software will help to identify practically relevant problems, thus spurring further research on the theoretical framework. More precisely, we develop our solutions along the following research cycle:

1. We study real-world code in order to identify typical iteration patterns and challenging complexity behaviour.

2. Generalizing from our observations, we identify an abstract program model which captures the observed implementation patterns.

3. We design algorithms which compute the complexity of programs formulated in the abstract program model.

4. We implement our algorithms and run experiments on real source code in order to find further interesting code bases which serve as our field-of-study in the next iteration.

By means of this methodology we iteratively adjust both our abstract program model and the algorithms until we can handle a large class of real-world code in a scalable and predictable manner.

Our goal can thus be further concretized as follows:

1. We identify a suitable abstract program model for bound analysis with desirable properties such as the decidability of the halting problem. Most of the complexity behaviour of real-world imperative programs shall be expressible in our abstract program model.
2. We devise correctness-proven algorithms for bound analysis and amortized complexity analysis of programs formulated in the abstract program model.
3. We implement our algorithms and abstraction mechanisms in a publicly available tool which will allow us to prove the practical relevance of our research by means of experiments on open-source code.

Our implementation is intended to handle programs written in the C programming language. The concepts, algorithms, etc., we develop shall, however, be general enough to be applicable to other imperative languages as well.

1.4 Motivation and Overview

In the following we motivate our algorithm and technical solutions by four typical challenges to bound analysis. For each challenge, the discussion is structured as follows: In a first subsection we motivate and explain the problem that is posed by the challenge. In a second subsection we state intuition on how our algorithm copes with the challenge. The third subsection refers to those parts of this thesis where the previously summarized technical solution is detailed.

In order to give the reader some general orientation, we outline the overall structure of the thesis in Section 1.5.

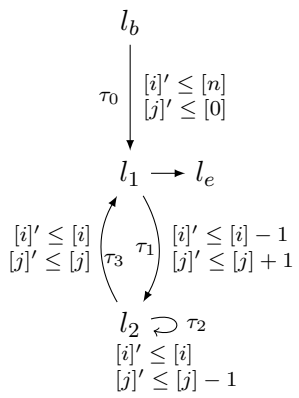
1.4.1 Challenge I: Amortized Complexity

Amortized complexity analysis [Tar85] aims at inferring the worst-case average cost over a sequence of calls to an operation or function rather than the worst-case cost of a single call. In (resource) bound analysis the difference between the single worst-case cost and the amortized cost is relevant, e.g., if a function `f` is called inside a loop: Assume the loop bound is n and the single worst-case cost of a call to `f` is also n. The cost of a

```
  void tarjan(uint n) {
     uint i = n;
     uint j = 0; //stack is empty
l1:  while (i > 0) {
        i--;
        j++;          //push;
l2:     while (j > 0 && ?) //pop
           j--;
     }
  }
```

Total Worst-Case Cost of "push" and "pop": $2n$

(a)



(b)

Figure 1.2: (a) Example `tarjan`: Model of Tarjan's classical example [Tar85] for amortized complexity analysis, with '?' denoting non-determinism. (b) *DCP* obtained by abstraction from `tarjan`

single call to `f` *amortized* over all n calls might, however, be lower than n, e.g., 2. In this case the total worst-case cost of iterating the loop is $2n$ rather than n^2. Note that in our non-recursive setting function calls can always be inlined. The amortized analysis problem thus boils down to the problem of inferring the cost of executing an inner loop averaged over all executions of the outer loop.

Tarjan [Tar85] motivates *amortized complexity analysis* on the example of a program which executes n stack operations `StackOp`. Each `StackOp` operation consists of a *push* instruction, adding an element to the stack, followed by a *pop* instruction, removing an arbitrary number of elements from the stack. Initially the stack is empty. The *cost* of a single *push* is 1 and the *cost* of a single *pop* is the number of elements removed from the stack. Tarjan points out that the worst-case cost of a single *pop* is n: the nth *pop* instruction may pop n elements (cost n) from the stack, if the previous *pop* instructions did not remove any elements from the stack. I.e., the worst-case cost of a single `StackOp` operation is $n + 1$. Nevertheless all n operations `StackOp` cannot cost more than $2n$ in total since we cannot remove more elements from the stack than have been added to the stack and thus the overall cost of the *pop* instructions is bounded by the total number of *push* instructions (n by assumption). The *amortized cost* of `StackOp`, i.e., the cost of `StackOp` averaged over the sequence of all n operations, is therefore 2.

Example `tarjan` in Figure 1.2 (a) shows a model of Tarjan's example in form of two nested loops: All calls to `StackOp`, *push* and *pop* are *inlined*, j models the stack size, the inner loop models the *pop* operation. The body of the outer loop models the `StackOp` operation. The symbol '?' denotes non-determinism. Considering Example `tarjan`, the problem posed by Tarjan amounts to the problem of inferring a bound on the number of

times the outer and the inner loop of the example may be executed.

In accordance with Tarjan's example, the inner loop can be executed n times on a single execution of the outer loop (the worst-case cost of a single *pop* is n). But averaged over all n executions of the outer loop, the inner loop can only be executed once at most per execution of the outer loop (the amortized cost of a single execution of the outer loop is 2). Current approaches to automated complexity and resource bound analysis from the literature [GG08, GMC09, GJK09, GZ10, BHHK10, ZGSV11, AGM13, BEF$^+$16] are not able to reason about the *amortized cost* of inner loop executions: For Example `tarjan` these approaches assume that the inner loop can be executed n times on *each* execution of the outer loop and therefore infer the worst-case cost of n^2 rather than $2n$.

We sketch next how our approach automatically infers the precise overall cost of $2n$ of Example `tarjan`.

1.4.2 Sketch of Our Analysis I: Transition Bounds

Our approach is based on the observation that - in most cases - the complexity of imperative programs evolves from the interaction of loop counters. More precisely, the number of times a given loop or nested loop construct can be executed depends on the interplay between loop counter decrements, increments and resets. This is, in fact, intuitive to every programmer: In imperative programs control is typically separated from data. Counter variables are used to control the number of repetitions of a code sequence (a loop), other variables are used to perform the actual data manipulations. We conclude that an effective complexity analysis for imperative code has to track decrements, increments, and resets of counter variables.

One of our our key insights is that *difference constraints* (*DCs*) provide a *natural abstraction* of the standard manipulations of counters in imperative programs. Difference constraints have been introduced by Ben-Amram for termination analysis in [Ben08], where they denote relational inequalities of the form $x' \leq y + c$, and describe that the value of x in the current state is at most the value of y in the previous state plus some constant $c \in \mathbb{Z}$. We call a program whose transitions are given by a set of difference constraints a *difference constraint program* (*DCP*). As we elaborate in Section 2.2, counter *increments and decrements*, i.e., $x := x + c$ resp. *resets*, i.e., $x := y$, can be modeled by the *DCs* $x' \leq x + c$ resp. $x' \leq y$. Our approach exploits the expressive strength of *DCs*, distinguishing between counter resets, counter increments and counter decrements in the reasoning.

We now sketch how our approach infers the linear cost for Example `tarjan`:

1. Program Abstraction. We abstract the program to a *DCP* over $\mathbb{N}$ as shown in Figure 1.2 (b). The abstract variable $[j]$ represents the program expression $\max(j, 0)$.
2. Finding Local Bounds. We identify $[j]$ as a variable that limits the number of executions of transition τ_2: $[j]$ decreases on each execution of τ_2 ($[j]$ takes values over $\mathbb{N}$). We call $[j]$ a *local bound* for τ_2. Accordingly we identify $[i]$ as a *local bound* for τ_1 and τ_3.

3. Bound Analysis. Our bound algorithm (Chapter 3) computes *transition bounds*, i.e., (symbolic) upper bounds on the number of times program transitions can be executed. The main idea of our transition bound algorithm is to estimate *how often* and by *how much* the value of the transition's local bound may increase during program run. This reasoning is implemented by our function $T\mathcal{B}(\tau)$ which computes a transition bound for transition τ. We give an intuition on how our function $T\mathcal{B}(\tau)$ computes transition bounds: Assume we want to infer a bound on the overall cost of the `StackOp` operation, i.e., a bound on the total number of times that transitions τ_1, modeling the push instruction, and τ_2, modeling the pop instruction, can be executed. Our algorithm computes

$$\begin{aligned} T\mathcal{B}(\tau_1) &\overset{(1)}{\to} T\mathcal{B}(\tau_0) \times [n] \\ &\overset{(2)}{\to} 1 \times [n] \\ &= [n] \end{aligned}$$

where
(1) because the local bound $[i]$ of τ_1 is initially (on τ_0) set to $[n]$ and never increased or reset, (2) because $T\mathcal{B}(\tau_0) = 1$ since transition τ_0 is not part of any loop and can thus only be executed once.

Our algorithm computes $T\mathcal{B}(\tau_2)$ as follows:

$$\begin{aligned} T\mathcal{B}(\tau_2) &\overset{(1)}{\to} [0] + T\mathcal{B}(\tau_1) \times 1 \\ &\overset{(2)}{\to} 0 + [n] \times 1 \\ &= [n] \end{aligned}$$

where
(1) because τ_2 has local bound $[j]$ and $[j]$ is initially set to $[0]$ on τ_0; further $[j]$ is incremented by 1 on τ_1, i.e., the value of the local bound $[j]$ of τ_2 can be incremented up to $T\mathcal{B}(\tau_1)$ times by 1,
(2) our algorithm obtains $T\mathcal{B}(\tau_1) = [n]$ by a recursive call.

We have $T\mathcal{B}(\tau_1) + T\mathcal{B}(\tau_2) = [n] + [n] = 2n$ because the parameter n of Example `tarjan` has type *unsigned*. We thus obtain $2n$ as a bound on the total cost of the `StackOp` operation.

1.4.3 Overview I

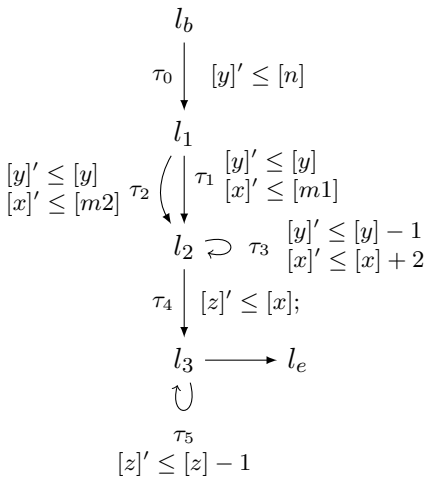
We formally define our abstract program model of *difference constraint programs* in Section 2.1. We discuss how we obtain abstract programs from concrete imperative code in Section 2.2.

Chapter 3 presents our bound algorithm. In Section 3.1 we give a formal definition of the problem that is solved by our algorithm and some basic but unavoidable definitions.

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

Bound of loop at l_3: $\max(m1, m2) + 2n$

(a)



(b)

Figure 1.3: (a) Example `twoSCCs`, (b) *DCP* obtained by abstraction from `twoSCCs`

In Section 3.2 we state our bound algorithm restricted to a special case of our abstract program model, namely *lossy vector addition systems* (VASS). A *DCP* is a VASS if variable updates are of form $x' \leq x + \mathtt{c}$. In a VASS, updates of form $x' \leq y + \mathtt{c}$ with $y \neq x$ are only allowed on the initial transition τ_0. E.g., the abstraction of Example `tarjan` in Figure 1.2 (b) is a VASS. In Section 3.2, we discuss in detail how our algorithm infers the precise cost of Example `tarjan`.

In Section 3.12 we discuss how our approach relates to Tarjan's classical amortized analysis technique by means of *potential functions*.

In Section 3.3 we generalize our bound algorithm to a broader set of abstract programs, allowing updates of form $x' \leq y + \mathtt{c}$ with $y \neq x$ if y is a *symbolic constant*.

In Section 3.6 we describe how we determine local bounds.

Next, we state an example of a general *DCP* and sketch how our bound algorithm infers bounds for general *DCP*s.

1.4.4 Challenge II: Invariants and Bound Analysis

Consider Example `twoSCCs` in Figure 1.3 (a). By '?' we denote non-determinism that arises from a condition which is not modeled in our analysis like, e.g., a function call (that cannot be inlined), a pointer dereference or some instruction accessing the heap. Note

that the abstracted *DCP* of Example `twoSCCs`, shown in Figure 1.3 (b), is a *general DCP*: The update $[z]' \leq [x]$ on τ_4 is not expressible in a VASS. Moreover, x is a variable rather than a symbolic constant.

Assume we want to compute a bound on the number of times that the second loop (at l_3) of Example `twoSCCs` can be executed. It is easy to infer x as a bound on the possible number of iterations of the second loop. However, when it comes to obtaining a bound in the *program parameters* the difficulty lies in finding an invariant of form $x \leq \texttt{expr}(n, m1, m2)$ where $\texttt{expr}(n, m1, m2)$ denotes an expression over the program parameters $n, m1, m2$. Here, the most precise invariant $x \leq \max(m1, m2) + 2n$ cannot be computed by standard abstract domains such as *octagon* or *polyhedra* [Min06]: these domains are *convex* and cannot express non-convex relations such as *maximum*. The most precise approximation of x in the polyhedra domain is $x \leq m1 + m2 + 2n$. Unfortunately, as is well-known, the polyhedra abstract domain does not scale to larger programs and needs to rely on heuristics for termination.

Our approach does not rely on abstract interpretation for inferring the bound $m1+m2+2n$ for the loop at l_3. Our analysis implements an alternative idea, inferring invariants of form $x \leq \texttt{expr}(\vec{n})$, where $\vec{n}$ denotes the program parameters, by means of bound analysis. We give an intuition on this idea in the next section, where we sketch how our bound algorithm infers bounds for *general DCPs* such as the abstraction of Example `twoSCCs` in Figure 1.3 (b).

1.4.5 Sketch of Our Analysis II: Variable Bounds

Besides *transition bounds*, our analysis also infers *variable bounds*, i.e., (symbolic) upper bounds on variable values. Similar to our reasoning on transition bounds, the main idea is to reason *how much* and *how often* the value of a variable may increase during program run.

Our core algorithm for general *DCP*s (Section 3.4) is based on a mutual recursion between variable bound analysis ("how much", function $V\mathcal{B}(\mathtt{v})$) and transition bound analysis ("how often, function $T\mathcal{B}(\tau)$). We sketch how our algorithm infers a bound for the loop at l_3 of Example `twoSCCs`:

Consider the *DCP* abstraction of Example `twoSCCs` shown in Figure 1.3 (b). Transition τ_5 corresponds to the loop at l_3. We have that $[z]$ is a *local bound* for τ_5 and $[y]$ is a local bound for τ_3.

Our algorithm computes a transition bound for τ_5 in Figure 1.3 (b) as follows:

$$\begin{aligned} T\mathcal{B}(\tau_5) &\overset{(1)}{\to} T\mathcal{B}(\tau_4) \times V\mathcal{B}([x]) \\ &\overset{(2)}{\to} 1 \times V\mathcal{B}([x]) \\ &= V\mathcal{B}([x]) \\ &\overset{(3)}{\to} T\mathcal{B}(\tau_3) \times 2 + \max([m_1], [m_2]) \end{aligned}$$

$$\begin{aligned}
&\overset{(4)}{\rightarrow} ([n] \times T\mathcal{B}(\tau_0)) \times 2 + \max([m_1], [m_2]) \\
&\overset{(5)}{\rightarrow} ([n] \times 1) \times 2 + \max([m_1], [m_2]) \\
&= 2 \times [n] + \max([m_1], [m_2])
\end{aligned}$$

We discuss the computation steps:
(1) The local bound $[z]$ of τ_5 is set to $[x]$ on each execution of τ_4.
(2) We compute $T\mathcal{B}(\tau_4) = 1$ since τ_4 cannot be repeated.
(3) Variable $[x]$ is incremented by 2 on each execution of τ_3. Further $[x]$ is initially set to either m_1 or m_2.
(4) We compute $T\mathcal{B}(\tau_3) = [n] \times T\mathcal{B}(\tau_0)$, because the local bound $[y]$ of τ_3 is set to $[n]$ on τ_0.
(5) We compute $T\mathcal{B}(\tau_0) = 1$ because τ_0 cannot be repeated.

Finally we get $m1 + m2 + 2n$ as bound for the loop at l_3 in Example `twoSCCs` because n, m_1, m_2 have type *unsigned* and therefore $[n] = n$, $[m_1] = m_1$ and $[m_2] = m_2$.

We point out the mutual recursion between $T\mathcal{B}$ and $V\mathcal{B}$: $T\mathcal{B}(\tau_5)$ calls $V\mathcal{B}(x)$, which in turn calls $T\mathcal{B}(\tau_3)$. We highlight that the variable bound $V\mathcal{B}(x)$ (corresponding to the invariant $x \leq \max(m_1, m_2) + 2n$) is established during the computation of $T\mathcal{B}(\tau_5)$.

1.4.6 Overview II

In Section 3.4 we state our bound algorithm for general *DCP*s. In Section 3.11 we discuss how our reasoning is substantially different from invariant analysis by abstract interpretation. In Section 3.10 we show how our analysis can be instrumented by a *uniform cost model* through the $V\mathcal{B}$ function. The general idea is to introduce a fresh counter c which is initially set to 0. We increment the counter c on a transition τ by the cost of executing the instructions and/or operations on τ. Note that this cost is determined by the cost model. Our *variable bound* method can then be applied to c (computing $V\mathcal{B}(c)$) in order to infer an upper bound on the overall *worst-case cost*, which is modeled by c.

We prove soundness of our bound algorithm in Section 7.1.

1.4.7 Challenge III: Real-World Amortized Analysis

Example `xnu` in Figure 1.4 (a) is a representative of a class of iteration patterns that we found in parsing and string matching routines during our experiments. In these loops the inner loop iterates over disjoint partitions of an array or string, where the partition sizes are determined by the program logic of the outer loop.

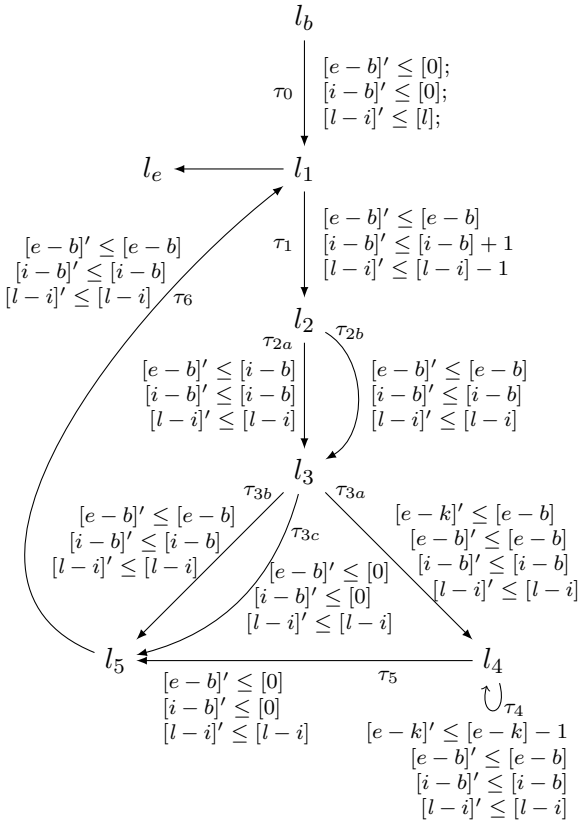
Example `xnu` is taken from the SPEC CPU2006 benchmark [spe]. It is a sliced version of function XNU in 456.hmmer/src/masks.c.

The outer loop of Example `xnu` partitions the interval $[0, len]$ into disjoint sub-intervals $[beg, end]$. The inner loop iterates over the sub-intervals. Therefore the inner loop has an

```
void xnu(int len) {
  int beg,end,i = 0;
l1 while(i < len) {
     i++;
l2   if (?)
       end = i;
l3   if (?) {
       int k = beg;
l4     while (k < end)
         k++;
       end = i;
       beg = end;
     } else if(?) {
       end = i;
       beg = end;
     }
l5 }
}
```

Complexity: $2 \times \max(len, 0)$

(a)



(b)

Figure 1.4: Example `xnu`: (a) Sliced version of function XNU in 456.hmmer/src/masks.c in SPEC CPU2006 benchmark [spe], (b) *DCP* obtained from Example `xnu`; we use the following short forms: l for *len*, b for *beg*, e for *end*

overall linear iteration count. Example `xnu` is a natural example for amortized complexity: Though a single visit to the inner loop can cost *len* (if $beg = 0$ and $end = len$), several visits can also not cost more than *len* since in each visit the loop iterates over a disjoint sub-interval. We therefore have: The *amortized cost* of a visit to the inner loop, i.e., the cost of executing the inner loop within an iteration of the outer loop averaged over all *len* iterations of the outer loop, is 1. Here, we refer by *cost* to the number of consecutive back jumps in the inner loop. But in general, any *resource consumption* inside the inner loop can, in total, only be repeated up to $\max(len, 0)$ times (see Discussion in Section 1.1).

Together with the loop bound $\max(len, 0)$ of the outer loop, our observation yields an overall complexity of $2 \times \max(len, 0)$.

1.4.8 Sketch of Our Analysis III: Reset Chains

Consider the *DCP* abstraction of Example `xnu` Figure 1.4 (b). We use the following short forms: l for *len*, b for *beg*, e for *end*. Recall that $[a]$ denotes $\max(a, 0)$.

We outline how the abstraction is obtained from Example `xnu`:
Based on the conditions $i < l$ and $k < e$ our abstraction algorithm forms the expressions $[l - i]$ and $[e - k]$. The abstraction (Figure 1.4 (b)) is now obtained by tracking how the value of each of these two expressions changes on the respective program transitions: For example, we have that k is set to b in the if-branch at l_3. This *reset* of k to b is modeled in the abstracted *DCP* by the predicate $[e - k]' \leq [e - b]$ on transition τ_{3a}. As a result, our abstraction algorithm now also tracks how the value of the expression $[e - b]$ is altered on each program transition, etc.

Next, we motivate a powerful feature of our analysis by which we obtain the precise bound *len* of the loop at l_4 of Example `xnu`: As discussed previously on Example `tarjan` and Example `twoSCCs`, our bound algorithm is applied on the abstract *DCP* representation (Figure 1.4 (b)). Here, the loop at l_4 is modeled by transition τ_4. It is easy to see that $[e - k]$ is a *local bound* for τ_4, $[l - i]$ is a local bound for all other transitions except τ_0 which can only be executed once. Our algorithm infers a bound for the loop at l_4 by computing $T\mathcal{B}(\tau_4)$ based on its local bound $[e - k]$.

However, our reasoning as sketched so far on Example `tarjan` and Example `twoSCCs`, is not sufficient for obtaining the *linear* bound of τ_4 (the loop at l_4): On the previous examples, our algorithm considered only the *increases* of a *local bound* or *variable*. For the linear bound of τ_4, it is, however, decisive to consider the *reset* of $[e - b]$ to $[0]$ on transition τ_5, as we argue next. If this reset were not on τ_5, transition τ_4 could indeed be executed a *quadratic* number of times: Note that the value of $[i - b]$ can be incremented to $\frac{[l]}{2}$ (l denotes the program parameter *len*) by executing the outer loop (at l_1) $\frac{[l]}{2}$ times ($[i - b]$ is incremented by 1 on τ_1). We can set $[e - b]$ to $\frac{[l]}{2}$ by executing τ_{2a}. Now, if $[e - b]$ was not reset to $[0]$ on τ_5, we could set the local bound $[e - k]$ of τ_4 to $\frac{[l]}{2}$ on each of the remaining $\frac{[l]}{2}$ executions of τ_{3a}. I.e., on each of the $\frac{[l]}{2}$ remaining executions of the outer loop, the loop at l_4 could be executed $\frac{[l]}{2}$ times, resulting in a total of $\frac{[l]^2}{4}$ executions.

For reasoning on *resets* that can *decrease* the maximum amount of iterations (as the reset of $[e - b]$ to $[0]$ on τ_5, which decreases the maximum amount of iterations of the loop at l_4), we introduce the concept of *reset chains*. We say that $[0] \xrightarrow{\tau_5} [e - b] \xrightarrow{\tau_{3a}} [e - k]$ is a *sound reset chain* because whenever $[e - k]$ is set to $[e - b]$ on τ_{3a}, $[e - b]$ will in turn be set to 0 on τ_5. Our algorithm now applies the same reasoning as outlined on the previous examples but based on *reset chains* rather than on single transitions. We state the details in Section 3.5.

In our experiments (Section 5.2.3) our implementation `loopus` (available at [looa]) was the only tool that inferred the *linear* complexity of Example `xnu`.

1.4.9 Overview III

We formally introduce our concept of *reset chains* in Section 3.5, where we also extend our algorithm to the reasoning based on *reset chains* rather than single transitions. We prove soundness of this reasoning in Section 7.2.

We state all details on the abstraction of Example `xnu` by our abstraction algorithm in Section 2.3. We show how our *bound algorithm* infers the complexity of Example `xnu` in Section 3.7.

1.4.10 Challenge IV: Tracking Increments *and* Decrements

Consider the example in Figure 1.5 (a). The loop at l_2 can be executed n times in total: Initially the loop counter x of the loop at l_2 is set to $n+1$. x is *decremented* before entering the loop at l_2 and *incremented* after leaving the loop. However, the increment of x cannot increase the number of executions of the loop since it only levels out the decrement of x which happened before:

Note that the worst-case bound n of the inner loop remains if we remove the *increment* of x on the outer loop (the statement '$\mathtt{x}++$'): We can still execute the inner loop n times on the first iteration of the outer loop.

In contrast, if x were not *decremented* on the outer loop (if we removed the statement '$\mathtt{x}--$' from the outer loop), the bound of the inner loop would in fact be $2n$: we would have one additional execution of the inner loop on the first iteration of the outer loop, also we would have one additional execution of the inner loop for *each* of the remaining $n-1$ executions of the outer loop due to the increment of x in the outer loop.

The example demonstrates that *decrements* can decrease the maximal number of iterations of a loop. In particular, decrements can level out *increments*.

1.4.11 Sketch of Our Analysis IV: Path-Sensitive Reasoning

Consider the *DCP* in Figure 1.5 (b), which we obtain by abstraction from the example in Figure 1.5 (a). We have that $[x]$ is a *local bound* for transition τ_2, which corresponds to the loop at l_2 of Figure 1.5 (a).

In order to obtain a bound for the loop at l_2, we apply our previously discussed method $T\mathcal{B}$ on τ_2, based on the local bound x.

However, for obtaining a *precise* bound, it is not sufficient to reason only about the *increases* of $[x]$. As argued previously, the bound of the loop at l_2 is $2n$ rather than n if the *decrement* of x is ignored.

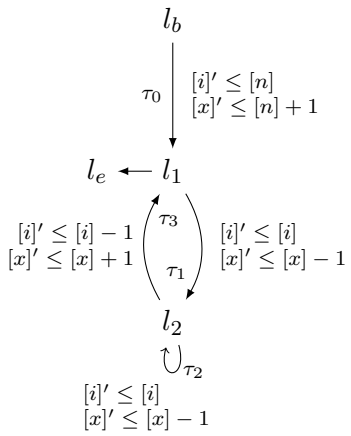
On example `xnu` we motivated a reasoning which considers *decreases* caused by *resets*. We now sketch how our reasoning takes into account *decreases* by *decrements*:

The essential idea is to consider increments and decrements of variables not only on single program transitions but on *cyclic paths* of the program. We therefore refer to

```
void foo(uint n) {
  int i = n;
  int x = n + 1;
l1 while(i > 0) {
     x--;
l2    while(x > 0 && ?)
        x--;
     x++;
     i--;
  }
}
```

Bound of Loop at l_2: n

(a)



(b)

Figure 1.5: (a) Minimal example for our path-sensitive analysis, (b) *DCP* obtained by abstraction

this reasoning as *path-sensitive* reasoning. The basic concept remains as motivated on Example `tarjan` and Example `twoSCCs`:

In the case of Figure 1.5 (b), our reasoning considers the following *cyclic* paths: $\pi_1 = \tau_1 \circ \tau_3$ and $\pi_2 = \tau_2$ (by $\tau_1 \circ \tau_3$ we denote the path that results from concatenating the transitions τ_1 and τ_3). When computing $T\mathcal{B}(\tau_2)$, our path-sensitive transition bound algorithm checks how the respective local bound is increased on π_1 and on π_2. We have that $[x]$ is increased neither on π_1 nor on π_2: On π_2 $[x]$ is only decremented, on π_1 the increment of $[x]$ on τ_3 is leveled out by the decrement of $[x]$ on τ_1.

Therefore, the maximal number of executions of τ_2 depends only on the *initial value* $[n] + 1$ of $[x]$. However, this value can flow to τ_2 only by taking the path $\tau_0 \circ \tau_1$. Since $[x]$ is decremented on τ_1, the total amount that flows into $[x]$ at l_2 is limited by $[n]$ rather than $[n] + 1$. We obtain $T\mathcal{B}(\tau_2) = [n]$.

Finally the precise bound n of the loop at l_2 is obtained since n has type *unsigned* and therefore $[n] = n$.

1.4.12 Overview IV

We formally define our path-sensitive bound algorithm in Section 3.8.3. As before, we develop our algorithm step-by-step: We first deal with the special case of a VASS. We then generalize our path-sensitive bound algorithm to full *DCP*s. We further discuss an interesting example which we found during our experiments. This discussion demonstrates the power of our path-sensitive reasoning for the complexity analysis of real source code.

In Section 3.10.1 we discuss the instrumentation of our path-sensitive reasoning for computing bounds on memory consumption.

In Section 7.3 we prove soundness of the path-sensitive reasoning.

1.5 Structure of the Work

The overall structure of our work follows our methodological approach: We start with the presentation of our abstract program model and appropriate abstraction techniques in Chapter 2. In Chapter 3, where we present our bound algorithm, we first limit ourselves to a restrictive sub-set of abstract programs, namely VASS. We then generalize our algorithm step by step as we summarized in Section 1.4.3. We present our full bound algorithm in Section 3.9. Our experiments in Section 5 demonstrate that our analysis can handle a large class of real-world code in a scalable and predictable manner.

To support comprehensibility it is our aim to keep the presentation of our bound analysis focused. To this end, we separate the essential insights and techniques from extensions and enhancements which we consider to be of a rather technical nature (Chapter 4). With respect to our abstraction mechanism such extensions are discussed in Section 4.1. In Section 4.2 we discuss extensions of our bound algorithm. We stress that these technical discussions are of high practical value and recommend to the reader the study of the respective sections.

In Section 5 we discuss some essential properties of our implementation. We further present the results of an evaluation and tool comparison on (a) a large benchmark of open source code (Section 5.2.1), (b) a benchmark of examples from the literature on bound analysis (Section 5.2.2), and (c) a benchmark of challenging loop iteration patterns gathered from real-world code bases (Section 5.2.3).

We summarize the results of our work in Section 6. In particular, we reflect on the contributions of our work to the field of automated complexity and resource bound analysis.

We present essential soundness proofs in Section 7. We prove soundness of our algorithm for general *DCP*s (Section 7.1), of our reasoning on reset chains (Section 7.2), and of the path-sensitive reasoning (Section 7.3).

We finally point out that, though we mainly discuss examples with *linear* bounds, our analysis is not limited to linear bounds. By placing the focus on examples with linear bounds, we stress the intrinsic difficulty of inferring such bounds in presence of nested loops or loops with multiple counters. Our analysis, however, can still infer (precise) bounds for all examples we discuss in this work if the respective example is placed inside of an outer loop, thereby obtaining an overall *quadratic* complexity.

```
void foo(uint n) {
   int i = n;
   int j = n;
   while(i > 0) {
     if (j > 0 && ?) {
        j--;
        i = n;
     } else
       i--;
   }
}
```

Bound of the loop: n^2

Figure 1.6: Example with two loop counters

1.6 Related Work and State of the Art

We start with a discussion of approaches from other research areas which are related to *automated complexity and resource bound analysis*. We discuss approaches from the fields of *termination analysis*, *worst-case execution time analysis*, and *performance profiling* (Section 1.6.1).

Afterward, we summarize recent work in the area of automated complexity and resource bound analysis. We categorize the different approaches in approaches based on *recurrence relations* (Section 1.6.2), approaches based on *termination proofs* (Section 1.6.3), approaches based on *amortized complexity analysis* techniques (Section 1.6.4), and approaches based on *program transformation* and *abstract interpretation* (Section 1.6.5).

We discuss *open issues* in the field of bound analysis in Section 1.6.6. In Section 1.6.7 we summarize our previous work in the area and point out the contributions that this thesis adds to our publications.

1.6.1 Related Research Areas

Complexity and resource bound analysis can be understood as a quantitative variant of *termination analysis*, which provides not only a qualitative "yes" answer, but also a symbolic upper bound on the run-time of the program. Whereas *(resource) bound analysis* has gained attention only recently, *termination analysis* has been intensively studied in the last decade.
Some approaches to bound analysis are based on termination analysis as we detail in Section 1.6.3. These approaches instrument termination proofs in form of *ranking functions*. Synthesis of *linear ranking functions* by means of **linear constraint solving** has been described in [PR04], amongst others. However, often a single linear ranking function does not exist: As an example consider the loop in Figure 1.6. A ranking

function for this loop is $n \times j + i$, but there is no linear ranking function. Note that our algorithm (Chapter 3) infers the precise loop bound n^2 for Figure 1.6.
[BMS05] generalized the constraint solving approach to the synthesis of *multi-dimensional* linear ranking functions. E.g., for the loop in Figure 1.6 we have that $\langle i, j \rangle$ is a ranking function over $\mathbb{N}^2$, where the well-founded order relation '$>$' over $\mathbb{N}$ is lifted to $\mathbb{N}^2$ lexicographically. But the constraint system generated by [BMS05] is *not linear*, which complicates the search for feasible solutions. The non-linear nature of the constraints is due to the fact that [BMS05] synthesizes the state invariants needed to infer the ranking function and the ranking function itself in one step. This has the advantage that no global invariant analysis is needed. An alternative is to run a **polyhedra analysis** upfront (as, e.g., in [ADFG10]) which, however, is also costly. We conclude that synthesis of ranking functions is a non-trivial task. Unlike other approaches to bound analysis (e.g., [BEF$^+$16, ADFG10, FH14, AAG$^+$12]), the approach we discuss in this work does not rely on general techniques for the synthesis of ranking functions.

However, our research benefits from research on termination analysis with respect to the investigation of abstract program models: The bound analysis in [ZGSV11] is based on the *size change abstraction* which was designed and investigated by Lee et al. [LJBA01] as an abstract program model for *termination analysis*.
Further, the termination paper [Ben08] shows that termination of *difference constraint programs* (*DCPs*) is undecidable in general but decidable for the natural syntactic subclass of *fan-in-free DCPs* (see Definition 4), which is the class of *DCPs* that we use as our abstract program model.

The field dedicated to *worst case execution time analysis* [WEE$^+$08] is closely related to the research on automated complexity and (resource) bound analysis. But unlike bound analysis, WCET analysis aims at predicting the timing behaviour of software *under consideration* of the underlying *hardware*. State-of-the-art WCET analysis techniques achieve a very high precision in computing time bounds by low-level modeling of architectural features such as caches, branch prediction and instruction pipelines. Such time bounds are needed, e.g., for proving correctness of *real-time systems*, where the result of a computation is required to be available within a certain time frame.

WCET analysis is, in general, not compatible with a uniform cost model (see Section 1.1), because such a cost model does not allow the timing behavior of hardware components such as caches to be modeled precisely: E.g., the performance of a cache or memory depends on physical constraints such as the temperature and can therefore increase or decrease over time when components are heating up.

For establishing loop bounds, WCET techniques usually either require user annotation, or use simple techniques based on pattern matching or numerical analysis. A symbolic loop bound analysis for WCET analysis has been suggested in [KKZ11]. The approach is, however, limited to the special case of for-loops with only one loop counter. E.g., [KKZ11] cannot handle the loop in Figure 1.6.

Recently, new *profiling* approaches have been proposed that apply curve fitting techniques

for deriving a cost function, which relates size measures on the program input to the measured program performance [ZH12, CDF12]. This idea can be understood as a *dynamic* bound analysis, which however, cannot give any guarantees on the correctness of the inferred bounds.

1.6.2 Bound Analysis Based on Recurrence Relations

Recurrence relations are the oldest methodology for computing bounds on *resource consumption*. [Weg75] automatically extracts recurrence relations from LISP programs and then computes closed expressions for the recurrences. [Mét88] first simplifies programs into ones with the same asymptotic complexity and then determines their complexity by checking a database for programs with the same recursion patterns.

In recent research, the recurrence relation approach is followed prominently by the COSTA project (e.g. [AAG+12, AAGP11, AGM13]). In this line of research, recurrence relations are obtained from so-called *cost equations* using **invariant analysis** based on the **polyhedra abstract domain** and **linear ranking functions**. Ranking functions are inferred through **linear programming** (as discussed in Section 1.6.1). Closed-form solutions for the obtained recurrence relations are inferred by means of **computer algebra**.

[BHHK10] also follows the recurrence based approach. [BHHK10] is limited to nested for-loops of a special shape (so-called ABC loops) but guarantees to infer a bound for such loop constructs. In particular, it is required that each loop has only one loop counter, resp. a second counter will be ignored. [BHHK10] cannot, e.g., handle the example in Figure 1.6. [BHHK10] instruments the innermost loop body by a new counter z which is increased on each iteration; next, a recurrence relation determining the counters value is generated. A polynomial *closed form* expression over the iteration variables is obtained by solving the recurrence. Due to the special shape of the for-loops, the recurrence is guaranteed to be of a form for which closed form solutions exist and can be inferred. A symbolic loop bound is obtained from the closed form expression by replacing the iteration variables by bounds on their value. These bounds are obtained by the same technique.

1.6.3 Bound Analysis Based on Termination Proofs

Another line of research obtains complexity bounds from termination proofs:

[ADFG10] over-approximates the *reachable states* by **abstract interpretation** based on the **polyhedra abstract domain**. This information is used for generating a **linear constraint problem** from which a **multi-dimensional linear ranking function** (see our discussion in Section 1.6.1) for each control location is obtained. Having inferred a ranking function rf_l for a control location l, the computed approximation of the reachable states is used for inferring a bound on the number of values which can be taken by the ranking function rf_l during program run. This bound is a *reachability bound* for l (a bound on the number of times location l can be visited during program run [GZ10]) since

the ranking function, by definition, takes a *different* value on each visit to l.
Importantly, the number of dimensions of the ranking function determines the degree of the bound polynomial. [ADFG10] therefore aims at inferring a ranking function with a *minimal* number of dimension. This is in contrast to [BMS05] which pioneered multi-dimensional ranking functions for *termination analysis*: for termination analysis the number of dimensions of the ranking function does not matter. Unlike [BMS05] [ADFG10] therefore depends on a *minimal* solution to the linear constraint problem which is obtained by **linear optimization** ([ADFG10] instruments the LP-solver with an *objective function*).

The bound analysis in [BEF$^+$16] applies approaches from the literature for **synthesizing ranking functions** (see our discussion in Section 1.6.1) thereby inferring bounds on the number of times the execution of isolated program parts can be repeated. These bounds, called *time bounds*, are then used to compute bounds on the absolute value of variables, so-called *variable size bounds*. Additional information is inferred through **abstract interpretation** based on the **octagon abstract domain**. An overall complexity bound is deduced by alternating between time bound and variable size bound analysis. In each alternation bounds for larger program parts are obtained based on the previously computed information. We experimentally compare to [BEF$^+$16] in Section 5.2.1 and Section 5.2.2. We give a detailed discussion on the comparison of our approach to [BEF$^+$16] in Section 6.4.

Our previous works [ZGSV11, SZV14a] also follow the termination proof-based approach: In [ZGSV11] a multi-dimensional ranking function is obtained based on the *size-change abstraction*, **invariant analysis** by **abstract interpretation** is employed to obtain upper bounds for each component of the multi-dimensional ranking function, multiplication of the obtained bounds results in an overall *complexity bound*. [SZV14a], one of our works on which this thesis is based, computes a multi-dimensional ranking function *efficiently* based on the abstract program models of *vector addition systems*. In contrast to [ZGSV11] no *global invariant analysis* is applied, a bound is obtained from the multi-dimensional ranking function by reasoning on how the transitions bounded by one component of the ranking function can increase other ranking function components. We state further aspects of [ZGSV11] and [SZV14a] in Section 1.6.7.

1.6.4 Bound Analysis Based on Amortized Analysis

An interesting line of research studies the *amortized complexity analysis* of first-order functional programs formulated as type rules over a template potential function with unknown coefficients (e.g. [HJ03, HAH12]); these coefficients are then found by *linear programming*. The core idea is the annotation of data structures by potentials that can be used to pay for executing program steps. The recent paper [CHS15] adapts this approach for imperative integer programs: Hoare-style proof rules are employed as an adequate *imperative* counterpart for the type rules of the functional approach. Similarly to the approach for functional programs, the rule system is applied for deriving a *linear constraint system*. The size of the generated problem is at least quadratic in the

program's size because any combination of two variables and/or constants is represented by a variable in the constraint system. Since bound analysis typically does not aim at *some* bound but tries to infer a *tight* bound, [CHS15] uses **linear optimization** (an LP-solver instrumented by an *objective function*) in order to obtain a *minimum solution* to the problem. In contrast to the older approach for functional programs, the imperative version of the approach can so far only derive *linear* resource bounds. We experimentally compare our implementation to [CHS15] in Section 5.2.2. We state further details on the comparison of our work to [CHS15] in Section 6.4.

An alternative approach for the amortized analysis of imperative programs is reported in [ABG12]. **Quantifier elimination** is applied for simplifying a constraint system over template cost functions. Since quantifier elimination is expensive, the technique does not scale to larger programs.

1.6.5 Bound Analysis Based on Program Transformation and Abstract Interpretation

Other approaches for bound computation employ **program transformation** and **abstract interpretation**. A straightforward approach is to introduce a counter, which is increased whenever the resource under consideration is consumed, and then to compute an upper bound on this counter with abstract interpretation techniques. [Ros89] was the first to implement this idea for first-order LISP programs. [GG08] proposes to extend the **polyhedra abstract domain** with the maximum operator and non-linear expressions. [GMC09] introduces multiple counters and exploits their dependencies such that upper bounds (inferred by **abstract interpretation**) have to be computed only for restricted program parts. [GJK09] proposes a program transformation (control-flow refinement) based on **abstract interpretation** that separates the different loop phases such that bounds can be computed for each phase in isolation. We discuss the potential benefit of control-flow refinement in the context of our approach in Section 4.1.3. [GZ10] employs proof rules for bound computation, combined with **disjunctive invariant generation** by abstract interpretation using **power set domains** for summarizing inner loops.

[FH14] combines **control-flow refinement**, **linear ranking functions**, **linear optimization** and **abstract interpretation** based on the **polyhedra abstract domain** to design a powerful bound analysis. [FH14] is based on the COSTA approach. In particular, [FH14] is formulated in terms of *cost equations*. Further [FH14], is inspired by the counter instrumentation-based approach [GMC09], by [GJK09], where control-flow refinement for bound analysis is discussed, and by termination analysis-based approaches to bound analysis like [SZV14a]. [FH14] applies the techniques [PR04, BMPZ12] for inferring **linear ranking functions** (see our discussion on the synthesis of ranking functions above).
We experimentally compare to [FH14] in Section 5.2.1 and Section 5.2.2. We state further details on the comparison of our approach to [FH14] in Section 6.4.

1.6.6 Conclusion: Drawbacks of Existing Solutions and Our Complementary Approach

Existing techniques to bound analysis are either limited to special cases ([BHHK10]) or based on the following general frameworks for *global invariant analysis* and *automated reasoning*:

- **abstract interpretation** (**polyhedra** or **octagon** abstract domain, [AAG+12, AAGP11, AGM13, FH14, BEF+16, ZGSV11, GG08, GMC09, GJK09]),
- **computer algebra** ([AAG+12, AAGP11, AGM13]),
- **synthesis of ranking functions** by **linear programming** and **invariant analysis** ([AAG+12, AAGP11, AGM13, FH14, ADFG10, BEF+16]),
- constraint solving through **linear optimization** ([FH14, HJ03, HAH12, CHS15]),
- **quantifier elimination** ([ABG12]),
- **program transformation** based on **abstract interpretation** ([GJK09]),
- **disjunctive invariant generation** using **power set abstract domains** ([GZ10]).

We think that the relatively poor performance characteristics of state-of-the-art bound analysis techniques (see also our experimental results in Section 5.2.1) is due to the massive deployment of general-purpose reasoning machinery.

Our work aims at a more scalable and more predictable solution to bound analysis. We take an orthogonal approach based on scalable static analysis techniques which complement previous research.

Further, our discussion in Section 6.4 shows, and our experimental results in Section 5.2.3 demonstrate that existing approaches have difficulties to infer tight bounds for a class of nested loop constructs which we identified during our experiments on real-world code. What the loop patterns in this class have in common is that the *amortized* worst-case cost of an inner loop, i.e., the cost of executing an inner loop averaged over the executions of its outer loop(s), is lower than the worst-case cost of a single execution of that inner loop (for an example see Section 1.4.1 and Section 1.4.7): [CHS15] is limited to *linear* bounds, [ADFG10] easily fails in case of *polynomial* bounds (see discussion in Section 6.4), [BEF+16, ZGSV11, FH14] often fail to infer *tight* bounds (for [BEF+16] and [FH14] this can be also be seen from the experimental results in Section 5.2.3).

In contrast, the approach we present in this work succeeds in inferring tight bounds for challenging cases (consider, e.g., Example `xnu` in Figure 1.4) while not being limited to the linear case. On the contrary, the same reasoning which infers tight *linear* bounds is also applied for inferring tight *polynomial* bounds (Consider, e.g., Figure 1.6 and Figure 6.1, page 127).

1.6.7 Relation to Our Previous Publications

We took first steps towards an adequate abstract program model for *bound analysis* in [ZGSV11]. [ZGSV11] proposes a bound analysis based on so-called *size-change constraints* of form $x' \lhd y$, where $\lhd \in \{<, \leq\}$.

The basic insights on which we elaborate in this thesis were published for the first time in [SZV14a] and [SZV15] and [SZV16].

In [SZV14a] we proposed a bound analysis based on constraints of the form $x' \leq x + c$, where c is either an integer or a symbolic constant. A system modeled by such constraints is called a parametrized *lossy vector addition system with states* (VASS) in the literature. However, [SZV14a] cannot model resets. As a result [SZV14a] cannot, e.g., infer the linear complexity of Example `xnu` (Figure 1.4). In [SZV15] we generalized our abstract program model to *difference constraints*, which allow to model counter resets and increments as discussed in Section 1.4.2. In [SZV16] we elaborate on the approach of [SZV15] by discussing the algorithm in more detail, stating additional experimental results and providing essential soundness proofs.

This thesis goes beyond [SZV14a, SZV15, SZV16]. In particular, a path-sensitive version of our algorithm (Section 3.8) is presented for the first time. Moreover, as we discuss in Section 6.4, our path-sensitive algorithm solves a strict superset of abstract programs compared to the VASS-based approach in [SZV14a]. For the first time we state the full soundness proof of our bound algorithm, including a proof of soundness of our path-sensitive reasoning (Chapter 7). The presentation of our work is, in many aspects, much more detailed than in [SZV14a] and [SZV15] and [SZV16]: We provide important technical insights regarding program abstraction (Section 4.1) and our main algorithm (Section 4.2). We hope that our discussions of challenging examples, which we present in this work, will help the reader to gain a deeper understanding of our approach and its potential, thereby further populating our core insights in the research community.

1.7 Contributions

The main contributions we make to the field of *automated complexity and resource bound analysis* are the following:

1. We demonstrate that *difference constraints* are a suitable abstract program model for automatic complexity and resource bound analysis. We develop appropriate techniques for abstracting imperative programs to *DCPs* in Section 2.2.

2. Our approach handles bound analysis problems of high practical relevance which current approaches cannot handle as we demonstrate by our experiments in Section 5.2. The results on a benchmark of challenging iteration patterns, which we found in real-world code, are presented in Section 5.2.3. We state, and discuss in detail, a number of examples in Chapter 3 and Chapter 4.

3. Our approach is more *scalable* than existing approaches. We give a detailed discussion in Section 6.1. The claim is further supported by our experimental comparison on real C code in Section 5.2.1.

4. At the same time, our approach is general and can handle most of the bound analysis problems which are discussed in the literature. This claim is supported by our experimental comparison on examples from the literature on bound analysis in Section 5.2.2.

5. Our approach is rigorous, we prove soundness of our bound algorithm, the reasoning on chained resets and the path-sensitive reasoning in Section 7.

We give a detailed discussion on our contributions in Chapter 6.

CHAPTER 2

Program Model and Abstraction

Definition 1 (Program)**.** *Let* Σ *be a set of* states*. A* program *over* Σ *is a directed labeled graph* $\mathcal{P} = (L, T, l_b, l_e)$*, where* L *is a finite set of* locations*,* $l_b \in L$ *is the entry location,* $l_e \in L$ *is the exit location and* $T \subseteq L \times 2^{\Sigma\times\Sigma} \times L$ *is a finite set of* transitions*. We write* $l_1 \xrightarrow{\lambda} l_2$ *to denote a transition* $(l_1, \lambda, l_2) \in T$*. We call* $\lambda \in 2^{\Sigma\times\Sigma}$ *a* transition relation*. A* path *of* $\mathcal{P}$ *is a sequence* $l_0 \xrightarrow{\lambda_0} l_1 \xrightarrow{\lambda_1} \cdots$ *with* $l_i \xrightarrow{\lambda_i} l_{i+1} \in E$ *for all* i*. A* run *of* $\mathcal{P}$ *is a sequence* $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ *such that* $l_b \xrightarrow{\lambda_0} l_1 \xrightarrow{\lambda_1} \cdots$ *is a path of* $\mathcal{P}$ *and for all* $0 < i$ *it holds that* $(\sigma_{i-1}, \sigma_i) \in \lambda_{i-1}$*.* ρ *is* complete *if it ends at* l_e*.*

Note that a *run* of $\mathcal{P} = (L, T, l_b, l_e)$ starts at location l_b. We will need the notion of a *complete* run only later in Section 3.8 when we discuss our path-sensitive bound algorithm.

Further note that we call an *edge* $l_1 \xrightarrow{\lambda} l_1 \in T$ of the program a *transition*, whereas λ is its *transition relation*. In the following we will refer to *transitions* by τ and to *transition relations* by λ.

Formal Program Representation. We use *labeled transition systems* (LTS) to represent programs. Figure 2.1 (b) shows the LTS representation of Example `xnu` (discussed in Section 1.4.7). (In the LTS of Example `xnu` we use l as a shorthand for the program variable *len*, accordingly b for *beg* and e for *end*.) Each edge in an LTS is labeled by a formula which encodes the transition relation. We define the semantic of an LTS in compliance with Definition 1 as follows: We assume that the state of the program is described by a mapping which assigns each variable its current value. Let $\sigma \in \Sigma$. By $\sigma(x)$ we denote the value of variable x in state σ. We further assume that variables take values over the *integers*. Consider, e.g., the edge from l_1 to l_2 in Figure 2.1 (b). It is labeled by the formula:

$$i < l \wedge b' = b \wedge e' = e \wedge i' = i + 1 \wedge l' = l$$

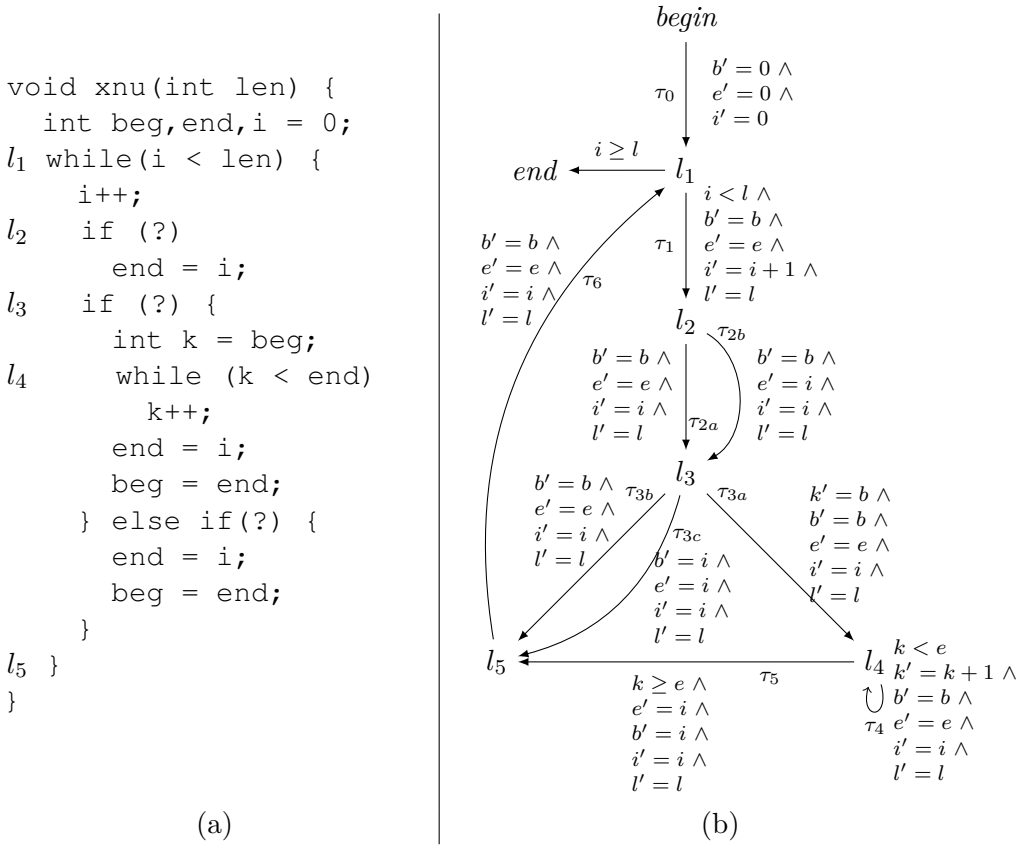


Figure 2.1: (a) Example `xnu` (b) Formal representation of `xnu` by an LTS

This formula encodes the transition relation $\lambda_1 =$

$$\{(\sigma, \sigma') \in \Sigma \times \Sigma \mid \sigma(i) < \sigma(l) \wedge \sigma'(b) = \sigma(b) \wedge \sigma'(e) = \sigma(e) \wedge \sigma'(i) = \sigma(i) + 1 \wedge \sigma'(l) = \sigma(l)\}.$$

2.1 Difference Constraint Programs

As discussed introductory, we base our algorithm on the abstract program model of *difference constraint programs* which we now formally define in Definition 4.

Definition 2 (Variables, Symbolic Constants, Atoms)**.** *By $\mathcal{V}$ we denote a finite set of* variables*. By $\mathcal{C}$ we denote a finite set of symbolic* constants*. $\mathcal{A} = \mathcal{V} \cup \mathcal{C}$ is the set of* atoms*.*

Definition 3 (Difference Constraints)**.** *A* difference constraint *over $\mathcal{A}$ is an inequality of form $x' \leq y + c$ with $x \in \mathcal{V}$, $y \in \mathcal{A}$ and $c \in \mathbb{Z}$. By $\mathcal{DC}(\mathcal{A})$ we denote the set of all difference constraints over $\mathcal{A}$.*

Notation: We often write $x' \leq y$ as a shorthand for the difference constraint $x' \leq y + \mathsf{c}$.

Definition 4 (Difference Constraint Program, Syntax)**.** *A* difference constraint program *(DCP) over $\mathcal{A}$ is a directed labeled graph $\Delta\mathcal{P} = (L, E, l_b, l_e)$, where L is a finite set of vertices, $l_b \in L$ and $l_e \in L$ and $E \subseteq L \times 2^{\mathcal{DC}(\mathcal{A})} \times L$ is a finite set of edges. We write $l_1 \xrightarrow{u} l_2$ to denote an edge $(l_1, u, l_2) \in E$ labeled by a set of difference constraints $u \in 2^{\mathcal{DC}(\mathcal{A})}$. We use the notation $l_1 \to l_2$ to denote an edge that is labeled by the empty set of difference constraints. $\Delta\mathcal{P}$ is* fan-in-free*, if for every edge $l_1 \xrightarrow{u} l_2 \in E$ and every $\mathtt{v} \in \mathcal{V}$ there is at most one $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u$.*

Example: Figure 2.2 (b) shows a fan-in free *DCP*.

Definition 5 (Difference Constraint Program, Semantics)**.** *The set of* valuations *of $\mathcal{A}$ is the set $Val_\mathcal{A} = \mathcal{A} \to \mathbb{N}$ of mappings from $\mathcal{A}$ to the natural numbers. Let $u \in 2^{\mathcal{DC}(\mathcal{A})}$. We define $[\![u]\!] \in 2^{(Val_\mathcal{A} \times Val_\mathcal{A})}$ s.t. $(\sigma, \sigma') \in [\![u]\!]$ iff for all $x' \leq y + c \in u$ it holds that (i) $\sigma'(x) \leq \sigma(y) + c$ and (ii) for all $\mathtt{s} \in \mathcal{C}$ $\sigma'(\mathtt{s}) = \sigma(\mathtt{s})$. A DCP $\Delta\mathcal{P} = (L, E, l_b, l_e)$ is a* program *over the set of states $Val_\mathcal{A}$ with locations L, entry location l_b, exit location l_e and transitions $T = \{l_1 \xrightarrow{[\![u]\!]} l_2 \mid l_1 \xrightarrow{u} l_2 \in E\}$.*

I.e., a *DCP* is a *program* (Definition 1) whose *transition relations* are solely specified by conjunctions of *difference constraints*.

Note that variables in difference constraint programs take values only over the *natural numbers*.

Further note that by u we refer to the syntactic representation of the transition relation in form of a set of difference constraints, whereas by $[\![u]\!]$ we refer to the transition relation itself.

Definition 6 (Well-defined *DCP*)**.** *Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a DCP.*
We say a variable x is defined at *l if $x \in \mathtt{def}(l)$, where $\mathtt{def} : L \to 2^\mathcal{A}$ is defined by*
$\mathtt{def}(l) = \bigcap_{l_1 \xrightarrow{u} l \in E} \{x \mid \exists y \in \mathcal{V}\ \exists \mathtt{c} \in \mathbb{Z}\ s.t.\ x' \leq y + \mathtt{c} \in u\} \cup \mathcal{C}$.

We say a variable x is used at *l if $x \in \mathtt{use}(l)$, where $\mathtt{use} : L \to 2^\mathcal{A}$ is defined by*
$\mathtt{use}(l) = \bigcup_{l \xrightarrow{u} l_1 \in E} \{y \mid \exists x \in \mathcal{A}\ \exists \mathtt{c} \in \mathbb{Z}\ s.t.\ x' \leq y + \mathtt{c} \in u\}$.

$\Delta\mathcal{P}$ is well-defined *iff l_b has no incoming edges and for all $l \in L$ it holds that $\mathtt{use}(l) \subseteq \mathtt{def}(l)$.*

Discussion. A program $\Delta\mathcal{P}$ is well-defined if l_b has no incoming edges and for all $\mathtt{v} \in \mathcal{V}$ it holds that $\mathtt{v}$ is defined at all locations at which $\mathtt{v}$ is used (symbolic constants are always defined). Note that for well-defined programs we in particular require $\mathtt{use}(l_b) \subseteq \mathtt{def}(l_b)$. Because l_b has no incoming edges we have $\mathtt{def}(l_b) = \mathcal{C}$. Thus only symbolic constants can be used at l_b.

Throughout this work we will only consider *DCP*s that are *fan-in free* and *well-defined*.

2.2 Program Abstraction

In the following we discuss how we abstract a given program to a *DCP*.

Definition 7 (Difference Constraint Invariants)**.** *Let $\mathcal{P}(L, T, l_e, l_b)$ be a* program *over states Σ. Let $e_1, e_2, e_3 \in \Sigma \to \mathbb{Z}$, and let $c \in \mathbb{Z}$ be some integer. We say $e_1' \leq e_2 + e_3$ is* invariant on *a transition $l_1 \xrightarrow{\lambda} l_2 \in T$, if $e_1(\sigma_2) \leq e_2(\sigma_1) + e_3(\sigma_1)$ holds for all $(\sigma_1, \sigma_2) \in \lambda$.*

Definition 8 (*DCP* Abstraction of a Program)**.** *Let $\mathcal{P} = (L, T, l_b, l_e)$ be a program and let N be a set of fucntions from the states to the natural numbers, i.e., $N \in 2^{\Sigma \to \mathbb{N}}$. A DCP $\Delta\mathcal{P} = (L, E, l_b, l_e)$ over atoms N is an abstraction of the program $\mathcal{P}$ iff for each transition $l_1 \xrightarrow{\lambda} l_2 \in T$ there is a transition $l_1 \xrightarrow{u} l_2 \in E$ s.t. every $e_1' \leq e_2 + c \in u$ is invariant on $l_1 \xrightarrow{\lambda} l_2$.*

Our abstraction mechanism proceeds in two steps: We first abstract a given concrete program to a *DCP* with integer semantics, in a second step we then further abstract the integer-*DCP* to a *DCP* over the natural numbers (in compliance with Definition 4).

2.2.1 Abstraction I: *DCP*s with Integer Semantics

We extend our abstract program model from Definition 4 to the non-well-founded domain $\mathbb{Z}$ by adding guards to the transitions of the program.

Syntax of *DCP*s with guards. The edges E of a *DCP with guards* $\Delta\mathcal{P}_G(L, E, l_b, l_e)$ are a subset of $L \times 2^{\mathcal{V}} \times 2^{\mathcal{DC}(\mathcal{A})} \times L$. I.e., an edge of a *DCP with guards* is of form $l_1 \xrightarrow{g,u} l_2$ with $l_1, l_2 \in L$, $g \in 2^{\mathcal{V}}$ and $u \in 2^{\mathcal{DC}(\mathcal{A})}$.

Example: See Figure 2.2 (a) on page 34 for an example.

Semantics of *DCP*s with guards. We extend the range of the *valuations* $Val_{\mathcal{A}}$ of $\mathcal{A}$ from $\mathbb{N}$ to $\mathbb{Z}$. Let $u \in 2^{\mathcal{DC}(\mathcal{A})}$. Let $[\![u]\!]$ be as defined in Definition 5. Let $g \in 2^{\mathcal{V}}$. We define $[\![g, u]\!] = \{(\sigma_1, \sigma_2) \in [\![u]\!] \mid \sigma_1(\mathtt{v}) > 0 \text{ for all } \mathtt{v} \in g\}$. A *guarded DCP* $\Delta\mathcal{P}_G = (L, E, l_b, l_e)$ is a *program* over the set of states $Val_{\mathcal{A}}$ with locations L, entry location l_b, exit location l_e and transitions $T = \{l_1 \xrightarrow{[\![g,u]\!]} l_2 \mid l_1 \xrightarrow{g,u} l_2 \in E\}$.

I.e., a transition $l_1 \xrightarrow{g,u} l_2$ of a *DCP with guards* can only be executed if the values of all $\mathtt{v} \in g$ are greater than 0.

Definition 9 (Norm)**.** *Let Σ be a set of* states*. A* norm *$e : \Sigma \to \mathbb{Z}$ over Σ is a function that maps the states to the integers.*

Definition 10 (Guard)**.** *Let $\mathcal{P}(L, T, l_e, l_b)$ be a* program *over states Σ. Let e be a norm, let $c \in \mathbb{Z}$. We say e is a* guard *of $l_1 \xrightarrow{\lambda} l_2 \in T$ if $e(\sigma_1) > 0$ holds for all $(\sigma_1, \sigma_2) \in \lambda$.*

We abstract a program $\mathcal{P} = (L, T, l_b, l_e)$ to a *DCP* with guards $\Delta\mathcal{P}_G = (L, E, l_b, l_e)$ as follows:

1. **Choosing an initial set of Norms.** We aim at creating a suitable abstract program for bound analysis. In our non-recursive setting complexity evolves from iterating loops. Therefore we search for expressions which limit the number of loop iterations. We consider conditions of form $a > b$ resp. $a \geq b$ found in loop headers or on loop-paths if they involve loop counter variables, i.e., variables which are incremented and/or decremented inside the loop. Such conditions are likely to limit the consecutive execution of single or multiple loop-paths. From each condition of form $a > b$ we create the integer expression $a - b$, from each condition of form $a \geq b$ we create the integer expression $a + 1 - b$. These expressions form our initial set of norms N. Note that on those transitions on which $a > b$ holds, $a - b > 0$ must hold, whereas with $a \geq b$ we have $a + 1 - b > 0$.
   In $\Delta\mathcal{P}_G$ we interpret a norm $e \in N$ from our initial set of norms N as *variable*, i.e., we have $e \in \mathcal{V}$ for all $e \in N$.

2. **Abstracting Transitions.** For each transition $l_1 \xrightarrow{\lambda} l_2 \in T$ we generate a set u_λ of difference constraints: Initially we set $u_\lambda = \emptyset$ for all transitions $l_1 \xrightarrow{\lambda} l_2 \in T$.

   We repeat the following construction *until* the set of norms N becomes *stable*:

   For each $e_1 \in N$ and for each $l_1 \xrightarrow{\lambda} l_2 \in T$, such that all variables in e_1 are defined at l_2, we check whether there is a difference constraint of form $e_1' \leq e_2 + \mathtt{c}$ with $e_2 \in N$ and $\mathtt{c} \in \mathbb{Z}$ in u_λ. If not, we derive a difference constraint $e_1' \leq e_2 + \mathtt{c}$ as follows: We symbolically execute λ for deriving e_1' from e_1: E.g., let $e_1 = x + y$ and assume x is assigned $x+1$ on $l_1 \xrightarrow{\lambda} l_2$ while y stays unchanged. We get $e_1' = x+1+y$ through symbolic execution. In order to keep the number of norms low, we first try

   a) to find a norm $e_2 \in N$ and $\mathtt{c} \in \mathbb{Z}$ s.t. $e_1' \leq e_2 + \mathtt{c}$ is invariant on $l_1 \xrightarrow{\lambda} l_2$ (see Definition 7). If we succeed we add the predicate $e_1' \leq e_2 + \mathtt{c}$ to u_λ. E.g., for $e_1 = x + y$ and $e_1' = x + 1 + y$ we get the transition invariant $(x + y)' \leq (x + y) + 1$ and will thus add $e_1' \leq e_1 + 1$ to u_λ. In general, we find a norm e_2 and a constant $\mathtt{c}$ by separating constant parts in the expression e_1' using associativity and commutativity, thereby forming an expression e_3 over variables and program parameters and an integer constant $\mathtt{c}$. We then search a norm $e_2 \in N$ with $e_2 = e_3$ where the check on equalitiy is performed modulo associativity and commutativity.

   b) If a) fails, i.e., no such $e_2 \in N$ exists, we add e_3 to N and derive the predicate $e_1' \leq e_3 + \mathtt{c}$. In $\Delta\mathcal{P}_G$ we interpret e_3 as atom, i.e., $e_3 \in \mathcal{A}$. E.g., given $e_1' = 5 + z$ we set $e_3 = z$ and $\mathtt{c} = 5$. We interpret e_3 as a symbolic constant, i.e., $e_3 \in \mathcal{C}$, only if e_3 is purely built over the programs input parameters and constants. Note that this step increases the number of norms.

3. **Inferring Guards** For each transition $l_1 \xrightarrow{\lambda} l_2$ we generate a set g_λ of guards: Initially we set $g_\lambda = \emptyset$ for all transitions $l_1 \xrightarrow{\lambda} l_2$. For each $e \in N$ and each transition $l_1 \xrightarrow{\lambda} l_2$ we check if e is a *guard* of $l_1 \xrightarrow{\lambda} l_2$. If so, we add e to g_λ. We use an SMT solver to perform this check. E.g., let $e = x + y$ and assume that $l_1 \xrightarrow{\lambda} l_2$ is guarded by the conditions $x \geq 0$ and $y > x$. An SMT solver supporting *linear arithmetic* proves that $x \geq 0 \wedge y > x$ implies $x + y > 0$ and we thus add $x + y$ to g_λ.

4. We set $E = \{l_1 \xrightarrow{g_\lambda, u_\lambda} l_2 \mid l_1 \xrightarrow{\lambda} l_2 \in T\}$.

Note that SMT solver reasoning is applied only *locally* to single transitions to check if an expression is greater than 0 on that transition.

Propagation of Guards. We improve the precision of our abstraction by *propagating guards*: Consider a transition $l_3 \xrightarrow{g_3, u_3} l_4$. Assume l_3 has the incoming edges $l_1 \xrightarrow{g_1, u_1} l_3$ and $l_2 \xrightarrow{g_2, u_2} l_3$. If $y \in g_1 \cap g_2$ (i.e., y is a guard on both incoming edges) *and* y does not decrease on the corresponding concrete transitions $l_1 \xrightarrow{\lambda_1} l_3$ and $l_1 \xrightarrow{\lambda_2} l_3$ (checked by symbolic execution) then y is also a guard on $l_3 \xrightarrow{g_3, u_3} l_4$ and we add y to g_3.

Well-defined and Fan-in free. *DCP*s generated by our algorithm are always *fan-in free* by construction: For each transition we get at most one predicate $e' \leq e_2 + \mathtt{c}$ for each $e \in N$: We check whether there is already a predicate for e before a predicate is inferred resp. added. We ensure *well-definedness* of our abstraction by a final *clean-up*: We iterate over all $l \in L$ and check if $\mathtt{use}(l) \subseteq \mathtt{def}(l)$ holds. If this check fails we remove all difference constraints $x' \leq y + \mathtt{c}$ with $y \in \mathtt{use}(l) \setminus \mathtt{def}(l)$ from all outgoing edges of l. We repeat this iteration until well-definedness is established, i.e., until $\mathtt{use}(l) \subseteq \mathtt{def}(l)$ holds for all $l \in L$.

Termination. We have to ensure the termination of our abstraction procedure, since case b) in step "*2. Abstracting Transitions*" triggers a recursive abstraction for the newly added norm: Note that we can always stop the abstraction process at any point, getting a sound abstraction of the original program. We therefore ensure termination of the abstraction algorithm by limiting the chain of recursive abstraction steps that is triggered by entering case 2.b).

Non-linear Iterations. We can handle counter updates such as $x' = 2x$ or $x' = x/2$ as follows: 1) We add the expression $\log x$ to our set of norms. 2) We derive the difference constraint $(\log x)' \leq (\log x) - 1$ from the update $x' = x/2$ if $x > 1$ holds. Symmetrically we get $(\log x)' \leq (\log x) + 1$ from the update $x' = 2x$ if $x > 0$ holds.

Data Structures. In previous publications [GLAS09, MTLT10] it has been described how to abstract programs with data structures to pure integer programs by making use of appropriate *norms* such as the length of a list or the number of elements in a tree. In

our implementation we follow these approaches using a light-weight abstraction based on optimistic aliasing assumptions (see Section 5.1). Once the program is transformed to an integer program, our abstraction algorithm is applied as described above for obtaining a difference constraint program.

2.2.2 Abstraction II: From the Integers to the Natural Numbers

We now discuss how we abstract a *DCP with guards* $\Delta\mathcal{P}_G = (L, E, l_b, l_e)$ to a *DCP* $\Delta\mathcal{P} = (L, E', l_b, l_e)$ over $\mathbb{N}$ (Definition 4):

Let $e \in N$. By $[e] : \Sigma \to \mathbb{N}$ we denote the function $[e](\sigma) = \max(e(\sigma), 0)$. Recall that e is interpreted as *atom* in $\Delta\mathcal{P}_G$, i.e., $e \in \mathcal{A}$. In $\Delta\mathcal{P}$, we interpret $[e]$ as *variable* (i.e., $[e] \in \mathcal{V}$) if $e \in \mathcal{V}$. We interpret $[e]$ as *symbolic constant* (i.e., $[e] \in \mathcal{C}$) if $e \in \mathcal{C}$.

Let $l_1 \xrightarrow{g,u} l_2 \in E$. We create a transition $l_1 \xrightarrow{u'} l_2 \in E'$ as follows: Let $e_1' \leq e_2 + \mathtt{c} \in u$. If $\mathtt{c} \geq 0$, we add $[e_1]' \leq [e_2] + \mathtt{c}$ to u'. If $\mathtt{c} < 0$ and $e_2 \in g$ we add the constraint $[e_1]' \leq [e_2] - 1$ to u'. If $\mathtt{c} < 0$ and $e_2 \notin g$ we add the constraint $[e_1]' \leq [e_2] + 0$ to u'.

Discussion. Soundness of Abstraction II is due to the following observation: Consider a transition $l_1 \xrightarrow{g,u} l_2$ of $\Delta\mathcal{P}_G$. Let $e_1' \leq e_2 + \mathtt{c} \in u$, i.e., $e_1' \leq e_2 + \mathtt{c}$ is *invariant* for the corresponding transition τ of the concrete program. Then $[e_1]' \leq [e_2] + 0$ is also invariant for τ. Further: If $\mathtt{c} \geq 0$ then $[e_1]' \leq [e_2] + \mathtt{c}$ is invariant for τ. And if $\mathtt{c} < 0$ and $e_2 \in g$ (i.e., $e_2 > 0$ must hold before executing τ), then $[e_1]' \leq [e_2] - 1$ is invariant for τ.

2.3 Example

We exemplify our abstraction algorithm on Example `xnu`. Consider the LTS representation of Example `xnu` in Figure 2.1 (b), we use l as a shorthand for the program variable *len*, accordingly b for *beg* and e for *end*.

We first apply abstraction step I discussed in Section 2.2.1:

1. **Choosing an initial set of Norms.** Our described heuristic adds the expressions $l - i$ and $e - k$ generated from the conditions $k < e$ and $i < l$ to the initial set of norms N. Thus our initial set of norms is $N = \{l - i, e - k\}$.
2. **Abstracting Transitions.**
   - We check how $l - i$ changes on the transitions $\tau_0, \tau_1, \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, , \tau_{3c}, \tau_4, \tau_5, \tau_6$:
     - τ_0: we derive $(l - i)' \leq l$ (reset), we add l to N. Since l is an input parameter we have $l \in \mathcal{C}$.
     - τ_1: we derive $(l - i)' \leq (l - i) - 1$ (decrement)
     - $\tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_{3c}, \tau_4, \tau_5, \tau_6$: $l - i$ unchanged
   - We check how $e - k$ changes on the transitions τ_{3a}, τ_4 (k is only defined at l_4):

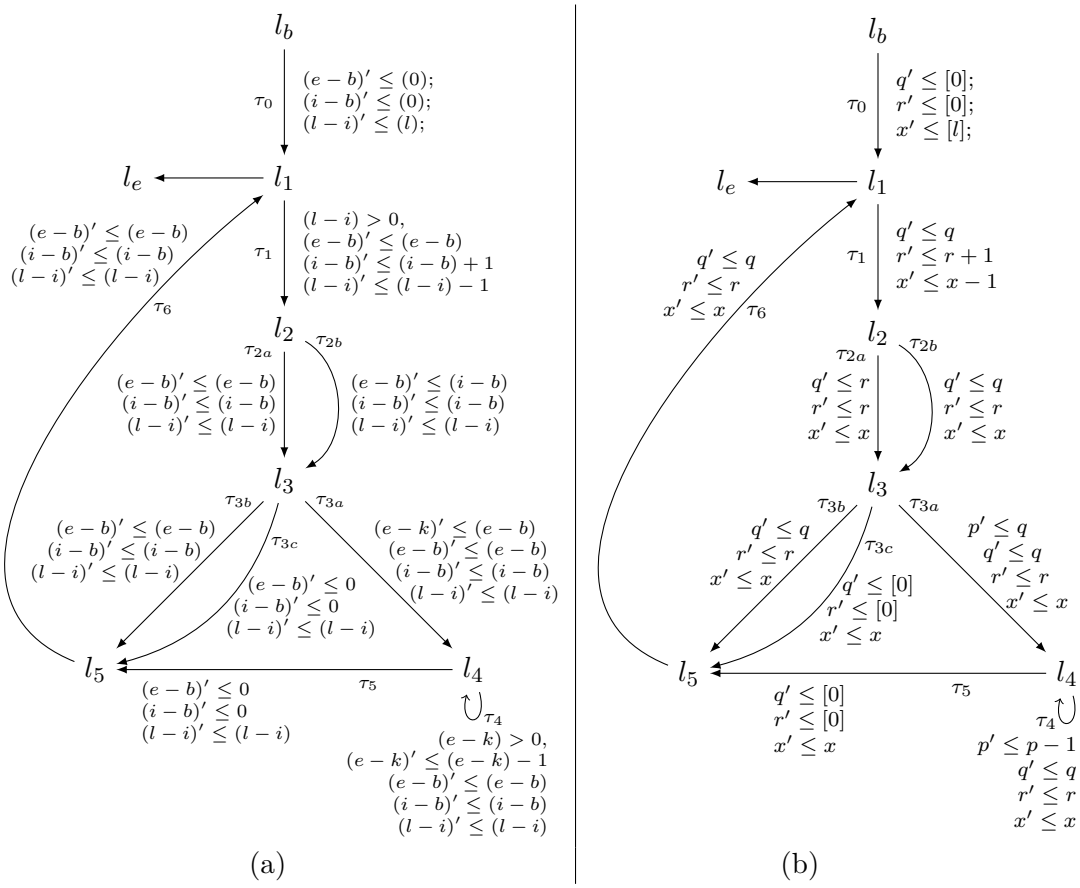


Figure 2.2: (a) Abstraction I: *DCP* with guards obtained from Example `xnu` (b) Abstraction II: *DCP* obtained from Example `xnu`; where $p, q, r, x \in \mathcal{V}$ and $[e-k] = p$, $[e-b] = q$, $[i-b] = r$, $[l-i] = x$

  - τ_{3a}: we derive $(e-k)' \leq (e-b)$ (reset), we add $e-b$ to N
  - τ_4: we derive $(e-k)' \leq (e-k)-1$ (decrement)
- We check how $e-b$ changes on the transitions $\tau_0, \tau_1, \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_{3c}, \tau_4, \tau_5, \tau_6$:
  - τ_0: we derive $(e-b)' \leq 0$ (reset), we add 0 to N. Since 0 is a constant we have $0 \in \mathcal{C}$.
  - τ_{2a}: we derive $(e-b)' \leq (i-b)$, we add $i-b$ to N.
  - τ_{3c}: we derive $(e-b)' \leq 0$ (reset)
  - τ_5: we derive $(e-b)' \leq 0$ (reset)
  - $\tau_1, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_4, \tau_6$: $e-b$ unchanged
- We check how $i-b$ changes on the transitions $\tau_0, \tau_1, \tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_{3c}, \tau_4, \tau_5, \tau_6$:
  - τ_0: we derive $(i-b)' \leq 0$ (reset)
  - τ_1: we derive $(i-b)' \leq (i-b)+1$ (increment)

  - τ_{3c}: we derive $(i-b)' \leq 0$ (reset)
  - τ_5: we derive $(i-b)' \leq 0$ (reset)
  - $\tau_{2a}, \tau_{2b}, \tau_{3a}, \tau_{3b}, \tau_4, \tau_6$: unchanged
- We have processed all norms in N

3. **Inferring Guards.** We add the guard $(l-i)$ to τ_1 in $\Delta\mathcal{P}_G$ because $l-i$ is a *guard* of τ_1 in $\mathcal{P}$ (due to the condition $i < l$), we add the guard $(e-k)$ to τ_4 in $\Delta\mathcal{P}_G$ because $e-k$ is a *guard* of τ_4 in $\mathcal{P}$ (due to the condition $k < e$).

4. The resulting *DCP* with guards is shown in Figure 2.2 (a).

Applying abstraction step II discussed in Section 2.2.2 gives us the *DCP* shown in Figure 2.2 (b). In the depiction of the abstraction we assume $p, q, r, x \in \mathcal{V}$ and $[e-k] = p$, $[e-b] = q$, $[i-b] = r$, $[l-i] = x$.

CHAPTER 3

Algorithm

In this chapter we develop our bound algorithm for *DCP*s step-by-step. We first present a bound algorithm for the special case of *vector addition systems* in Section 3.2 (motivated in Section 1.4.1). We then generalize this algorithm to *DCP*s with only constant resets (Section 3.3). In Section 3.4 we finally generalize our algorithm to full *DCP*s (motivated in Section 1.4.1).

We present our reasoning based on *chained resets* (consecutive resets of counters $c_n, c_{n-1}, \ldots c_1$ such that c_1 is set to c_2 which is set to c_3 ,etc.), motivated in Section 1.4.7, in Section 3.5.

In Section 3.6 we present our analysis from inferring *local bounds*.

In Section 3.7 we state a complete example.

In Section 3.8 we discuss our *path-sensitive reasoning*.

In Section 3.9 we combine the presented ideas to form our full bound algorithm.

In Section 3.10 we discuss how our reasoning can be instrumented for computing *resource bounds*. In Section 3.10.1 we give further details on how bounds on *memory consumption* or memory-like resources can be inferred by our analysis.

In Section 3.11 we reflect on the relation of our analysis to classical *invariant analysis*.

In Section 3.12 we discuss how our reasoning is connected to classical *amortized complexity analysis*.

We start by sketching the principal idea that is underlying our analysis.

Reconsider Tarjan's example for *amortized complexity* discussed in Section 1.4. We restate the example in Figure 3.1 (a). Assume we want to compute a bound on the overall cost of the *pop* operation, modeled by the inner loop (see discussion in Section 1.4).

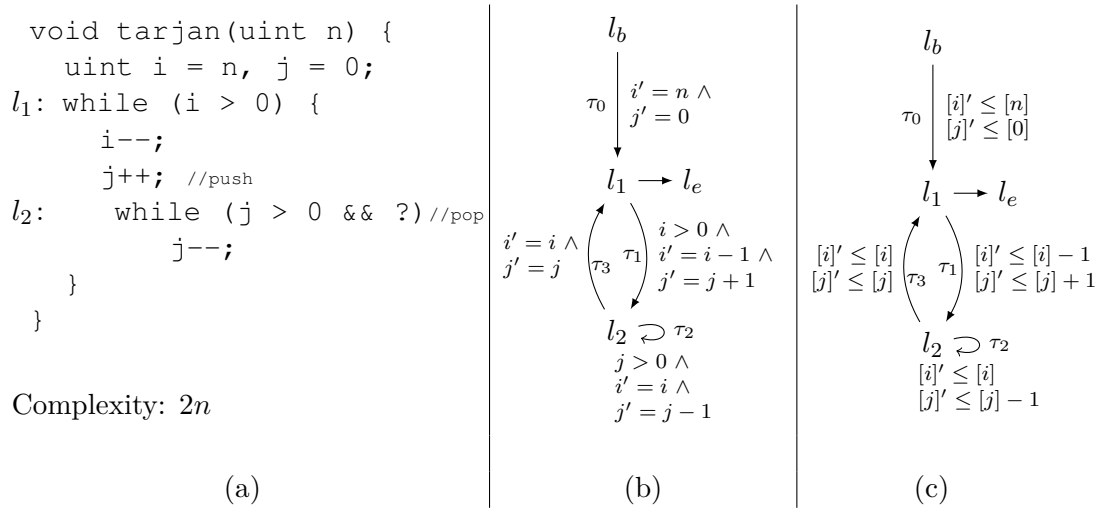


Figure 3.1: Example `tarjan` (b) LTS of Example `tarjan` (c) *DCP* obtained by abstraction from `tarjan`

Local Bound. Note that j decreases with each execution of the inner loop and that the inner loop can only be repeated if $j > 0$ holds. Thus the number of consecutive executions of the inner loop is limited by the value of j. j is, however, not a *bound* on the *overall* executions of the inner loop, because the value of j is incremented on the outer loop. We therefore call j a *local bound*.

Bound Computation. Our algorithm, which we present next, infers a *bound* on the overall number of executions of the inner loop by reasoning *how often* and by *how much* the inner loop's *local bound* j may increase during program run. Since j is incremented by 1 on the outer loop, this reasoning triggers a recursive bound computation for the outer loop. The bound n of the outer loop multiplied by the increment 1 of j then leads to the loop bound n for the inner loop. By this reasoning, we conclude that the overall cost of the push operation is limited by n.

3.1 Formal Problem Statement and Basic Definitions

Transition bounds are at the core of our analysis: We infer bounds on the number of loop iterations, on computational complexity, on resource consumption, etc., by computing bounds on the number of times that one or several transitions can be executed.

Before we formally define our notion of a *transition bound* we have to introduce some notation.

Definition 11 (Counter Notation I)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over Σ. Let $\tau \in T$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $\mathcal{P}$. By $\sharp(\tau, \rho)$ we denote the number of times that τ occurs on ρ.*

In the following, we denote by '∞' a value s.t. $a < \infty$ for all $a \in \mathbb{Z}$ (infinity).

Definition 12 (Transition Bound)**.** *Let $\mathcal{P} = (L, T, l_b, l_e)$ be program over states Σ. Let $\tau \in T$. A value $\mathtt{b} \in \mathbb{N}_0 \cup \{\infty\}$ is a bound for τ on a run $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} (l_2, \sigma_2) \xrightarrow{\lambda_2} \dots$ of $\mathcal{P}$ iff $\sharp(\tau, \rho) \leq \mathtt{b}$, i.e., iff τ appears not more than $\mathtt{b}$ times on ρ. A function $\mathtt{b} : \Sigma \to \mathbb{N}_0 \cup \{\infty\}$ is a* bound *for τ iff for all runs ρ of $\mathcal{P}$ it holds that $\mathtt{b}(\sigma_0)$ is a bound for τ on ρ, where σ_0 denotes the initial state of ρ.*

Our aim is to design a procedure $\mathtt{TB}(\tau)$ which, given a program transition τ, returns a *bound* for τ. If possible, the bound computed by our procedure should be *precise* or *tight*, in particular the trivial bound $\Sigma \to \infty$ is (most often) of no value to us.

Definition 13 (Precise Transition Bound)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over states Σ. Let $\tau \in T$. We say that a transition bound $\mathtt{b} : \Sigma \to \mathbb{N}_0 \cup \{\infty\}$ for τ is* precise *iff for each $\sigma_0 \in \Sigma$ there is a run $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ such that $\sharp(\tau, \rho) = \mathtt{b}(\sigma_0)$.*

Informally: A transition bound is *precise* if it can be reached for all initial states σ_0. Note that there is only exactly *one* precise transition bound.

Definition 14 (Tight Transition Bound)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over states Σ. Let $\tau \in T$. We say that a transition bound $\mathtt{b} : \Sigma \to \mathbb{N}_0 \cup \{\infty\}$ is* tight *iff there is a $c > 0$ such that either 1) for all $\sigma \in \Sigma$ we have $\mathtt{b}(\sigma) < c$ ($\mathtt{b}$ is bounded), or 2) there is a family of states $(\sigma_i)_{i \in \mathbb{N}}$ with $\lim_{i \mapsto \infty} \mathtt{b}(\sigma_i) = \infty$ ($\mathtt{b}$ is unbounded) such that for all σ_i there is a run ρ_i starting in σ_i with $\mathtt{b}(\sigma_i) \leq c \times \sharp(\tau, \rho_i)$.*

Informally: A transition bound is *tight* if it is in the same *asymptotic class* as the *precise* transition bound: Let $\tau \in T$. For the special case $\Sigma = \mathbb{N}$ we have the following: Let $f : \mathbb{N} \to \mathbb{N}$ denote the *precise* transition bound for τ. Let $g : \mathbb{N} \to \mathbb{N}$ be *some* transition bound for τ. Trivially $f \in O(g)$ (f does not grow faster than g). Now, g is *tight* if also $f \in \Omega(g)$ (f does not always grow slower than g). With $f \in O(g)$ and $f \in \Omega(g)$ we have that $f \in \Theta(g)$. The same can be formulated for general state sets Σ by mapping Σ to the natural numbers.

We now formally define the notion *local bound*.

Definition 15 (Counter Notation II)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over Σ. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \cdots$ be a run of $\mathcal{P}$. Let $e : \Sigma \to \mathbb{Z}$ be a norm. By $\downarrow(e, \rho)$ we denote the number of times that the value of e decreases on ρ, i.e.,*
$\downarrow(e, \rho) = |\{i \mid e(\sigma_i) > e(\sigma_{i+1})\}|$.

Definition 16 (Local Bound)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over Σ. Let $\tau \in T$. Let $e : \Sigma \to \mathbb{N}$ be a* norm *that takes values in the* natural numbers*. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ be a run of $\mathcal{P}$. e is a* local bound *for τ on ρ if it holds that $\sharp(\tau, \rho) \leq \downarrow(e, \rho)$.*

Discussion. A natural number valued norm e is a *local bound* for τ on a run ρ if τ appears not more often on ρ than the number of times the value of e decreases. I.e., a *local* bound e for τ limits the number of executions of τ on a run ρ as long as certain program parts (those were e increases) are not executed. We argue in Section 3.12 that, in our analysis, *local bounds* play the role of *potential functions* in classical *amortized complexity analysis* [Tar85].

We discuss how we obtain local bounds in Section 3.6.

Following our methodological approach (Section 1.3) we design our algorithm on the level of our *abstract program model* of *difference constraint programs*. We discussed in Section 2.2 how we obtain *DCP*s from concrete programs.

Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. Our bound algorithm, which we start to develop in the next section, computes a *transition bound* for a given transition $\tau \in E$ in form of an expression over $\mathcal{A}$ which involves the operators $+,\times,/$,min,max and the floor function $\lfloor\cdot\rfloor$. Note, however, that the *norms*, which are treated as *atoms* (elements of $\mathcal{A}$) in the abstraction, can involve arbitrary operators (see Section 2.2).

Definition 17 (Expressions over $\mathcal{A}$)**.** *By $Expr(\mathcal{A})$ we denote the set of expressions over $\mathcal{A}\cup\mathbb{Z}\cup\infty$ that are formed using the arithmetical operators addition ($+$), multiplication ($\times$), division ($/$), maximum (*max*), minimum (*min*) and integer division of form $\lfloor\frac{\texttt{expr}}{\texttt{c}}\rfloor$ where $\texttt{expr} \in Expr(\mathcal{A})$ and $\texttt{c} \in \mathbb{N}$. The semantics function $[\![\cdot]\!] : Expr(\mathcal{A}) \to (Val_{\mathcal{A}} \to \mathbb{Z} \cup \{\infty\})$ evaluates an expression $\texttt{expr} \in Expr(\mathcal{A})$ over a state $\sigma \in Val_{\mathcal{A}}$ using the usual operator semantics (we have $a + \infty = \infty$, $\min(a, \infty) = a$, etc.).*

We now introduce the key ingredients of our bound algorithm.

Our bound algorithm computes a bound for a given transition $\tau \in E$ based on a mapping (called *local bound mapping*) which assigns each $\tau \in E$ either 1) a *bound* for τ in form of an expression over the symbolic constants ($\zeta(\tau) \in Expr(\mathcal{C})$) or 2) a *local bound* for τ in form of a variable ($\zeta(\tau) \in \mathcal{V}$). Note that $\mathcal{V} \cap Expr(\mathcal{C}) = \emptyset$. In case 1) our algorithm (Definition 22) returns $T\mathcal{B}(\tau) = \zeta(\tau)$. In case 2) a transition bound $T\mathcal{B}(\tau) \in Expr(\mathcal{C})$ is computed by inferring *how often* and by *how much* the local transition bound $\zeta(\tau) \in \mathcal{V}$ of τ may increase during program run.

Definition 18 (Local Bound Mapping)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ be a run of $\Delta\mathcal{P}$. We call a function $\zeta : E \to Expr(\mathcal{A})$ a* local bound mapping *for ρ if for all $\tau \in E$ it holds that either*
1) $\zeta(\tau) \in Expr(\mathcal{C})$ and $[\![\zeta(\tau)]\!](\sigma_0)$ is a bound *for τ on ρ or*
2) $\zeta(\tau) \in \mathcal{V}$ and $[\![\zeta(\tau)]\!]$ is a local bound *for τ on ρ.*
We say that ζ is a local bound mapping *for $\Delta\mathcal{P}$ if ζ is a local bound mapping for all runs of $\Delta\mathcal{P}$.*

Further, our bound algorithm is based on a syntactic distinction between two kinds of updates that may modify the value of a given variable $\texttt{v} \in \mathcal{V}$: We identify transitions which *increment* v and transitions which *reset* v.

Definition 19 (Resets and Increments)**.** *Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\mathtt{v} \in \mathcal{V}$. We define the* resets $\mathcal{R}(\mathtt{v})$ *and* increments $\mathcal{I}(\mathtt{v})$ *of $\mathtt{v}$ as follows:*

$\mathcal{R}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in E \times \mathcal{A} \times \mathbb{Z} \mid \mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u, \mathtt{a} \neq \mathtt{v}\}$

$\mathcal{I}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in E \times \mathbb{N} \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u, \mathtt{c} > 0\}$

Given a path π of $\Delta\mathcal{P}$ we say that $\mathtt{v}$ is reset *on π if there is a transition τ on π such that $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ for some $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$. We say that $\mathtt{v}$ is* incremented *on π if there is a transition τ on π such that $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ for some $\mathtt{c} \in \mathbb{N}$.*

I.e., we have that $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ if variable $\mathtt{v}$ is reset to a value smaller or equal to $\mathtt{a} + \mathtt{c}$ when executing the transition τ. Accordingly we have $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ if variable $\mathtt{v}$ is incremented by a value smaller or equal to $\mathtt{c}$ when executing the transition τ.

3.2 Bound Algorithm for Lossy Vector Addition Systems

Definition 20 (Monotone Difference Constraints.)**.** *Let $x' \leq y + \mathtt{c} \in \mathcal{DC}(\mathcal{A})$ be a difference constraint over $\mathcal{A}$. We say that $x' \leq y + \mathtt{c}$ is* monotone *if $x = y$. We denote the set of all monotone difference constraints over $\mathcal{V}$ by $\mathcal{MDC}(\mathcal{V})$.*

I.e., *monotone difference constraints* are a strict syntactic sub-class of *difference constraints*, we have $\mathcal{MDC}(\mathcal{V}) \subset \mathcal{DC}(\mathcal{A})$.

Definition 21 (Lossy Vector Addition System with States.)**.** *A DCP $\Delta\mathcal{P}(L, E, l_b, l_e)$ is a* lossy vector addition system with states *(VASS) iff (1) for all $l_1 \xrightarrow{u} l_2 \in E$ with $l_1 \neq l_b$ $u \in 2^{\mathcal{MDC}(\mathcal{V})}$, (2) there is exactly one $l_2 \in L$ s.t. $l_b \xrightarrow{u} l_2 \in E$.*

I.e., *lossy vector addition systems with states* (see, e.g., [BM99]) are a strict syntactic sub-class of *difference constraint programs*. Non-monotone difference constraints are allowed only on the single initial transition.

Example: The *DCP* shown in Figure 3.1 (b) is a VASS. Figure 3.1 (b) is obtained by our abstraction procedure (Chapter 2) from Figure 3.1 (a).

Definition 22 (Bound Algorithm for VASS.)**.** *Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$. We define $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

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

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

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_2)$	$\to \texttt{Incr}([j]) + \max([0]+0,0)$ $= \texttt{Incr}([j])$ $\to [n]$	$\zeta(\tau_2) = [j]$, $\mathcal{R}([j]) = (\tau_0, [0], 0)$, $\texttt{Incr}([j])$
$\texttt{Incr}([j])$	$\to T\mathcal{B}(\tau_1) \times 1$ $\to [n] \times 1$ $= [n]$	$\mathcal{I}([j]) = (\tau_1, 1)$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([i]) + \max([n]+0,0)$ $\to 0 + \max([n]+0,0)$ $= [n]$	$\zeta(\tau_1) = [i]$, $\mathcal{R}([i]) = (\tau_0, [n], 0)$, $\texttt{Incr}([i])$
$\texttt{Incr}([i])$	$\to 0$	$\mathcal{I}([i]) = \emptyset$

Table 3.1: Computation of $T\mathcal{B}(\tau_2)$ and $T\mathcal{B}(\tau_1)$ for Example `tarjan` (Figure 3.1) by Definition 22

Discussion. We first explain the subroutine $\texttt{Incr}(\texttt{v})$: With $(\tau, \texttt{c}) \in \mathcal{I}(\texttt{v})$ we have that a single execution of τ *increments* the value of $\texttt{v}$ by not more than $\texttt{c}$. $\texttt{Incr}(\texttt{v})$ multiplies the bound for τ with the increment $\texttt{c}$ in order to summarize the total amount by which $\texttt{v}$ may be incremented over all executions of τ. $\texttt{Incr}(\texttt{v})$ thus computes a bound on the total amount by which the value of $\texttt{v}$ may be *incremented* during program run. The function $T\mathcal{B}(\tau)$ computes a transition bound for τ based on the following reasoning: (1) The total amount by which the local bound $\zeta(\tau)$ of transition τ can be *incremented* is bounded by $\texttt{Incr}(\zeta(\tau))$. (2) By Definition 21 we have that $\zeta(\tau) \in \mathcal{V}$ can only be *reset* on the *single* initial transition $l_b \xrightarrow{u} l_2$. By well-definedness of $\Delta\mathcal{P}$ l_b has no incoming edges. Thus $l_b \xrightarrow{u} l_2$ can be executed exactly once. Further $\mathcal{R}(\zeta(\tau)) \neq \emptyset$ because by well-definedness of $\Delta\mathcal{P}$ $\zeta(\tau)$ *must* be reset on $l_b \xrightarrow{u} l_2$.

Example. Consider the VASS in Figure 3.1 (c) which models Tarjan's example for amortized complexity discussed in Section 1.4.1: Variable j models the stack size. The *push* instruction is modeled by increasing the stack size j by 1. The *pop* instruction is modeled by decreasing the stack size. We want to compute a bound on the overall cost of the `StackOp` operation, that is, a bound on the total number of times that transitions τ_1 and τ_2 can be executed. See Table 3.1 for details on the computation. We get $T\mathcal{B}(\tau_1) = [n]$ and $T\mathcal{B}(\tau_2) = [n]$. We thus get $2n$ as upper bound on the total cost of the `StackOp` operation (n has type *unsigned*). Recall from the discussion in Section 1.4.1 that $2n$ is in fact the *precise* upper bound for the cost of the `StackOp` operation.

Termination. Our algorithm does not terminate iff recursive calls cycle, i.e., if a call to $T\mathcal{B}(\tau)$ (indirectly) leads to a recursive call to $T\mathcal{B}(\tau)$. This can be easily detected, we return the expression '∞'. A cyclic computation occurs iff there is a transition τ_1 with local bound x that increases the local bound y of a transition τ_2 which in turn increases x. We conclude that absence of a cyclic computation is ensured if for all

strongly connected components (SCC) $\texttt{SCC}$ of $\Delta\mathcal{P}$ we can find an ordering $\tau_1, \ldots \tau_n$ of the transitions of $\texttt{SCC}$ s.t. the local bound of transition τ_i is not increased on any transition τ_j with $n \geq j > i \geq 1$. Note that the existence of such an ordering for each SCC of $\Delta\mathcal{P}$ proves termination of $\Delta\mathcal{P}$: it allows to directly compose a termination proof in form of a *lexicographic ranking function* [BMS05] by ordering the respective local transition bounds accordingly.

Complexity. Our algorithm can be efficiently (polynomial in the number of variables and transitions of the abstract program) implemented using a cache (dynamic programming): We set $\zeta(\tau) = T\mathcal{B}(\tau)$ after having computed $T\mathcal{B}(\tau)$.

Theorem 1 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free VASS over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $\tau \in E$. Let $T\mathcal{B}$ be as defined in Definition 22. Then $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ.*

Proof: Definition 22 is a special case of Definition 27. We prove soundness of Definition 27 in Section 7.1.1.

3.3 Bound Algorithm for *DCP*s with Constant Resets

In Figure 3.2 (a) we show a simplified version of an example which we found during our experiments. We discuss how our algorithm handles the full version of Example `SingleLinkSimple` (Example `SingleLinkCluster` in Figure 4.13) in Section 4.2.2. In Figure 3.2 (b) the *DCP* obtained from Example `SingleLinkSimple` by our abstraction procedure from Section 2.2 is shown. Note that the *DCP* in Figure 3.2 (b) is not a VASS: The variable i is *reset* to a value lower or equal to $n-1$ with each execution of transition τ_2, where $n \in \mathcal{C}$ is a *symbolic constant*.

In this section we discuss how we extend our bound algorithm such that it supports *constant resets*.

Definition 23 (*DCP* with only Constant Resets.)**.** *A DCP $\Delta\mathcal{P}(L, E, l_b, l_e)$ over $\mathcal{A}$ is a DCP* with only constant resets *iff for all $l_1 \xrightarrow{u} l_2 \in E$ and all $x' \leq y + \mathtt{c} \in u$ it holds that either $x' \leq y + \mathtt{c} \in \mathcal{MDC}(\mathcal{V})$, i.e., $y = x$, or $y \in \mathcal{C}$.*

See Figure 3.2 (b) for an example.

Definition 24 (Bound Algorithm for *DCP*s with only Constant Resets)**.** *Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$. We define $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

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

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

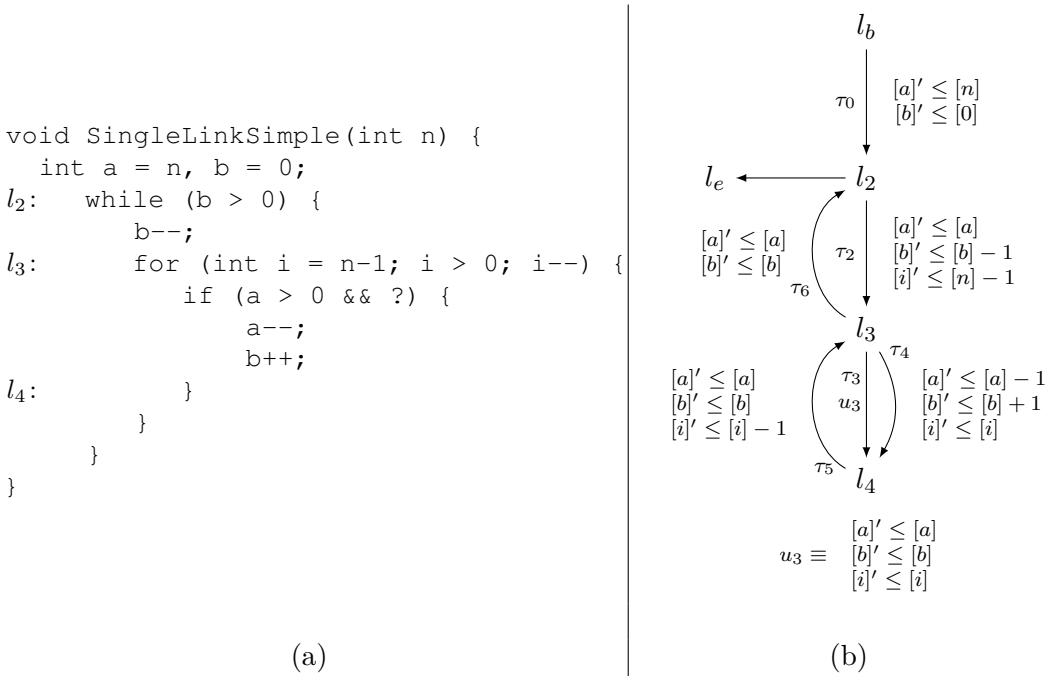


Figure 3.2: (a) Example `SingleLinkSimple` is a simplified version of Example `SingleLinkCluster` which we discuss in Section 4.2.2, (b) *DCP* obtained by abstraction from `SingleLinkSimple`

Discussion. The only difference to Definition 22 (our algorithm for VASS) is that we multiply a reset $(t, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))$ by the number of times it may occur: Consider a reset $(t, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))$. Note that $\mathtt{a} \in \mathcal{C}$ if $\Delta\mathcal{P}$ is a *DCP* with constant resets. In the worst case, a single execution of t resets the local bound $\zeta(t)$ to $\mathtt{a} + \mathtt{c}$, adding $\max(\mathtt{a} + \mathtt{c}, 0)$ to the potential number of executions of t; in total all $T\mathcal{B}(t)$ possible executions of t add up to $T\mathcal{B}(t) \times \max(\mathtt{a} + \mathtt{c}, 0)$ to the potential number of executions of t.

Theorem 2 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP* with only constant resets *over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $\tau \in E$. Let $T\mathcal{B}$ be as defined in Definition 24. Then $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ.*

Proof: Definition 24 is a special case of Definition 27. We prove soundness of Definition 27 in Section 7.1.1.

Example. Assume we want to compute a bound on the number of times that the inner loop at l_3 of Figure 3.2 can be executed. We thus compute a transition bound for τ_5 (the single back-edge of the inner loop). For details on the computation see Table 3.2. We get $T\mathcal{B}(\tau_5) = [n] \times \max([n] - 1, 0)$. We thus have that $\max(n, 0) \times \max(n - 1, 0)$ is a loop

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_5)$	$\to \texttt{Incr}([i]) + T\mathcal{B}(\tau_2) \times \max([n]-1,0)$ $\to 0 + T\mathcal{B}(\tau_2) \times \max([n]-1,0)$ $\to 0 + [n] \times \max([n]-1,0)$ $= [n] \times \max([n]-1,0)$	$\zeta(\tau_5) = [i]$, $\mathcal{R}([i]) = \{(\tau_2, [n], -1)\}$, $\texttt{Incr}([i])$, $T\mathcal{B}(\tau_2)$
$\texttt{Incr}([i])$	$\to 0$	$\mathcal{I}([i]) = \emptyset$
$T\mathcal{B}(\tau_2)$	$\to \texttt{Incr}([b]) + T\mathcal{B}(\tau_0) \times \max([0]+0,0)$ $= \texttt{Incr}([b])$ $\to [n]$	$\zeta(\tau_2) = [b]$, $\mathcal{R}([b]) = \{(\tau_0, [0], 0)\}$, $\texttt{Incr}([b])$
$\texttt{Incr}([b])$	$\to T\mathcal{B}(\tau_4) \times 1$, $\to [n] \times 1$ $= [n]$	$\mathcal{I}([b]) = \{(\tau_4, 1)\}$ $T\mathcal{B}(\tau_4)$
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}([a]) + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + 1 \times \max([n],0)$ $= [n]$	$\zeta(\tau_4) = [a]$, $\mathcal{R}([a]) = \{(\tau_0, [n], 0)\}$, $\texttt{Incr}([a])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([a])$	$\to 0$	$\mathcal{I}([a]) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

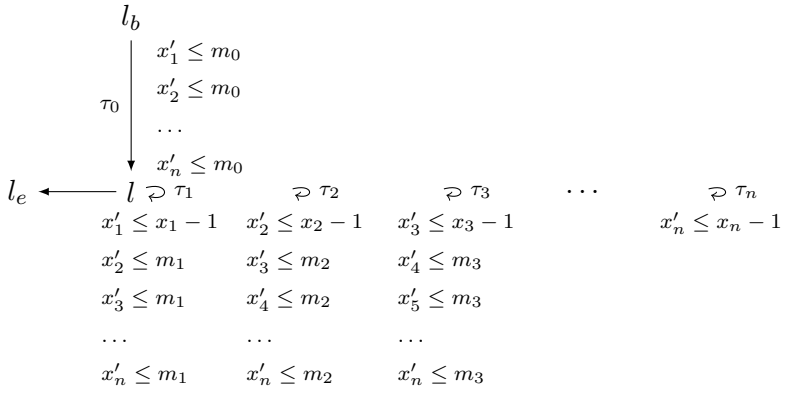
Table 3.2: Computation of $T\mathcal{B}(\tau_5)$ for Example `SingleLinkSimple` (Figure 3.2) by Definition 24

bound for the inner loop. In fact, $\max(n,0) \times \max(n-1,0)$ is the *precise* loop bound for the inner loop of Figure 3.2.

Termination. The same condition applies as for Definition 22 (see Section 3.2). We return the expression "∞" if a cyclic computation occurs.

Complexity. The time complexity of our algorithm remains polynomial if implemented by *dynamic programming* (see discussion in Section 3.2). The computed bound expressions, however, can now be of exponential size: Consider the *DCP* $\Delta\mathcal{P} = (\{l_b, l\}, \{\tau_0, \tau_1, \ldots, \tau_n\}, l_b, l_e)$ over variables $\{x_1, x_2, \ldots, x_n\}$ and constants $\{m_1, m_2, \ldots, m_n\}$ shown in Figure 3.3. Our algorithm computes:

$$
\begin{aligned}
T\mathcal{B}(\tau_n) = m_0 &+ \\
& m_0 \times m_1 + \\
& (m_0 + m_0 \times m_1) \times m_2 + \\
& (m_0 + m_0 \times m_2 + (m_0 + m_0 \times m_1) \times m_2) \times m_3 \\
\cdots & \\
= & \sum_{S \in 2^{\{1,2,\ldots,n\}}} m_0 \times \prod_{i \in S} m_i
\end{aligned}
$$



$$T\mathcal{B}(\tau_n) = \sum_{S \in 2^{\{1,2,\ldots,n\}}} m_0 \times \prod_{i \in S} m_i$$

Figure 3.3: Example for which we get a bound expression of *exponential size*, transitions $\tau_{1 \leq i \leq n}$ have source- and target-location l

We thus obtain a bound expression of *exponential size* (exponential in the number of transitions and variables of $\Delta\mathcal{P}$). In fact, $T\mathcal{B}(\tau_n) = \sum_{S \in 2^{\{1,2,\ldots,n\}}} m_0 \times \prod_{i \in S} m_i$ is *precise* for Figure 3.3. The example, however, is artificial. To our experience, the computed bound expressions can, in practice, be reduced to human readable size by applying basic rules of arithmetic.

3.4 Bound Algorithm for *DCP*s

In this Section we discuss how our bound algorithm is extended to support *non-constant* resets, i.e., difference constraints of form $z' \leq x + \mathtt{c}$ with $x \in \mathcal{V}$. Let us sketch our basic idea on the Example `twoSCCs` which we discussed in Section 1.4.4: In Figure 3.4 (a) we restate Example `twoSCCs`. In Figure 3.4 (b) the abstraction of Example `twoSCCs` is shown as obtained by our abstraction procedure (discussed in Section 2.2). On transition τ_2 of Figure 3.4 (b) $[z]$ is set to a value lower or equal than $[x]$, where $[x]$ is a variable of the abstract program. We reduce the problem of reasoning about this *non-constant* reset to the problem of reasoning about a *constant* reset by computing an invariant $x \leq \mathtt{expr}(n, m1, m2)$, where $\mathtt{expr}(n, m1, m2)$ is an expression over the programs parameters n, $m1$ and $m2$, i.e., an expression which value does not change during program run.

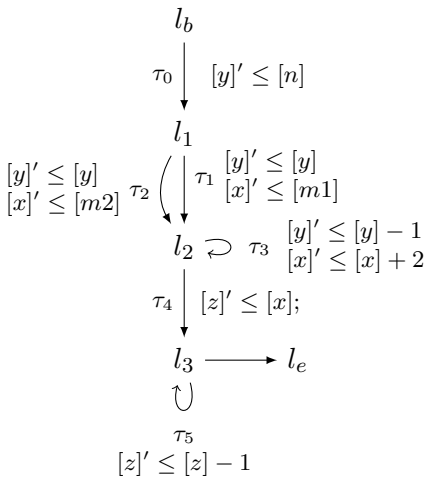
We call $\mathtt{expr}(n, m1, m2)$ such that $x \leq \mathtt{expr}(n, m1, m2)$ an *upper bound invariant* for x.

We compute *upper bound invariants* by means of bound analysis: We bound the number of times that x can be incremented by 2 in the loop at l_2 by computing a loop bound for l_2. I.e., we design a mutual recursive algorithm consisting of the function $T\mathcal{B}$ for computing transition bounds and a new function $V\mathcal{B}$ for inferring *variable bounds*, a

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

Bound of loop at l_3: $\max(m1, m2) + 2n$

(a)



(b)

Figure 3.4: (a) Example `twoSCCs` discussed in Section 1.4.4, (b) *DCP* obtained by abstraction from `twoSCCs`

special case of *upper bound invariants*. $T\mathcal{B}$ calls $V\mathcal{B}$ for over-approximating non-constant resets by constant expressions and $V\mathcal{B}$ calls $T\mathcal{B}$ for over-approximating the number of times that a given variable is incremented on a given transition.

Definition 25 (Upper Bound Invariant)**.** *Let $\mathcal{P}(L, T, l_b, l_e)$ be a program over Σ. Let $e : \Sigma \to \mathbb{Z}$. Let $l \in L$. Let $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} (l_2, \sigma_2) \xrightarrow{\lambda_2} \dots$ be a run of $\mathcal{P}$. $\mathtt{b} \in \mathbb{Z} \cup \{\infty\}$ is an* upper bound invariant *for e at l on ρ iff $e(\sigma_i) \leq \mathtt{b}$ holds for all i on ρ with $l_i = l$. A function $\mathtt{b} : \Sigma \to \mathbb{Z} \cup \{\infty\}$ is an* upper bound invariant *for e at l iff for all runs ρ of $\mathcal{P}$ it holds that $\mathtt{b}(\sigma_0)$ is an* upper bound invariant *for e at l on ρ, where σ_0 denotes the initial state of ρ.*

Definition 27 computes a special case of an upper bound invariant that we call a *variable bound*.

Definition 26 (Variable Bound)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\mathtt{a} \in \mathcal{A}$. We call $\mathtt{b}$ s.t. $\mathtt{b}$ is an* upper bound invariant *for $[\![\mathtt{a}]\!]$ at all $l \in L$ with $\mathtt{a} \in \mathtt{def}(l)$ a* variable bound *for $\mathtt{a}$.*

Let variable x of the abstract program represent the expression `expr` of the concrete program. Note that by computing an *variable bound* for x in the abstract program, we compute an *upper bound invariant* for `expr` in the concrete program.

As motivated above, we extend our algorithm by a function $V\mathcal{B}$ which computes *variable bounds*.

Definition 27 (Bound Algorithm)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$.*
Let $\zeta : E \to Expr(\mathcal{A})$. We define $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ and $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:

$$V\mathcal{B}(\mathtt{a}) = \mathtt{a},\ \textit{if}\ \mathtt{a} \in \mathcal{C},\ \textit{else}$$
$$V\mathcal{B}(\mathtt{v}) = \mathtt{Incr}(\mathtt{v}) + \max_{(\_,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} (V\mathcal{B}(\mathtt{a}) + \mathtt{c})$$

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

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

Discussion. The function $V\mathcal{B}(\mathtt{v})$ computes a variable bound for $\mathtt{v}$: After executing a transition τ with $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$, the value of $\mathtt{v}$ is bounded by $V\mathcal{B}(\mathtt{a}) + \mathtt{c}$. As long as $\mathtt{v}$ is not *reset*, its value cannot increase by more than $\mathtt{Incr}(\mathtt{v})$.
The function $T\mathcal{B}(\tau)$ is defined similar as in Definition 24 (Section 3.3), the only difference is that we over-approximate the value of a reset $(t, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))$ by $V\mathcal{B}(\mathtt{a})$. Note that $V\mathcal{B}(\mathtt{a}) = \mathtt{a}$ if $\mathtt{a} \in \mathcal{C}$. Thus, Definition 27 is identical to Definition 24 for the special case of a *DCP* with only constant resets.

Termination. The same condition applies as for Definition 22 (our algorithm for VASS, Section 3.2), and we return the expression "∞" if a cyclic computation occurs. In comparison to Definition 22 there are, however, two additional reasons which may lead to cyclic computations: (1) $V\mathcal{B}(\mathtt{v})$ cycles: There is a variable $\mathtt{v} \in \mathcal{V}$ s.t. the computation of $V\mathcal{B}(\mathtt{v})$ ends up calling $V\mathcal{B}(\mathtt{v})$ over a number of recursive calls to $V\mathcal{B}$. (2) $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\mathtt{v})$ cycle mutually: There is a variable $\mathtt{v} \in \mathcal{V}$ and a transition $\tau \in E$ s.t. the computation of $T\mathcal{B}(\tau)$ calls $V\mathcal{B}(\mathtt{v})$ which in turn ends up calling $T\mathcal{B}(\tau)$ over a number of recursive calls to $V\mathcal{B}$ and $T\mathcal{B}$.
Case (1) occurs iff there is a cycle in the *reset graph* (Definition 28, page 52) of $\Delta\mathcal{P}$. In Section 3.5.3 (page 59) we discuss a preprocessing that ensures absence of cycles in the reset graph and thus absence of case (1) by renaming the program variables appropriately.
An example for Case (2) is given in Figure 3.5: Consider the *DCP* abstraction in Figure 3.5 (b). τ_1 is the transition on which y is reset to a. τ_2 is the single transition of the inner loop. Assume we want to compute a loop bound for the inner loop, i.e., a transition bound for τ_2 with local bound $[y]$. This triggers a variable bound computation for $[a]$ because $[y]$ is reset to $[a]$. Since $[a]$ is incremented on τ_2, the variable bound computation for $[a]$ will in turn trigger a transition bound computation for τ_2. Note, however, that the loop bound of the inner loop (the transition bound of τ_2) is *exponential* (2^n). We

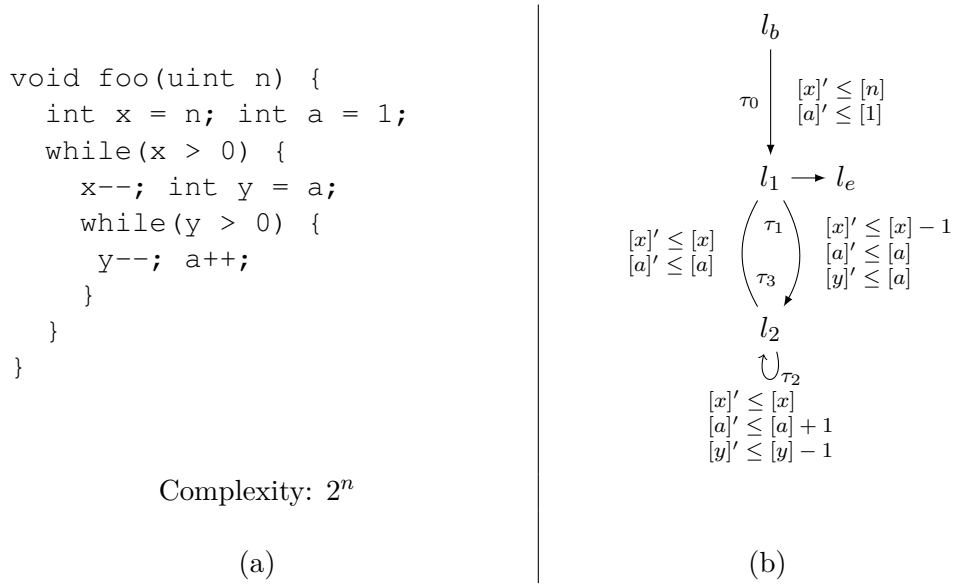


Figure 3.5: (a) Example with exponential loop bound, (b) *DCP* obtained by abstraction

consider exponential loop bounds to be very rare, we did not encounter an exponential loop bound during our experiments.

Complexity. The time complexity of our algorithm remains polynomial if implemented by *dynamic programming*: Similar to the discussed caching for $T\mathcal{B}(\tau)$ (see discussion in Section 3.2), we can introduce a cache to store the result of a $V\mathcal{B}(\mathtt{v})$ computation. When $V\mathcal{B}(\mathtt{v})$ is called we first check if the result is already in the cache before performing the computation.

Theorem 3 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $\mathtt{a} \in \mathcal{A}$ and $\tau \in E$. Let $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\mathtt{a})$ be as defined in Definition 27. We have: (1) $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ. (2) $[\![V\mathcal{B}(\mathtt{a})]\!]$ is a* variable bound *for* a.

Proof: We prove soundness of Definition 27 in Section 7.1.1.

Example. We want to infer a loop bound for the loop at l_3 in Figure 3.4. We thus compute a transition bound for τ_5 (the single back edge of the loop at l_3). See Table 3.3 for details on the computation. We get $T\mathcal{B}(\tau_5) = \max([m1], [m2]) + [n] \times 2$. Thus $\max(m1, m2) + 2n$ is a bound for the loop at l_3 (n, *m1* and *m2* have type *unsigned*). Recall that $\max(m1, m2) + 2n$ is in fact the *precise* bound of the loop at l_3 (see discussion in Section 1.4.4).

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_5)$	$\to \texttt{Incr}([z]) + T\mathcal{B}(\tau_4) \times \max(V\mathcal{B}([x]) + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_4) \times \max(V\mathcal{B}([x]) + 0, 0)$ $\to 0 + 1 \times \max(V\mathcal{B}([x]) + 0, 0)$ $\to 0 + 1 \times \max($ $\quad [n] \times 2 + \max([m1], [m2]) + 0,$ $\quad 0)$ $= \max([m1], [m2]) + [n] \times 2$	$\zeta(\tau_5) = [z]$, $\mathcal{R}([z]) = \{(\tau_4, [x], 0)\}$, $\texttt{Incr}([z])$, $T\mathcal{B}(\tau_4)$, $V\mathcal{B}([x])$
$\texttt{Incr}([z])$	$\to 0$	$\mathcal{I}([z]) = \emptyset$
$T\mathcal{B}(\tau_4)$	$\to 1$	$\zeta(\tau_4) = 1$
$V\mathcal{B}([x])$	$\to \texttt{Incr}([x]) + \max([m1] + 0, [m2] + 0)$ $\to [n] \times 2 + \max([m1] + 0, [m2] + 0)$ $= [n] \times 2 + \max([m1], [m2])$	$\mathcal{R}(x) = \{(\tau_1, [m1], 0),$ $(\tau_2, [m2], 0)\}$, $[m1], [m2] \in \mathcal{C}$, $\texttt{Incr}([x])$
$\texttt{Incr}([x])$	$\to T\mathcal{B}(\tau_3) \times 2$ $\to [n] \times 2$	$\mathcal{I}([x]) = \{(\tau_3, 2)\}$, $T\mathcal{B}(\tau_3)$
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}([y]) + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_3) = [y]$, $\mathcal{R}([y]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$, $\texttt{Incr}([y])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([y])$	$\to 0$	$\mathcal{I}([y]) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 3.3: Computation of $T\mathcal{B}(\tau_5)$ for Example `twoSCCs` (Figure 3.4) by Definition 27

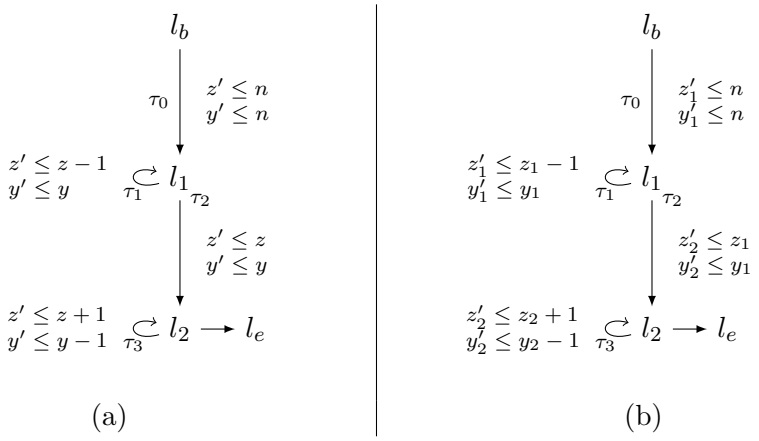


Figure 3.6: (a) *DCP* before variable renaming, (b) *DCP* obtained by variable renaming (Section 3.5.3)

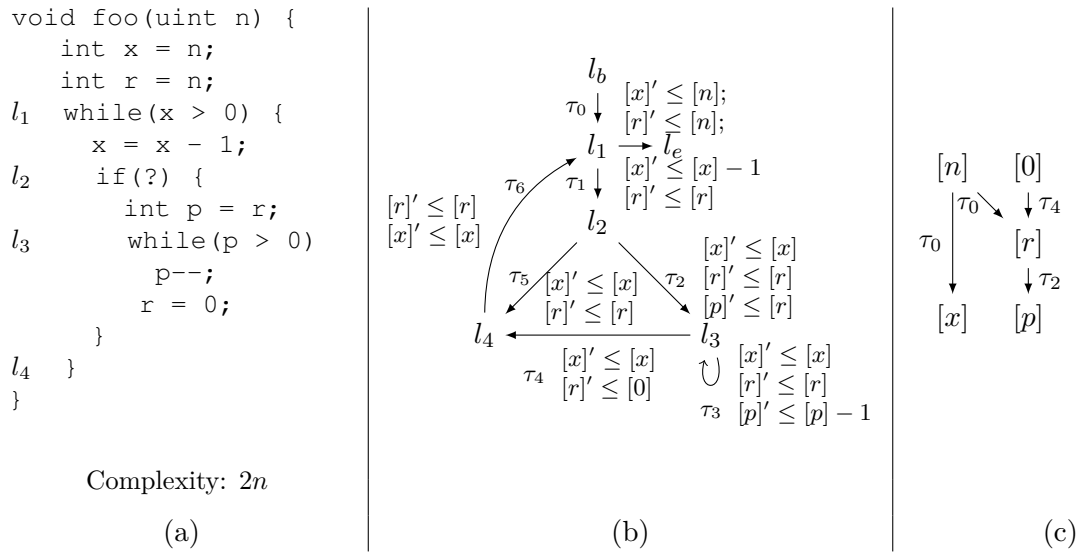


Figure 3.7: (a) Example, (b) Abstraction, (c) Reset Graph

Flow-Sensitive Reasoning. Our algorithm, as formulated in Definition 27, is *flow-insensitive*. This can lead to coarse over-approximations even for simple cases such as the one in Figure 3.6 (a). Here our algorithm does not take into account that z at l_2 can never flow into z at l_1 and thus computes $T\mathcal{B}(\tau_1) = 2 \times n$ whereas the precise transition bound for τ_1 is n. However, thanks to our simple program abstraction, we obtain a *flow-sensitive* bound algorithm through a simple program transformation: By a preprocessing, which we discuss in Section 3.5.3, we obtain the semantically equivalent *DCP* in Figure 3.6 (b) from Figure 3.6 (a) through variable renaming (z at l_1 is renamed to z_1, z at l_2 is renamed to z_2, etc.). For Figure 3.6 (b) we now obtain $T\mathcal{B}(\tau_1) = n$. I.e., by renaming the program variables appropriately the precision of our algorithm is enhanced. Note that Figure 3.6 (b) is very similar to the *static single assignment* (SSA) form. In program analysis, the transformation of programs into the SSA form is a well-known trick for adding some flow-sensitivity to a flow-insensitive analysis.

3.5 Reasoning Based on Reset Chains

Consider Figure 3.7. The precise bound for the loop at l_3 is n: Initially r has value n, after we have iterated the loop at l_3, r is set to 0. Thus the loop can only be executed in at most one iteration of the outer loop. However, our algorithm from Definition 27 infers a quadratic bound for the loop at l_3: As shown in Table 3.4 we have $T\mathcal{B}(\tau_3) = [n] \times [n]$. We thus get n^2 (n has type *unsigned*) as bound for the loop at l_3 in the concrete program.

Our algorithm from Definition 27 does not take into account that r is reset to 0 after executing the loop at l_3. In the following we discuss an extension of our algorithm which overcomes this imprecision by taking the *context* under which a transition is executed

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to T\mathcal{B}(\tau_2) \times \max(V\mathcal{B}([r]) + 0, 0)$ $\to [n] \times \max(V\mathcal{B}([r]) + 0, 0)$ $\to [n] \times \max([n] + 0, 0)$ $= [n] \times [n]$	$\zeta(\tau_3) = [p]$, $\mathcal{R}([p]) = \{(\tau_2, [r], 0)\}$, $T\mathcal{B}(\tau_2)$, $V\mathcal{B}([r])$
$T\mathcal{B}(\tau_2)$	$\to T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_2) = [x]$, $\mathcal{R}([x]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$, $T\mathcal{B}(\tau_0)$
$V\mathcal{B}([r])$	$\to \max([n] + 0, [0] + 0)$ $= \max([n], [0])$ $= [n]$	$\mathcal{R}([r]) = \{(\tau_0, [n], 0),$ $(\tau_4, [0], 0)\}$, $[n], [0] \in \mathcal{C}$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 3.4: Computation of $T\mathcal{B}(\tau_3)$ for Figure 3.7 (b) by Definition 27 (without calls to `Incr` because $\mathcal{I}(\mathtt{v}) = \emptyset$ and thus $\mathtt{Incr}(\mathtt{v}) = 0$ for Figure 3.7 (b))

into account: We say that a transition τ_2 is executed under *context* τ_1 if transition τ_1 was executed before the current execution of τ_2 and after the previous execution of τ_2 (if any).

As an example, consider Figure 3.7 (b), the abstraction of Figure 3.7 (a). We have that τ_2 is always executed either under context τ_0 or under context τ_4. When executing τ_2 under context τ_0, p is set to n. But when executing τ_2 under context τ_4, p is set to 0. Moreover, τ_2 can only be executed once under context τ_0 because τ_0 is executed only once.

We define the notion of a *reset graph* as a means to reason systematically about the context under which *resets* can be executed.

Definition 28 (Reset Chain Graph)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. The* reset chain graph *or* reset graph *of $\Delta\mathcal{P}$ is the directed graph $\mathcal{G}$ with node set $\mathcal{A}$ and edges $\mathcal{E} = \{(y, \tau, \mathtt{c}, x) \mid (\tau, y, \mathtt{c}) \in \mathcal{R}(x)\} \subseteq \mathcal{A} \times E \times \mathbb{Z} \times \mathcal{V}$, i.e., each edge has a label in $E \times \mathbb{Z}$. We call $\mathcal{G}(\mathcal{A}, \mathcal{E})$ a* reset chain DAG *or* reset DAG *if $\mathcal{G}(\mathcal{A}, \mathcal{E})$ is acyclic. We call $\mathcal{G}(\mathcal{A}, \mathcal{E})$ a* reset chain forest *or* reset forest *if the sub-graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a forest. We call a* finite *path $\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \dots \mathtt{a}_0$ in $\mathcal{G}$ with $n > 0$ a* reset chain *of $\Delta\mathcal{P}$. We say that κ is a reset chain from $\mathtt{a}_n$ to $\mathtt{a}_0$. Let $n \geq i \geq j \geq 0$. By $\kappa_{[i,j]}$ we denote the sub-path of κ that starts at $\mathtt{a}_i$ and ends at $\mathtt{a}_j$. We define $in(\kappa) = \mathtt{a}_n$, $c(\kappa) = \sum_{i=1}^{n} c_i$, $trn(\kappa) = \{\tau_n, \tau_{n-1} \dots, \tau_1\}$, and $atm(\kappa) = \{a_{n-1} \dots, a_0\}$. κ is* sound *if for all $1 \leq i < n$ it holds that $\mathtt{a}_i$ is* reset *on all paths from the target location of τ_1 to the source location of τ_i in $\Delta\mathcal{P}$. κ is* optimal *if κ is sound and there is no sound reset chain $\varkappa$ of length $n + 1$ s.t. $\varkappa_{[n,0]} = \kappa$. Let $\mathtt{v} \in \mathcal{V}$, by $\mathfrak{R}(\mathtt{v})$ we denote the set of* optimal reset chains *ending in* v.

Example: Figure 3.7 (c) shows the reset graph of Figure 3.7 (b).

We elaborate on the notions *sound* and *optimal* below. Let us first state a basic intuition on how we employ reset chains to enhance the precision of our reasoning:

For a given reset $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$, the reset graph determines which atom flows into variable $\mathtt{v}$ under which context. For example, consider Figure 3.7 (b) and its reset graph in Figure 3.7 (c): When executing the reset $(\tau_2, [r], 0) \in \mathcal{R}([p])$ under the context τ_4, $[p]$ is set to $[0]$, if the same reset is executed under the context τ_0, $[p]$ is set to $[n]$. Note that the reset graph does not represent *increments* of variables. We discuss how we handle increments in Section 3.5.1.

Let $\mathtt{v} \in \mathcal{V}$. Given a reset chain κ of length n that ends at $\mathtt{v}$, we say that $(trn(\kappa), in(\kappa), c(\kappa))$ is a reset of $\mathtt{v}$ with context of length $n-1$. I.e., $\mathcal{R}(\mathtt{v})$ from Definition 19 is the set of *context-free* resets of $\mathtt{v}$ (context of length 0), because $(trn(\kappa), in(\kappa), c(\kappa)) \in \mathcal{R}(\mathtt{v})$ iff κ ends at $\mathtt{v}$ and has length 1. Our previously defined algorithm from Definition 27 uses only *context-free* resets, we say that it reasons *context free*. For reasoning with context, we substitute the term

$$\sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\zeta(\tau))} T\mathcal{B}(t) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$$

in Definition 27 by the term

$$\sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0).$$

Note that we can compute a bound on the number of times that a sequence $\tau_1, \tau_2, \ldots \tau_n$ of transitions may occur on a run by computing $\min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$.

We now discuss how our algorithm infers the *linear* bound for τ_3 of Figure 3.7 when applying the described modification to Definition 27: The reset graph of Figure 3.7 (b) is shown in Figure 3.7 (c). There are 3 reset chains ending in $[p]$: $\kappa_1 = [0] \xrightarrow{\tau_4,0} [r] \xrightarrow{\tau_2,0} [p]$, $\kappa_2 = [n] \xrightarrow{\tau_0,0} [r] \xrightarrow{\tau_2,0} [p]$ and $\kappa_3 = [r] \xrightarrow{\tau_2,0} [p]$. However, κ_3 is a sub-path of κ_1 and κ_2. Note that κ_1 and κ_2 are *sound* by Definition 28 because $[r]$ is reset on all paths from the target location l_3 of τ_2 to the source location l_2 of τ_2 in Figure 3.7 (b) (namely on τ_4). κ_1 and κ_2 are both *optimal* because they are sound and of maximal length (we discuss the notions *sound* and *optimal* next). Thus $\mathfrak{R}([p]) = \{\kappa_1, \kappa_2\}$. Basing our analysis on $\mathfrak{R}([p])$ rather than $\mathcal{R}([p])$ our approach reasons as shown in Table 3.5. We get $T\mathcal{B}(\tau_3) = [n]$, i.e., we get the bound n (n has type *unsigned*) for the loop at l_3 in the concrete program (Figure 3.7 (a)).

Sound and Optimal Reset Chains. A given reset chain $\mathtt{a}_n \xrightarrow{\tau_n,\mathtt{c}_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1},\mathtt{c}_{n-1}} \ldots \xrightarrow{\tau_1,\mathtt{c}_1} \mathtt{a}_0$ is *sound* if in between any two executions of τ_1 all atoms on the path (but

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to T\mathcal{B}(\{\tau_4,\tau_2\}) \times \max([0]+0,0) +$ $T\mathcal{B}(\{\tau_0,\tau_2\}) \times \max([n]+0,0)$ $= 0 + T\mathcal{B}(\{\tau_0,\tau_2\}) \times \max([n]+0,0)$ $\to 0 + \min(1,[n]) \times \max([n]+0,0)$ $= \min(1,[n]) \times [n]$ $= [n]$	$\zeta(\tau_3) = [p]$, $\mathfrak{R}([p]) = \{[0] \xrightarrow{\tau_4} [r] \xrightarrow{\tau_2} [p],$ $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]\}$, $[n], [0] \in \mathcal{C}$, $T\mathcal{B}(\{\tau_0,\tau_2\})$
$T\mathcal{B}(\{\tau_0,\tau_2\})$	$\to \min(T\mathcal{B}(\tau_0), T\mathcal{B}(\tau_2))$ $\to \min(1,[n])$	$T\mathcal{B}(\tau_0)$, $T\mathcal{B}(\tau_2)$
$T\mathcal{B}(\tau_2\}$	$\to T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 1 \times \max([n]+0,0)$ $= [n]$	$\zeta(\tau_2) = [x]$, $\mathfrak{R}([x]) = \{[n] \xrightarrow{\tau_0} [x]\}$, $[n] \in \mathcal{C}$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 3.5: Computation of $T\mathcal{B}(\tau_3)$ for Figure 3.7 (b) by Definition 29 (without calls to `Incr` because $\mathcal{I}(\mathtt{v}) = \emptyset$ and thus $\mathtt{Incr}(\mathtt{v}) = 0$ for Figure 3.7 (b))

not necessarily $\mathtt{a}_n$ where the path starts and $\mathtt{a}_0$ where it ends) are *reset*: Assume that r in Figure 3.7 (a) would not be reset after executing the inner loop. Then we could repeat the reset of p to r without resetting r to 0, and the inner loop would have a *quadratic* loop bound. For the abstract program the described modification amounts to replacing the constraint $[r]' \leq [0]$ on τ_4 in Figure 3.7 (b) by $[r]' \leq [r]$. In the modified program $[r]$ is not reset between any two executions of τ_2. Our algorithm must therefore reason based on the reset chain $[r] \xrightarrow{\tau_2} [p]$ in order to obtain the quadratic bound for τ_3: $T\mathcal{B}(\tau_3) = T\mathcal{B}(\tau_2) \times V\mathcal{B}(r) = [n] \times [n]$. I.e., if r is not *reset* on the outer loop this is modeled in our analysis by considering the reset chain $[r] \xrightarrow{\tau_2} [p]$ rather than the maximal reset chain $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]$. Considering the *maximal* reset chain $[n] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]$ would be unsound in the described scenario: $\min(T\mathcal{B}(\tau_0), T\mathcal{B}(\tau_2)) \times [n] = [n]$ is *not* a valid transition bound for τ_3 if r is not reset to 0 between two executions of the inner loop. The *optimal* reset chains are the sound reset chains with *maximal* context, i.e., those reset chains that are sound and cannot be extended without becoming *unsound*.

3.5.1 Algorithm Based on Reset Chain Forests

In the presence of cycles in the reset graph we get infinitely many reset chains. Let us for now assume that the given program has a *reset forest*, i.e., that the sub-graph of the reset graph which has nodes only in $\mathcal{V}$ is a forest (Definition 28). Then also the complete reset graph is *acyclic* because the nodes in $\mathcal{C}$ cannot have incoming edges (Definition 28).

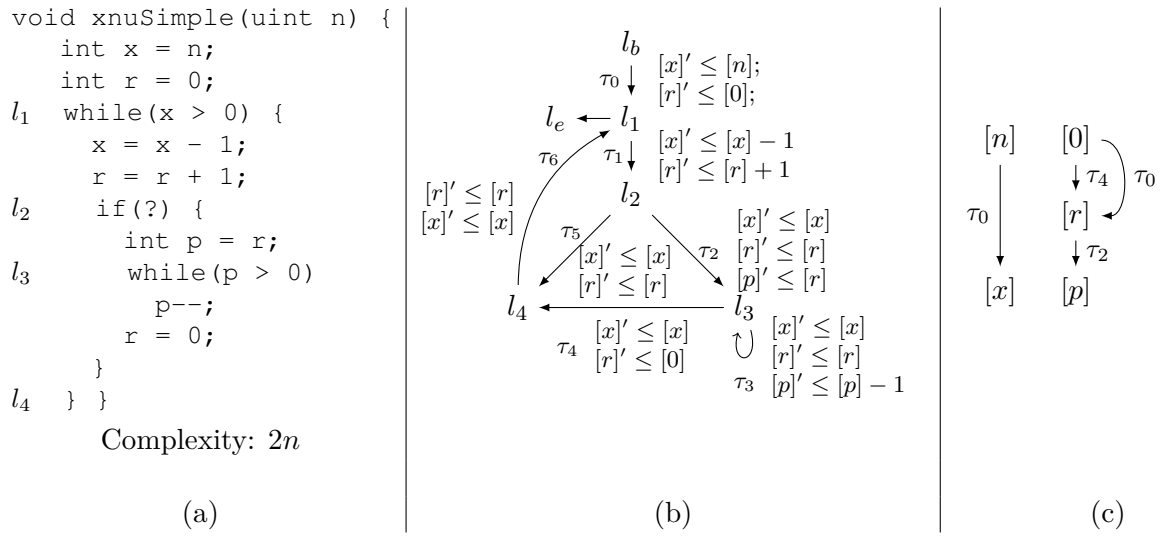


Figure 3.8: (a) Example `xnuSimple`, (b) *DCP* obtained by abstraction from `xnuSimple` (c) Reset Graph

Definition 29 (Bound Algorithm using Reset Chains (reset forest))**.** *Let* $\zeta : E \to Expr(\mathcal{A})$ *be a* local bound mapping *for* $\Delta\mathcal{P}$*. Let* $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ *be as defined in Definition 27. We override the definition of* $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ *in Definition 27 by stating:*

$$T\mathcal{B}(\tau) = \zeta(\tau), \text{ if } \zeta(\tau) \notin \mathcal{V}, \text{ else}$$
$$T\mathcal{B}(\tau) = \texttt{Incr}\left(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm(\kappa)\right) + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0)$$

where $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$ *and*
$\texttt{Incr}(\{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n\}) = \sum_{1 \leq i \leq n} \texttt{Incr}(\mathtt{a}_i)$ *with* $\texttt{Incr}(\emptyset) = 0$

Discussion and Example. We have discussed above why we replace the term $T\mathcal{B}(t) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$ from Definition 27 by the term $T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0)$. We further discuss the term $\texttt{Incr}(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm(\kappa))$ which replaces $\texttt{Incr}(\zeta(\tau))$ from Definition 27: Consider Example `xnuSimple` in Figure 3.8. Note that r may be incremented on τ_1 between the reset of r to 0 on τ_0 resp. τ_4 and the reset of p to r on τ_2. The term $\texttt{Incr}(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm(\kappa))$ takes care of such increments which may increase the value that finally flows into $\zeta(\tau)$ (in the example p) when the last transition on κ (in the example τ_2) is executed. In Table 3.6 the details of the bound computation are given. We get $T\mathcal{B}(\tau_3) = [n]$, i.e, we have the bound n for the loop at l_3 in the concrete program (Figure 3.8 (a), n has type *unsigned*).

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}(\{[r],[p]\}) +$ $T\mathcal{B}(\{\tau_4,\tau_2\}) \times \max([0]+0,0) +$ $T\mathcal{B}(\{\tau_0,\tau_2\}) \times \max([0]+0,0)$ $= \texttt{Incr}(\{[r],[p]\}) + 0 + 0$ $\to [n] + 0 + 0$ $= [n]$	$\zeta(\tau_3) = [p]$, $\mathfrak{R}([p]) = \{[0] \xrightarrow{\tau_4} [r] \xrightarrow{\tau_2} [p],$ $[0] \xrightarrow{\tau_0} [r] \xrightarrow{\tau_2} [p]\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}(\{[r],[p]\})$
$\texttt{Incr}(\{[r],[p]\})$	$\to \texttt{Incr}([r]) + \texttt{Incr}([p])$ $\to [n] + 0$ $= [n]$	$\texttt{Incr}([r])$, $\texttt{Incr}([p])$
$\texttt{Incr}([r])$	$\to T\mathcal{B}(\tau_1) \times 1$ $\to [n] \times 1$ $= [n]$	$\mathcal{I}([p]) = \{(\tau_1, 1)\}$, $T\mathcal{B}(\tau_1)$
$\texttt{Incr}([p])$	$\to 0$	$\mathcal{I}([p]) = \emptyset$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([x]) + T\mathcal{B}(\tau_0) \times [n]$ $\to 0 + T\mathcal{B}(\tau_0) \times [n]$ $\to 0 + 1 \times [n]$ $= [n]$	$\zeta(\tau_1) = [x]$, $\mathfrak{R}([x]) = \{[n] \xrightarrow{\tau_0} [x]\}$, $[n] \in \mathcal{C}$, $\texttt{Incr}([x])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([x])$	$\to 0$	$\mathcal{I}([x]) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 3.6: Computation of $T\mathcal{B}(\tau_3)$ for Example `xnuSimple` (Figure 3.8) by Definition 29

Soundness. Definition 29 for *DCP*s with a reset forest is a special case of Definition 31 for *DCP*s with a reset DAG. We prove soundness of Definition 31 in Section 7.2.1.

Complexity. The nodes of a reset forest are the variables and constants of the abstract program (the elements of $\mathcal{A}$). Since the number of paths of a forest is polynomial in the number of nodes, the run time of our algorithm remains polynomial.

3.5.2 Algorithm Based on Reset Chain DAGs

The examples we considered so far had reset forests. Note that the definition of a *reset forest* (Definition 28) only requires the sub-graph over the variables (i.e., the reset graph without the nodes that are symbolic constants) to be a *forest*. In the following we generalize Definition 29 to reset DAGs. We discuss in Section 3.5.3 how we ensure that the reset graph is *acyclic*.

Consider the Example shown in Figure 3.9. The outer loop (at l_1) can be executed n times. The loop at l_4 resp. transition τ_6 can be executed $2n$ times, e.g., by executing the program as depicted in Table 3.7: The first row counts the number of iterations of the outer loop, the second row shows the transitions that are executed and in the

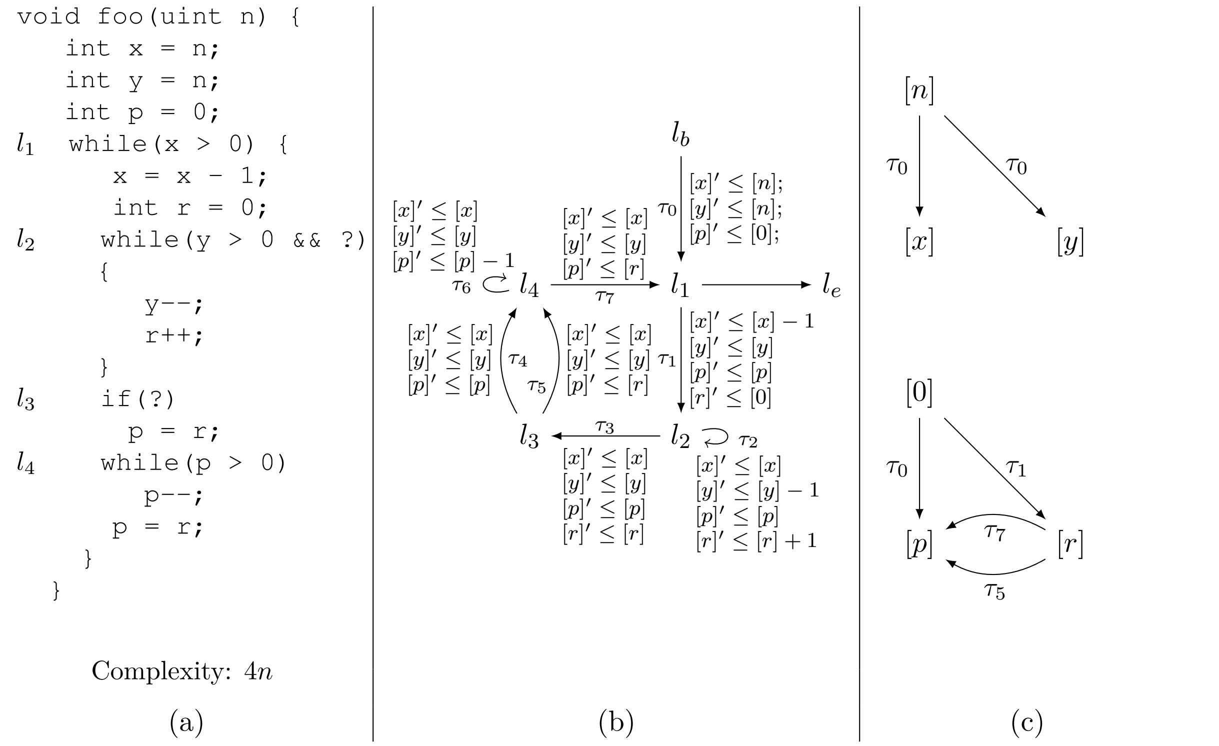


Figure 3.9: (a) Example, (b) Abstraction, (c) Reset Graph

	1								2						3								4						...
τ_0	τ_1	τ_2	τ_2	τ_3	τ_5	$\boldsymbol{\tau_6}$	$\boldsymbol{\tau_6}$	τ_7	τ_1	τ_3	τ_4	$\boldsymbol{\tau_6}$	$\boldsymbol{\tau_6}$	τ_7	τ_1	τ_2	τ_2	τ_3	τ_5	$\boldsymbol{\tau_6}$	$\boldsymbol{\tau_6}$	τ_7	τ_1	τ_3	τ_4	$\boldsymbol{\tau_6}$	$\boldsymbol{\tau_6}$	τ_7	...
r	0	1	2	2	2	2	2	2	0	0	0	0	0	0	0	1	2	2	2	2	2	2	0	0	0	0	0	0	...
p	0	0	0	0	2	1	0	2	2	2	2	1	0	0	0	0	0	0	2	1	0	2	2	2	2	1	0	0	...

Table 3.7: Run of Figure 3.9

last two rows the values of r resp. p are tracked. The execution switches between two iteration schemes of the outer loop: an uneven iteration increments r twice (by executing τ_2 twice) and afterward assigns r to p by executing τ_5. We can then execute τ_6 two times. Afterward the value of r is "saved" in p for the next (even) iteration of the outer loop before r is set to 0 on τ_1. Therefore τ_6 can be executed again two times in the next, even iteration though r is not incremented on that iteration.

Consider the abstracted *DCP* in Figure 3.9 (b) and its reset graph in Figure 3.9 (c). We have that $\kappa_2 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_5} [p]$ and $\kappa_3 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_7} [p]$ are two reset chains ending in $[p]$ (see Figure 3.9 (c)). Observe that both are sound, i.e., between any two executions of τ_7 resp. τ_5 $[r]$ is reset. However, $[r]$ is *not* necessarily reset between the execution of τ_5 and τ_7, therefore the accumulated value 2 of r is used twice to increase the local bound $[p]$ of τ_6.

I.e., since there are two paths from $[r]$ to $[p]$ in the reset graph (Figure 3.9 (c)) we have to count the increments of $[r]$ twice: once for κ_2 and once for κ_3. Definition 30 distinguishes between nodes that have a single resp. multiple path(s) to a given variable in the reset graph. This is used in Definition 31 for a sound handling of the latter case.

Definition 30 ($atm_1(\kappa)$ and $atm_2(\kappa)$)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $P(\mathtt{a}, \mathtt{v})$ denote the set of paths from $\mathtt{a}$ to $\mathtt{v}$ in the* reset graph *of $\Delta\mathcal{P}$. Let $\mathtt{v} \in \mathcal{V}$. Let κ be a reset chain ending in $\mathtt{v}$. We define $atm_1(\kappa) = \{\mathtt{a} \in atm(\kappa) \mid |P(\mathtt{a}, \mathtt{v})| \leq 1\}$ and $atm_2(\kappa) = \{\mathtt{a} \in atm(\kappa) \mid |P(\mathtt{a}, \mathtt{v})| > 1\}$, where $|S|$ denotes the number of elements in S.*

Definition 31 (Bound Algorithm using Reset Chains (reset DAG))**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ be as defined in Definition 27. We override the definition of $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ in Definition 27 by stating:*

$$T\mathcal{B}(\tau) = \zeta(\tau), \text{ if } \zeta(\tau) \notin \mathcal{V}, \text{ else}$$
$$T\mathcal{B}(\tau) = \mathtt{Incr}\left(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)\right) + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} T\mathcal{B}(trn(\kappa)) \times \max(V\mathcal{B}(in(\kappa)) + c(\kappa), 0) + \mathtt{Incr}(atm_2(\kappa))$$

where $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \leq i \leq n} T\mathcal{B}(\tau_i)$ *and*
$\mathtt{Incr}(\{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n\}) = \sum_{1 \leq i \leq n} \mathtt{Incr}(\mathtt{a}_i)$ *with* $\mathtt{Incr}(\emptyset) = 0$

Discussion. If $atm_2(\kappa) = \emptyset$ for all reset chains κ, Definition 31 is equal to Definition 29. This is the case for all *DCP*s with a reset forest (all examples in this thesis except Figure 3.9). Definition 31 thus is a generalization of Definition 29.

Example. As shown in Table 3.8 we get $T\mathcal{B}(\tau_6) = [n] + [n]$ for Figure 3.9 by Definition 31. I.e., we get the precise bound $2n$ for the loop at l_4 in Figure 3.9 (n has type *unsigned*).

Theorem 4 (Soundness of Bound Algorithm using Reset Chains)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$. Let $T\mathcal{B}$ and $V\mathcal{B}$ be defined as in Definition 31. Let $\tau \in E$ and $\mathtt{a} \in \mathcal{A}$. If $\Delta\mathcal{P}$ has a reset DAG then (1) $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ and (2) $[\![V\mathcal{B}(\mathtt{a})]\!]$ is a* variable bound *for $\mathtt{a}$.*

Proof: See Section 7.2.1.

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_6)$	$\to \texttt{Incr}([p]) +$ $T\mathcal{B}(\{\tau_0\}) \times \max([0]+0,0) + 0 +$ $T\mathcal{B}(\{\tau_1,\tau_7\}) \times \max([0]+0,0) +$ $\texttt{Incr}([r]) +$ $T\mathcal{B}(\{\tau_1,\tau_5\}) \times \max([0]+0,0) +$ $\texttt{Incr}([r])$ $= \texttt{Incr}([p]) + 0 + 0 + 0 + \texttt{Incr}([r]) +$ $0 + \texttt{Incr}([r])$ $\to 0 + 0 + 0 + 0 + [n] + 0 + [n]$ $= [n] + [n]$	$\zeta(\tau_6) = [p]$, $\kappa_1 = [0] \xrightarrow{\tau_0} [p]$, $\kappa_2 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_7} [p]$, $\kappa_3 = [0] \xrightarrow{\tau_1} [r] \xrightarrow{\tau_5} [p]$, $\mathfrak{R}([p]) = \{\kappa_1, \kappa_2, \kappa_3\}$, $atm_1(\kappa_1) = \{[p]\}$, $atm_1(\kappa_2) = \{[p]\}$, $atm_1(\kappa_3) = \{[p]\}$, $atm_2(\kappa_1) = \emptyset$, $atm_2(\kappa_2) = \{[r]\}$, $atm_2(\kappa_3) = \{[r]\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}([p])$, $\texttt{Incr}([r])$
$\texttt{Incr}([p])$	$\to 0$	$\mathcal{I}([p]) = \emptyset$
$\texttt{Incr}([r])$	$\to T\mathcal{B}(\tau_2) \times 1$ $\to [n] \times 1$ $= [n]$	$\mathcal{I}([r]) = \{(\tau_2, 1)\}$, $T\mathcal{B}(\tau_2)$
$T\mathcal{B}(\tau_2)$	$\to T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 1 \times \max([n]+0,0)$ $= [n]$	$\zeta(\tau_2) = [y]$, $\mathfrak{R}([y]) = \{[n] \xrightarrow{\tau_0} [y]\}$, $[n] \in \mathcal{C}$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 3.8: Computation of $T\mathcal{B}(\tau_6)$ for Figure 3.9 by Definition 31

Complexity. A DAG can have exponentially many paths in the number of nodes. Thus there can be exponentially many reset chains in $\mathfrak{R}(\mathtt{v})$ (exponential in the number of variables and constants of the abstract program, i.e., the norms generated during the abstraction process, see Section 2.2). However, in our experiments enumeration of (optimal) reset chains did not affect performance.

3.5.3 Preprocessing: Transforming a Reset Graph into a Reset DAG

Consider the *DCP* shown in Figure 3.10 (a). Figure 3.10 (a) has a cyclic reset graph as shown in Figure 3.10 (b). In the following we describe an algorithm which transforms Figure 3.10 (a) into Figure 3.10 (d) by renaming the program variables. Figure 3.10 (d) has an acyclic reset graph (a reset DAG).

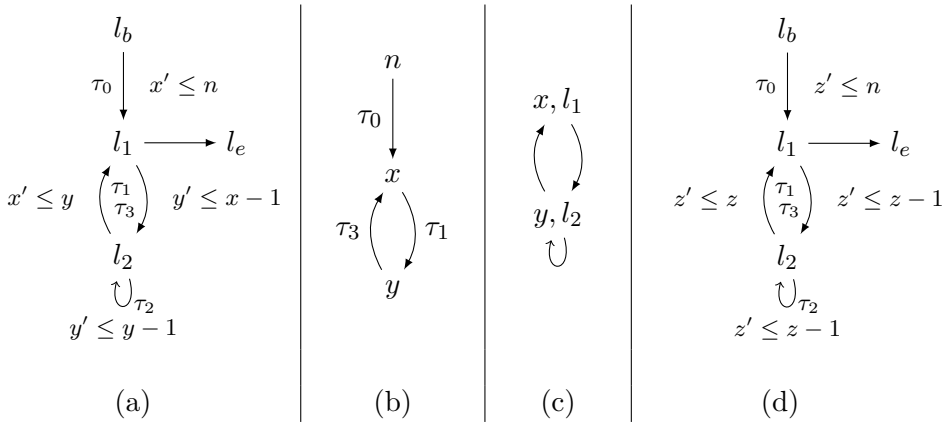


Figure 3.10: (a) Example, (b) Reset Graph, (c) Variable Flow Graph, (d) Example after renaming variables

Definition 32 (Variable Flow Graph)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. We call the graph with node set $\mathcal{V} \times L$ and edge set*

$$\{(y, l_1) \to (x, l_2) \mid l_1 \xrightarrow{u} l_2 \in E \wedge x' \leq y + \mathtt{c} \in u \text{ with } x, y \in \mathcal{V}\}$$

the variable flow graph*.*

Example: Figure 3.10 (c).

Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a *DCP*. Let $\{\mathtt{SCC}_1, \mathtt{SCC}_2, \ldots, \mathtt{SCC}_n\}$ be the *strongly connected components* of its *variable flow graph*. For each SCC $\mathtt{SCC}_i$ we choose a fresh variable $\mathtt{v}_i \in \mathcal{V}$. Let $\varsigma : \mathcal{V} \times L \mapsto \mathcal{V}$ be the mapping $\varsigma(\mathtt{v}, l) = \mathtt{v}_i$, where i s.t. $(\mathtt{v}, l) \in \mathtt{SCC}_i$. We extend ς to $\mathcal{A} \times L \mapsto \mathcal{A}$ by defining $\varsigma(\mathtt{s}, l) = \mathtt{s}$ for all $l \in L$ and $\mathtt{s} \in \mathcal{C}$.

We obtain $\Delta\mathcal{P}'(L, E', l_b, l_e)$ from $\Delta\mathcal{P}$ by setting $E' = \{l_1 \xrightarrow{u'} l_2 \mid l_1 \xrightarrow{u} l_2 \in E\}$, where u' is obtained from u by generating the constraint $\varsigma(x, l_2)' \leq \varsigma(y, l_1) + \mathtt{c}$ from a constraint $x' \leq y + \mathtt{c} \in u$.

Examples. Figure 3.10 (d) is obtained from Figure 3.10 (a) by applying the described transformation using the mapping $\varsigma(x, l_1) = \varsigma(y, l_2) = z$. Figure 3.6 (b) is obtained from Figure 3.6 (a) by applying the described transformation using the mapping $\varsigma(z, l_1) = z_1$, $\varsigma(z, l_2) = z_2$, $\varsigma(y, l_1) = y_1$, $\varsigma(y, l_2) = y_2$.

Soundness. Soundness of the described variable renaming is obvious if there are no two (different) variables $\mathtt{v}_1$ and $\mathtt{v}_2$ that are renamed to the same fresh variable at some location l. This is the case if in each SCC of the *variable flow graph* each location $l \in L$ appears at most once, i.e., if there is no SCC $\mathtt{SCC}$ in the *variable flow graph* of the program s.t. there is a location $l \in L$ and variables $\mathtt{v}_1, \mathtt{v}_2 \in \mathcal{V}$ with $\mathtt{v}_1 \neq \mathtt{v}_2$ and $(l, \mathtt{v}_1) \in \mathtt{SCC}$ and $(l, \mathtt{v}_2) \in \mathtt{SCC}$. In the literature, a program with this property is called *stratifiable* (e.g., [BAL07]).

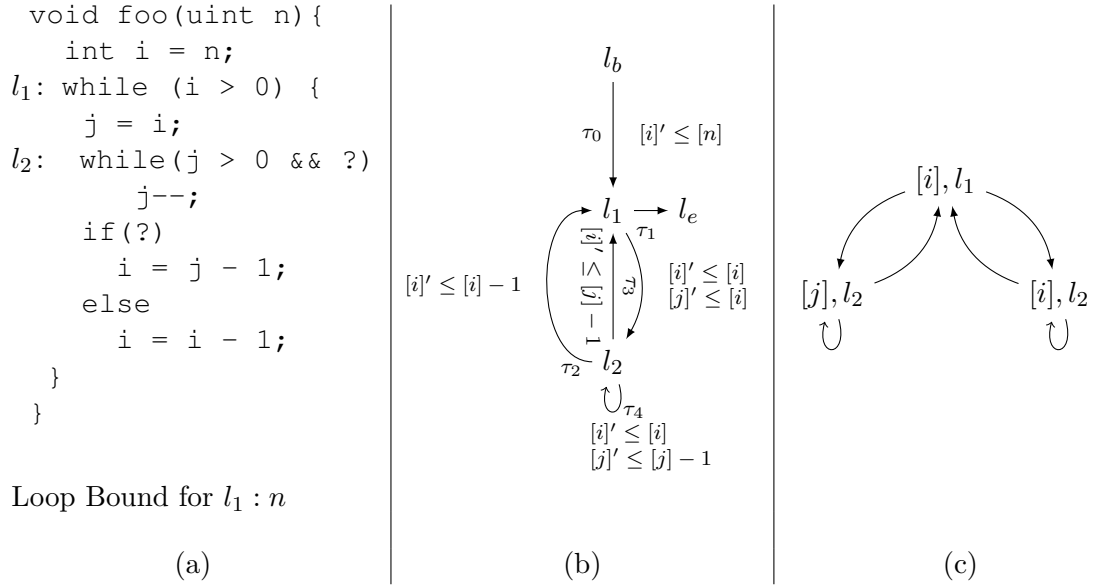


Figure 3.11: (a) Example, (b) *DCP* obtained by abstraction, (c) Variable Flow Graph

Ensuring Stratifiability

A *fan-in free DCP* that is *not stratifiable* can be transformed into a *stratifiable* and *fan-in free DCP* by introducing appropriate case distinctions into the control flow of the program. We state an example next. In the worst-case, however, this transformation can cause an exponential blow up of the number of transitions in the program (the size of the control flow graph).

Example. Consider the program in Figure 3.11 (a), in Figure 3.11 (b) the *DCP* as obtained by our abstraction procedure is shown. Note that the reset graph of Figure 3.11 (b) is *cyclic* ($[i]$ flows into $[j]$ on τ_1, on τ_3 $[j]$ flows into $[i]$). For ensuring termination of our algorithm we want to apply our previously discussed variable renaming on Figure 3.11 (b). However, Figure 3.11 (b) is *not stratifiable*: The *variable flow graph* is shown in Figure 3.11 (c). There is an SCC that contains the nodes "$[i], l_2$" and "$[j], l_2$". Renaming $[i]$ and $[j]$ at location l_2 to the same fresh variable would obviously alter the semantics of the program. We thus have to transform Figure 3.11 (b) into a (semantically equivalent) *stratifiable DCP* before applying our variable renaming.
We do so by introducing the following case distinction into the control-flow of the program: We distinguish between the case that a) transition τ_2 and b) transition τ_3 is executed after visiting l_2. For this purpose we split the control location l_2 into the two control locations l_{2a} and l_{2b}. The resulting *DCP* is shown in Figure 3.12 (a), its variable flow graph is shown in Figure 3.12 (b). The variable flow graph has 3 SCCs: $\mathtt{SCC}_1 = \{([i], l_1), ([i], l_{2a}), ([j], l_{2b})\}$, $\mathtt{SCC}_2 = \{([j], l_{2a})\}$, $\mathtt{SCC}_3 = \{([i], l_{2b})\}$. We conclude that Figure 3.12 (a) is indeed *stratifiable* because in all three SCCs each location appears

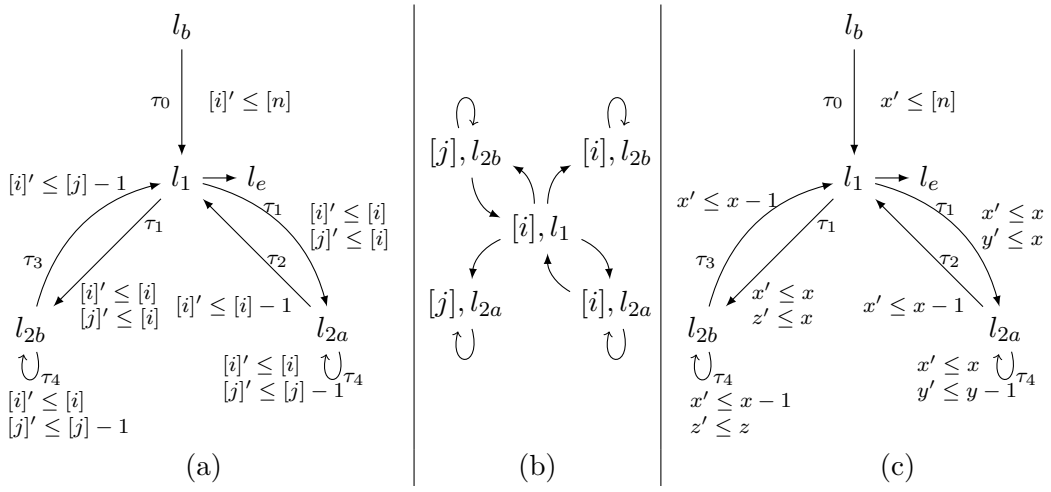


Figure 3.12: (a) *DCP* refined, (b) Variable Flow Graph, (c) *DCP* obtained by variable renaming

at most once. However, the reset graph of Figure 3.12 (a) remains cyclic, $[i]$ flows into $[j]$ and vice versa.
In accordance with our previously described procedure we therefore apply the following variable renaming to Figure 3.12 (a): The variables in $\mathtt{SCC}_1$ are renamed to x at the respective locations, i.e., $\varsigma([i], l_1) = \varsigma([i], l_{2a}) = \varsigma([j], l_{2b}) = x$. The variables in $\mathtt{SCC}_2$ are renamed to y at the respective locations, i.e., $\varsigma([j], l_{2a}) = y$. The variables in $\mathtt{SCC}_3$ are renamed to z at the respective locations, i.e., $\varsigma([i], l_{2b}) = z$. The resulting *DCP* is shown in Figure 3.12 (c).
In Figure 3.12 (c) there are two back-edges that end at location l_1, namely τ_2 and τ_3. By our bound algorithm (Definition 27) we obtain $T\mathcal{B}(\tau_2) = [n]$ and $T\mathcal{B}(\tau_3) = [n]$. We thus get the bound $2n$ (n has type *unsigned*) for the loop at l_1 of the program in Figure 3.11 (a).

Note, however, that the precise bound of the loop at l_1 in Figure 3.11 (a) is n. In Section 4.2.2 we discuss an extension of our bound algorithm by which we obtain the precise bound n. The extension exploits the fact that both transitions, τ_2 and τ_3 have the same *local bound* (namely x).

3.6 Finding Local Bounds

In this section we describe our algorithm for finding local bounds.

Intuition. Let $\tau = l_1 \xrightarrow{u} l_2 \in E$ and $\mathtt{v} \in \mathcal{V}$. Clearly, $\mathtt{v}$ is a *local bound* for τ if $\mathtt{v}$ decreases when executing τ, i.e., if $\mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u$ for some $\mathtt{c} < 0$. Moreover, $\mathtt{v}$ is a *local bound* for τ, if every time τ is executed also some other transition $t \in E$ is executed and

$\mathtt{v}$ is a local bound for t. This is, e.g., the case if t is always executed either before each execution of τ or after each execution of τ.

Algorithm. The above intuition can be turned into a simple three-step algorithm. Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a *DCP*. (1) We set $\zeta(\tau) = 1$ for all transitions τ that do not belong to a *strongly connected component* (SCC) of $\Delta\mathcal{P}$. (2) Let $\mathtt{v} \in \mathcal{V}$. We define $\xi(\mathtt{v}) \subseteq E$ to be the set of all transitions $\tau = l_1 \xrightarrow{u} l_2 \in E$ such that $\mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u$ for some $\mathtt{c} < 0$. For all $\tau \in \xi(\mathtt{v})$ we set $\zeta(\tau) = \mathtt{v}$. (3) Let $\mathtt{v} \in \mathcal{V}$ and $\tau \in E$. Assume τ was not yet assigned a local bound by (1) or (2). We set $\zeta(\tau) = \mathtt{v}$, if τ does not belong to a *strongly connected component* (SCC) of the directed graph (L, E') where $E' = E \setminus \{\xi(\mathtt{v})\}$ (the control flow graph of $\Delta\mathcal{P}$ without the transitions in $\xi(\mathtt{v})$).
If there are $\mathtt{v}_1 \neq \mathtt{v}_2$ s.t. $\tau \in \xi(\mathtt{v}_1) \cap \xi(\mathtt{v}_2)$ then $\zeta(\tau)$ is assigned either $\mathtt{v}_1$ or $\mathtt{v}_2$ non-deterministically. An alternative way of handling this case is as follows: We generate two local bound mappings, ζ_1 and ζ_2 where $\zeta_1(\tau) = \mathtt{v}_1$ and $\zeta_2(\tau) = \mathtt{v}_2$. This way we can systematically enumerate all possible choices, finally we apply our bound algorithm once based on ζ_1, based on ζ_2, etc., and finally take the minimum over all computed bounds. In our implementation, however, we follow the aforementioned greedy approach based on non-deterministic choice.

Discussion on Soundness. Soundness of step (1) and (2) is obvious. We discuss soundness of step (3): Let $\tau \in E$. If τ does not belong to an SCC of $(L, E \setminus \{\xi(\mathtt{v})\})$ we have that some transition in $\xi(\mathtt{v})$ (which decreases $\mathtt{v}$) has to be executed in between any two executions of τ. It remains to ensure that there is a decrease of $\mathtt{v}$ also for the last execution of τ: For special cases this is unfortunately not the case. Consider Figure 4.12 (b) (page 98). The above stated algorithm sets $\zeta(\tau_1) = [x]$. However, $[x]$ is not a *local bound* for τ_1 of Figure 4.12 (b) because there is no decrease of $[x]$ for the last execution of τ_1 (before executing τ_3).
It is straightforward to ensure soundness of the algorithm: Adding an edge from l_e to l_b forces the algorithm to take the last execution of a transition into account. I.e., we set $E' = E \cup \{l_e \xrightarrow{\emptyset} l_b\} \setminus \{\xi(\mathtt{v})\}$. Now our algorithm fails to find a local bound for τ_1 of Figure 4.12 (b), which is sound. We discuss how we handle the example in Figure 4.12 in Section 4.2.1.

Complexity. Step (1) and (2): can be implemented in linear time. Step (3): For each $\mathtt{v} \in \mathcal{V}$ we need to compute the SCCs of $(L, E \setminus \xi(\mathtt{v}))$. It is well known that SCCs can be computed in linear time (linear in the number of transitions and nodes). Since we need to perform one SCC computation per variable, step (3) is quadratic.

3.7 Example

We discussed in Section 2.2 how we obtain the *DCP* in Figure 2.2 (b) as an abstraction of Example `xnu` (Figure 2.1). Let us now exemplify our algorithm for finding *local bounds*

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}(\{p,q,r\})$ $+\ T\mathcal{B}(\{\tau_5,\tau_{2a},\tau_{3a}\})\times\max([0]+0,0)$ $+\ T\mathcal{B}(\{\tau_{3c},\tau_{2a},\tau_{3a}\})\times\max([0]+0,0)$ $+\ T\mathcal{B}(\{\tau_0,\tau_{2a},\tau_{3a}\})\times\max([0]+0,0)$ $+\ T\mathcal{B}(\{\tau_0,\tau_{3a}\})\times\max([0]+0,0)$ $+\ T\mathcal{B}(\{\tau_5,\tau_{3a}\})\times\max([0]+0,0)$ $+\ T\mathcal{B}(\{\tau_{3c},\tau_{3a}\})\times\max([0]+0,0)$ $=\texttt{Incr}(\{p,q,r\})+0+0+0+0+0+0$ $\to [l]+0+0+0+0+0+0$ $=[l]$	$\zeta(\tau_4)=p,$ $\mathfrak{R}(p)=\{$ $[0]\xrightarrow{\tau_0} r\xrightarrow{\tau_{2a}} q\xrightarrow{\tau_{3a}} p,$ $[0]\xrightarrow{\tau_{3c}} r\xrightarrow{\tau_{2a}} q\xrightarrow{\tau_{3a}} p,$ $[0]\xrightarrow{\tau_5} r\xrightarrow{\tau_{2a}} q\xrightarrow{\tau_{3a}} p,$ $[0]\xrightarrow{\tau_0} q\xrightarrow{\tau_{3a}} p,$ $[0]\xrightarrow{\tau_{3c}} q\xrightarrow{\tau_{3a}} p,$ $[0]\xrightarrow{\tau_5} q\xrightarrow{\tau_{3a}} p\}$ $[0]\in\mathcal{C},$ $\texttt{Incr}(\{p,q,r\})$
$\texttt{Incr}(\{p,q,r\})$	$\to \texttt{Incr}(p)+\texttt{Incr}(q)+\texttt{Incr}(r)$ $\to 0+0+[l]$ $=[l]$	$\texttt{Incr}(p),$ $\texttt{Incr}(q),$ $\texttt{Incr}(r)$
$\texttt{Incr}(p)$	$\to 0$	$\mathcal{I}(p)=\emptyset$
$\texttt{Incr}(q)$	$\to 0$	$\mathcal{I}(q)=\emptyset$
$\texttt{Incr}(r)$	$\to T\mathcal{B}(\tau_1)\times 1$ $\to [l]\times 1$ $=[l]$	$\mathcal{I}([p])=\{(\tau_1,1)\},$ $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}(x)+T\mathcal{B}(\tau_0)\times[l]$ $\to 0+T\mathcal{B}(\tau_0)\times[l]$ $\to 0+1\times[l]$ $=[l]$	$\zeta(\tau_1)=x,$ $\mathfrak{R}(x)=\{[l]\xrightarrow{\tau_0} x\},$ $[l]\in\mathcal{C},\ \texttt{Incr}(x),\ T\mathcal{B}(\tau_0)$
$\texttt{Incr}(x)$	$\to 0$	$\mathcal{I}(x)=\emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0)=1$

Table 3.9: Computation of $T\mathcal{B}(\tau_4)$ for Example xnu (Figure 2.1) by Definition 29 resp. Definition 31

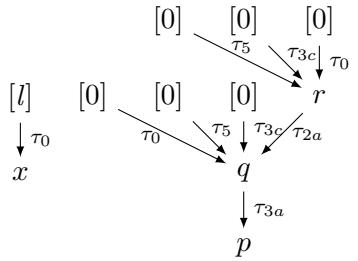


Figure 3.13: Reset Graph of Example xnu (Figure 2.2 (b))

(Section 3.6) and our bound algorithm (Section 3.5) on Figure 2.2 (b). We first apply our algorithm for finding local bounds:

- τ_0 does not belong to a *strongly connected component* of Figure 2.2 (b). We set $\zeta(\tau_0) = 1$.
- We have $\xi(q) = \emptyset$, $\xi(r) = \emptyset$, $\xi(x) = \{\tau_1\}$ and $\xi(p) = \{\tau_4\}$. We therefore set $\zeta(\tau_1) = x$ and $\zeta(\tau_4) = p$.
- We did not yet assign *local bounds* to the transitions τ_{2a},τ_{2b},τ_{3a},τ_{3b},τ_{3c},τ_5 and τ_6. We now remove $\xi(x)$, i.e., τ_1 from Figure 2.2 (b). Observe that after τ_1 is removed τ_{2a},τ_{2b},τ_{3a},τ_{3b},τ_{3c},τ_5 and τ_6 do no longer belong to a *strongly connect component*. We therefore set $\zeta(\tau_{2a}) = \zeta(\tau_{2b}) = \zeta(\tau_{3a}) = \zeta(\tau_{3b}) = \zeta(\tau_{3c}) = \zeta(\tau_5) = \zeta(\tau_6) = x$.

We discussed in Section 1.4.7 that the inner loop at l_4 of Example `xnu` has a *linear* bound, namely $\max(len, 0)$. Our algorithm *infers* the *linear* bound of the inner loop at l_4 based on the abstracted *DCP* in Figure 2.2 (b) and the *local bound mapping* $\zeta(\tau_0) = 1$, $\zeta(\tau_1) = x$, $\zeta(\tau_4) = p$, $\zeta(\tau_{2a}) = x$, $\zeta(\tau_{2b}) = x$, $\zeta(\tau_{3a}) = x$, $\zeta(\tau_{3b}) = x$, $\zeta(\tau_{3c}) = x$, $\zeta(\tau_5) = x$, and $\zeta(\tau_6) = x$: Table 3.9 states how our bound algorithm from Section 3.5 computes $T\mathcal{B}(\tau_4) = [l]$ for τ_4, the single transition of the loop at l_4. In Figure 3.13 the reset graph of Figure 2.2 (b) is shown. Recall, that the abstract variable $[l]$ represents the expression $\max(len, 0)$ of the concrete program.

Note that the reset graph of Figure 2.2 (b) (Shown in Figure 3.13) is a *reset forest*. Therefore $atm_1(\kappa) = atm(\kappa)$ for all reset chains κ of Figure 2.2 (b), as discussed in Section 3.5.2. The computation traces of Definition 29 and Definition 31 are thus equivalent for Figure 2.2 (b).

3.8 Path-Sensitive Reasoning

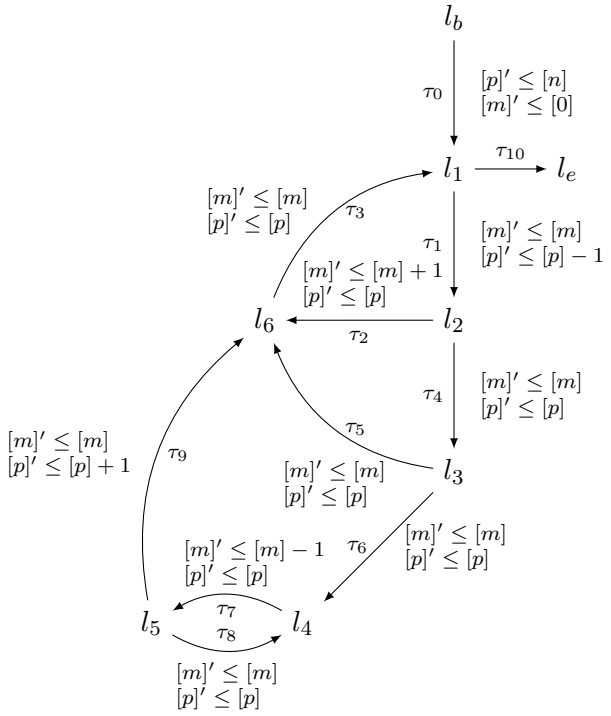
Example `s_SFD_process` shown in Figure 3.14 (a) is a simplified version of a nested loop in the function *s_SFD_process* in file *office_ghostscript/src/sfilter1.c* of the *cBench* Benchmark [cbe]. Example `s_SFD_process` poses the following challenge: We have that the condition $p > 0$ must hold before each execution of the outer loop. Further p is decremented on each execution of the outer loop. But there is also an execution of the outer loop in which p is incremented (when entering the else-branch). A closer look reveals that in case p is incremented $m > 0$ holds and that m is decremented. But, m can also be incremented on the outer loop. I..e, we have a *cyclic dependency* between the two loop counters p and m: If p decrements, m may increment and vice versa. Nevertheless the example terminates and the outer loop cannot be executed more than $2n - 1$ times as we show in the course of the following discussion.

In Figure 3.14 (b) the *DCP* is shown as it results from applying our abstraction procedure (Section 2.2) to Example `s_SFD_process`. Note that Figure 3.14 (b) is a VASS (Definition 21).

```
void s_SFD_process(uint n)
{
     uint p = n;
     uint m = 0;
l1   while(p > 0) {
       p--;
l2     if(?)
          m++;
l3     else if(m > 0) {
l4       do {
            m--;
l5        } while(m > 0 && ?)
          p++;
        }
l6   }
}
```

Complexity: $2n - 1$

(a)



(b)

Figure 3.14: (a) Example `s_SFD_process`, (b) *DCP* obtained by abstraction

Assume we want to compute a bound for loop at location l_4 of Example `s_SFD_process`, i.e., a transition bound for its single back-edge τ_8. Our bound algorithm from Definition 22 computes $T\mathcal{B}(\tau_8) = \infty$: We have that $[m]$ is a local bound for τ_8 and for τ_9. Variable $[p]$ is a local bound for τ_2. Variable $[m]$ is incremented on τ_2. Variable $[p]$ is incremented on τ_9. Due to this cyclic dependency between the increment of $[m]$ and the increment of $[p]$ our algorithm from Definition 22 returns '∞' when reasoning about the transition bound of τ_8.

We now discuss a *path-sensitive* extension of our bound algorithm which infers the tight transition bound n for τ_8. The precise transition bound for τ_8 is $n - 2$.

As for the path-insensitive case, we develop our path-sensitive algorithm step-by-step: We first present a path-sensitive algorithm for VASS in Section 3.8.1. We generalize this algorithm to *DCP*s with only constant resets in Section 3.8.2. Finally we present our path-sensitive bound algorithm for full *DCP*s in Section 3.8.3.

Note that we do not restate our insights on reasoning with reset chains (Section 3.5) in the context of our path-sensitive reasoning. It is straightforward to combine both ideas for obtaining one powerful bound algorithm. Our full bound algorithm is stated in Section 3.9.

Intuition. The underlying idea of our path-sensitive extension is to cancel out increments with decrements that lay on the same cyclic path. E.g., in Example `s_SFD_process` every increment of p on τ_9 is preceded by a decrement of p on τ_1. Let $\mathtt{v} \in \mathcal{V}$. For canceling out increments of $\mathtt{v}$ with decrements of $\mathtt{v}$ we (1) enumerate the cyclic paths (Definition 33) and (2) sum up the increments and decrements of $\mathtt{v}$ on each cyclic path (Definition 35). The total amount by which a variable $\mathtt{v}$ is incremented on a run ρ is then calculated by multiplying the number of times that a cyclic path that increments $\mathtt{v}$ is executed on ρ with the respective increment. We implement this reasoning through the re-definition of $\mathtt{Incr}(\mathtt{v})$ in Definition 36.

Definition 33 (Simple and (A)Cyclic Paths)**.** *Let $\Delta\mathcal{P}(L, E, l_e, l_b)$ be a DCP over $\mathcal{A}$. Let $\pi = l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} \dots l_n$ be a* finite *path of $\Delta\mathcal{P}$. By $trn(\pi)$ we denote the set $\{l_1 \xrightarrow{u_1} l_2, l_2 \xrightarrow{u_2} l_3, \dots\}$ of transitions of π. π is* acyclic *if π does not visit any location twice (i.e., $l_i \neq l_j$ for all $1 \leq i, j \leq n$ with $i \neq j$),* cyclic *else. π is* simple *if its prefix $\pi_{[1,n-1]}$ is* acyclic*. I.e., π is* simple and cyclic *iff it has the same start- and end-location and does not visit a location twice except for the start- and end-location. Given two simple and cyclic paths π_1 and π_2 we say that π_1 is* equivalent *to π_2 and write $\pi_1 \approx \pi_2$ iff $trn(\pi_1) = trn(\pi_2)$. Let $l_1, l_2 \in L$. By $\mathcal{S}(l_1, l_2)$ we denote the set of* simple paths *from $l_1 \in L$ to $l_2 \in L$. By $\mathcal{S}_a(l_1, l_2)$ we denote the set of* acyclic paths *from $l_1 \in L$ to $l_2 \in L$. Note that $\mathcal{S}_a(l_1, l_2) = \mathcal{S}(l_1, l_2)$ if $l_1 \neq l_2$ and $\mathcal{S}_a(l_1, l_2) = \emptyset$ else.*

Notation: In the following we denote by $\tau_1 \circ \tau_2$ the path that results from concatenating the transitions τ_1 and τ_2.

Example: For Figure 3.14 (b) we have: The set of *simple and cyclic* paths is $\mathcal{S}(l_1, l_1) \cup \mathcal{S}(l_4, l_4)$ where $\mathcal{S}(l_1, l_1) = \{\tau_1 \circ \tau_2 \circ \tau_3, \tau_1 \circ \tau_4 \circ \tau_5 \circ \tau_3, \tau_1 \circ \tau_4 \circ \tau_6 \circ \tau_7 \circ \tau_9 \circ \tau_3\}$ and $\mathcal{S}(l_4, l_4) = \{\tau_7 \circ \tau_8\}$. We have $\tau_7 \circ \tau_8 \approx \tau_8 \circ \tau_7$ because $trn(\tau_7 \circ \tau_8) = \{\tau_7, \tau_8\} = trn(\tau_8 \circ \tau_7)$. Further $\mathcal{S}(l_b, l_e) = \{\tau_0 \circ \tau_{10}\}$ is the set of *simple* paths from l_b to l_e.

Definition 34 (Decrements)**.** *Let $\mathtt{v} \in \mathcal{V}$. The set*

$$\mathcal{D}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in E \times \mathbb{Z} \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u, \mathtt{c} < 0\}$$

is the set of decrements *of* $\mathtt{v}$.

Example: For Figure 3.14 (b) we have: $\mathcal{D}([m]) = \{(\tau_7, -1)\}$, $\mathcal{D}([p]) = \{(\tau_1, -1)\}$.

Definition 35 (Incrementing and Decrementing Paths)**.** *Let $\Delta\mathcal{P}(L, E, l_e, l_b)$ be a DCP over $\mathcal{A}$. Let $\pi = l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} \dots l_n$ be a* finite *path of $\Delta\mathcal{P}$. Let $\mathtt{v} \in \mathcal{V}$. Let*

$$\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) = \sum_{(\tau,\mathtt{c}) \in \mathcal{I}(\mathtt{v}) \cup \mathcal{D}(\mathtt{v}) \ s.t. \ \tau \in trn(\pi)} \mathtt{c}$$

denote the sum of all increments and decrements of $\mathtt{v}$ on π. We define

$$\mathtt{C}^+(\mathtt{v}) = \{\pi \in \bigcup_{l \in L} \mathcal{S}(l, l) \cup \mathcal{S}(l_b, l_e) \mid \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) > 0\}$$

to denote the set of incrementing *paths of* $\mathtt{v}$. *Accordingly*

$$\mathtt{C}^-(\mathtt{v}) = \{\pi \in \bigcup_{l \in L} \mathcal{S}(l,l) \cup \mathcal{S}(l_b, l_e) \mid \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) < 0\}$$

denotes the set of decrementing *paths of* $\mathtt{v}$.

Example: For Figure 3.14 (b) we have: $\mathtt{C}^+([m]) = \{\tau_1 \circ \tau_2 \circ \tau_3\}$, $\mathtt{C}^-([m]) = \{\tau_7 \circ \tau_8\}$, $\mathtt{C}^+([p]) = \emptyset$ and $\mathtt{C}^-([p]) = \{\tau_1 \circ \tau_2 \circ \tau_3, \tau_1 \circ \tau_4 \circ \tau_5 \circ \tau_3\}$.

Discussion. Note that $\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ sums up all increments and decrements of $\mathtt{v}$ on π, *ignoring* possible *resets* of $\mathtt{v}$ on π.
Further note that in the definition of $\mathtt{C}^+(\mathtt{v})$ resp. $\mathtt{C}^-(\mathtt{v})$ also the *simple* paths from l_b to l_e are considered, beside the *simple and cyclic* paths. This ensures that we consider also those *increments* resp. *decrements* of $\mathtt{v}$ which are not on a *cyclic* path in $\Delta\mathcal{P}$.

3.8.1 Path-Sensitive Reasoning for VASS

Definition 36 (Bound Algorithm for VASS (path-sensitive))**.** *Let* $\Delta\mathcal{P} = (L, E, l_b, l_e)$ *be a DCP over* $\mathcal{A}$. *Let* $\zeta : E \to Expr(\mathcal{A})$. *We define* $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ *as:*

$T\mathcal{B}(\tau) = \zeta(\tau)$, *if* $\zeta(\tau) \notin \mathcal{V}$, *else*
$T\mathcal{B}(\tau) = \mathtt{Incr}(\zeta(\tau)) + \sum_{(\_,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\zeta(\tau))} \max(\mathtt{a} + \mathtt{c}, 0)$

where

1. $\mathtt{Incr}(\mathtt{v}) = \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} T\mathcal{B}(trn(\pi)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ *(we set* $\mathtt{Incr}(\mathtt{v}) = 0$ *for* $\mathtt{C}^+(\mathtt{v}) = \emptyset$*)*
2. $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \le i \le n} T\mathcal{B}(\tau_i)$

Discussion. $T\mathcal{B}(\tau)$ is defined exactly as for the path-insensitive case of our algorithm for VASS (Definition 22). We implement path-sensitivity in our algorithm by redefining the sub-routine $\mathtt{Incr}(\mathtt{v})$: $\mathtt{Incr}(\mathtt{v})$ is defined over the incrementing *paths* $\mathtt{C}^+(\mathtt{v})$ of $\mathtt{v}$ rather than over the incrementing *transitions* $\mathcal{I}(\mathtt{v})$ of $\mathtt{v}$. We over-approximate the number of times that a given cyclic path $\pi \in \mathtt{C}^+(\mathtt{v})$ may appear on a run by $T\mathcal{B}(trn(\pi)) = \min_{\tau \in trn(\pi)} T\mathcal{B}(\tau)$: A path in $\Delta\mathcal{P}$ cannot be taken more often than the number of times that any transitions on the path can be taken.

Example I. Consider Example `s_SFD_process` in Figure 3.14. We want to compute a loop bound for inner loop at l_4, i.e., a bound for τ_8, the single back-edge of the loop. Our algorithm from Definition 36 computes $T\mathcal{B}(\tau_8) = [n]$, i.e., we get the tight bound n for the inner loop of Example `s_SFD_process` (n has type *unsigned*). The details are given in Table 3.10. Note that the *precise loop bound* for the loop at l_4 is $\max(n-2, 0)$.

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_8)$	$\to \texttt{Incr}([m]) + \max([0]+0, 0)$ $= \texttt{Incr}([m])$ $\to [n]$	$\zeta(\tau_8) = [m]$, $\mathcal{R}([m]) = (\tau_0, [0], 0)$, $[0] \in \mathcal{C}$, $\texttt{Incr}([m])$
$\texttt{Incr}([m])$	$\to T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\}) \times 1$ $\to [n] \times 1$ $= [n]$	$\texttt{C}^+([m]) = \{\tau_1 \circ \tau_2 \circ \tau_3\}$, $T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\})$
$T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\})$	$\to \min(T\mathcal{B}(\tau_1), T\mathcal{B}(\tau_2), T\mathcal{B}(\tau_3))$ $\to \min(\infty, [n], \infty)$ $= [n]$	$T\mathcal{B}(\tau_1)$, $T\mathcal{B}(\tau_2)$, $T\mathcal{B}(\tau_3)$
$T\mathcal{B}(\tau_1)$	$\to \infty$	$\zeta(\tau_1) = \infty$
$T\mathcal{B}(\tau_2)$	$\to \texttt{Incr}([p]) + \max([n]+0, 0)$ $\to 0 + \max([n]+0, 0)$ $= [n]$	$\zeta(\tau_2) = [p]$, $\mathcal{R}([p]) = (\tau_0, [n], 0)$, $[n] \in \mathcal{C}$, $\texttt{Incr}([p])$
$\texttt{Incr}([p])$	$\to 0$	$\texttt{C}^+([p]) = \emptyset$
$T\mathcal{B}(\tau_3)$	$\to \infty$	$\zeta(\tau_3) = \infty$

Table 3.10: Computation of $T\mathcal{B}(\tau_8)$ for Example `s_SFD_process` (Figure 3.14 (b)) by Definition 36

Definition 37 (Simple Paths that contain a given Transition)**.** *Let $\Delta\mathcal{P}(L, E, l_e, l_b)$ be a DCP. Let $\tau \in E$. We define $C(\tau) = \{\pi \in \bigcup_{l \in L} \mathcal{S}(l, l) \cup \mathcal{S}(l_b, l_e) \mid \tau \in trn(\pi)\}$.*

Example: Consider Figure 3.14 (b): We have $C(\tau_1) = \mathcal{S}(l_1, l_1)$ because τ_1 is on all simple paths that start and end at l_1.

Theorem 5 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free VASS over $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for $\Delta\mathcal{P}$ s.t. for all $\tau \in E$ with $\zeta(\tau) \in \mathcal{V}$ it holds that $C(\tau) \subseteq \texttt{C}^-(\zeta(\tau))$. Let $\tau \in E$. Let $T\mathcal{B}(\tau)$ be as defined in Definition 36. Let ρ be a* complete *run of $\Delta\mathcal{P}$. Then $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ on ρ.*

Proof: Definition 5 is a special case of Definition 40. Soundness of Definition 40 is proven in Section 7.3.3.

Discussion. In comparison to previous soundness theorems we have two restrictions in Theorem 5:
1) For the local bound mapping ζ it is required that "for all $\tau \in E$ with $\zeta(\tau) \in \mathcal{V}$ it holds that $C(\tau) \subseteq \texttt{C}^-(\zeta(\tau))$". We call such a *local bound mapping* a *path-sensitive local*

bound mapping (Definition 38). The restriction to path-sensitive local bounds is justified by the following observation: Consider τ_1 in Figure 3.14 (b). Obviously $[p]$ is a *local bound* for τ_1. Note that $C(\tau_1) \not\subseteq \mathtt{C}^-([p])$ because $\tau_1 \circ \tau_4 \circ \tau_6 \circ \tau_7 \circ \tau_9 \circ \tau_3 \in C(\tau_1)$ but $\tau_1 \circ \tau_4 \circ \tau_6 \circ \tau_7 \circ \tau_9 \circ \tau_3 \notin \mathtt{C}^-([p])$. I.e., $[p]$ is not a *path-sensitive local bound* for τ_1. If we run Definition 36 based on $\zeta(\tau) = [p]$ we obtain $T\mathcal{B}(\tau_1) = [n]$ because $\mathtt{C}^+([p]) = \emptyset$. However, $[n]$ is *not* a valid bound for τ_1: the reasoning misses that $[p]$ is *incremented* when executing τ_9 which allows for additional executions of τ_1. In Table 3.10 we ensure soundness of the reasoning by setting $\zeta(\tau_1) = \infty$. We discuss next how we generate *path-sensitive local bound mappings*. In Section 4.2.1 we show how our algorithm infers a *bound* for τ_1 based on assigning τ_1 a *set of path-sensitive local bounds* (Table 4.2).
2) Soundness is only guaranteed for *complete* runs of $\Delta\mathcal{P}$, i.e., for runs that end at l_e. The restriction is necessary because we cancel out increments of $\mathtt{v} \in \mathcal{V}$ with decrements of $\mathtt{v}$ on *cyclic paths*: this reasoning assumes that cyclic paths are completely executed during program run. We take care of the case that a loop is left during execution (break-statements) by considering the simple paths from l_b to l_e in Definition 35 (see discussion under Definition 35).

Definition 38 (Path Sensitive Local Bound Mapping)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over atoms $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$. ζ is a* path-sensitive local bound mapping *for $\Delta\mathcal{P}$ if ζ is a* local bound mapping *for $\Delta\mathcal{P}$ and for all $\tau \in E$ with $\zeta(\tau) \in \mathcal{V}$ it holds that $C(\tau) \subseteq \mathtt{C}^-(\zeta(\tau))$.*

I.e., a path-sensitive local bound mapping is a local bound mapping s.t. all simple and cyclic paths that contain τ are *decrementing* paths of $\zeta(\tau)$.

Finding Path Sensitive Local Bounds. Let $\tau \in E$. We infer a *path-sensitive local bound* for τ as follows: For each $\mathtt{v} \in \mathcal{V}$ s.t. $\mathtt{v}$ is a *local bound* for τ (see Section 3.6 on how we determine such $\mathtt{v}$) we check whether $C(\tau) \subseteq \mathtt{C}^-(\mathtt{v})$. For the first such local bound $\mathtt{v}$ for τ we set $\zeta(\tau) = \mathtt{v}$. If the check fails for all local bounds for τ we set $\zeta(\tau) = \infty$.

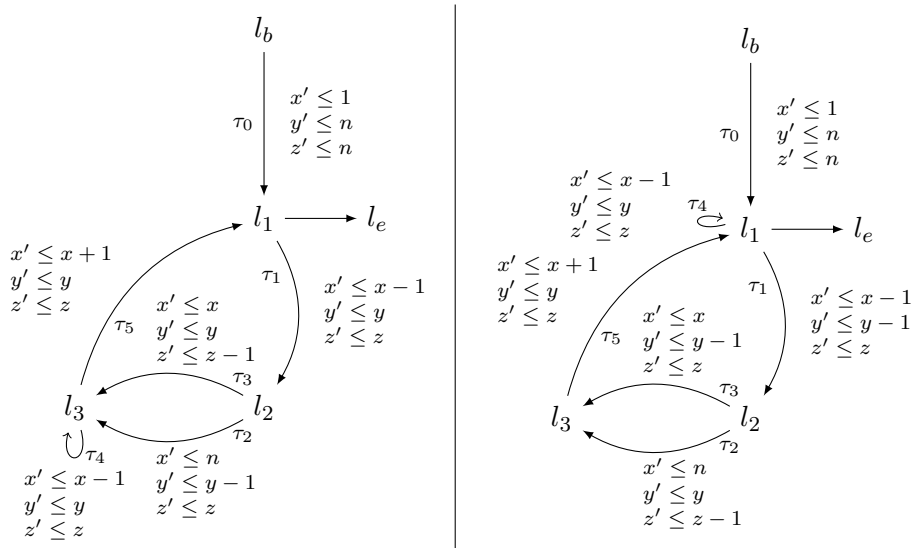
3.8.2 Path-Sensitive Reasoning for *DCP*s with only constant resets.

Definition 39 (Bound Algorithm for *DCP*s with only constant resets (path-sensitive))**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$.*
We define $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:

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

where

1. $\mathtt{Incr}(\mathtt{v}) = \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} T\mathcal{B}(trn(\pi)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ *(we set* $\mathtt{Incr}(\mathtt{v}) = 0$ *for* $\mathtt{C}^+(\mathtt{v}) = \emptyset$*)*
2. $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \le i \le n} T\mathcal{B}(\tau_i)$



(a) Bound for τ_4: n^2

(b) Bound for τ_4: $n^2 + 1$

Figure 3.15: Examples for path sensitive reasoning with constant resets

Discussion. $T\mathcal{B}(\tau)$ is defined exactly as for the path-insensitive case of our algorithm for *DCP*s with only constant resets (Definition 24). We implement path-sensitivity in our bound algorithm for *DCP*s with only constant resets (Definition 24) by re-defining `Incr(v)` as discussed for VASS in Section 3.8.1. In the following we give intuition for the soundness of Definition 39 for *DCP*s with only constant resets by means of the examples in Figure 3.15. Soundness is formally stated in Theorem 6.

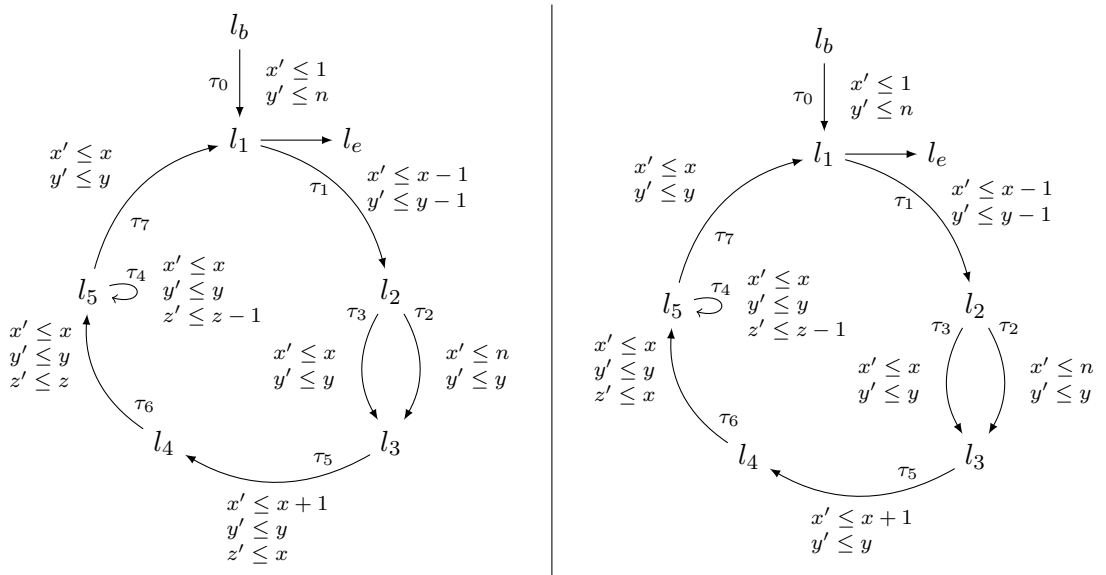
Example I. Consider the *DCP* in Figure 3.15 (a). For the ease of the discussion we assume that the symbolic constant '1' to which x is reset on τ_0 is evaluated to the natural number 1. Consider the transition τ_4. Variable x is a local bound for τ_4. The highest value that x can reach during program run (the precise variable bound of x) is $n + 1$: We reset x to n by executing τ_2, afterward we execute τ_5, incrementing x by 1. Given that the increment of x on τ_1 *does* affect the upper bound of x, it may thus seem inadmissible to cancel out the increment of x on τ_5 with the decrement of x on τ_1. But at location l_3, where τ_4 is situated, x can have at most the value n: x has value n after executing τ_2. If we increment x by executing τ_5, we have to execute τ_1 in order to reach l_3. On τ_1 variable x is decremented. Thus, τ_4 can be executed at most n times consecutively. Since τ_2 (local bound y), where x is set to n, can be executed n times in total, we get the precise transition bound n^2 for τ_4. Our path-sensitive algorithm for *DCP*s with constant resets (Definition 39) computes the tight bound $T\mathcal{B}(\tau_4) = n \times n + 1 = n^2 + 1$. The details are given in Table 3.11. Note that our path-insensitive algorithm (Definition 24) ignores the decrement of x by 1 on τ_1 and therefore infers the transition bound $(n \times 1 + 1 \times n) + n \times n + 1 \times 1 = 2n + n^2 + 1$ for τ_4.

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}(x) + T\mathcal{B}(\tau_2) \times \max(n+0,0) + T\mathcal{B}(\tau_0) \times \max(1+0,0)$ $= \texttt{Incr}(x) + T\mathcal{B}(\tau_2) \times n + T\mathcal{B}(\tau_0) \times 1$ $\to 0 + T\mathcal{B}(\tau_2) \times n + T\mathcal{B}(\tau_0) \times 1$ $\to 0 + n \times n + T\mathcal{B}(\tau_0) \times 1$ $\to 0 + n \times n + 1 \times 1$ $= n \times n + 1$	$\zeta(\tau_4) = x$, $\mathcal{R}(x) = \{(\tau_2, n, 0), (\tau_0, 1, 0)\}$, $n, 1 \in \mathcal{C}$, $\texttt{Incr}(x)$, $T\mathcal{B}(\tau_2)$, $T\mathcal{B}(\tau_1)$
$\texttt{Incr}(x)$	$\to 0$	$\texttt{C}^+(x) = \emptyset$
$T\mathcal{B}(\tau_2)$	$\to \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times \max(n+0,0)$ $= \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times n$ $\to 0 + T\mathcal{B}(\tau_0) \times n$ $\to 0 + 1 \times n$ $= n$	$\zeta(\tau_2) = y$, $\mathcal{R}(y) = \{(\tau_0, n, 0)\}$, $n \in \mathcal{C}$, $\texttt{Incr}(y)$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$\texttt{Incr}(y)$	$\to 0$	$\texttt{C}^+(y) = \emptyset$

Table 3.11: Computation of $T\mathcal{B}(\tau_4)$ for Figure 3.15 (a) and (b) by Definition 39

Example II. Consider the *DCP* in Figure 3.15 (b). The only difference to Figure 3.15 (a) is that the simple loop τ_4 is now located at l_1 rather than l_3. We have discussed before that x (the local bound of τ_4) can reach the value $n+1$ at l_1. I.e., in contrast to Figure 3.15 (a), transition τ_4 can be executed $n+1$ times consecutively. Nevertheless, τ_4 can be executed at most n^2+1 times in total during program run: If we execute τ_4 $n+1$ times, x has value 0 and we cannot execute the outer loop any further. I.e., only after the last execution of the outer loop τ_4 can be executed $n+1$ times, on all other executions of the outer loop we can execute τ_4 only n times. Given that τ_2 (which resets x to n) can be executed n times in total, we get the bound $(n-1 \times n) + (1 \times n + 1) = n^2 + 1$ for τ_4. As shown in Table 3.11 our path-sensitive algorithm (Definition 39) infers the *precise* transition bound for τ_4 in Figure 3.15 (b): $T\mathcal{B}(\tau_4) = n \times n + 1 = n^2 + 1$. We conclude that the number of times τ_4 can be executed is in fact not affected by the increment of its local bound x on τ_5. The additional execution of τ_4 after the last execution of the outer loop is due to the initial value 1 of x.

Discussion. Our discussion of Figure 3.15 (a) and (b) shows that the precise transition bound of τ_4 depends on the location at which τ_4 is situated. In case τ_4 starts and ends at location l_3 (Figure 3.15 (a)) it can be executed n^2 times in total, if situated at location l_1 (Figure 3.15 (b)) τ_4 can be executed n^2+1 times in total. The reasoning of our algorithm (Definition 39), however, is identical for τ_4 in Figure 3.15 (a) and in Figure 3.15 (b) (Table 3.11). While paths are enumerated in $\texttt{Incr}(\texttt{v})$, our algorithm does not take the start-location of τ_4 on a given path into account and computes $T\mathcal{B}(\tau_4) = n^2 + 1$ for both, Figure 3.15 (a) and Figure 3.15 (b).



(a) Bound for τ_4: n^2

(b) Bound for τ_4: $n^2 + n$

Figure 3.16: Examples for path sensitive reasoning for *DCP*s

Theorem 6 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and deterministic DCP* with only constant resets *over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* path-sensitive local bound mapping *for $\Delta\mathcal{P}$. Let $\tau \in E$. Let $T\mathcal{B}$ be as defined in Definition 39. Let ρ be a* complete *run of $\Delta\mathcal{P}$. Then $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ on ρ.*

Proof: Definition 39 is a special case of Definition 40. Soundness of Definition 40 is proven in Section 7.3.3.

Discussion. Soundness is restricted to *path-sensitive* local bound mappings and *complete* runs for the same reasons already discussed for the case of VASS (Section 3.8.1).

3.8.3 Path-Sensitive Reasoning for *DCP*s

Consider the *DCP* in Figure 3.16 (a). Figure 3.16 (a) is similar to the examples from Figure 3.15: Again we are interested in the transition bound of τ_4. Transition τ_4 has local bound z. Variable z, however, is set to x on τ_5. The number of times τ_4 can be executed consecutively thus depends on the upper bound of x. As in our previous examples (Figure 3.15, discussed in Section 3.8.2) the precise upper bound invariant for x is $n + 1$. But, at l_3, from where x flows into a, x can have at most the value n. In total, transition τ_5 (local bound y), on which z is set to x, can be executed n times. We thus get the precise transition bound n^2 for τ_4 in Figure 3.16 (a).

Consider the *DCP* in Figure 3.16 (b). The only difference to Figure 3.16 (a) is that z is reset to x on τ_6 rather than on τ_5. I.e., x at l_4 now flows into z. At l_4 the precise variable bound of x is $n+1$. Since τ_6 (local bound y) can be executed n times in total, we get the precise transition bound $n \times (n+1) = n^2 + n$ for τ_4.

The examples in Figure 3.16 demonstrate, that computing precise transition bounds requires to compute *upper bound invariants* per *location*: the precise upper bound invariant for x at l_3 is n, whereas at l_4 it is $n+1$. This small and apparently negligible increase of 1 in the upper bound of x allows for n additional executions of τ_4. This is in contrast to the examples in Figure 3.15 (*DCP*s with constant resets), where x is the local bound of τ_4: Here an increase of 1 in the upper bound of x allows for only one additional execution of τ_4.

Our observation motivates the new definition of $V\mathcal{B}$ in Definition 40: $V\mathcal{B}(\mathtt{v}, l)$ infers an *upper bound invariant* for $\mathtt{v}$ at location l.

Definition 40 (Bound Algorithm (path-sensitive))**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$. We define $V\mathcal{B} : (\mathcal{A} \times L) \mapsto Expr(\mathcal{A})$ and $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

$$
\begin{aligned}
&V\mathcal{B}(\mathtt{a}, l) = \mathtt{a}, \textit{ if } \mathtt{a} \in \mathcal{A} \setminus \mathcal{V}, \textit{ else}\\
&V\mathcal{B}(\mathtt{v}, l) = \mathtt{Incr}(\mathtt{v}) + \max_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} (V\mathcal{B}(\mathtt{a}, l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}))\\
&\qquad \textit{(we set } \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) = 0 \textit{ for } \mathcal{S}_a(l_2, l) = \emptyset)\\
&T\mathcal{B}(\tau) = \zeta(\tau), \textit{ if } \zeta(\tau) \in \mathcal{C}, \textit{ else}\\
&T\mathcal{B}(\tau) = \mathtt{Incr}(\zeta(\tau)) + \sum_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} T\mathcal{B}(l_1 \xrightarrow{u} l_2) \times \max(V\mathcal{B}(\mathtt{a}, l_1) + \mathtt{c}, 0)
\end{aligned}
$$

where

1. $\mathtt{Incr}(\mathtt{v}) = \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} T\mathcal{B}(trn(\pi)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ *(we set* $\mathtt{Incr}(\mathtt{v}) = 0$ *for* $\mathtt{C}^+(\mathtt{v}) = \emptyset$*)*
2. $T\mathcal{B}(\{\tau_1, \tau_2, \ldots, \tau_n\}) = \min_{1 \le i \le n} T\mathcal{B}(\tau_i)$

Discussion. We pass the additional parameter l to $V\mathcal{B}$. l denotes the location for which the variable bound of $\mathtt{v}$ is to be computed: The term $\max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ adds the total amount by which $\mathtt{v}$ may be incremented on any acyclic path from the reset of $\mathtt{v}$ at l_2 to l, the location for which the variable bound is to be computed: In Figure 3.16 (b) x is reset to n on τ_2, but between l_3 and l_4 x is incremented by 1. We have $\max_{\pi \in \mathcal{S}_a(l_3, l_4)} \mathtt{Sum}\mathcal{ID}(\pi)(x) = \mathtt{Sum}\mathcal{ID}(l_3 \xrightarrow{u_5} l_4)(x) = 1$. We thus get $V\mathcal{B}(x, l_4) = n + 1$, see Table 3.13 for details.
The definition of $T\mathcal{B}$ in Definition 40 is nearly identical to Definition 27, we only adjust

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}(z) +$ $\quad T\mathcal{B}(\tau_5) \times \max(V\mathcal{B}(x, l_3) + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_5) \times \max(V\mathcal{B}(x, l_3) + 0, 0)$ $\to 0 + n \times \max(V\mathcal{B}(x, l_3) + 0, 0)$ $\to 0 + n \times \max(n + 0, 0)$ $= n \times n$	$\zeta(\tau_4) = z,$ $\mathcal{R}(z) = \{(\tau_5, x, 0)\},$ $\texttt{Incr}(z),$ $T\mathcal{B}(\tau_5),$ $V\mathcal{B}(x, l_3)$
$\texttt{Incr}(z)$	$\to 0$	$\texttt{C}^+(z) = \emptyset$
$T\mathcal{B}(\tau_5)$	$\to \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times \max(n + 0, 0)$ $= \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times n$ $\to 0 + T\mathcal{B}(\tau_0) \times n$ $\to 0 + 1 \times n$ $= n$	$\zeta(\tau_2) = y,$ $\mathcal{R}(y) = \{(\tau_0, n, 0)\},$ $n \in \mathcal{C},$ $\texttt{Incr}(y),$ $T\mathcal{B}(\tau_0)$
$V\mathcal{B}(x, l_3)$	$\to \texttt{Incr}(x) +$ $\quad \max(n + 0 + \max_{\pi \in \mathcal{S}_a(l_3, l_3)} \texttt{Sum}\mathcal{ID}(\pi)(x),$ $\qquad 1 + 0 + \max_{\pi \in \mathcal{S}_a(l_1, l_3)} \texttt{Sum}\mathcal{ID}(\pi)(x))$ $\to 0 + \max(n + 0 + 0,$ $\quad 1 + 0 + \max_{\pi \in \mathcal{S}_a(l_1, l_3)} \texttt{Sum}\mathcal{ID}(\pi)(x))$ $\to 0 + \max(n + 0 + 0, 1 + 0 + \max(-1, -1))$ $= n$	$\mathcal{R}(x) =$ $\quad \{(\tau_2, n, 0), (\tau_0, 1, 0)\},$ $n, 1 \in \mathcal{C},$ $\texttt{Incr}(x),$ $\mathcal{S}_a(l_3, l_3) = \emptyset,$ $\mathcal{S}_a(l_1, l_3) =$ $\qquad \{l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} l_3,$ $\qquad l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_3} l_3\}$
$\texttt{Incr}(y)$	$\to 0$	$\texttt{C}^+(y) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$\texttt{Incr}(x)$	$\to 0$	$\texttt{C}^+(x) = \emptyset$

Table 3.12: Computation of $T\mathcal{B}(\tau_4)$ for Figure 3.16 (a) by Definition 40

the call to $V\mathcal{B}$: we pass the location l_2 at which the variable bound for `a` is needed. As discussed in Section 3.8.1 for the case of VASS, path-sensitivity is implemented into our algorithm by defining $\texttt{Incr}(\texttt{v})$ over the incrementing *paths* $\texttt{C}^+(\texttt{v})$ rather than over the incrementing *transitions* $\mathcal{I}(\texttt{v})$.

Example I Consider Figure 3.16 (a). Our algorithm from Definition 40 infers the *precise* transition bound $T\mathcal{B}(\tau_4) = n \times n$ for τ_4. The details are given in Table 3.12.

Example II Consider Figure 3.16 (b). Our algorithm from Definition 40 infers the *precise* transition bound $T\mathcal{B}(\tau_4) = n \times n + n$ for τ_4. The details are given in Table 3.13.

Computing *upper bound invariants* per location. In the definition of $V\mathcal{B}(\texttt{v}, l)$ the location l is only used in the term $\max_{\pi \in \mathcal{S}_a(l_2, l)} \texttt{Sum}\mathcal{ID}(\pi)(\texttt{v})$. This term, however, is

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}(z) +$ $\quad T\mathcal{B}(\tau_6) \times \max(V\mathcal{B}(x, l_3) + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_6) \times \max(V\mathcal{B}(x, l_3) + 0, 0)$ $\to 0 + n \times \max(V\mathcal{B}(x, l_4) + 0, 0)$ $\to 0 + n \times \max(n + 1, 0)$ $= n \times (n + 1)$	$\zeta(\tau_4) = z$, $\mathcal{R}(z) = \{(\tau_6, x, 0)\}$, $\texttt{Incr}(z)$, $T\mathcal{B}(\tau_6)$, $V\mathcal{B}(x, l_4)$
$\texttt{Incr}(z)$	$\to 0$	$\texttt{C}^+(z) = \emptyset$
$T\mathcal{B}(\tau_6)$	$\to \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times \max(n + 0, 0)$ $= \texttt{Incr}(y) + T\mathcal{B}(\tau_0) \times n$ $\to 0 + T\mathcal{B}(\tau_0) \times n$ $\to 0 + 1 \times n$ $= n$	$\zeta(\tau_2) = y$, $\mathcal{R}(y) = \{(\tau_0, n, 0)\}$, $n \in \mathcal{C}$, $\texttt{Incr}(y)$, $T\mathcal{B}(\tau_0)$
$V\mathcal{B}(x, l_4)$	$\to \texttt{Incr}(x) +$ $\max(n + 0 + \max_{\pi \in \mathcal{S}_a(l_3, l_4)} \texttt{Sum}\mathcal{ID}(\pi)(x),$ $\quad 1 + 0 + \max_{\pi \in \mathcal{S}_a(l_1, l_4)} \texttt{Sum}\mathcal{ID}(\pi)(x))$ $\to 0 + \max(n + 0 + 1,$ $\quad 1 + 0 + \max_{\pi \in \mathcal{S}_a(l_1, l_4)} \texttt{Sum}\mathcal{ID}(\pi)(x))$ $\to 0 + \max(n + 0 + 1, 1 + 0 + \max(0, 0))$ $= \max(n + 1, 1)$ $= n + 1$	$\mathcal{R}(x) = \{(\tau_2, n, 0),$ $\quad (\tau_0, 1, 0)\}$, $n, 1 \in \mathcal{C}$, $\texttt{Incr}(x)$, $\mathcal{S}_a(l_3, l_4) = \{l_1 \xrightarrow{u_5} l_4\}$, $\mathcal{S}_a(l_1, l_4) = \{$ $\quad l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} l_3 \xrightarrow{u_5} l_4,$ $\quad l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_3} l_3 \xrightarrow{u_5} l_4\}$
$\texttt{Incr}(y)$	$\to 0$	$\texttt{C}^+(y) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$\texttt{Incr}(x)$	$\to 0$	$\texttt{C}^+(x) = \emptyset$

Table 3.13: Computation of $T\mathcal{B}(\tau_4)$ for Figure 3.16 (b) by Definition 40

crucial for the overall soundness of the algorithm: Consider the computation of $T\mathcal{B}(\tau_4)$ for Figure 3.16 (b) in Table 3.13. Consider the row computing $V\mathcal{B}(x, l_4)$. The term $\max_{\pi \in \mathcal{S}_a(l_3, l_4)} \texttt{Sum}\mathcal{ID}(\pi)(x)$ evaluates to 1 because x is increased by 1 on $l_3 \xrightarrow{u_5} l_4 \in \mathcal{S}_a(l_3, l_4)$ in Figure 3.16 (b). We therefore get $V\mathcal{B}(x, l_4) = n + 1$. As a result we obtain the precise (see discussion above) transition bound $n \times (n + 1)$ for τ_4. In contrast, if we ignored the increment of x by 1 on τ_5, we would obtain $n \times n$ as a bound for τ_4. As discussed previously, $n \times n$ is *not* a transition bound for τ_4 in Figure 3.16 (b).

Theorem 7 (Soundness)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and deterministic DCP over atoms $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* path-sensitive local bound mapping *for $\Delta\mathcal{P}$. Let $\texttt{a} \in \mathcal{A}$ and $\tau \in E$. Let $l \in L$ be s.t. $\texttt{a} \in \texttt{def}(l)$. Let $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\texttt{a}, l)$ be as defined in Definition 40. Let ρ be a* complete *run of $\Delta\mathcal{P}$. We have: (1) $[\![T\mathcal{B}(\tau)]\!]$ is a* transition bound *for τ on ρ. (2) $[\![V\mathcal{B}(\texttt{a}, l)]\!]$ is an* upper bound invariant *for $\texttt{a}$ at l on ρ.*

Proof: See Section 7.3.3.

3.9 Full Bound Algorithm

Our full bound algorithm finally results from combining the reasoning on reset chains (Definition 31) and the path-sensitive reasoning (Definition 40).

Definition 41 (Notation)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\kappa = \mathtt{a}_n \xrightarrow{\tau_n, c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1}, c_{n-1}} \dots \mathtt{a}_0$ be a* reset chain *of $\Delta\mathcal{P}$. By $sl(\kappa)$ we denote the source location of transition τ_n, i.e., if $\tau_n = l_1 \xrightarrow{u} l_2$ then $sl(\kappa) = l_1$.*

Definition 42 (Full Bound Algorithm)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \mapsto Expr(\mathcal{A})$. We define $V\mathcal{B} : (\mathcal{A} \times L) \mapsto Expr(\mathcal{A})$ and $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

$$V\mathcal{B}(\mathtt{a}, l) = \mathtt{a},\ \textit{if}\ \mathtt{a} \in \mathcal{A} \setminus \mathcal{V},\ \textit{else}$$

$$V\mathcal{B}(\mathtt{v}, l) = \mathtt{Incr}(\mathtt{v}) + \max_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} \left(V\mathcal{B}(\mathtt{a}, l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right)$$

$$\textit{(we set}\ \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) = 0\ \textit{for}\ \mathcal{S}_a(l_2, l) = \emptyset\textit{)}$$

$$T\mathcal{B}(\tau) = \zeta(\tau),\ \textit{if}\ \zeta(\tau) \in \mathcal{C},\ \textit{else}$$

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

where

1. $\mathtt{Incr}(\{\mathtt{a}_1, \mathtt{a}_2, \dots, \mathtt{a}_n\}) = \sum_{1 \le i \le n} \mathtt{Incr}(\mathtt{a}_i)$ *with* $\mathtt{Incr}(\emptyset) = 0$
2. $\mathtt{Incr}(\mathtt{v}) = \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} T\mathcal{B}(trn(\pi)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ *(we set* $\mathtt{Incr}(\mathtt{v}) = 0$ *for* $\mathtt{C}^+(\mathtt{v}) = \emptyset$*)*
3. $T\mathcal{B}(\{\tau_1, \tau_2, \dots, \tau_n\}) = \min_{1 \le i \le n} T\mathcal{B}(\tau_i)$

We introduced and discussed the terms from which Definition 42 is composed step-by-step in this chapter.

Soundness. Soundness of our full bound algorithm (Definition 42) results from combining the soundness proof for the reasoning on reset chains (Section 7.2) and the soundness proof for the path-sensitive reasoning (Section 7.3).

3.10 Parametrization by a Cost Model

Recall our discussion on complexity and resource bounds in Section 1.1. So far, our examples have always assumed the *back-edge* metric as *cost model*. We now discuss an instrumentation of our analysis by other *uniform* cost models:

Let $\mathcal{P}(L, T, l_b, l_e)$ be a program. Let $\mathcal{CM}$ denote a *uniform cost model*. I.e., $\mathcal{CM}$ assigns a *cost* in $\mathbb{Z}$ to each instruction or operation in $\mathcal{P}$. We obtain a *cost* for a transition $\tau \in T$ of $\mathcal{P}$, by summing up the cost of all instructions and operations on τ. We thereby obtain a function $\mathcal{CM} : T \to \mathbb{Z}$.

Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ denote the *DCP* abstraction of $\mathcal{P}$ as obtained by our algorithm from Section 2.2.

In order to infer bounds with respect to the cost model $\mathcal{CM}$, we apply the following modification to $\Delta\mathcal{P}$: We choose a fresh variable $\mathtt{v}_{cost} \in \mathcal{V}$ ($\mathtt{v}_{cost}$ does not yet appear in $\Delta\mathcal{P}$). We add the difference constraint $\mathtt{v}'_{cost} \leq 0$ to all *outgoing* transitions of l_b, i.e., to all *initial* transitions. For all other transitions $\tau \in E$ we now add the constraint $\mathtt{v}'_{cost} \leq \mathtt{v}_{cost} + \mathcal{CM}(\tau)$, thereby increasing $\mathtt{v}_{cost}$ up to the cost of executing τ.

We then compute $V\mathcal{B}(\mathtt{v}_{cost})$ in order to infer a bound on the overall cost of executing $\mathcal{P}$ with respect to the cost model $\mathcal{CM}$.

3.10.1 Computing Bounds on Memory Consumption

Note that the variable $\mathtt{v}_{cost}$ maybe *decremented* in presence of *negative* costs. For example, the *deallocation* of a resource such as *memory* is usually modeled by negative costs.

For reasoning about *decrements*, we have introduced the path-sensitive reasoning in Section 3.8. Our path-sensitive variable bound algorithm, however, computes bounds on variable values at a given control location $l \in L$. But the above described instrumentation of our algorithm by a *cost model* requires to infer a *global* bound on the value of $\mathtt{v}_{cost}$.

We now show how our path-sensitive variable bound algorithm can be generalized for inferring *global variable bounds*.

Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. Let $\mathtt{v} \in \mathcal{V}$. We obtain a *variable bound* for $\mathtt{v}$, i.e., an upper bound invariant for $\mathtt{v}$ which is valid at all locations $l \in L$ s.t. $\mathtt{v} \in \mathtt{def}(l)$, by computing $\max_{l \in L \text{ s.t. } \mathtt{v} \in \mathtt{def}(l)} V\mathcal{B}(\mathtt{v}, l)$. I.e., we have:

$$V\mathcal{B}(\mathtt{v}) = \max_{l \in L \text{ s.t. } \mathtt{v} \in \mathtt{def}(l)} V\mathcal{B}(\mathtt{v}, l)$$

As an example consider variable p in Example `s_SFD_process` (Figure 3.14 (a)). Assume we want to compute a *global* upper bound on p. As usual, our computation is performed on the *DCP* obtained by abstraction (Figure 3.14 (a)). Since $\mathtt{C}^+([p]) = \emptyset$ we have that $\mathtt{Incr}([p]) = 0$. With $\mathcal{R}([p]) = \{(\tau_0, [n], 0)\}$ and $[n] \in \mathcal{C}$ we compute:

$$V\mathcal{B}([p]) \to \max_{l \in L \text{ s.t. } [p] \in \mathtt{def}(l)} V\mathcal{B}([p], l)$$

$$
\begin{aligned}
&\to \max_{l\in L \text{ s.t. } [p]\in \mathtt{def}(l)} \mathtt{Incr}([p]) + [n] + 0 + \max_{\pi\in\mathcal{S}_a(l_1,l)} \mathtt{Sum}\mathcal{ID}(\pi)([p]) \\
&= \mathtt{Incr}([p]) + [n] + 0 + \max_{l\in L \text{ s.t. } [p]\in \mathtt{def}(l)} \max_{\pi\in\mathcal{S}_a(l_1,l)} \mathtt{Sum}\mathcal{ID}(\pi)([p]) \\
&\to 0 + [n] + 0 + \max_{l\in\{l_2,l_3,l_4,l_5,l_6\}} \max_{\pi\in\mathcal{S}_a(l_1,l)} \mathtt{Sum}\mathcal{ID}(\pi)([p]) \\
&\to 0 + [n] + 0 + 0 \\
&= [n]
\end{aligned}
$$

We have that $\max_{l\in\{l_2,l_3,l_4,l_5,l_6\}} \max_{\pi\in\mathcal{S}_a(l_1,l)} \mathtt{Sum}\mathcal{ID}(\pi)([p]) = 0$ because $\mathtt{Sum}\mathcal{ID}(\pi)([p]) \leq 0$ for all paths π with start location l_1. We therefore obtain the precise bound n (n has type *unsigned*) as an upper bound on p. In contrast, our path-insensitive algorithm infers the bound $2n$ since it ignores the decrement of p.

3.11 Comparison to Invariant Analysis

In this section we compare our bound algorithm to classical invariant analysis by abstract interpretation.

3.11.1 Variable Bounds and Classical Invariant Analysis

Reconsider our discussion of Example `twoSCCs` in Section 3.4. Table 3.3 (page 50) shows how our algorithm computes $V\mathcal{B}([x]) = [n] \times 2 + \max([m1], [m2])$, thereby obtaining the *invariant* $x \leq \max(m1, m2) + 2n$ (n, *m1* and *m2* have type *unsigned*).

Note that this invariant cannot be computed by standard abstract domains such as *octagon* or *polyhedra* [Min06]: these domains are *convex* and cannot express non-convex relations such as *maximum*. The most precise approximation of x in the polyhedra domain is $x \leq m1 + m2 + 2n$. Unfortunately, it is well-known that the polyhedra abstract domain does not scale to larger programs and needs to rely on heuristics for termination.

We further point out how our algorithm differs *methodologically* from classical invariant analysis: Standard *abstract domains* such as *octagon* [Min06] or *polyhedra* propagate information *forward* until a fixed point is reached, *greedily* computing all possible invariants expressible in the abstract domain at every location of the program. In contrast, $V\mathcal{B}(x)$ infers the invariant $x \leq \max(m1, m2) + 2n$ by *modular reasoning*: *local information* about the program (increments/resets of variables and local bounds) is combined to a *global* program property.

Moreover, our variable and transition bound analysis is *demand-driven*: our algorithm performs only those recursive calls that are indeed needed to derive the desired bound. We believe that our analysis complements existing techniques for invariant analysis and will find applications outside of bound analysis.

3.11.2 Transition Bounds and Classical Invariant Analysis

We now contrast our approach for computing a loop bound for the loop at l_3 in Example `xnuSimple` (Figure 3.8, page 55) with classical invariant analysis: Assume that we have added a counter c, which counts the number of inner loop iterations (i.e., c is initialized to 0 and incremented in the inner loop). For inferring $c <= n$ through invariant analysis the invariant $c + x + r <= n$ is needed for the outer loop, and the invariant $c + x + p <= n$ for the inner loop. Both relate 3 variables and cannot be expressed as (parametrized) octagons (e.g., [SGS14]). Further, the expressions $c + x + r$ and $c + x + p$ do not appear in the program, which is challenging for template based approaches to invariant analysis.

3.12 Relation to Amortized Complexity Analysis

In the following we discuss how our approach relates to *amortized complexity analysis* as introduced by Tarjan in his influential paper [Tar85]. We recall Tarjan's idea of using *potential functions* for amortized analysis in Section 3.12.1. In Section 3.12.2 we explain how our approach can be viewed as an instantiation of amortized analysis via potential functions.

3.12.1 Amortized Analysis using Potential Functions.

We refer the reader to Section 1.4.1 for an example on *amortized complexity analysis*.

Potential Function. As a means to reason about the amortized cost of an operation or a sequence of operations, Tarjan introduces the notion of a *potential function*. A potential function is a function $\Phi : \Sigma \to \mathbb{Z}$ from the program states to the integers. Let $\mathcal{C}_{\mathtt{op}}(\sigma)$ denote the cost of executing operation $\mathtt{op}$ at program state $\sigma \in \Sigma$. Let Φ be a potential function. Tarjan defines the *amortized cost* $\mathcal{C}^{\mathcal{A}}_{\mathtt{op}}(\sigma)$ as $\mathcal{C}^{\mathcal{A}}_{\mathtt{op}}(\sigma) = \mathcal{C}_{\mathtt{op}}(\sigma) + \Phi(\sigma') - \Phi(\sigma)$ where σ denotes the program state before and σ' denotes the program state after executing $\mathtt{op}$. I.e., the amortized cost is the cost plus the decrease resp. increase in the value of the potential. Consider a sequence of n operations, let $\mathtt{op}_i$ denote the ith operation in the sequence. Let σ_i denote the program state before executing operation $\mathtt{op}_i$, σ_{i+1} is the program state after executing $\mathtt{op}_i$. In general, the total cost of executing all n operations is:

$$\sum_{i=1}^{n} \mathcal{C}_{\mathtt{op}_i}(\sigma_i) = \sum_{i=1}^{n} \mathcal{C}^{\mathcal{A}}_{\mathtt{op}_i}(\sigma_i) - \Phi(\sigma_{i+1}) + \Phi(\sigma_i) = \Phi(\sigma_1) - \Phi(\sigma_{n+1}) + \sum_{i=1}^{n} \mathcal{C}^{\mathcal{A}}_{\mathtt{op}_i}(\sigma_i) \quad (3.1)$$

"*That is, the total time of the operations equals the sum of their amortized times plus the net decrease in potential from the initial to the final configuration. [...] In most cases of interest, the initial potential is zero and the potential is always non-negative. In such a situation the total amortized time is an upper bound on the total time.*" [Tar85]. I.e., if

$\Phi_i \geq 0$ and $\Phi_0 = 0$ then

$$\sum_{i=1}^{n} \mathcal{C}_{\mathtt{op}_i}(\sigma_i) \leq \sum_{i=1}^{n} \mathcal{C}^{\mathcal{A}}_{\mathtt{op}_i}(\sigma_i) \tag{3.2}$$

Reconsider Tarjan's example of a sequence of n executions of operation `StackOp` (Section 1.4.1). Let j denote the stack size, i.e., $\sigma_i(j)$ is the size of the stack in program state σ_i. The cost of executing `StackOp` in program state σ_i is $\mathcal{C}_{\mathtt{StackOp}}(\sigma_i) = 1+(\sigma_i(j)+1-\sigma_{i+1}(j))$ (where $\sigma_i(j) + 1 - \sigma_{i+1}(j)$ is the cost of the pop operation). Tarjan proposes to use the stack size j as a potential function, i.e. we choose $\Phi(\sigma_i) = \sigma_i(j)$. We have

$$\begin{aligned}
\mathcal{C}^{\mathcal{A}}_{\mathtt{StackOp}}(\sigma_i) &= \mathcal{C}_{\mathtt{StackOp}}(\sigma_i) + \Phi(\sigma_{i+1}) - \Phi(\sigma_i) \\
&= \mathcal{C}_{\mathtt{StackOp}}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) \\
&= 1 + (\sigma_i(j) + 1 - \sigma_{i+1}(j) + \sigma_{i+1}(j) - \sigma_i(j) \\
&= 2
\end{aligned}$$

With (3.2) we get:

$$\sum_{i=1}^{n} \mathcal{C}_{\mathtt{StackOp}_i}(\sigma_i) \leq \sum_{i=1}^{n} \mathcal{C}^{\mathcal{A}}_{\mathtt{StackOp}_i}(\sigma_i) = \sum_{i=1}^{n} 2 = 2n$$

3.12.2 Amortized Analysis in our Algorithm.

Consider the *labeled transition system* of Example `tarjan` shown in Figure 3.1 (b). We have that transition τ_1 models the *push* instruction, increasing the stack size j by 1, a sequence of transitions τ_2 models the *pop* instruction, decreasing the stack by an arbitrary number of elements. A *complete* run ρ of Example `tarjan` can be decomposed into the initial transition τ_0 and a number of sub-runs $\rho_{[i_k,i_{k+1}]}$ with $1 \leq i_1 < i_2 \ldots$ s.t. each $\rho_{[i_k,i_{k+1}]}$ consists of a single transition τ_1 (*push*) followed by a sequence of transitions τ_2 (*pop*), followed by a single execution of transition τ_3. Each sub-run $\rho_{[i_k,i_{k+1}]}$ models Tarjan's `StackOp` operation. We thus have that the *amortized cost* of a sub-run $\rho_{[i_k,i_{k+1}]}$ is 2. Given that τ_1 cannot be executed more than n times and each $\rho_{[i_k,i_{k+1}]}$ contains exactly one τ_1, we get that the overall cost of executing Example `tarjan` is bounded by $n \times 2 = 2n$.

In the following we discuss that our transition bound algorithm $T\mathcal{B}$ can be viewed as an instantiation of amortized analysis using *potential functions*. We base our discussion on the concrete semantics of Example `tarjan` given by the LTS in Figure 3.1 (b). Note, however, that our algorithm is run on the abstracted *DCP* in Figure 3.1 (c) where the same reasoning applies: Suppose we want to compute the transition bound of transition τ_2 in order to compute the total cost of the *pop* instructions. Let $\rho = (\sigma_0, l_0) \xrightarrow{\lambda_0} (\sigma_1, l_1) \xrightarrow{\lambda_1} \ldots$ be a run of Example `tarjan`. Let $len(\rho)$ denote the *length* of ρ (i.e, total number of transitions on ρ). We define the cost of executing τ_2 in program state σ_i as $C_{\tau_2}(\sigma_i) = 1$ and the cost of executing τ_1 and τ_3 as $C_{\tau_1}(\sigma_i) = C_{\tau_3}(\sigma_i) = 0$ since we are only interested in τ_2. We have

$$\sharp(\tau_2, \rho) = \sum_{i=1}^{len(\rho)-1} C_{\rho(i)}(\sigma_i)$$

where $\rho(i)$ denotes the $i+1$th transition $l_i \xrightarrow{u_i} l_{i+1}$ on ρ. Our algorithm reduces the question "how often can τ_2 be executed" to the question "how often can the local bound 'j' of τ_2 be increased on τ_1". This reasoning uses the *local bound* j of τ_2 as a *potential function*, as we show next: We get the following *amortized* costs for executing τ_1, τ_2 and τ_3 respectively:

$$C^{\mathcal{A}}_{\tau_2}(\sigma_i) = C_{\tau_2}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 1 + \sigma_{i+1}(j) - \sigma_i(j) = 0$$
$$C^{\mathcal{A}}_{\tau_1}(\sigma_i) = C_{\tau_1}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 0 + \sigma_{i+1}(j) - \sigma_i(j) = 1$$
$$C^{\mathcal{A}}_{\tau_3}(\sigma_i) = C_{\tau_3}(\sigma_i) + \sigma_{i+1}(j) - \sigma_i(j) = 0 + \sigma_{i+1}(j) - \sigma_i(j) = 0$$

With $\sigma_i(j) \geq 0$, $\sigma_1(j) = 0$ and (3.2) we have:

$$\sharp(\tau_2, \rho) = \sum_{i=1}^{len(\rho)-1} C_{\rho(i)}(\sigma_i) \leq \sum_{i=1}^{len(\rho)-1} C^{\mathcal{A}}_{\rho(i)}(\sigma_i) = \sharp(\tau_1, \rho) \times 1$$

We point out that choosing the *local bound* j of τ_2 as potential function causes the amortized cost of executing τ_2 to be 0 and reduces the question of how often τ_2 can be executed to how often the potential j can be incremented on τ_1.
Using $\sharp(\tau_1, \rho) \leq \sharp(\tau_0, \rho) \times n = n$ one obtains the upper bound n for the total cost of the *pop* instructions.

CHAPTER 4

Extensions

In this chapter we present a number of rather technical, but practically very useful extensions of the abstraction procedure and of the bound algorithm.

4.1 Extensions of the Abstraction Procedure

In this Section we show how to keep some of the information that is lost by our basic abstraction algorithm (Section 2.2).

4.1.1 Modeling arbitrary Decrements

Consider the example in Figure 4.1 (a). The loop at l_2 can be executed $\lfloor \frac{n+1}{2} \rfloor$ times because i is incremented n times by 1 when iterating the loop at l_1, afterward i is decreased by 2 on each execution of the loop at l_2. Figure 4.1 (b) shows the *DCP* with guards as obtained by *Abstraction I* (Section 2.2.1). Figure 4.1 (c) shows the *DCP* as obtained by *Abstraction II* (Section 2.2.2). Note that in Figure 4.1 (c) the decrease of i by 2 on τ_3 is over-approximated by a decrease of $[i]$ by at least 1. Recall that the variable $[i]$ represents the program expression $\max(i, 0)$. It is unsound to add the predicate $[i]' \leq [i] - 2$ to transition τ_3 of the *DCP*: E.g., consider the case where $n = 1$. Then $\max(i, 0)$ indeed only decreases by 1 when executing τ_3 (whereas i decreases by 2).

As a result, transition τ_3 of the *DCP* in Figure 4.1 (c) can be executed n times in total. Basing our analysis on the *DCP* in Figure 4.1 (c) we thus over-approximate the bound for the loop at l_2 in Figure 4.1 (a) by n (the precise bound is $\lfloor \frac{n+1}{2} \rfloor$).

In the following we discuss how to obtain a more precise *DCP* abstraction for programs where loop counter variables are *decremented* by a constant greater than 1 such as i in Figure 4.1 (a). Observe that the *DCP* shown in Figure 4.2 is a valid abstraction of Figure 4.1 (a): In comparison to Figure 4.1 (b) the abstraction in Figure 4.2 is based

on the additional norm $[i+1]$. We have that $i > 0$ is a *guard* of τ_3 (see Figure 4.1 (b)), thus $i + 1 > 1$ holds when executing τ_3. Therefore $[i+1]$ $(= \max(i+1, 0))$ decreases by 2 when executing τ_3. Transition τ_3 of Figure 4.2 can be executed up to $\lfloor \frac{[n]+1}{2} \rfloor$ times.

Based on our observation we suggest the following algorithm for obtaining a *DCP* abstraction that preserves decrements by 2: Let $\mathcal{P}$ be a *program*, let $\Delta\mathcal{P}_G(L, E, l_b, l_e)$ be the *DCP with guards* obtained from $\mathcal{P}$ by *Abstraction I* (Section 2.2.1), let $\Delta\mathcal{P}(L, E', l_b, l_e)$ be the *DCP* obtained from $\Delta\mathcal{P}_G$ by *Abstraction II* (Section 2.2.2). We refine our *DCP* abstraction $\Delta\mathcal{P}$ as follows: Let $\tau = l_1 \xrightarrow{g,u} l_2 \in E$ be a transition of $\Delta\mathcal{P}_G$. Let $\tau' = l_1 \xrightarrow{u'} l_2 \in E'$ be the corresponding transition of $\Delta\mathcal{P}$. Let $e'_1 \leq e_1 - 2 \in u$ for some norm e_1. If $e_1 \in g$, we i) add the predicate $[e_1 + 1]' \leq [e_1 + 1] - 2$ to u', ii) for each $\tau'' = l_3 \xrightarrow{u''} l_4 \in E'$ with $\tau'' \neq \tau'$ and $[e_1]' \leq [e_1] + \mathtt{c} \in u''$ for some $\mathtt{c} \in \mathbb{Z}$ we add the predicate $[e_1 + 1]' \leq [e_1 + 1] + \mathtt{c}$ to u'', iii) for each $\tau'' = l_3 \xrightarrow{u''} l_4 \in E'$ with $\tau'' \neq \tau'$ and $[e_1]' \leq [e_2] + \mathtt{c} \in u''$ for some $\mathtt{c} \in \mathbb{Z}$ and some $e_2 \neq e_1$ we add the predicate $[e_1 + 1]' \leq [e_2] + (\mathtt{c} + 1)$ to u''.
We can handle decrements greater than 2 accordingly: E.g., if $e'_1 \leq e_1 - 3 \in u$ we add the predicate $[e_1 + 2]' \leq [e_1 + 2] - 3$ to u', etc.

Example: By the described algorithm the refined *DCP* abstraction shown in Figure 4.2 is obtained from the *DCP with guards* in Figure 4.1 (b) and the *DCP* in Figure 4.1 (c).

Soundness. Soundness of the suggested refinement follows by the following observations: Let τ be a transition of $\mathcal{P}$. If e_1 is a *guard* of τ ($e_1 > 0$ before executing τ) and $e'_1 \leq e_1 - 2$ is *invariant* on τ we have that $[e_1 + 1]' \leq [e_1 + 1] - 2$ is invariant on τ (as previously discussed on the example in Figure 4.1 above). Further: Let $\tau' \neq \tau$ be some transition of the concrete program. If $[e_1]' \leq [e_1] + \mathtt{c}$ is invariant on τ' then $[e_1 + 1]' \leq [e_1 + 1] + \mathtt{c}$ is also invariant on τ'. Let $e_1 \neq e_2$. If $[e_1]' \leq [e_2] + \mathtt{c}$ is invariant on τ' then $[e_1 + 1]' \leq [e_2] + (\mathtt{c} + 1)$ is also invariant on τ'.

Note that our refinement algorithm is a procedure that performs simple syntactic manipulations based on purely syntactic information. It can therefore be performed efficiently (linear in the number of transitions of $\Delta\mathcal{P}_G$), no re-run of the abstraction procedure is performed. The refinement uses $\Delta\mathcal{P}_G$ and $\Delta\mathcal{P}$ as obtained by Abstraction I and II.

We discuss in Section 4.2.3 (page 102) how our bound algorithm infers the correct bound for the loop at l_2 in Figure 4.1 (a) based on the abstraction in Figure 4.2.

4.1.2 Modeling Flags

Consider Figure 4.3 (a) which is an example of a loop with two paths. The path interleaving is controlled by the boolean variable b. Such loop constructs are common in imperative programs, boolean control variables such as b in Figure 4.3 (a) are usually called *flags*. Figure 4.3 (a) has loop bound $2n - 1$: the *if*-branch of the loop can be executed n times (i is initially n), and for each but the last execution of the *if*-branch there is one execution of the *else*-branch, i.e., we have $n + n - 1 = 2n - 1$ executions.

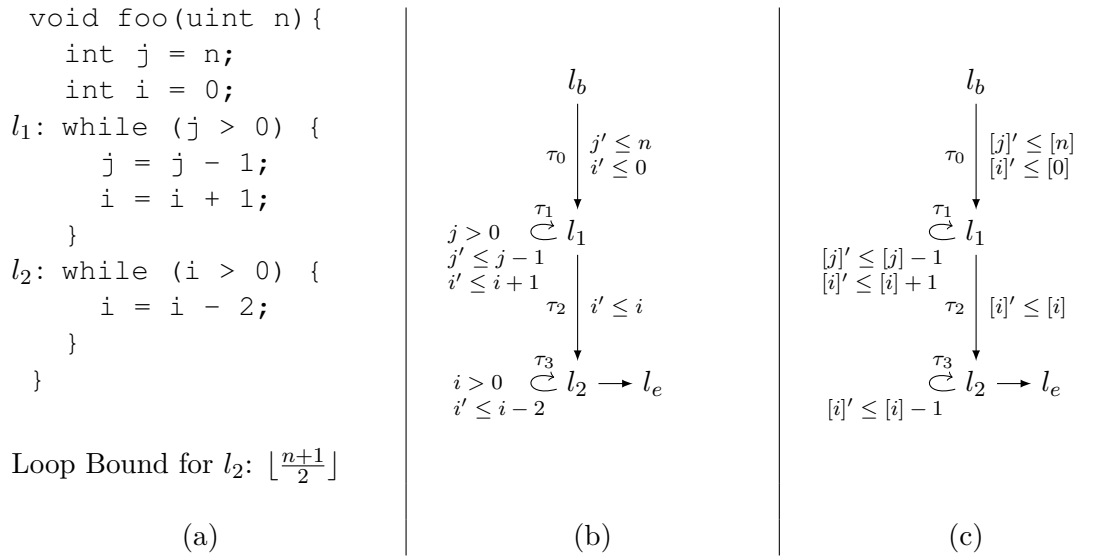


Figure 4.1: (a) Example, (b) *DCP* with guards (Abstraction I), (c) *DCP* (Abstraction II)

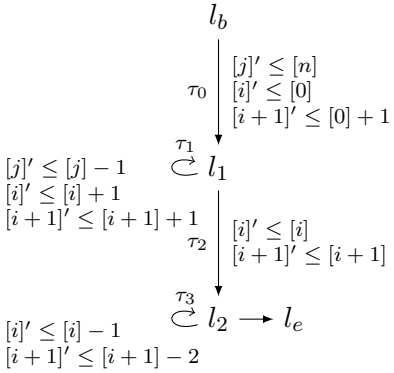


Figure 4.2: *DCP* abstraction obtained for Figure 4.1 (a) when adding the norm $[i + 1]$

We can handle the boolean variable b as an integer variable (a common treatment in program analysis). By a direct application of our abstraction procedure (Section 2.2), we obtain the abstraction shown in Figure 4.3 (b). The *DCP* in Figure 4.3 (b), however, does not terminate: Even if we interpret the symbolic constants [0] and [1] as the natural numbers 0 and 1 respectively, τ_2 can be executed infinitely often.

In our implementation, we abstract an update of form $b :=$ `false` by $b' \leq b - 1$ if we can establish that $b =$ `true` holds before the update, correspondingly for an update of form $b :=$ `true`. Example: Applying this heuristic during the abstraction of Figure 4.3 (a), we obtain the *DCP* shown in Figure 4.3 (c).

Based on the *DCP* abstraction in Figure 4.3 (c), our bound algorithm (Chapter 3) now easily obtains the tight bound $2n$ for Figure 4.3 (a). (The precise bound is $2n - 1$.)

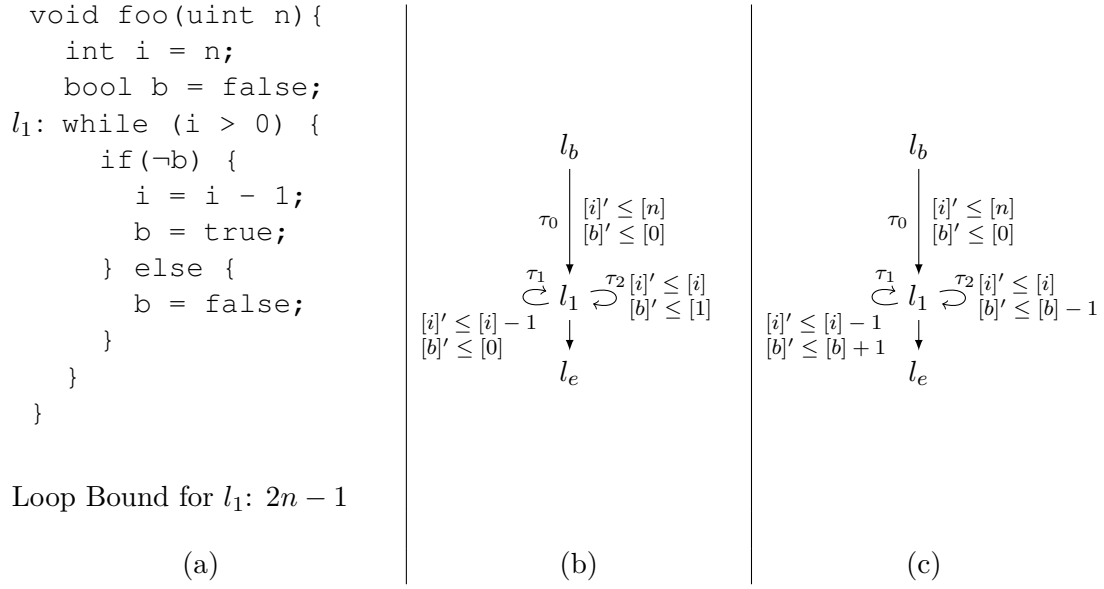


Figure 4.3: (a) Example, (b) *DCP* obtained by abstraction, (c) alternative *DCP* abstraction

4.1.3 Control-Flow Refinement

The program shown in Figure 4.4 (a) is another example of a loop which phases are controlled by a flag variable. The loop has loop bound $2n-2$: In the worst case we first count i down from initially $n-1$ to 1, then we set b to `false` and count i up from 0 to n. With our technique for abstracting *flags* (see Section 4.1.2) we obtain the *DCP* shown in Figure 4.5 (a) by our abtraction procedure. This *DCP*, however, does not terminate: We can infinitely often execute $\tau_1, \tau_2, \tau_4, \tau_1, \tau_2, \tau_4, \ldots$. The information, that altering between the execution of the `if` and the `else` branch is not possible, is lost during the abstraction.

In our implementation we use *control-flow refinement* for handling such cases. The idea of *control-flow refinement* is to encode *semantic* properties of the program into its control-flow structure, thereby *refining* the control-flow. E.g., consider the *labeled transition system* (LTS) of the program in Figure 4.4 (a) shown in Figure 4.4 (b). In Figure 4.4 (b) there is a path that takes transition τ_4 and afterwards transition τ_1. However, this path will never be executed during program run: Executing τ_4 requires that $\neg b$ holds, whereas executing τ_1 requires that b holds. Variable b, however, stays `false` if it is set to `false` once. Similarly, we cannot execute τ_4 directly after executing τ_0, we first will have to execute τ_3.
These observations are made explicit by the LTS in Figure 4.4 (c). (by λ_i we refer to the transition relation of transition τ_i in Figure 4.4 (b).) Note that Figure 4.4 (b) is semantically equivalent to Figure 4.4 (c).

Applying our abstraction algorithm to the control-flow refined LTS shown in Figure 4.4 (c)

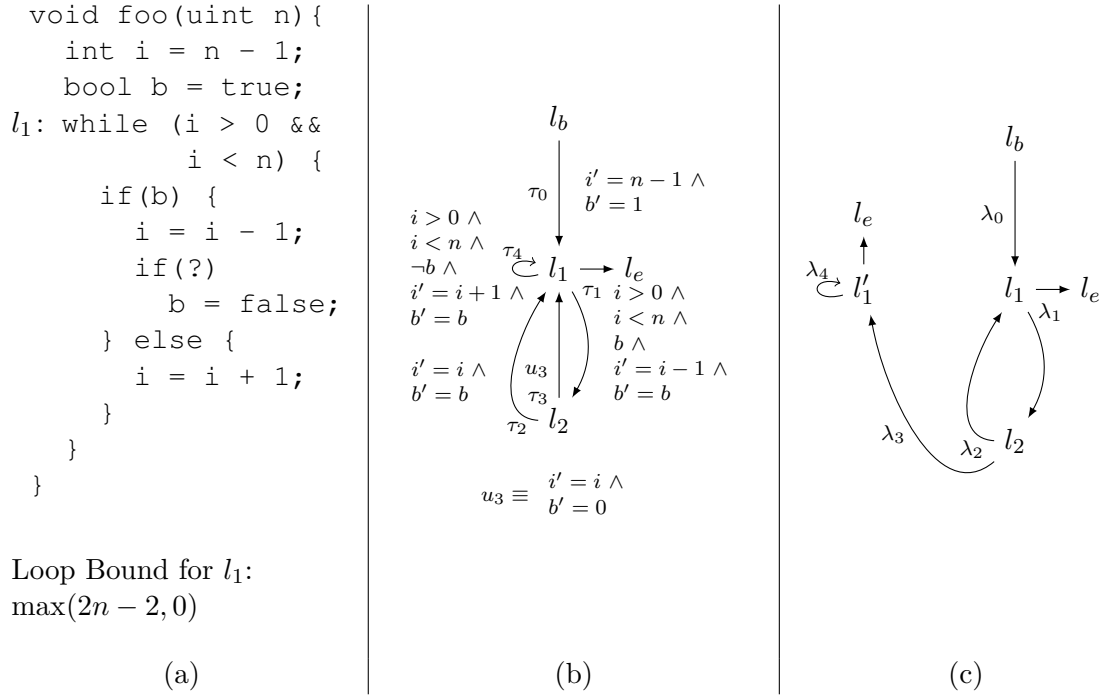


Figure 4.4: (a) Example, (b) Labeled Transition System, (c) Control-Flow Refined LTS, by λ_i we denote the transition relation of transition τ_i of the original LTS

results in the *DCP* shown in Figure 4.5 (b). In contrast to the *DCP* in Figure 4.5 (a), Figure 4.5 (b) *does* terminate: On each execution of the cycle τ_1, τ_2 variable $[i]$ decreases whereas on each execution of τ_4 variable $[n-i]$ decreases. Our bound algorithm (Chapter 3) infers the bound $2n-1$ for Figure 4.5 (b). (Note that, as a preprocessing, we rename the program variables as we discussed in Section 3.5.3.)

Control-flow refinement techniques are, e.g., discussed in [GJK09] and [ZGSV11] (for imperative programs), and in [MV06] (for functional programs). In [FH14] control-flow refinement is applied to *cost equations*.

Contextualization, Infeasible Path Elimination and Loop Unrolling

In our implementation we use the conrol-flow refinement technique *contextualization* introduced in [ZGSV11].

In our framework *contextualization* can be defined as follows: Given a program $\mathcal{P}(L, T, l_b, l_e)$ we create a new program $\mathcal{P}'(L', T', l'_b, l'_e)$ where

- $L' = \{l_\tau \mid \tau \in T\}$ and
- $T' = \{l_{\tau_1} \xrightarrow{\lambda_1} l_{\tau_2} \mid \tau_1 = l_1 \xrightarrow{\lambda_1} l_2 \in T \wedge \tau_2 = l_2 \xrightarrow{\lambda_2} l_3 \in T \text{ and } \lambda_1 \circ \lambda_2 \neq \emptyset\}$.

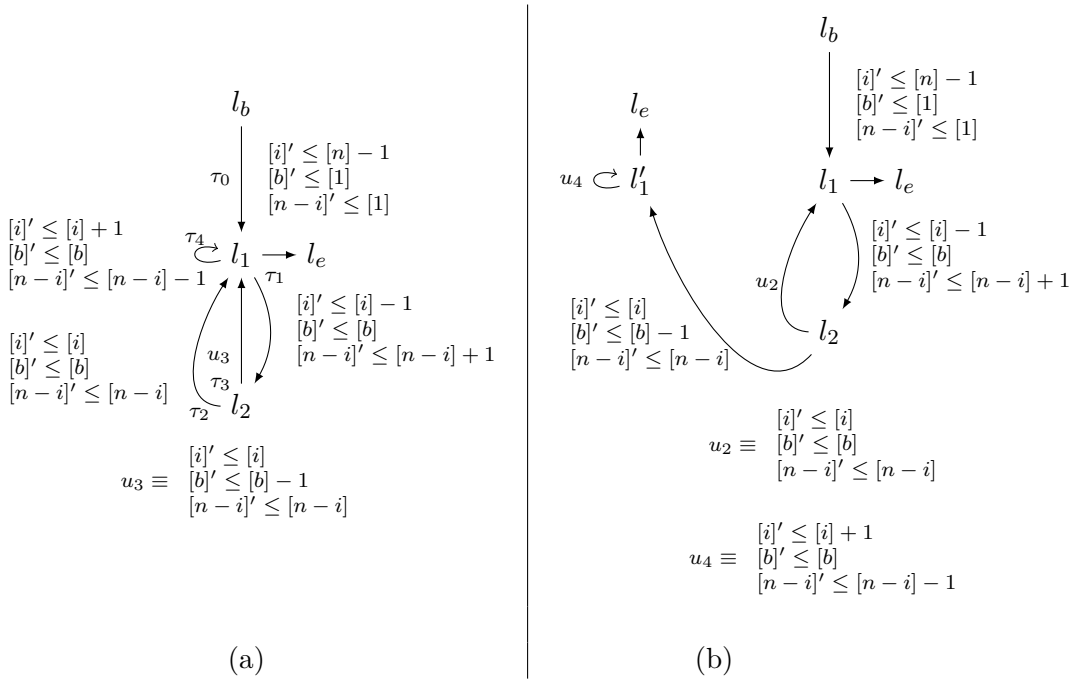


Figure 4.5: Abstractions of Figure 4.4 (a): (a) *DCP* obtained by abstraction from original LTS, (b) *DCP* obtained by abstraction from control-flow refined LTS

Here $\lambda_1 \circ \lambda_2$ denotes the transition relation that results form *concatenating* the transition relations λ_1 and λ_2. E.g., we have $\{(\sigma_a, \sigma_b), (\sigma_a, \sigma_c)\} \circ \{(\sigma_c, \sigma_d), (\sigma_e, \sigma_f)\} = \{(\sigma_a, \sigma_d)\}$. The check whether $\lambda_1 \circ \lambda_2 \neq \emptyset$ can be performed by an SMT solver (as in our implementation): Assume that λ_1 is given by formula ϕ_1 and λ_2 is given by formula ϕ_2. We ask the SMT solver whether $\phi_1 \wedge \phi_2'$ is *satisfiable*, where ϕ_2' results from ϕ_2 by adding an additional $'$ to all variables in the formula. E.g., in the LTS stated in Figure 4.6 (b) the transition relation of τ_1 is stated by the formula $i > 0 \wedge i' = i$, the transition relation of τ_4 is given by $i \leq 0 \wedge i' = i$. We thus ask the SMT solver whether $i > 0 \wedge i' = i \wedge i' \leq 0 \wedge i'' = i'$ is satisfiable. Since this is not the case, there is no transition from l_{τ_1} to l_{τ_4} in the contextualized LTS in Figure 4.6 (c).

Since, in general, first-order logic is *undecidable*, we sometimes have to overapproximate λ_1 respectively λ_2 by removing conjuncts that involve undecidable theories (e.g., non-linear arithmetic) from the respective formulas. For obivious reasons this does not affect soundness of *contextualization*, we still obtain a *semantically* equivalent program.

It remains to define the start- and end-location of the contextualized program. We set $l_b' = l_\tau$ for $\tau = l_b \xrightarrow{\lambda} l_1$ with $l_1 \in L$. If there are several such $\tau \in E$ we add a new location l_b' and edges from l_b' to all such l_τ. Accordingly we set $l_e' = l_\tau$ for $\tau = l_1 \xrightarrow{\lambda} l_e$ with $l_1 \in L$. We handle the case that there are several such $\tau \in E$ in the same way as for l_b.

Example. Consider the example in Figure 4.6 (a), in Figure 4.6 (b) the labeled transition system is shown, in Figure 4.7 (a) we show the *DCP* as obtained by our abstraction procedure. Note that this *DCP* (Figure 4.7 (a)) does not terminate: We can infinitely often execute the path τ_1, τ_4. However, in the original program (Figure 4.6 (b)), this execution is not possible: Since τ_1 is guarded by $i > 0$ and τ_4 is guarded by $i \leq 0$, i must decrease between executing τ_1 and executing τ_4. This ensures termination of the original program. We make this information available in the abstract program by applying *contextualization* before computing the abstraction: In Figure 4.6 (c) we show the LTS that results from applying *contextualization* to the original LTS (Figure 4.6 (b)). In Figure 4.7 (b) the *DCP* is shown that results from applying our abstraction procedure to the refined LTS in Figure 4.6 (c). Note that the *DCP* in Figure 4.7 (b) terminates: $[i]$ never increases but decreases on all cyclic paths of the program. By our bound algorithm (Chapter 3) we are able to infer the bound n for the outer loop (at l_1) as well as for the inner loop (at l_2).

Applying Contextualization iteratively vs. Large-Block Encoding. Note that we can apply *contextualization* again to the refined program $\mathcal{P}'$, thereby increasing the precision of our refinement. E.g., we have to apply contextualization twice to the LTS in Figure 4.4 (b) in order to exclude the execution τ_2, τ_4.
To our observation, at most one application of contextualization is sufficient in practice if *large-block encoding* is used appropriately: Large-Block encoding is a well-known technique for increasing the precision of static analysis methods. It is also highly effective in our context. In contrast to contextualization large-block encoding *reduces* the number of control locations. Control locations are removed by pairwise concatenating incoming and outgoing transitions. E.g., in our implementation we only consider the loop headers as *control locations*. Thus the transitions τ_1, τ_2 and τ_1, τ_3 of Figure 4.4 (b) are encoded into one transition respectively. Let $\tau_{1,2}$ be the transition that encodes the execution τ_1, τ_2. Now the execution $\tau_{1,2}, \tau_4$ is excluded after only one application of contextualization because the guard b of $\tau_{1,2}$ contradicts the guard $\neg b$ of τ_4 (b stays unaltered when executing $\tau_{1,2}$).
Large-block encoding may, however, blow up the number of transitions *exponentially*. To our experience, using only loop headers as control locations is a good trade off between precision and complexity of our analysis.

Infeasible Path Elimination. The approach in [SZV14a] *refines* the control-flow of a given program by removing *infeasible* paths from the control flow graph. A path is *infeasible* if concatenating the transition relations along the path results in the empty relation. E.g. in Figure 4.4 (b) the cyclic path τ_3, τ_1 is infeasible. We call this refinement technique *infeasible path elemination*. In fact, *contextualization* is a more general technique which implements *infeasible path elemination*: E.g., by applying *contextualization* to Figure 4.4 (b) we obtain an LTS similar to the one shown in Figure 4.4 (c). Here the path τ_3, τ_1 is removed, while the feasible execution τ_1, τ_3 still remains.

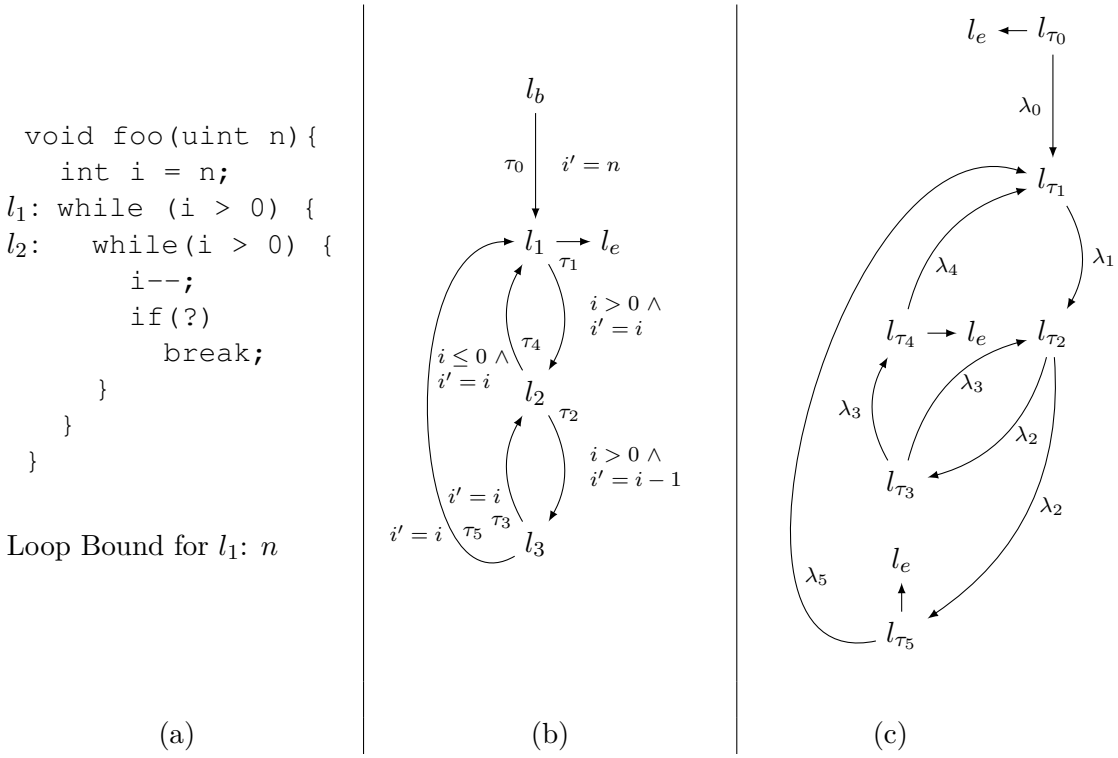


Figure 4.6: (a) Example, (b) Labeled Transition System, (c) LTS after *Contextualization*, we denote by λ_i the transition relation of transition τ_i of the original LTS

Loop Unrolling. Note that in the case of Figure 4.6 (b) *contextualization* essentially *unrolls* the inner loop once. *Loop unrolling* is a *control-flow refinement* technique which detects whether an inner loop must be executed within an execution of the outer loop. If so, the inner loop is *unrolled* once. *Contextualization* can be understood as a more general technique which implements *loop unrolling*.

Unfolding

In our implementation we only apply the presented *contextualization* technique. We can, of course, apply *any* control-flow refinement technique as a preprocessing to our abstraction procedure, thereby enriching the abstraction with additional information. We have demonstrated the effectiveness and power of *contextualization* on the discussed examples. In order to demonstrate that our approach can also benefit from other control-flow refinement techniques, we consider an adaption of the technique introduced in [GJK09]. We call this technique *unfolding*.

Consider the program in Figure 4.8 (a). Proving termination of the outer loop is challenging: b is set to `true` before reaching the inner loop, if b remains `true` the *break* statement is reached and the execution of the outer loop is stopped. If, however, b is set

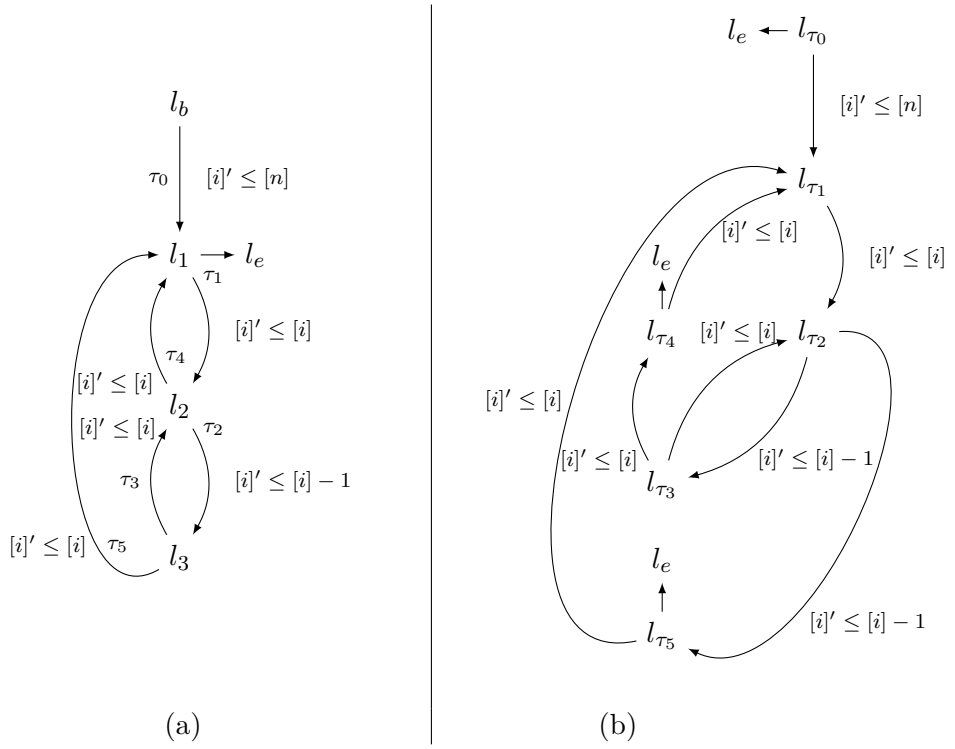


Figure 4.7: Abstractions of Figure 4.6 (a): (a) *DCP* obtained by abstraction from original LTS, (b) *DCP* obtained by abstraction from LTS after contextualization

to `false` inside of the inner loop, i is decreased by at least 1. This ensures termination of the outer loop. We conclude, that proving termination of the example requires to analyze the interplay between the executions of the inner and the outer loop. In the worst-case the outer loop can be executed n times: On each execution of the outer loop we enter the if-branch of the inner loop exactly once, thereby setting b to `false` and decreasing i by 1.

Figure 4.8 (b) shows the LTS of the example. In order to reduce the number of control locations, thereby simplifying the subsequent discussion, we use *large-block encoding* here (e.g., we have only one transition per loop-path of the inner loop). As discussed, large-block encoding is also used in our implementation.

In Figure 4.8 (c) we depict the *DCP* as obtained when applying our abstraction procedure directly to the LTS in Figure 4.8 (b). This *DCP* does in fact not terminate: We can execute the loop-path τ_1, τ_4 *infinitely* often. The reason is, that the previously discussed relation between the executions of the inner and the outer loop is lost during the abstraction process.

For the case of Figure 4.8 (b) *contextualization* does not solve the issue: By contextualization we introduce a case distinction on which transition is executed *next* by creating corresponding new control locations. This case distinction does not allow to prove

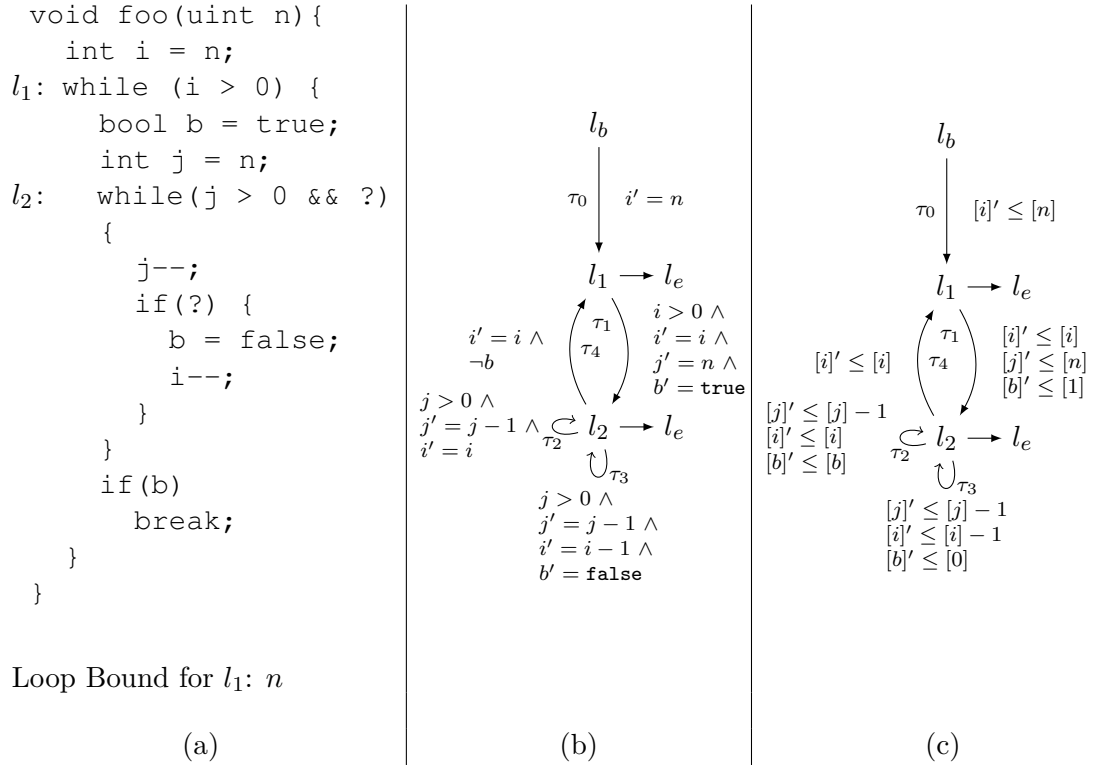


Figure 4.8: (a) Example, (b) Labeled Transition System, (c) *DCP* obtained by abstraction

termination of the example. We rather need a different kind of case distinction here: We need to distinguish between the case where b is set to `false` inside of the inner loop and the case where b stays unaltered during the iteration of the inner loop. I.e., we need to distinguish whether τ_3 (which sets b to `false`) is executed *at least* once within one iteration of the outer loop or not. This case distinction is implemented by the control-flow refinement technique that was introduced in [GJK09] and which we call *unfolding*. We adapt *unfolding* to our framework by stating an appropriate formalization:

Given a program $\mathcal{P}(L, T, l_b, l_e)$ we create a new program $\mathcal{P}'(L', T', l'_b, l'_e)$ where

- $L' = \{l^S \mid l \in L \text{ and } S \in 2^T\}$ and
- $T' = \{l_1^{S_1} \xrightarrow{\lambda_1} l_2^{S_1 \cup \tau_1} \mid \tau_1 = l_1 \xrightarrow{\lambda_1} l_2 \in E\}$.

We set $l'_b = l_b^{\emptyset}$. We set $l'_e = l_e^{\emptyset}$. We add edges $l_e^S \to l_e^{\emptyset}$ for all $S \in 2^T$ with $S \neq \emptyset$ to T'.

We can apply *unfolding* also to only part of a program. E.g., for the LTS in Figure 4.8 (b) it is sufficient to apply *unfolding* only to the inner loop. The resulting LTS is shown in Figure 4.9 (a).

After *unfolding* we apply *dead transition elimination* which we discuss next.

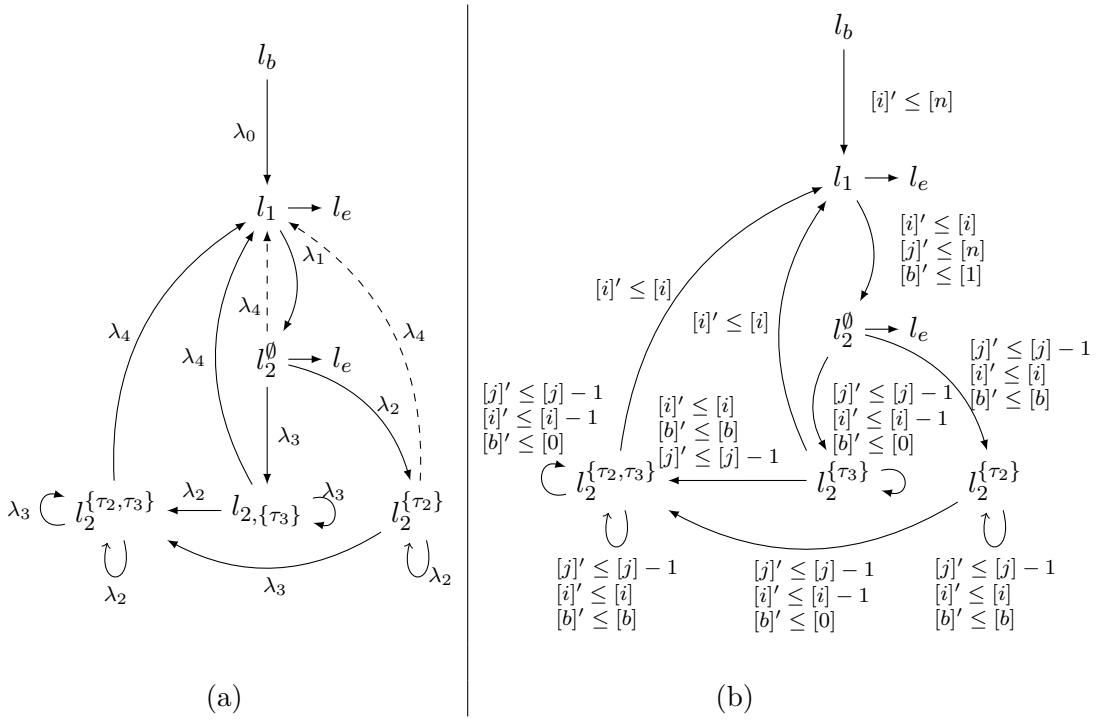


Figure 4.9: (a) LTS after *unfolding* of inner loop (by λ_i we refer to the transition relation of transition τ_i of the original LTS in Figure 4.8 (b)), dashed transitions are removed by *dead transition elimination*, (b) *DCP* obtained by abstraction

Dead Transition Elimination

Definition 43 (Reachable States)**.** *Let* $\mathcal{P}(L, T, l_e, l_b)$ *be a* program *over states* Σ*. By* $\texttt{Reach} : L \to 2^{\Sigma}$ *be denote the set of states that is* reachable *at* l*. Let* $l \in L$*, we define* $\texttt{Reach}(l) =$
$\{\sigma \in \Sigma \mid \exists$ *run* $\rho = (l_b, \sigma_0) \xrightarrow{\lambda_0} (l_1, \sigma_1) \xrightarrow{\lambda_1} \ldots$ *of* $\mathcal{P}$ *with* $l_i = l$ *and* $\sigma_i = \sigma$ *for some* $i\}$*.*

Dead transition elimination removes those transitions from a given LTS which can never be taken during program run. A transition $l_1 \xrightarrow{\lambda} l_2 \in T$ is *dead* iff $\{(s_1, s_2) \in \lambda \mid s_1 \in \texttt{Reach}(l_1)\} = \emptyset$, where $\texttt{Reach}(l_1)$ denotes the states that are *reachable* at l_1 (Definition 43). Obviously a transition $l_1 \xrightarrow{\lambda} l_2 \in T$ can never be executed iff no state $s \in \Sigma$ with $(s, \_) \in \lambda$ is reachable at its source location l_1. Note that it is sound to approximate "deadness" using an over-approximation of $\texttt{Reach}(l)$. Over-approximations of the reachable states can be computed by *invariant analysis*.

Example. A simple invariant analysis suffices for inferring that b is invariant at $l_2^{\{\tau_2\}}$ in Figure 4.9 (a). Since transition τ_4 is guarded by $\neg b$, over-approximating $\texttt{Reach}(l_2^{\{\tau_2\}})$

by the set of all states in which b is true, is sufficient for concluding that the transition $l_2^{\{\tau_2\}} \xrightarrow{\tau_4} l_1$ is *dead*. We thus eliminate $l_2^{\{\tau_2\}} \xrightarrow{\tau_4} l_1$ from the LTS. In the same way the transition $l_2^{\emptyset} \xrightarrow{\tau_4} l_1$ is eliminated.

We obtain the *DCP* depicted in Figure 4.9 (b) by applying our abstraction procedure to the refined LTS after dead transition elimination. Note that this *DCP terminates*: $[i]$ decreases on all cyclic paths of the outer loop. We now obtain the correct bound n for the outer loop of Figure 4.9 (b) by our bound algorithm (Chapter 3).

4.1.4 Symbolic Increments

Consider the example in Figure 4.10 (a). The update $i := i + c$ cannot be directly approximated by a *difference constraint* (Definition 3) because $c \notin \mathbb{Z}$. In Figure 4.10 (b) we show the *DCP* that we obtain from Figure 4.10 (a) by our abstraction algorithm (Section 2.2).

Note that Figure 4.10 (b) does not terminate.

Nevertheless examples such as the one in Figure 4.10 (a) are, in principle, within reach of our bound algorithm (Chapter 3). We now introduce a straightforward extension of our basic abstract program model which allows to naturally over-approximate updates of form $i := i + \texttt{expr}_c$, where $\texttt{expr}_c$ is a (symbolic) expression over the programs parameters and constants (i.e., a *constant* expression).

Definition 44 (Difference Constraints with Symbolic Increments)**.** *We denote an inequality of form $x' \leq y + c$ with $x \in \mathcal{V}$ and $y \in \mathcal{A}$ a* difference constraint with symbolic increment *(over $\mathcal{A}$) if $c \in \mathcal{C}$. Recall that $\mathcal{C}$ denotes the set of* symbolic constants*. Note that $x' \leq y + c$ is a* difference constraint *iff $c \in \mathbb{Z}$.*

We extend our abstract program model of *difference constraint programs* (Definition 4 and Definition 5) to *difference constraint programs with symbolic increments* by extending the domain of the edge labeling accordingly. For an example see Figure 4.10 (c).

We now discuss how we can exploit the expressive power of *DCP*s with symbolic increments by extending our abstraction algorithm appropriately: In *Abstraction I* (Section 2.2.1), step "*2. Abstracting Transitions*" we allow the constant $\texttt{c}$ to be any expression formed over $\mathbb{Z}$, parameters and constants of the program. In *Abstraction II* (Section 2.2.2) we generate the constraint $y' \leq x + [\texttt{c}]$ from a constraint $y' \leq x + \texttt{c}$ if $\texttt{c} \notin \mathbb{Z}$. This is obviously sound because $\texttt{c} \leq [\texttt{c}]$.

We discuss in Section 4.2.4 how our bound algorithm infers the correct bound $n \times \texttt{c}$ for the loop at l_2 in Figure 4.10 (a) based on the abstraction in Figure 4.10 (c).

We further discuss in Section 4.2.4 how we express updates of form $x := x + \texttt{expr}$, where $\texttt{expr}$ is an expression over the programs parameters *and variables* (e.g., the update $x := x + i$ in Figure 4.14), in our extended program model of *DCPs with symbolic increments* by *interleaving* bound analysis and program abstraction.

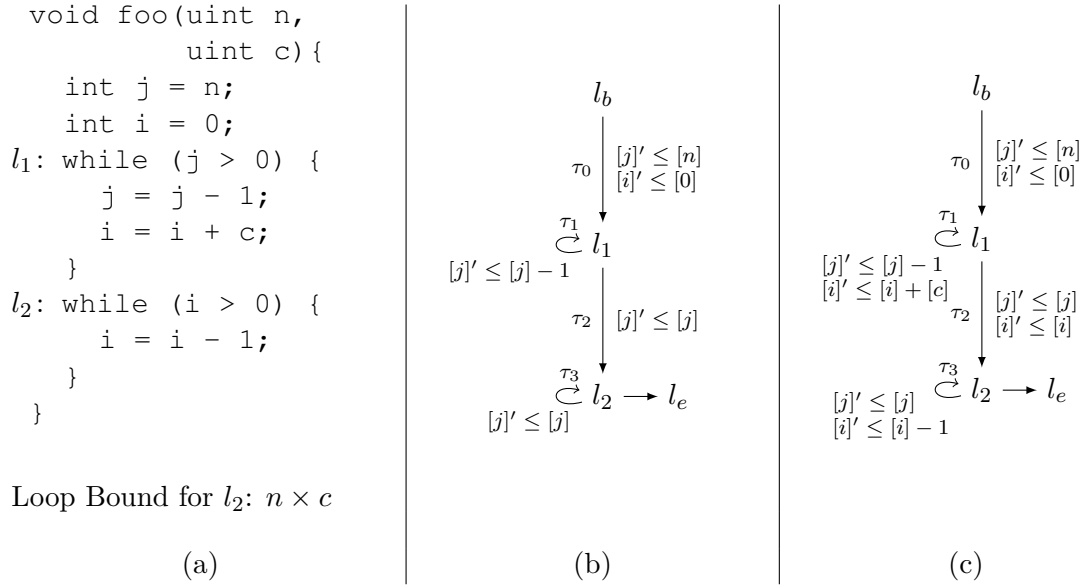


Figure 4.10: (a) Example, (b) *DCP* as obtained by our standard abstraction algorithm, (c) *DCP* with symbolic increments as obtained by our extended abstraction algorithm

We note that iteration patterns such as the one in Figure 4.10 (a) or Figure 4.14 seem to be rare in real-world C code. During our experiments we found only a few such iterations.

4.2 Extensions of the Bound Algorithm

In the following we extend our bound algorithm (presented in Chapter 3) by various features: In Section 4.2.1 we present an extension that handles special cases such as *non-linear control flow* (break statements). In Section 4.2.2 and Section 4.2.3 we introduce extensions by which we obtain *lower constant factors* in the computed bound expressions. In Section 4.2.4 we extend our bound algorithm to handle *symbolic increments*.

4.2.1 Generalizing Local Bounds to Sets of Local Bounds

Consider the example in Figure 4.11. In Figure 4.11 (b) the program is shown in form of a labeled transition system. We have that x is a local bound for τ_1 and y is a local bound for τ_2. However, it is not straightforward to find a local bound for τ_3: In order to form a local bound for τ_3 we need to combine x and y to a linear combination, e.g., $2x + y$. It is unclear how to automatically come up with such expressions.

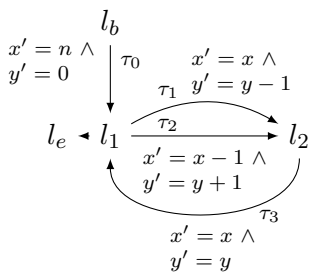
In the following we discuss a simple generalization of our algorithm by which we avoid an explicit composition of local bounds.

We generalize the local bound mapping $\zeta : E \to \mathit{Expr}(\mathcal{A})$ (Definition 18) to a *local bound set mapping* $\zeta : E \to 2^{\mathit{Expr}(\mathcal{A})}$.

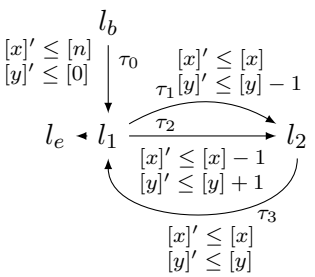
```
void foo(uint n) {
  int x = n; int y = 0;
  while(x > 0) {
    if(y > 0 && ?)
      y = y - 1;
    else {
      x = x - 1;
      y = y + 1;
    } } }
```

Complexity: $2n$

(a)



(b)



(c)

Figure 4.11: (a) Example, (b) Labeled Transition System, (c) *DCP* obtained by abstraction

Definition 45 (Local Bound Set Mapping)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\rho = (l_b, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ be a run of $\Delta\mathcal{P}$. We call a function $\zeta : E \to 2^{Expr(\mathcal{A})}$ a* local bound set mapping *for ρ if for all $\tau \in E$ it holds that*
$\sharp(\tau, \rho) \leq (\sum_{\mathtt{v} \in \zeta(\tau) \cap \mathcal{V}} \downarrow(\mathtt{v}, \rho)) + \sum_{\mathtt{expr} \in \zeta(\tau) \setminus \mathcal{V}} [\![\mathtt{expr}]\!](\sigma_0)$.
We say that ζ is a local bound set mapping *for $\Delta\mathcal{P}$ if ζ is a local bound set mapping for all runs of $\Delta\mathcal{P}$.*

Example: For Figure 4.11 (c) we have that $\zeta : E \to 2^{Expr(\mathcal{A})}$ with $\zeta(\tau_0) = \{1\}$, $\zeta(\tau_1) = \{[y]\}$, $\zeta(\tau_2) = \{[x]\}$ and $\zeta(\tau_3) = \{[x], [y]\}$ is a *local bound set mapping.*

We generalize the transition bound algorithm $T\mathcal{B}$ to local bound set mappings by summing up over all $\mathtt{expr} \in \zeta(\tau)$. We exemplify the generalization by extending Definition 27. Note that, in particular, our full bound algorithm (Definition 42) can be extended the same way. We discuss the specifics of dealing with sets of *path-sensitive* local bounds (path-sensitive local bounds were introduced in Section 3.8.1) below.

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to T\mathcal{B}([x]) + T\mathcal{B}([y])$ $\to [n] + T\mathcal{B}([y])$ $\to [n] + [n]$ $= 2 \times [n]$	$\zeta(\tau_3) = \{[x], [y]\}$, $T\mathcal{B}([x])$, $T\mathcal{B}([y])$
$T\mathcal{B}([x])$	$\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 1 \times \max([n] + 0, 0)$ $= [n]$	$\mathcal{R}([x]) = \{(\tau_0, [n], 0)\}$, $\mathcal{I}([x]) = \emptyset$, $[n] \in \mathcal{C}$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}([y])$	$\to T\mathcal{B}(\tau_1) \times 1 +$ $\quad T\mathcal{B}(\tau_0) \times \max([0] + 0, 0)$ $= T\mathcal{B}(\tau_1) \times 1 + 0$ $\to [n] \times 1 + 0$ $= [n]$	$\mathcal{I}([y]) = \{(\tau_1, 1)\}$, $\mathcal{R}([y]) = \{(\tau_0, [0], 0)\}$, $[0] \in \mathcal{C}$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_1)$	$\to T\mathcal{B}([x])$ $= [n]$	$\zeta(\tau_1) = \{[x]\}$, $T\mathcal{B}([x])$
$T\mathcal{B}(\tau_0)$	$\to T\mathcal{B}(1)$ $\to 1$	$\zeta(\tau_0) = \{1\}$

Table 4.1: Computation of $T\mathcal{B}(\tau_3)$ for Figure 4.11 (c) by Definition 46

Definition 46 (Bound Algorithm based on Local Bound Sets)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to 2^{Expr(\mathcal{A})}$. Let $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ be defined as in Definition 27. We define $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

$$T\mathcal{B}(\tau) = \sum_{\mathtt{lb} \in \zeta(\tau)} T\mathcal{B}(\mathtt{lb})$$

$$T\mathcal{B}(\mathtt{lb}) = \mathtt{lb}, \text{ if } \mathtt{lb} \notin \mathcal{V}, \text{ else}$$

$$T\mathcal{B}(\mathtt{lb}) = \mathtt{Incr}(\mathtt{lb}) + \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{lb})} T\mathcal{B}(t) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$$

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

Example: For Figure 4.11 we get $T\mathcal{B}(\tau_3) = 2n$, details are shown Table 4.1 ($[n] = n$ because n has type *unsigned*).

Inferring a Local Bound Set Mapping. Our algorithm for *finding local bounds* can be easily extended for finding *local bound sets*: Steps (1) and (2) remain unchanged. Step (3) is generalized as follows: Let $\mathtt{v}_1, \ldots, \mathtt{v}_k \in \mathcal{V}$ and $\tau = l_1 \xrightarrow{u} l_2 \in E$. We set $\zeta(\tau) = \{\mathtt{v}_1, \ldots, \mathtt{v}_k\}$ if it holds that for each execution of τ a transition in $\xi(\mathtt{v}_1) \cup \cdots \cup \xi(\mathtt{v}_k)$

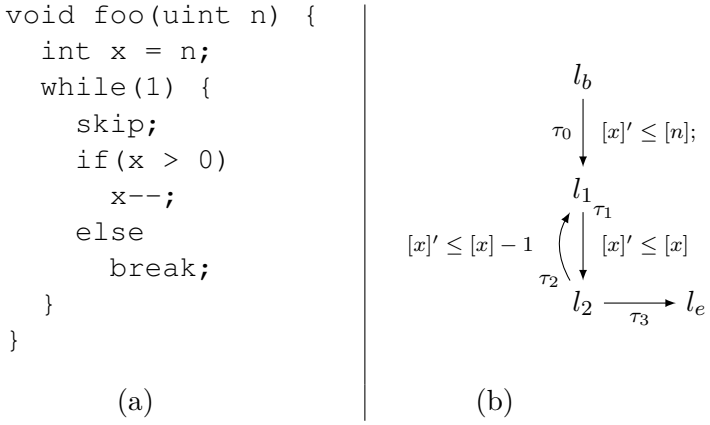


Figure 4.12: (a) Example with a "break"-statement, (b) *DCP* obtained by abstraction

is executed. This can be implemented by checking, if τ does not belong to a *strongly connected component* (SCC) `SCC` of the direct graph (L, E') where
$E' = E \cup \{l_e \xrightarrow{\emptyset} l_b\} \setminus (\xi(\mathtt{v}_1) \cup \cdots \cup \xi(\mathtt{v}_k))$.
Note that step (3) is parametrized in the number $k \in \mathbb{N}$ of variables considered. For obvious reasons it is preferable to find local bound sets of *minimal size*. Given a transition τ, we therefore first try to find a local bound set of size $k = 1$ for τ and increment k only if the search fails. With a fixed limit for k the complexity of our procedure for finding local bounds remains polynomial. To our experience limiting k to 3 is sufficient in practice.

Handling *break* statements Consider Figure 4.12 (a). The loop (resp. its back-edge) can be executed n times, the *skip* instruction (a placeholder for some code of interest), however, can be executed $n + 1$ times. Consider the abstraction shown in Figure 4.12 (b). Our algorithm for finding *local bounds*, as we discussed it so far, fails to find a *local bound (set)* for τ_1 (modeling the *skip* instruction). We extend the algorithm as follows: We set $\xi(1) = \{\tau \in E \mid \tau \text{ is not part of any SCC}\}$. I.e., for Figure 4.12 (b) we set $\xi(1) = \{\tau_0, \tau_3\}$. We add $1 \in \mathit{Expr}(\mathcal{A})$ to the set of "variables" $\mathtt{v}_1, \ldots, \mathtt{v}_k$. I.e., for our example we have $\mathtt{v}_1 = [x]$ and $\mathtt{v}_2 = 1$. The algorithm now computes $\zeta(\tau_3) = \{[x], 1\}$ for $k = 2$ given that $\xi([x]) = \{\tau_2\}$. Based on $\zeta(\tau_3) = \{[x], 1\}$ our algorithm from Definition 46 correctly infers $T\mathcal{B}(\tau_3) = [n] + 1 = n + 1$ (n has type *unsigned*).

Sets of Path-Sensitive Local Bounds

We shortly discuss how our path-sensitive bound algorithms is extended by *sets of local bounds*.

Applying the previously discussed modifications to Definition 36 results in Definition 47.

Definition 47 (Bound Algorithm for VASS based on Local Bound Sets (path-sensitive))**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to 2^{Expr(\mathcal{A})}$.*
We define $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:

$$T\mathcal{B}(\tau) = \sum_{\mathtt{lb} \in \zeta(\tau)} T\mathcal{B}(\mathtt{lb})$$
$$T\mathcal{B}(\mathtt{lb}) = \mathtt{lb}, \textit{ if } \mathtt{lb} \notin \mathcal{V}, \textit{ else}$$
$$T\mathcal{B}(\mathtt{lb}) = \mathtt{Incr}(\mathtt{lb}) + \sum_{(\_,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{lb})} \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$$

where

1. $\mathtt{Incr}(\mathtt{v}) = \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} T\mathcal{B}(trn(\pi)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ *(we set $\mathtt{Incr}(\mathtt{v}) = 0$ for $\mathtt{C}^+(\mathtt{v}) = \emptyset$)*

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

Finding Path-Sensitive Local Bound Sets. Recall, that our path-sensitive reasoning is based on path-sensitive local bounds (Definition 38). Accordingly, Definition 47 requires sets of *path-sensitive* local bounds. We now discuss how to determine such a set of local bounds for a given transition $\tau \in E$.
Let $\tau \in E$. Assume we have inferred that $\{\mathtt{v}_1, \ldots \mathtt{v}_k\}$ with $\mathtt{v}_i \in \mathcal{V}$ is a *local bound set* for τ by the algorithm discussed in Section 4.2.1. We infer a *path-sensitive local bound set* for τ as follows: We check whether $C(\tau) \subseteq \bigcup_{i=1}^{k} \mathtt{C}^-(\mathtt{v}_i)$. If so, we set $\zeta(\tau) = \{\mathtt{v}_1, \ldots \mathtt{v}_k\}$. If the check fails we set $\zeta(\tau) = \emptyset$.
Generalization to the case where the *local bound set* for τ is of form
$\{\mathtt{v}_1, \ldots \mathtt{v}_k, \mathtt{expr}_1, \ldots, \mathtt{expr}_n\}$
with $\mathtt{v}_i \in \mathcal{V}$ and $\mathtt{expr}_i \notin \mathcal{V}$ is straightforward: Similar to Definition 45 we require that $\sum_{i=1}^{n} \mathtt{expr}_i$ is a bound on the number of times that the paths $C(\tau) \setminus \bigcup_{i=1}^{k} \mathtt{C}^-(\mathtt{v}_i)$ can be executed, i.e., a *path bound*. We infer path bounds based on the following observation: Let $\mathtt{tb}(\tau)$ denote a *bound* for τ. Let π be a path. We have that in particular $\min_{\tau \in trn(\pi)} \mathtt{tb}(\tau)$ is a path bound for π.

Example II. Consider Example `s_SFD_process` in Figure 3.14 (page 66). Assume we want to compute a loop bound for the outer loop at l_1, i.e., a bound for its single back-edge τ_3. By the algorithm sketched in the previous paragraph we determine the *local bound set* $\{[p], [m]\}$ for τ_3, i.e., we set $\zeta(\tau_3) = \{[p], [m]\}$. Our algorithm from Definition 47 infers $T\mathcal{B}(\tau_3) = [n] + [n]$. The details are given in Table 4.2. We thus obtain the tight loop bound $2n$ (n has type *unsigned*) for the outer loop at l_1 of Example `s_SFD_process`. (The precise loop bound for the outer loop is $2n - 1$).

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to T\mathcal{B}([p]) + T\mathcal{B}([m])$ $\to [n] + T\mathcal{B}([m])$ $\to [n] + [n]$	$\zeta(\tau_3) = \{[p], [m]\}$, $T\mathcal{B}([p])$, $T\mathcal{B}([m])$
$T\mathcal{B}([p])$	$\to \texttt{Incr}([p]) + \max([n] + 0, 0)$ $\to 0 + \max([n] + 0, 0)$ $= [n]$	$\mathcal{R}([p]) = (\tau_0, [n], 0)$, $[n] \in \mathcal{C}$, $\texttt{Incr}([p])$
$T\mathcal{B}([m])$	$\to \texttt{Incr}([m]) + \max([0] + 0, 0)$ $= \texttt{Incr}([m])$ $\to [n]$	$\mathcal{R}([m]) = (\tau_0, [0], 0)$, $[0] \in \mathcal{C}$, $\texttt{Incr}([m])$
$\texttt{Incr}([m])$	$\to T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\}) \times 1$ $\to [n] \times 1$ $= [n]$	$\texttt{C}^+([m]) = \{\tau_1 \circ \tau_2 \circ \tau_3\}$, $T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\})$
$T\mathcal{B}(\{\tau_1, \tau_2, \tau_3\})$	$\to \min(T\mathcal{B}(\tau_1), T\mathcal{B}(\tau_2), T\mathcal{B}(\tau_3))$ $\to \min([n] + [n], [n], [n] + [n])$ $= [n]$	$T\mathcal{B}(\tau_1)$, $T\mathcal{B}(\tau_2)$, $T\mathcal{B}(\tau_3)$
$T\mathcal{B}(\tau_1)$	$\to T\mathcal{B}([p]) + T\mathcal{B}([m])$ $\to [n] + T\mathcal{B}([m])$ $\to [n] + [n]$	$\zeta(\tau_1) = \{[p], [m]\}$, $T\mathcal{B}([p])$, $T\mathcal{B}([m])$
$T\mathcal{B}(\tau_2)$	$\to \texttt{Incr}([p]) + \max([n] + 0, 0)$ $\to 0 + \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_2) = [p]$, $\mathcal{R}([p]) = (\tau_0, [n], 0)$, $[n] \in \mathcal{C}$, $\texttt{Incr}([p])$
$\texttt{Incr}([p])$	$\to 0$	$\texttt{C}^+([p]) = \emptyset$

Table 4.2: Computation of $T\mathcal{B}(\tau_3)$ for Example `s_SFD_process` (Figure 3.14 (b)) by Definition 47

4.2.2 Multiple Transitions with the same Local Bound

Example `SingleLinkCluster` in Figure 4.13 (a) is an example which we encountered during our experiments, it models the function *SingleLinkCluster* in file *456.hmmer/src/weight.c* of the SPEC CPU 2006 Benchmark [spe].

Example `SingleLinkCluster` has 3 nested loops with the loop counters a, b and i. Similar to Example `tarjan` (discussed in Section 1.4.1) we have that in Example `SingleLinkCluster` the counter b of the middle loop is incremented on the outer loop. But moreover, b is also incremented in the innermost loop with counter i. I.e., executing the innermost loop may increase the number of executions of the middle loop which in turn increases the number of executions of the inner loop.

In Figure 4.13 (b) the *DCP*, obtained by applying our abstraction procedure from

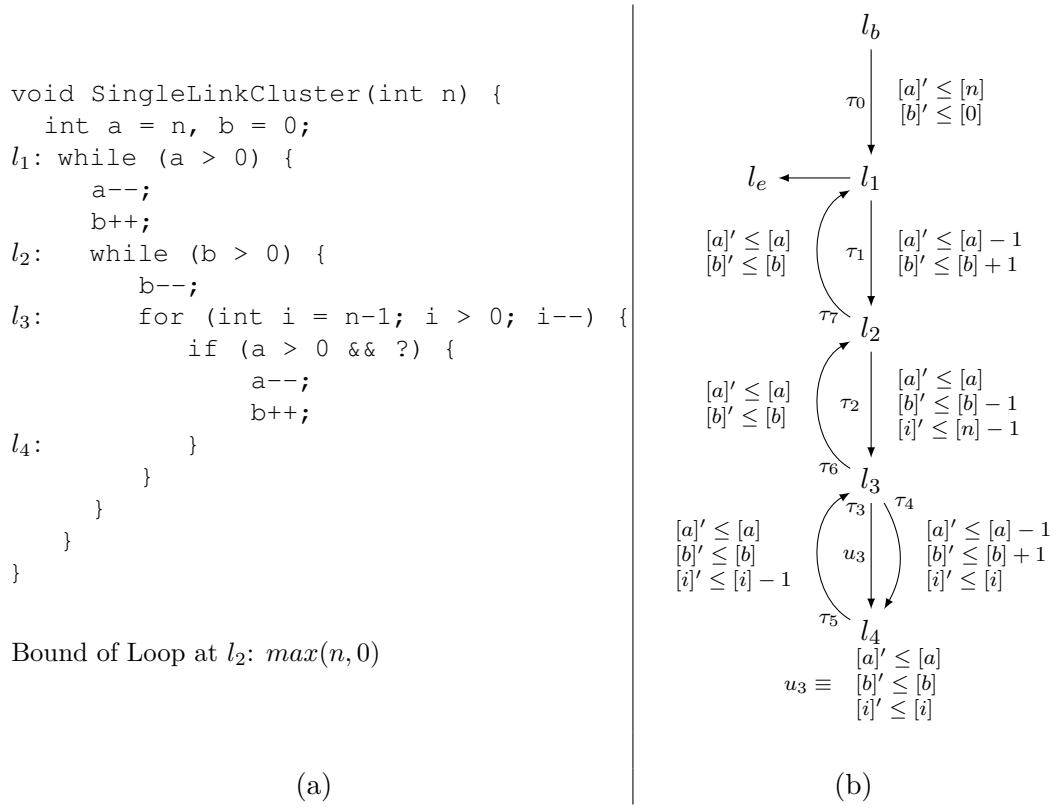


Figure 4.13: (a) Example `SingleLinkCluster`, (b) *DCP* obtained by abstraction

Section 2.2 to Example `SingleLinkCluster`, is shown.

Example I. Assume we want to compute a bound on the number of times that the middle loop at l_2 of Example `SingleLinkCluster` can be executed. We thus compute a bound for τ_6 (the single back-edge of the middle loop). We obtain $T\mathcal{B}(\tau_6) = 2 \times [n]$. The details are given in Table 4.3. (For simplicity we apply our algorithm from Definition 27, our complete algorithm from Definition 42 infers the same bound.) We thus get $2 \times \max(n, 0)$ as a loop bound for the loop at l_2. The loop at l_2, however, can only be executed up to $\max(n, 0)$ times: Initially b, the local bound of τ_6, is 0, b is increased by 1 when executing τ_1 or τ_4. On both transitions, however, a is decreased. I.e., both, τ_1 and τ_4, have local bound a. Since a is initially n the total amount by which the potential of executing τ_6 can be increased is n.

We generalize our observation: Let $\tau_1, \tau_2 \in E$ be two transitions with the same local bound, i.e., $\zeta(\tau_1) = \zeta(\tau_2)$. If τ_1 and τ_2 cannot be executed without decreasing the common local bound $\zeta(\tau_1)$ *twice*, once for τ_1 and once for τ_2 (e.g., τ_1 and τ_4 in `SingleLinkCluster`), we have that $\sharp(\tau_1, \rho) + \sharp(\tau_2, \rho) \leq [\![T\mathcal{B}(\tau_1)]\!](\sigma_0) = [\![T\mathcal{B}(\tau_2)]\!](\sigma_0)$. Thus, $T\mathcal{B}(\tau_1)$ is a bound on the number of times that τ_1 *and* τ_2 can be executed on any run of $\Delta\mathcal{P}$. We exploit

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_6)$	$\to \texttt{Incr}([b]) + T\mathcal{B}(\tau_0) \times \max([0]+0,0)$ $= \texttt{Incr}([b])$ $\to 2 \times [n]$	$\zeta(\tau_6) = [b]$, $\mathcal{R}([b]) = \{(\tau_0, [0], 0)\}$, $\texttt{Incr}([b])$
$\texttt{Incr}([b])$	$\to T\mathcal{B}(\tau_1) \times 1 + T\mathcal{B}(\tau_4) \times 1$ $\to [n] \times 1 + T\mathcal{B}(\tau_4) \times 1$ $\to [n] \times 1 + [n] \times 1$ $= 2 \times [n]$	$\mathcal{I}([b]) = \{(\tau_1, 1), (\tau_4, 1)\}$, $T\mathcal{B}(\tau_1)$, $T\mathcal{B}(\tau_4)$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([a]) + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + 1 \times \max([n]+0,0)$ $= [n]$	$\zeta(\tau_1) = [a]$, $\mathcal{R}([a]) = \{(\tau_0, [n], 0)\}$, $\texttt{Incr}([a])$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_4)$	$\to \texttt{Incr}([a]) + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + 1 \times \max([n],0)$ $= [n]$	$\zeta(\tau_4) = [a]$, $\mathcal{R}([a]) = \{(\tau_0, [n], 0)\}$, $\texttt{Incr}([a])$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$\texttt{Incr}([a])$	$\to 0$	$\mathcal{I}([a]) = \emptyset$

Table 4.3: Computation of $T\mathcal{B}(\tau_5)$ for Example `SingleLinkCluster` (Figure 4.13) by Definition 27

this observation: Assume some $\mathtt{v} \in \mathcal{V}$ is incremented by $\mathtt{c}_1$ on τ_1 and by $\mathtt{c}_2$ on τ_2. For computing $\texttt{Incr}(\mathtt{v})$ we only add $T\mathcal{B}(\tau_1) \times \max\{\mathtt{c}_1, \mathtt{c}_2\}$ instead of $T\mathcal{B}(\tau_1) \times \mathtt{c}_1 + T\mathcal{B}(\tau_2) \times \mathtt{c}_2$. This idea can be further generalized to multiple transitions.

Example II. Applying the described optimization to our algorithm we obtain $T\mathcal{B}(\tau_6) = [n]$ for Example `SingleLinkCluster` as shown in Table 4.4. Our algorithm thus infers the *precise* loop bound $\max(n, 0)$ for the loop at l_2.

4.2.3 Bounds involving Integer Division

Reconsider the example from Figure 4.1 (a) (page 85). As discussed in Section 4.1.1 the loop at l_2 of Figure 4.1 (a) (corresponds to transition τ_3 in the abstraction) has loop bound $\lfloor \frac{n+1}{2} \rfloor$. However, for the abstraction in Figure 4.2 our bound algorithm from Definition 27 computes either $T\mathcal{B}(\tau_3) = n$ if we choose $\zeta(\tau_3) = [i]$ or $T\mathcal{B}(\tau_3) = n + 1$ if we choose $\zeta(\tau_3) = [i + 1]$ (see Table 4.5 for details). We make the following observation:

Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a *DCP* over $\mathcal{A}$. Let $\tau \in E$. Let $\mathtt{v} \in \mathcal{V}$ be a *local bound* for τ, i.e., for all runs ρ of $\Delta\mathcal{P}$ we have that $\sharp(\tau, \rho) \leq \downarrow(\mathtt{v}, \rho)$. Let $\mathtt{c} \in \mathbb{N}$. Let $\downarrow(\mathtt{v}, \mathtt{c}, \rho)$ denote the number of times that the value of $\mathtt{v}$ decreases on a run ρ of $\Delta\mathcal{P}$ by at least $\mathtt{c}$

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_6)$	$\to \texttt{Incr}([b]) + T\mathcal{B}(\tau_0) \times \max([0]+0,0)$ $= \texttt{Incr}([b])$ $[n]$	$\zeta(\tau_6) = [b]$, $\mathcal{R}([b]) = \{(\tau_0, [0], 0)\}$, $\texttt{Incr}([b])$
$\texttt{Incr}([b])$	$\to T\mathcal{B}(\tau_1) \times \max(1,1)$ $\to [n] \times \max(1,1)$ $= [n]$	$\mathcal{I}([b]) = \{(\tau_1, 1), (\tau_4, 1)\}$, $T\mathcal{B}(\tau_1)$, $\zeta(\tau_1) = \zeta(\tau_4) = [a]$, τ_1, τ_4 are in distinct loops
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([a]) + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + 1 \times \max([n]+0,0)$ $= [n]$	$\zeta(\tau_1) = [a]$, $\mathcal{R}([a]) = \{(\tau_0, [n], 0)\}$, $\texttt{Incr}([a])$, $T\mathcal{B}(\tau_0)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$\texttt{Incr}([a])$	$\to 0$	$\mathcal{I}([a]) = \emptyset$

Table 4.4: Computation of $T\mathcal{B}(\tau_5)$ for Example `SingleLinkCluster` (Figure 4.13) using the optimization for "multiple transitions with the same local bound" (Section 4.2.2)

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}([i+1]) + T\mathcal{B}(\tau_0) \times \max([0]+1,0)$ $\to [n] + T\mathcal{B}(\tau_0) \times \max([0]+1,0)$ $\to [n] + 1 \times \max([0]+1,0)$ $= [n]+1$	$\zeta(\tau_3) = [i+1]$, $\mathcal{R}([i{+}1]) = \{(\tau_0, [0], 1)\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}([i+1])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([i+1])$	$\to T\mathcal{B}(\tau_1) \times 1$ $\to [n] \times 1$ $= [n]$	$\mathcal{I}([i+1]) = \{(\tau_1, 1)\}$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([j]) + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n]+0,0)$ $\to 0 + 1 \times \max([n]+0,0)$ $= [n]$	$\zeta(\tau_1) = [j]$, $\mathcal{R}([j]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$ $\texttt{Incr}([j])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([j])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$

Table 4.5: Computation of $T\mathcal{B}(\tau_3)$ for Figure 4.2 by Definition 27

(refines Definition 15). If for all runs ρ of $\Delta\mathcal{P}$ we have that $\sharp(\tau,\rho) \leq \downarrow(\mathtt{v},\mathtt{c},\rho)$ (refines Definition 16) then $[\![\lfloor \frac{T\mathcal{B}(\tau)}{\mathtt{c}} \rfloor]\!]$ is a *bound* for τ (assuming $\zeta(\tau) = \mathtt{v}$).

In our simple abstract program model, $\mathtt{c} \in \mathbb{N}$ is obtained syntactically from a constraint $\mathtt{v}' \leq \mathtt{v} - \mathtt{c}$. See Section 3.6 on how we determine relevant constraints.

Example. For the *DCP* in Figure 4.2 (the abstraction of Figure 4.1 (a)) we infer that $[i+1]$ is a *local bound* for τ_3 by the algorithm discussed in Section 3.6. We thus set $\zeta(\tau_3) = [i+1]$. We obtain $\sharp(\tau_3,\rho) \leq \downarrow([i+1],2,\rho)$ for all runs ρ of Figure 4.2 directly from the constraint $[i+1] \leq [i+1] - 2$ on τ_3. We thus conclude that $[\![\lfloor \frac{T\mathcal{B}(\tau_3)}{2} \rfloor]\!]$ is a *bound* for τ. With $T\mathcal{B}(\tau_3) = n+1$ (see Table 4.5) we obtain the precise bound $\lfloor \frac{n+1}{2} \rfloor$ for τ_3.

4.2.4 Symbolic Increments

We discussed in Section 4.1.4 an extension of our program model (*DCP*s with symbolic increments) that allows to represent updates of form $i = i + \mathtt{c}$ where $\mathtt{c}$ is some symbolic expression over the program parameters. Reconsider the program in Figure 4.10 (a). By the extension of our abstraction algorithm that we discussed in Section 4.1.4 we obtain the *DCP with symbolic increments* shown in Figure 4.10 (c).
We now discuss how our bound algorithm computes the precise loop bound $n \times c$ for the loop at l_2 of Figure 4.10 (a): Since the symbolic constant $[c]$ of the abstract program can take any value in $\mathbb{Z}$, in particular a positive one, we consider the predicate $[i]' \leq [i] + [c]$ in Figure 4.10 (c) to be an *increment* of $[i]$ and thus set $\mathcal{I}([i]) = \{\tau_1, [c]\}$. Table 4.6 shows in detail how our algorithm obtains $T\mathcal{B}(\tau_3) = [n] \times [c]$ (transition τ_3 corresponds to the loop at l_2) based on $\mathcal{I}([i]) = \{\tau_1, [c]\}$.

We generalize our algorithm to *DCPs with symbolic increments* by generalizing Definition 19 to Definition 48.

Definition 48 (Resets and Increments for *DCP*s with symbolic increments)**.** *Let $\Delta\mathcal{P} = (L, E, l_b, l_e)$ be a DCP with symbolic increments over $\mathcal{A}$. Let $\mathtt{v} \in \mathcal{V}$. We define the* resets $\mathcal{R}(\mathtt{v})$ *and* increments $\mathcal{I}(\mathtt{v})$ *of* $\mathtt{v}$ *as follows:*

$$\mathcal{R}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in E \times \mathcal{A} \times (\mathbb{Z} \cup \mathcal{C}) \mid \mathtt{v}' \leq \mathtt{a} + \mathtt{c} \in u, \mathtt{a} \neq \mathtt{v}\}$$
$$\mathcal{I}(\mathtt{v}) = \{(l_1 \xrightarrow{u} l_2, \mathtt{c}) \in E \times (\mathbb{N} \cup \mathcal{C}) \mid \mathtt{v}' \leq \mathtt{v} + \mathtt{c} \in u \text{ where } \mathtt{c} \in \mathbb{N} \text{ and } \mathtt{c} > 0 \text{ or } \mathtt{c} \in \mathcal{C}\}$$

Now consider the example shown in Figure 4.14 (a) which is similar to the running example of [BEF+16]. The precise loop bound for the loop at l_2 is

$$\sum_{i=1}^{n} 2i = 2 \times \sum_{i=1}^{n} i = 2 \times \frac{n^2+n}{2} = n^2 + n.$$

Even in our extended program model of *DCP*s with symbolic increments we cannot express the update $x = x + i$ of x in the loop at l_1, since i is not a *program parameter*,

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}([i]) + T\mathcal{B}(\tau_0) \times \max([0], 0)$ $= \texttt{Incr}([i])$ $\to [n] \times [c]$	$\zeta(\tau_3) = [i]$, $\mathcal{R}([i]) = \{(\tau_0, [0], 0)\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}([i])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([i])$	$\to T\mathcal{B}(\tau_1) \times [c]$ $\to [n] \times [c]$	$\mathcal{I}([i]) = \{(\tau_1, [c])\}$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([j]) + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_1) = [j]$, $\mathcal{R}([j]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$, $\texttt{Incr}([j])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([j])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$

Table 4.6: Computation of $T\mathcal{B}(\tau_3)$ for Figure 4.10 (c) by Definition 27, extended to *DCP*s with symbolic increments by Definition 48

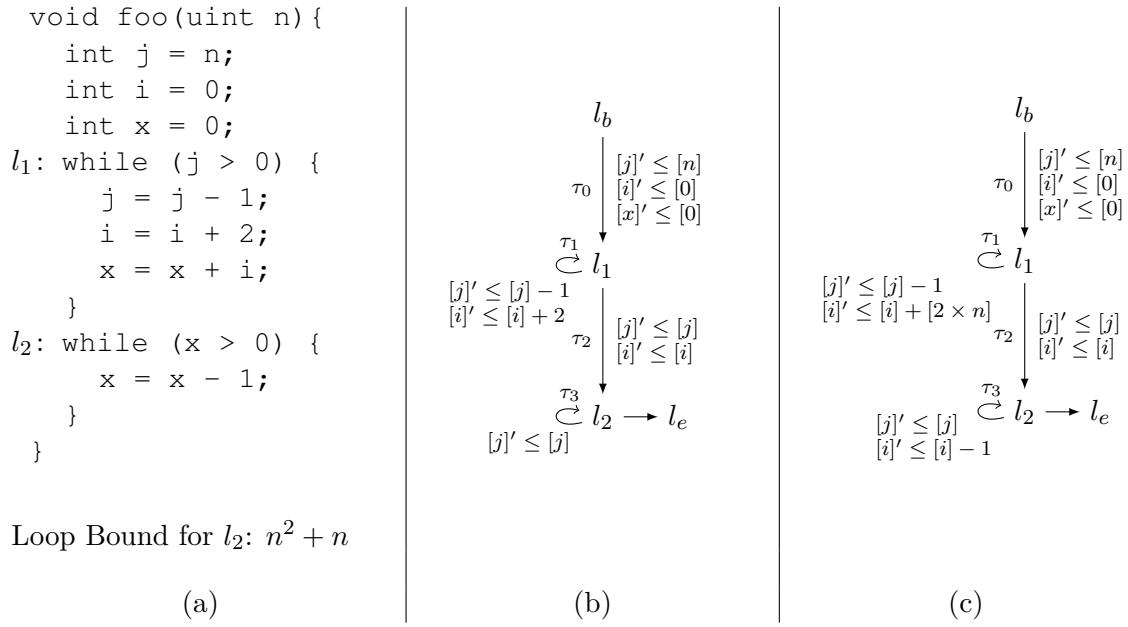


Figure 4.14: (a) Example, (b) *DCP* as obtained by our abstraction algorithm, (c) *DCP* with symbolic increments as obtained by our extended abstraction algorithm, using the *upper bound invariant* $i \leq 2 \times n$

Call	Evaluation and Simplification	Using
$V\mathcal{B}([i])$	$\to \texttt{Incr}([i]) + \max([0], 0)$ $= \texttt{Incr}([i])$ $\to [n] \times 2$	$\mathcal{R}([i]) = \{(\tau_0, [0], 0)\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}([i])$
$\texttt{Incr}([i])$	$\to T\mathcal{B}(\tau_1) \times 2$ $\to [n] \times 2$	$\mathcal{I}([i]) = \{(\tau_1, 2)\}$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([j]) + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_1) = [j]$, $\mathcal{R}([j]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$, $\texttt{Incr}([j])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([j])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$

Table 4.7: Computation of $V\mathcal{B}([i])$ for Figure 4.14 (b) by Definition 27

Call	Evaluation and Simplification	Using
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}([x]) + T\mathcal{B}(\tau_0) \times \max([0], 0)$ $= \texttt{Incr}([x])$ $\to [n] \times [2 \times n]$	$\zeta(\tau_3) = [x]$, $\mathcal{R}([x]) = \{(\tau_0, [0], 0)\}$, $[0] \in \mathcal{C}$, $\texttt{Incr}([x])$
$\texttt{Incr}([x])$	$\to T\mathcal{B}(\tau_1) \times [2 \times n]$ $\to [n] \times [2 \times n]$	$\mathcal{I}([x]) = \{(\tau_1, [2 \times n])\}$, $T\mathcal{B}(\tau_1)$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([j]) + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_1) = [j]$, $\mathcal{R}([j]) = \{(\tau_0, [n], 0)\}$, $[n] \in \mathcal{C}$, $\texttt{Incr}([j])$, $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([j])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$

Table 4.8: Computation of $T\mathcal{B}(\tau_3)$ for Figure 4.14 (c) by Definition 27, extended to *DCP*s with symbolic increments by Definition 48.

but a variable which value changes during program run. We thus obtain the *DCP* shown in Figure 4.14 (b) from our (extended) abstraction algorithm.

However, considering Figure 4.14 (b) we can obtain an *upper bound invariant* for $\max(i, 0)$, namely $[i] \leq 2 \times [n]$, by applying our *variable bound* algorithm: We get $V\mathcal{B}([i]) = 2 \times [n]$ as shown in Table 4.7.

Given that $i \leq [i] \leq 2 \times [n] = [2 \times n]$ we now over-approximate i in the update $x = x + i$ by $[2 \times n]$ and get the predicate $x' \leq x + [2 \times n]$ which is a *difference constraint with symbolic increment*. We thus obtain the *DCP with symbolic increments* shown in Figure 4.14 (c) as a valid abstraction of the original program in Figure 4.14 (a).

Applying our bound algorithm on Figure 4.14 (c) we obtain $T\mathcal{B}(\tau_3) = [n] \times [2 \times n]$ (see Table 4.8 for details), given that n is of type *unsigned* we thus obtain the tight loop bound $2n^2$ for the loop at l_2 in Figure 4.14 (a). The precise loop bound is $n^2 + n$.

In the same way the running example of [BEF+16] is handled by our approach.

In general, we abstract an update of form $x = x + \texttt{expr}$ where `expr` is some expression over variables, constants and program parameters by 1) inferring an *upper bound invariant* $\texttt{expr} \leq \texttt{expr}_c$, where $\texttt{expr}_c$ is an expression over constants and program parameters only. Based on $\texttt{expr} \leq \texttt{expr}_c$, we 2) obtain a *DCP with symbolic increments* as discussed in Section 4.1.4. We obtain the *upper bound invariant* $\texttt{expr} \leq \texttt{expr}_c$ by 1a) abstracting the program under scrutiny based on the norm $[\texttt{expr}]$ and 1b) computing $V\mathcal{B}([\texttt{expr}])$ (Definition 27) on the resulting abstraction. We fail to abstract the update $x = x + \texttt{expr}$ if either step 1a) or step 1b) fails.

4.2.5 More Precise Variable Bounds

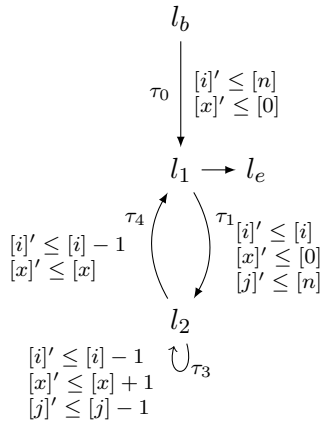
Consider the example in Figure 4.15 (a). It is easy to see that the assertion at the end of the example holds: Variable x is set to 0 before each execution of the inner loop, and the inner loop may be executed at most n times within one iteration of the outer loop.

However, our algorithm computes $V\mathcal{B}([x]) = [n] \times [n]$ as depicted in Table 4.9 (we show the computation as performed by Definition 27, however, our full algorithm from Definition 42 obtains the same result). The reason is, that our variable bound algorithm ignores the *reset* of $[x]$ to $[0]$ on transition τ_1. We show how this problem can be solved by a simple extension of our algorithm.

For simplicity we exemplify our extension on Definition 27. It can, however, be applied in the same way to our full bound algorithm from Definition 42.

```
void foo(uint n) {
  int i = n;
  uint x = 0;
  while(i > 0) {
    int j = n;
    x = 0;
    while(j > 0 && ?) {
      j--;
      x++;
    }
    i--;
  }
  assert(x <= n);
}
```

(a)



(b)

Figure 4.15: (a) Example, (b) *DCP* abstraction

Call	Evaluation and Simplification	Using
$V\mathcal{B}([x])$	$\to \texttt{Incr}([x]) + T\mathcal{B}(\tau_1) \times \max([0], 0)$ $+ T\mathcal{B}(\tau_0) \times \max([0], 0)$ $= \texttt{Incr}([x])$ $\to [n] \times [n]$	$\mathcal{R}([x]) = \{(\tau_1, [0], 0),$ $(\tau_1, [0], 0)\},$ $[0] \in \mathcal{C},$ $\texttt{Incr}([x])$
$\texttt{Incr}([x])$	$\to T\mathcal{B}(\tau_3) \times 1$ $\to ([n] \times [n]) \times 1$ $= [n] \times [n]$	$\mathcal{I}([x]) = \{(\tau_3, 1\},$ $T\mathcal{B}(\tau_3)$
$T\mathcal{B}(\tau_3)$	$\to \texttt{Incr}([j]) + T\mathcal{B}(\tau_1) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_1) \times \max([n] + 0, 0)$ $\to 0 + [n] \times \max([n] + 0, 0)$ $= [n] \times [n]$	$\zeta(\tau_3) = [j],$ $\mathcal{R}([j]) = \{(\tau_1, [n], 0)\},$ $[n] \in \mathcal{C},$ $\texttt{Incr}([j]),$ $T\mathcal{B}(\tau_1)$
$\texttt{Incr}([j])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$
$T\mathcal{B}(\tau_1)$	$\to \texttt{Incr}([i]) + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_0) \times \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta(\tau_1) = [i],$ $\mathcal{R}([i]) = \{(\tau_0, [n], 0)\},$ $[n] \in \mathcal{C},$ $\texttt{Incr}([i]),$ $T\mathcal{B}(\tau_0)$
$\texttt{Incr}([i])$	$\to 0$	$\mathcal{I}([i]) = \emptyset$
$T\mathcal{B}(\tau_0)$	$\to 1$	$\zeta(\tau_0) = 1$

Table 4.9: Computation of $V\mathcal{B}([x])$ for Figure 4.15 (b) by Definition 27

Call	Evaluation and Simplification	Using
$V\mathcal{B}([x])$	$\to \texttt{Incr}([x], \zeta[\tau_1 \mapsto 1])$ $\quad + \ T\mathcal{B}(\tau_1) \times \max([0], 0)$ $\quad + \ T\mathcal{B}(\tau_0) \times \max([0], 0)$ $= \texttt{Incr}([x], \zeta[\tau_1 \mapsto 1])$ $\to [n]$	$\mathcal{R}([x]) = \{(\tau_1, [0], 0),$ $\quad (\tau_0, [0], 0)\},$ $[0] \in \mathcal{C},$ $\texttt{Incr}([x], \zeta[\tau_1 \mapsto 1])$
$\texttt{Incr}([x], \zeta[\tau_1 \mapsto 1])$	$\to T\mathcal{B}(\tau_3, \zeta[\tau_1 \mapsto 1]) \times 1$ $\to [n] \times 1$ $= [n]$	$\mathcal{I}([x]) = \{(\tau_3, 1\},$ $T\mathcal{B}(\tau_3, \zeta[\tau_1 \mapsto 1])$
$T\mathcal{B}(\tau_3, \zeta[\tau_1 \mapsto 1])$	$\to \texttt{Incr}([j], \zeta[\tau_1 \mapsto 1])$ $\quad + \ T\mathcal{B}(\tau_1, \zeta[\tau_1 \mapsto 1])$ $\quad\quad \times \ \max([n] + 0, 0)$ $\to 0 + T\mathcal{B}(\tau_1, \zeta[\tau_1 \mapsto 1])$ $\quad\quad \times \ \max([n] + 0, 0)$ $\to 0 + 1 \times \max([n] + 0, 0)$ $= [n]$	$\zeta[\tau_1 \mapsto 1](\tau_3) = [j],$ $\mathcal{R}([j]) = \{(\tau_1, [n], 0)\},$ $[n] \in \mathcal{C},$ $\texttt{Incr}([j]),$ $T\mathcal{B}(\tau_1, \zeta[\tau_1 \mapsto 1])$
$\texttt{Incr}([j], \zeta[\tau_1 \mapsto 1])$	$\to 0$	$\mathcal{I}([j]) = \emptyset$
$T\mathcal{B}(\tau_1, \zeta[\tau_1 \mapsto 1])$	$\to 1$	$\zeta[\tau_1 \mapsto 1](\tau_1) = 1$

Table 4.10: Computation of $V\mathcal{B}([x])$ for Figure 4.15 (b) by Definition 49

Definition 49 (Bound Algorithm)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\zeta : E \to Expr(\mathcal{A})$. We define $V\mathcal{B} : \mathcal{A} \mapsto Expr(\mathcal{A})$ and $T\mathcal{B} : E \mapsto Expr(\mathcal{A})$ as:*

$$V\mathcal{B}(\mathtt{a}) = \mathtt{a}, \textit{ if } \mathtt{a} \in \mathcal{C}, \textit{ else}$$
$$V\mathcal{B}(\mathtt{v}) = \texttt{Incr}(\mathtt{v}, \zeta[\{\tau \mapsto 1 \mid (\tau, \_, \_) \in \mathcal{R}(\mathtt{v})\}]) + \max_{(\_, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} (V\mathcal{B}(\mathtt{a}) + \mathtt{c})$$

$$T\mathcal{B}(\tau) = T\mathcal{B}(\tau, \zeta)$$

$$T\mathcal{B}(\tau, \xi) = \xi(\tau), \textit{ if } \xi(\tau) \notin \mathcal{V}, \textit{ else}$$
$$T\mathcal{B}(\tau, \xi) = \texttt{Incr}(\xi(\tau), \xi) + \sum_{(t, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\xi(\tau))} T\mathcal{B}(t, \xi) \times \max(V\mathcal{B}(\mathtt{a}) + \mathtt{c}, 0)$$

where

1. $\texttt{Incr}(\mathtt{v}, \xi) = \sum_{(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})} T\mathcal{B}(\tau, \xi) \times \mathtt{c}$ *(we set* $\texttt{Incr}(\mathtt{v}, \xi) = 0$ *for* $\mathcal{I}(\mathtt{v}) = \emptyset$*)*

2. $\zeta[\{\tau \mapsto 1 \mid (\tau, \_, \_) \in \mathcal{R}(\mathtt{v})\}]$ *denotes a mapping that results from ζ by assigning all τ with $(\tau, \_, \_) \in \mathcal{R}(\mathtt{v})$ the expression* 1

Discussion. Definition 49 differs from Definition 27 by 1) the new, second parameter ξ of $T\mathcal{B}$ and `Incr` (note that by default ξ is set to the local bound mappign ζ), and 2) setting the parameter ξ to $\zeta[\{\tau \mapsto 1 \mid (\tau, \_, \_) \in \mathcal{R}(\mathtt{v})\}]$ when calling `Incr` from $V\mathcal{B}$. Note that ξ is used as local bound mapping in the definition of $T\mathcal{B}$. The notation $\zeta[\{\tau \mapsto 1 \mid (\tau, \_, \_) \in \mathcal{R}(\mathtt{v})\}]$ denotes a mapping that results from ζ by assigning all transitions τ with $(\tau, \_, \_) \in \mathcal{R}(\mathtt{v})$ the expression '1'. I.e., when `Incr` is called from within $V\mathcal{B}(\mathtt{v})$, all transitions which reset `v` are assigned the constant transition bound '1'. This has the effect, that only one repetition of these transitions is considered when reasoning *how often* a transition that *increments* `v` can be executed. The single execution of a transition that resets `v` models the initialization of `v`, before `v` is incremented.

Example. Reconsider the example in Figure 4.15. Table 4.10 shows how we obtain $V\mathcal{B}([x]) = [n]$, i.e., the upper bound invariant $x \leq n$ (x and n have type *unsigned*), by our algorithm from Definition 49.

CHAPTER 5

Evaluation

5.1 Implementation

We have implemented the presented algorithm into our prototype tool `loopus` [looa]. `loopus` reads in the LLVM [LA04] intermediate representation and performs an intra-procedural analysis. We designed `loopus` to work with the LLVM intermediate representation as it typically results from compiling *C code* with the *clang* compiler. We do not have experience with LLVM intermediate representations obtained from other programming languages or by other compilers.

`loopus` is capable of computing bounds for loops as well as analyzing the complexity of non-recursive functions. In both cases we assume the *back-edge metric* (see Section 1) as *cost model*.

In our implementation we perform program abstraction on-demand during bound computation. I.e., we abstract program parts in the moment they turn out to be relevant for inferring the required bound. This has the obvious advantage that no time is spent in abstracting irrelevant parts of the program.

`loopus` models integers as mathematical integers (not bit-vectors), which is the standard approach in the bound analysis literature. We use the Z3 SMT solver [dMB08] for performing control-flow refinement and program abstraction. Our implementation considers only *loop headers* as *program locations*. I.e., we apply *large-block encoding* as discussed in Section 4.1.3.

By default `loopus` inlines calls to functions which are not recursive nor contain loops.

By a command-line option the tool can be instructed to compute bounds not in terms of the function parameters but in terms of the variables which are defined at the header of the *strongly connected component* (SCC) of the control flow graph in which the loop is situated.

# paths	1	2	3 - 9	10 - 99	100 - 299	300 - 1999	2000 - 4999	≥ 5000
unsliced	1174	578	616	310	66	44	11	34
sliced	1623	512	424	183	37	30	8	16
merged	1766	429	415	186	24	10	1	2
refined	1766	429	414	187	24	10	1	2

Table 5.1: Number of simple and cyclic paths with respective number of SCCs in the cBench [cbe] benchmark

We discussed in Section 3.2 that an successful answer of our bound algorithm implies that the *local bounds* can be ordered to form a *lexicographic ranking function* which proves termination of the loop or function under scrutiny. In our implementation we *explicitly* check if this condition applies before starting the bound algorithm. This allows for an *early* failure of our analysis in case no lexicographic ranking function can be generated. We could restart the analysis with a different choice of local bounds, but this feature is not yet implemented. By a command-line option the tool can be asked to print out the inferred ranking function, thereby proving termination.

5.1.1 Handling of Pointers and Data Structures

By default `loopus` soundly abstracts from all instructions which cannot directly be modeled in terms of integer valued expressions. Real C code, however, often contains pointers and (recursive) data structures. A typical loop iteration pattern is the iteration over a *list* or a *tree*. As a means to test the potential of our tool and its performance, and in order to find interesting examples, we implemented heuristic methods for handling non-integer code. These heuristics can be activated by command-line parameters. If the corresponding command-line parameter is set, `loopus` infers bounds on loops iterating over arrays or recursive data structures by introducing *shadow variables* that represent norms such as the length of a list or the size of an array. Further, `loopus` makes the following optimistic assumptions if the corresponding command-line parameters are set: 1) pointers do not alias; 2) a recursive data structure is acyclic if a loop iterates over it; 3) a loop iterating over an array of characters is assumed to be terminating if an inequality check on the string termination character '\0' is found[1]; 4) given a loop condition of form '$a \neq 0$' `loopus` heuristically decides to either assume '$a > 0$' or '$a < 0$' as loop-invariant; 5) similarly, `loopus` assumes '$x > 0$' when an update of a loop counter of the form '$x = x * 2$' or '$x = x/2$' is detected. These assumptions are reported to the user if they were applied while computing the bound. The validity of the reported assumptions can, in principle, be checked by an external tool.

5.1.2 Slicing, Path Reduction and Control-Flow Refinement

As discussed, the potentially *exponential* complexity of our analysis evolves from combining transitions to paths as it is necessary for our reasoning on *reset chains* (Section 3.5)

[1] This assumption is often necessary to infer a bound since the type system of C does not distinguish between an array of characters and a string

	Succ.	1	n	n^2	n^3	$n^{>3}$	2^n	Total Time	# Time Outs
`loopus'15`	806	205	489	97	13	2	0	15m	6
`loopus'14`	431	200	188	43	0	0	0	40m	20
KoAT	430	253	138	35	2	0	2	5.6h	161
CoFloCo	386	200	148	38	0	0	0	4.7h	217

Table 5.2: Tool Results on analyzing the complexity of 1659 functions in the cBench benchmark, none of the tools infers *log* bounds.

	Succ.	1	n	n^2	n^3	$n^{>3}$	2^n
`loopus'15`	753	196	466	84	7	0	0
`loopus'14`	414	192	181	41	0	0	0
KoAT	420	245	136	35	2	0	2
CoFloCo	382	198	146	38	0	0	0

Table 5.3: Tool Results on analyzing the complexity of the subset of those functions in the cBench benchmark on which no tool timed out.

and for our *path-sensitive* reasoning (Section 3.8). We therefore take several measures in order to reduce the number of paths that must be considered: First of all, `loopus` applies *program slicing* with regard to the loop exit conditions, i.e., we delete all program behavior that cannot affect the number of loop iterations. In the next step, we exclude duplicates of simple and cyclic paths (see Definition 33) through syntactic comparison. On loops with more than 250 simple and cyclic paths left, we apply what we call *path merging*: For each simple and cyclic path p the conjunction c_p over all its predicates is built. Paths which assign syntactically identical expressions to the loop counters are grouped. The simple and cyclic paths in the same group G are substituted by a new simple and cyclic path with the single predicate $\bigvee_{p \in G} c_p$ (we simplify this predicate by standard techniques from propositional logic). The merged path over-approximates all paths in G. Though some path sensitivity is lost by this technique, we can still bound 81 (31%) of the 259 loops to which path merging is applied when running our tool on the cBench [cbe] benchmark. After the second path reduction step we apply the control-flow refinement techniques discussed in Section 4.1.3. Table 5.1 states the number of SCCs in the cBench benchmark with the respective number of paths in the original program (first row), in the sliced program (second row), after deleting path duplicates and applying path merging (third row), and after applying control-flow refinement (last row). We state the paths per SCC because in our implementation all loops in one SCC are processed at once. Our statistics (Table 5.1) demonstrate that slicing and merging significantly reduce the number of paths while control-flow refinement does not lead to any problematic increase in the path count.

5.2 Experiments

In the following we discuss three experimental setups and tool comparisons. Our first experiment, which we discuss in Section 5.2.1 is performed on a benchmark of open-source C programs. For our second experiment (Section 5.2.2), we assembled a benchmark of challenging programs from the literature on automatic bound analysis. The third experiment was performed on a set of interesting loop iteration patterns that we found in real source code. The different iteration patterns in this benchmark are instances of the *amortized complexity problem*, like, e.g., Example `xnu` (see Section 1.4.7).

5.2.1 Evaluation on Real-World C Code

Experimental Setup. For our experimental comparison we used the program and compiler optimization benchmark *Collective Benchmark* [cbe] (cBench), which contains a total of 1027 different C files (after removing code duplicates) with 211.892 lines of code. We set up the first comparison of complexity analysis tools on real-world code. For comparing our new tool (`loopus`'15) we chose the 3 most promising tools from recent publications: the tool KoAT implementing the approach of [BEF$^+$16], the tool CoFloCo implementing [FH14] and our own earlier implementation (`loopus`'14) [SZV14a]. Note that we compared against the most recent versions of KoAT and CoFloCo (download 01/23/15).[2] We were not able to evaluate *Rank* and *C4B* on our benchmark because both tools support only a limited subset of C. The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor and 16 GB memory. The task was to perform a complexity analysis on function level. We used the following experimental set up:
1) We compiled all 1027 C files in the benchmark into the LLVM intermediate representation using clang.
2) We extracted all 1751 functions which contain at least one loop using the tool *llvm-extract* (comes with the LLVM tool suite). Extracting the functions to single files guarantees an intra-procedural setting for all tools.
3) We used the tool *llvm2kittel* [llv] to translate the 1751 LLVM modules into 1751 text files in the integer transition system (ITS) format that is read in by KoAT.
4) We used the transformation described in [FH14] to translate the ITS format of KoAT into the cost equations representation that is read in by CoFloCo. This last step is necessary because there exists no direct way for translating C or the LLVM intermediate representation into the CoFloCo input format.
5) We decided to exclude the 91 recursive functions from the benchmark set because we were not able to run CoFloCo on these examples (the transformation tool does not support recursion), KoAT was not successful on any of them, and `loopus` does not support recursion.

In total our example set thus comprises 1659 functions.

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

	Succ.	1	n	n^2	n^3	$n^{>3}$	2^n	Time w/o TO	# Time Outs
`loopus'15`	86	2	51	27	1	5	0	4s	0
`loopus'14`	86	2	50	28	2	4	0	4s	0
CoFloCo	87	3	45	34	2	3	0	1m40s	1
Rank	78	3	49	21	3	2	0	20s	0
C4B	36	0	36	0	0	0	0	6s	0
KoAT	90	3	43	36	3	5	0	3m50s	3

Table 5.4: Tool Results on analyzing examples from the literature, none of the tools infers *log* bounds. The time out was 120 seconds. A higher time out did not yield additional results.

Evaluation. Table 5.2 shows the results of all 4 tools on our benchmark using a time out of 60 seconds. The first column shows the number of functions which were successfully bounded by the respective tool, the last column shows the number of time outs, on the remaining examples (not shown in the table) the respective tool did not time out but was also not able to compute a bound. The column *Time* shows the total time used by the respective tool to process the benchmark. `loopus'15` computes the complexity for about twice as many functions as KoAT, CoFloCo, and `loopus'14` while needing an order of magnitude less time than KoAT and CoFloCo, and significantly less time than `loopus'14`. We conclude that our implementation is both more scalable and, on real C code, more successful than implementations of other state-of-the-art approaches.

Pointers and Shapes. Even `loopus'15` computed bounds for only about half of the functions in the benchmark. Studying the benchmark code we concluded that for many functions pointer alias and/or shape analysis is needed for inferring functional complexity. In our experimental comparison such information was not available to the tools. Using optimistic (but unsound) assumptions on pointer aliasing and heap layout (see our discussion in Section 5.1), our tool `loopus'15` was able to compute the complexity for in total 1185 out of the 1659 functions in the benchmark, using 28 minutes total time. For the reasons of failure see our discussion in Section 5.3.

The benchmark and further experimental results can be found on [looa] where our tool is also offered for download.

5.2.2 Evaluation on Examples from the Literature

In order to evaluate the precision of our approach on a number of known challenges to bound analysis, we performed a tool comparison on 110 examples from the literature. Our example set comprises those examples from the tool evaluation in [BEF+16] and [SZV14a] that were available as imperative code (C or pseudo code, in total 89 examples), and additionally the examples used for the evaluation of [CHS15] (15 examples) as well as the running examples of [SZV15] (6 examples). We added the tools *Rank* (implementing [ADFG10]) and *C4B* (implementing [CHS15]) to the comparison, because we were

able to formulate the examples over the restricted C subset that is supported by the two tools (this was not possible for our experiment on real-world code).

The results of our evaluation are shown in Table 5.4. Our two tools `loopus`'15 and `loopus`'14 compute the highest number of *linear* bounds and are also significantly faster than the other tools, in particular than KoAT and CoFloCo. On the other hand, KoAT computes the highest number of bounds in total (4 more than `loopus`). CoFloCo computes, in total, 1 bound more than our tool. The comparable low number of bounds computed by C4B is also due to the fact that the approach implemented in C4B is limited to linear bounds.

In summary, our second evaluation shows that our approach is not only successful on the class of problems on which we focused in this thesis, but solves also many other bound analysis problems from the literature. Note, that in contrast to our first evaluation, our second benchmark contains small examples from academia (1293 LOC, in average 12 lines per file). On these examples our implementation is comparable in strength to the implementation of other state-of-the-art approaches to bound analysis. Since not all features of our analysis are implemented in our prototype tool (see Section 5.3), there is room for further improvement. More details on the results computed by each tool can be found on [loob].

5.2.3 Evaluation on Challenging Iteration Patterns from Real Code

Scanning through two C-code benchmarks (*cBench* [cbe] and *SPEC CPU 2006* [spe]), we found a number of 23 *different* loop iteration patterns which we consider to be particular challenging for state-of-the-art bound analyses. The 23 patterns have the following property in common: (1) There is an inner loop L with loop counter c, such that c is *increased* on an outer loop of L. (2) Nevertheless, the *amortized cost of* L (the overall worst-case cost of executing L, averaged over the number of executions of its outer loop) is lower than the worst-case cost of a *single* execution (a single instance of *consecutive* iterations) of L.

E.g., Example `xnu` (discussed in Section 1.4.7) is an instance of such a loop iteration pattern.

We ran the most recent version of `loopus`, available at [looa], on the benchmark.

The complete benchmark is available at [looc]. For each pattern we link its origin in the header of the respective file. Note that for some patterns we found several instances.

Table 5.5 states the results that were obtained by `loopus`, *CoFloCo*, *KoAT*, *Rank* and *C4B*:
'✓' denotes that the bound computed by the respective tool is *tight* (in the same asymptotic class, Definition 14),
'$\mathbf{O(n^x)}$' denotes that the respective tool did not infer a *tight* bound but a bound in the asymptotic class $\mathbf{O(n^x)}$,
'✗' denotes that no bound was inferred,

'**TO**' denotes that the tool timed out (the time out limit was 20 minutes, a longer time out did not yield additional results),
'◯' denotes that we were not able to translate the example into the input format of the tool.
We determined the asymptotic complexity (on base of the back-edge metric) for each file by hand, it is annotated behind the filename in Table 5.5.

We explain the last 5 rows of Table 5.5:
Total Tight states the number of examples for which the respective tool inferred a *tight* bound (see Definition 14).
Total Overapprox. states the number of examples for which the respective tool inferred a bound that is *not* tight.
Total Fail states the number of examples for which the respective tool did not report a bound, but returned within the time out limit of 20 minutes.
Total Timed Out states the total number of examples on which the respective tool timed out (the time out limit was 20 minutes).
Total Time states the overall time consumed by the respective tool for processing the complete benchmark.
Total Time w/o TO states the overall time consumed by the respective tool on those examples on which the tool did not time out.

`loopus` fails to infer a *tight* bound only for `Configure.c` and `analyse_other.c`. For both files the reason is that our improvement for obtaining *more precise variable bounds* (discussed in Section 4.2.5) is not yet implemented in our tool. Our analysis, as presented in this work, is capable of inferring tight bounds for both files.

`loopus` is far more successful in inferring tight bounds for the examples than any of the competitors. `loopus` infers 21, *Rank* 7, *C4B* 6, *CoFloCo* 6 and *KoAT* 2 tight bounds.

There are 9 examples for which *only* `loopus` infers a tight bound:
`cf_decode_eol.c`, `PackBitsEncode.c`, `s_SFD_process.c`, `send_tree.c`, `subsetdump.c`, `ParseFile.c`, `SingleLinkCluster.c`, `xdr3dfcoord.c`, and `XNU.c`.

The experiment demonstrates, that our bound analysis complements the state-of-the-art, by inferring tight bounds for a class of real-world loop iterations, on which existing techniques mostly fail or obtain coarse over-approximations.

Technical remarks. (1) We counted the time needed by the tool *Aspic* (a pre-processor for *Rank* which performs *invariant generation*) into the time of the bound analysis performed by *Rank*. (2) For the examples `s_SFD_process.c`, `load_mems.c` and `SingelLinkCluster.c`, *Rank* reported an unsound bound and the error message ("`count_points: ? infinite domain`"). On these examples we therefore assessed *Rank*'s return value as *fail*.

		loopus	CoFloCo	KoAT	Rank	C4B
cf_decode_eol.c	$O(n)$	✓	✗	$\mathbf{O(n^2)}$	✗	✗
cryptRandWriteFile.c	$O(n)$	✓	✓	$\mathbf{O(n^2)}$	✓	✓
encode_mcu_AC_refine.c	$O(n)$	✓	✗	$\mathbf{O(n^2)}$	✓	✗
hc_compute.c	$O(n^2)$	✓	$\mathbf{O(n^3)}$	**TO**	✓	✗
inflated_stored.c	$O(n)$	✓	✓	✗	✗	✗
PackBitsEncode.c	$O(n)$	✓	**TO**	$\mathbf{O(n^2)}$	◯	✗
s_SFD_process.c	$O(n)$	✓	✗	$\mathbf{O(n^2)}$	✗	✗
send_tree.c	$O(n)$	✓	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	✗
sendMTFValues.c	$O(n)$	✓	$\mathbf{O(n^2)}$	✓	$\mathbf{O(n^2)}$	✓
set_color_ht.c	$O(n^2)$	✓	✓	$\mathbf{O(n^3)}$	✗	✗
subsetdump.c	$O(n)$	✓	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	✗	✗
zwritehexstring_at.c	$O(n)$	✓	✓	$\mathbf{O(n^2)}$	✓	✓
analyse_other.c	$O(n^3)$	$\mathbf{O(n^4)}$	✗	$\mathbf{O(n^{13})}$	✗	✗
ApplyBndRobin.c	$O(n^4)$	✓	✗	✓	◯	◯
asctoeg.c	$O(n^2)$	✓	✓	$\mathbf{O(n^3)}$	✓	✗
Configure.c	$O(n)$	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	✓	✗
load_mems.c	$O(n)$	✓	✗	$\mathbf{O(n^3)}$	✗	✓
local_alloc.c	$O(n)$	✓	✓	$\mathbf{O(n^2)}$	✓	✓
ParseFile.c	$O(n)$	✓	**TO**	$\mathbf{O(n^3)}$	◯	✗
Perl_scan_vstring.c	$O(n)$	✓	✗	$\mathbf{O(n^2)}$	✗	✓
SingleLinkCluster.c	$O(n^2)$	✓	✗	**TO**	✗	✗
xdr3dfcoord.c	$O(n)$	✓	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	◯	✗
XNU.c	$O(n)$	✓	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$	✗	✗
Total Tight		*21*	*6*	*2*	*7*	*6*
Total Overapprox.		*2*	*7*	*18*	*2*	*0*
Total Fail		*0*	*8*	*1*	*10*	*16*
Total Timed Out		*0*	*2*	*2*	*0*	*0*
Total Time		*5s*	*41m16s*	*74m31s*	*28s*	*19m55s*
Total Time w/o TO		*5s*	*1m16s*	*34m31s*	*28s*	*19m55s*

Table 5.5: Tool Results on 23 challenging loop iteration patterns from *cBench* and *SPEC CPU 2006* Benchmarks. The time out was 20 minutes, a longer time out did not yield additional results.

5.3 Limitations of Our Implementation

The following features of our analysis are not yet implemented into our prototype tool:

- Control-flow refinement by *unfolding* (Section 4.1.3)
- Bounds involving *integer division* (Section 4.2.3)
- Bounds involving *logarithm* (Section 2.2.1)
- Enhanced precision for *variable bounds* (Section 4.2.5)
- Back-tracking for choosing an alternative *local bound mapping* in case of failure (Section 5.1 and Section 3.6)
- Optimization in presence of *multiple transitions with the same local bound* (Section 4.2.2)

Further, Control-flow refinement by *contextualization* (Section 4.1.3) is not completely implemented and currently applied very limited.

Our tool can currently only process *reducible control flow*. As discussed, this is not a general limitation of our approach.

5.3.1 Reasons for Failure

Even if the heuristics we discussed in Section 5.1.1 are activated, our tool fails to infer a bound in certain cases. We report on numbers from an experimental run of our tool on the cBench benchmark [cbe]. In order to facilitate the manual inspection of the errors, we instructed the tool to compute loop bounds in terms of the variables defined at the respective SCC header. In total the tool failed to infer bounds for 1005 loops out of the 4210 loops in the benchmark (after removing code duplicates).

(1) No local bound.
For 903 loops our analysis failed because there was a back-edge (a transition which implements a back jump) for which no local bound could be inferred. We inspected a random sample of 50 loops out of the 903 loops. On these 50 samples, the failures turned out to be due to insufficient modeling features of our implementation: bitwise operations (not modeled, 5 cases), function inlining (applied very restrictive, 4 cases), unsigned integers (modeled as integers, 2 cases), external functions (not modeled, 4 cases). In 5 cases termination is conditional and cannot be proven (e.g., a character stream is assumed to contain the line-break character). In 7 cases function pointers need to be resolved. In 27 cases invariant analysis is needed (7 array invariants, 20 arithmetic invariants).

(2) Forming a lexicographic ranking function failed.
Recall that proving termination by ordering the local bounds to a lexicographic ranking function is a precondition for the success of our bound algorithm (see discussion on

termination of our algorithm in Section 3.2, page 42). As discussed, this condition is *explicitly* checked in our implementation before applying our bound algorithm. For a total of 64 loops our analysis found a local bound for every back-edge but was not able to form a lexicographic ranking function. However, a manual inspection of the 64 loops revealed that only in 4 cases our technique was insufficient to show termination. For the other 60 loops the real reason for failure is that our greedy implementation chose the wrong local bound for a transition (for the same reasons discussed in (1)); the failure was thus caused by variables that were actually not relevant for the termination of the loop. Our implementation could be extended by back-tracking such that an alternative local bound is tried in case of failure. This would allow to handle these 60 cases. In the 4 remaining cases a more sophisticated implementation of control-flow refinement by *contextualization* (Section 4.1.3) would allow to obtain a bound by our technique.

(3) Failure of $V\mathcal{B}(\mathtt{v})$.
For the 38 remaining loops (1005 - 967) the main reason for not being able to compute a bound was that our variable bound method $V\mathcal{B}(\mathtt{v})$ failed to infer a bound on the value of some *reset*: E.g., a variable x is reset to some non-constant expression e and our abstraction techniques in combination with our variable bound method failed to infer an upper bound on e. This typically happened due to a missing heap invariant (e.g., all array elements are smaller than n).

We conclude that, in the case of our experimental run, the reasons of failure were rather due to technical reasons than due to general limitations of our analysis.

CHAPTER 6

Conclusion

In Section 1.7 we announced five key contributions to the field of automated complexity and resource bound analysis. Based on the presented material, we now give a detailed discussion on each of our contributions.

1. We have demonstrated that *difference constraints* are a suitable abstract program model for automatic complexity and resource bound analysis. Despite their syntactic simplicity, difference constraints are expressive enough to model the complexity-related aspects of many imperative programs. In particular, difference constraints allow to model *amortized complexity* problems such as the bound analysis challenge posed by Example `xnu`(discussed in Section 1.4.7). We developed appropriate techniques for abstracting imperative programs to *DCPs* (Chapter 2): We described how to extract *norms* (integer-valued expressions over the program state) from imperative programs and showed how to use these norms as variables in *DCPs*.

We think that the abstract program model of difference constraint programs is worth further investigation: Given that difference constraints can model standard counter manipulations (counter increments, decrements and resets), a further research on complexity analysis of difference constraint programs is of high value. We consider *DCP*s to be a very suitable program model for studying the principle challenges of automated complexity and resource bound analysis for imperative programs.

2. We presented a new approach to bound analysis and automatic complexity analysis. Our approach complements existing approaches in several aspects. (We draw a detailed comparison to the state-of-the-art in bound analysis in Section 6.4.) Our analysis handles bound analysis problems of high practical relevance which current approaches cannot handle: Current techniques [SZV14a, BEF$^+$16, CHS15, FH14] fail on Figure 2.1 and similar problems. We have argued that such problems, e.g., occur naturally in parsing and string matching routines. During our experiments on real-world source code, we found 23 different iteration patterns that pose a challenge for similar reasons as Example `xnu`:

In these patterns, the worst-case cost of a single inner loop execution is lower than the worst-case cost of the inner loop averaged over the iterations of the outer loop. Our analysis (Chapter 3) can handle all of these iteration patterns.

3. We stress that our approach is more *scalable* than existing approaches. We presented empirical evidence of the good performance characteristics of our analysis by a large experiment and tool comparison on real source code in Section 5.2.1. We discuss the main technical reasons for scalability of our analysis in Section 6.1.

4. Our approach deals with many challenges bound analysis is known to be confronted: In Section 5.2.2 we compared our tool on a benchmark of challenging problems from publications on bound analysis. The results show that our prototype implementation can handle most of these problems. Here, our implementation, while comparable in terms of strengths to other implementations of state-of-the-art bound analysis techniques, performs the task significantly faster than the competitors. The results obtained by our prototype tool could be further enhanced by completing the implementation (see Section 5.3).

5. In Chapter 7 we prove *soundness* of our bound algorithm (Section 7.1), of the reasoning on reset chains (Section 7.2), and of the path-sensitive reasoning (Section 7.3). Note that the sets $\mathcal{I}(\mathtt{v})$ and $\mathcal{R}(\mathtt{v})$ capture *simple local properties*, it is therefore straightforward to convince oneself of their correct statement in the computation tables (e.g., Table 3.3, page 50). The same applies for the local bound mapping ζ.

6.1 Discussion on the Scalability of Our Approach

Having completed the technical discussion of our analysis (Chapter 2 and Chapter 3), we have gathered the necessary details to reflect the performance characteristics of our approach. In the following we state what we consider to be the main technical reasons that make our analysis scale.

First of all, we achieve scalability by *local* reasoning: Note that our abstraction procedure relies on purely *local information*, i.e., information that is available on *single* program transitions. In particular, we do not apply any *global invariant analysis*. Further, the sets $\mathcal{I}(\mathtt{v})$ and $\mathcal{R}(\mathtt{v})$, by which our main algorithm is parametrized, are built by sorting the difference constraints on *single* (abstract) program transitions based on simple syntactic criteria. Our algorithm for computing the mapping ζ (Section 3.6) is *linear* and remains *polynomial* even in the generalized case (Section 4.2.1).

We use bound analysis to infer *bounds on variable values* (variable bounds). Unlike classical invariant analysis this approach is *demand-driven* and does *not* perform a *fixed point* iteration (see Discussion in Section 3.4).

Note that the only general purpose reasoner we employ is an SMT solver. Further, the SMT solver is only employed in the program abstraction phase. In terms of size, the problems we feed to the SMT solver are *small*, namely simple linear arithmetic

formulas, composed of the arithmetic of single transitions. Our approach instruments the SMT solver only for *yes/no* answers, no *optimal* solution (e.g., minimum or minimal unsatisfiable core) is required.

Our main algorithm (Definition 27) runs in *polynomial time*. The reasoning on reset chains (Definition 31) and the path-sensitive reasoning (Definition 40), however, have *exponential* worst-case complexity. We did not experience this to be an issue in practice. In both cases, exponential worst-case complexity results from the potentially exponential number of paths in the program (exponential in the number of program transitions). Thanks to the *simplicity* of our abstract program model straightforward engineering measures can be taken: Program slicing reduces the number of paths in the program *significantly*, further, *merging* of similar paths or transitions can be applied (Section 5.1.2).

6.2 Reflection on Research Methodology

We want to stress that our research methodology (Section 1.3) has proven beneficial in the development of the presented ideas: In our previous work [ZGSV11] we identified the size change abstraction (SCA) [LJBA01] as a suitable program model for bound analysis. We implemented a bound analysis based on SCA into our tool `loopus`. By extending our experiments to a larger code base we identified *amortized complexity analysis* as a relevant task for computing *precise* bounds for many C implementations. Our further research led to *vector additon systems* (VASS) as an abstract program model. In [SZV14a] we presented an algorithm for analyzing amortized program complexity using VASS as abstract program model. Further experiments on open source C code yielded a number of interesting examples which we were not able to model as VASS. Some of these examples were discussed in this work. Our findings revealed that an extension of our abstract program model is necessary to handle certain iteration patterns that are common in C code. In [SZV15] we generalized our VASS based approach to *difference constraint programs*. Our approach is now able to obtain tight bounds for a large class of real-world C programs, as we have discussed in this work.

6.3 Open Issues and Future work

Combining our *variable bound analysis* with a symmetrical *lower bound analysis* would result in a new kind of scalable *invariant analysis* which, importantly, could infer *polynomial* relations between program variables, a problem not tackled by standard abstract interpretation techniques. We plan to investigate the feasibility of this idea.

The bounds that are inferred by our approach are not guaranteed to be *precise*, meaning that it is not guaranteed that a program execution exists which reaches the respective bound. This is a general problem. Existing bound analysis techniques are either limited to very specific cases like for-loops with only one loop counter, or cannot give any guarantee on the precision of the inferred bounds. Naturally, bound analysis explicitly or implicitly *abstracts* from the concrete program behaviour in order to deal with a generally

intractable problem. However, observe that the computation trace of our algorithm (depicted in *computation tables*, e.g., Table 3.3, page 50) *does* provide information on how the transitions of the programs have to be executed in order to reach the inferred bound. We plan to automatically check if the program execution implicitly underlying the inferred bound is feasible in practice. In case the corresponding execution turns out to be infeasible, additional constraints could be taken into account for a re-computation which will then lead to a more precise bound. This idea could result in a CEGAR [CGJ+00] like approach for inferring *precise (resource) bounds*. A similar idea was recently proposed for WCET analysis [ČHK+15].

Our approach does not handle exponential loop bounds. Exponential loop bounds are, however, very rare in practice. Nevertheless an extension of our approach to cases as the one shown in Figure 3.3 (a) (page 46) is of interest: Since *termination* is decidable for *fan-in free DCP*s ([Ben08]), it is an interesting question whether there is a complete algorithm for *bound analysis* of fan-in free *DCPs*.

Given that loops iterating over data structures such as lists or trees are omnipresent in imperative code, the *sound* handling of recursive data structures and pointers for bound analysis is a challenging, but highly relevant task.

We further plan to investigate the applicability of our approach for the analysis of *functional* languages. Here an extension to *recursive programs* will be essential. Though tail-recursion can be handled, and many functions with a single recursive call turn out to be *tail-recursive* after *program slicing*, our approach cannot yet handle functions with *multiple recursive calls* (multiple recursion) such as *divide and conquer algorithms* (e.g., *quick sort*). In the literature a number of approaches dealing with such recursion patterns exist (e.g., [ZZ89, HAH12, AGM13]), and it is of interest to formulate these ideas within our framework. Our abstract program model could, e.g., be extended by a stack in order to model recursion directly.

The multi-core architecture of modern processors has given momentum to the investigation of *concurrency*. Bound analysis techniques for *concurrent programs* are therefore an interesting project for future work.

In many scenarios, the *average* resource consumption, rather than the worst-case consumption, will be of interest. Consider, e.g., a cloud provider seeking to estimate how much computation power a new application will consume *in average*. An answer to such kind of questions could be provided by *probabilistic bound analysis*, i.e., bound analysis techniques that take into account the probability by which program transitions are executed. We believe that our algorithm can be extended by such a reasoning.

Another interesting question is how the *abduction principle*, successfully applied in the area of termination analysis (e.g., [DU15]), can be made productive for bound analysis.

6.4 Detailed Comparison to Related Work

We presented an overview on the state of the art in bound analysis in Section 1.6. Recall that current approaches to bound analysis apply general frameworks for *global invariant analysis* and *automated reasoning* such as **abstract interpretation**, **computer algebra**, **linear optimization** and **quantifier elimination**. In contrast, our bound analysis is based on simple *static analysis* and *local reasoning*. In Section 6.1 we discussed the resulting advantages in performance and scalability, empirical evidence of which was given in Section 5.

Based on the complete presentation of our analysis, we are now ready to state a detailed comparison of our approach to the most related works [ZGSV11, SZV14a, SZV15, FH14, BEF+16, CHS15, ADFG10]. Naturally, this discussion also involves technical details.

In the following, we point out specific weaknesses or strengths of the respective bound analysis in comparison to our approach. Rather than on performance, our discussion focuses on functional aspects.

6.4.1 Our Previous Approaches

The bound analysis [ZGSV11] is based on the *size change abstraction*. Size-change constraints form a strict syntactic subclass of *difference constraints* which are at the core of the analysis we presented in this work. Whereas for *DCP*s termination is *undecidable* in general and *decidable* for the sub-class of *fan-in free DCP*s [Ben08], termination is decidable even for size-change programs which are not *fan-in free* and a complete algorithm for deciding the complexity of size-change programs has been developed [CDZ14].
For reasoning about *inner loops* [ZGSV11], computes *disjunctive loop summaries*, while such summaries are not computed by the approach discussed in this work. We have demonstrated by means of numerous examples that our approach nevertheless reasons about the interaction of outer and inner loops very effectively.

Our previous work [SZV14a] presents a bound analysis that uses *vector addition systems* as *abstract program model*. We formulated *vector addition systems* as a syntactic sub-class of our abstract program model of *difference constraint programs* in Section 3.2.
Our path-sensitive reasoning (Definition 36, page 68) can be understood as a re-formulation of the algorithm discussed in [SZV14a]: By defining $\texttt{Incr}(\texttt{v})$ over the cyclic paths $\texttt{C}^+(\texttt{v})$ rather than over the transitions $\mathcal{I}(\texttt{v})$, Definition 36 implements *control-flow abstraction* introduced in [SZV14a]. Our *local bounds* play the role of *local ranking functions* in [SZV14a]: If $\texttt{v} \in \mathcal{V}$ is a path sensitive local bound for $\tau \in E$, then $\texttt{v}$ is a *local ranking function* for all cyclic paths on which τ is situated (for all $\pi \in C(\tau)$).
The *lexicographic ranking function* that is explicitly computed in [SZV14a] is implicitly underlying Definition 36: As discussed in Section 3.2 (see paragraph on *termination*), our bound algorithm terminates only if the local bounds can be ordered to a *lexicographic ranking function*. We can thus run Algorithm 2 from [SZV14a] up-front for ensuring termination of Definition 36.

[SZV14a] defines the notion of a *path bound*, the bounds that are computed by [SZV14a] are bounds on the number of times a given cyclic path can be executed during program run. In contrast, we compute *transition bounds*. In Definition 36 a path bound is computed in terms of transition bounds: We compute the minimum over the bounds for all transitions which lie on the path. Computing path bounds in terms of transition bounds allows to handle *irreducible control flow*, which cannot be handled by [SZV14a]. [SZV14a] is limited to *reducible control-flow*. The approach we presented in this thesis does not have this restriction.

We crucially extended our *DCP* based bound algorithm from [SZV15] by *path-sensitive* reasoning (Section 3.8). The approach we presented in this work merges the insights of [SZV14a] and [SZV15], resulting in one uniform analysis that joins both forces and is therefore more powerful than [SZV15] and [SZV14a]: [SZV14a] in particular fails to infer the *linear* bound for Example `xnuSimple` (Figure 3.8, page 55) and for Example `xnu` (Figure 1.4, page 14). On the other hand, [SZV15] does not support *path sensitive* analysis and therefore computes an imprecise bound for Example `s_SFD_process` (Figure 3.14, page 66).

6.4.2 Other Related Approaches

In Section 5.2 we experimentally compared our implementation against the recent approaches to automated complexity analysis [FH14, BEF+16, CHS15, ADFG10].

[BEF+16] is, in one aspect, conceptually similar to our approach: While our algorithm is built over a mutual recursion between transition bound and variable bound analysis, [BEF+16] interleaves *time bound* and *variable size bound* analysis. A similar idea is also present in the works [GJ09] and [SZV14b] (the extended version of [SZV14a]).
In contrast to our approach, [BEF+16] computes upper bound invariants only for the *absolute* values of variables; for many cases, this does not allow to distinguish between variable increments and decrements: Consider the program `foo(int x, int y) {while(y > 0) {x--; y--;} while(x > 0) x--;}`. The algorithm described in [BEF+16] infers the bound $|x| + |y|$ for the second loop, whereas our analysis infers the bound $\max(x, 0)$.
[BEF+16] *depends* on *global invariant analysis*. E.g., given a decrement $x := x - 1$, [BEF+16] needs to check whether $x \geq 0$ holds. If $x \geq 0$ cannot be ensured, the *decrement* can actually *increment* the *absolute value* of x, and will thus be interpreted as $|x| = |x|+1$. This can either lead to gross over-approximations or failure of bound computation if the increment of $|x|$ cannot be bounded. Since our approach does not track the *absolute* value but the value, it is not concerned with this problem.
[BEF+16] does not support *amortized analysis*: E.g., [BEF+16] fails to compute the *linear bounds* for Example `tarjan`, Example `xnuSimple`, Example `xnu`, the inner while loop of Example `SingleLinkCluster`, Example `s_SFD_process` and other examples we discussed in this work.
[BEF+16] can infer bounds for functions with multiple recursive calls which is not supported by our approach.

```
   void foo(uint n) {
      uint k = n, j = 0;
      while(k > 0) {
        int i = n;
        while (i > 0) {
           i--;
           j++;
l3         while (j > 0 && ?)
             j--;
        }
        k--;
      }
   }
```

Bound of the loop at l_3: n^2

(a)

```
   void foo(uint n) {
     int i = n, k = 0;
     while(i > 0) {
       int j = n;
l2     while(j > 0) {
         k++;
         j--;
       }
       i--;
     }
l3   while(k > 0) {
       k--;
     }
   }
```

Bound of the loop at l_3: n^2

(b)

Figure 6.1: (a) Example `tarjan` (Figure 1.2, page 8) with an additional outer loop (loop counter k), (b) Invariant $k \leq n^2$ is not expressible in *polyhedra abstract domain*

[CHS15] is restricted to linear bounds, while our approach derives bounds which are polynomial (e.g., the bound n^2 of the loop at l_3 in Figure 6.1 (a) and (b)).
Further, [CHS15] cannot infer bounds involving the *maximum* operator, [CHS15] thus over-approximates the bound $\max(m1, m2)+2n$ of the second loop of Example `twoSCCs` (Figure 1.3, page 11) by $m1 + m2 + 2n$.
[CHS15] fails to deduce the linear bound of Example `xnu` (discussed in Section 1.4.7). The reason is that [CHS15] needs the invariant $k \leq e \leq i$ in combination with the *Q:WEAK*-rule (see [CHS15]) in order to find a bound for Example `xnu`. However, for efficiency reasons [CHS15] does not apply any global invariant analysis and for the same reason the *Q:WEAK*-rule ("*Q:WEAK* is not syntax directed." [CHS15]) is only applied heuristically. In contrast, our approach infers the linear bound for Example `xnu` without relying on invariant analysis (see Section 3.7).
Unlike our analysis, [CHS15] does support general recursion, including *mutual* recursion. Another strength of [CHS15] is that its reasoning includes *decrements* of variables also across different loops. E.g., [CHS15] obtains the bound n for the function
`foo(uint n) {while(n > 0 && ?)  n--; while(n > 0 && ?)  n--;}`
whereas our approach infers the bound $2n$ for this example.

Recall our discussion on [ADFG10] from Section 1.6. For a number of cases the approximation of the reachable states in shape of a polyhedron does not allow to conclude a bound on the number of values in the co-domain of the ranking function: As an example consider Figure 6.1 (a), which shows a version of Example `tarjan` (Figure 1.2, page 8)

with an additional outer loop. For the innermost loop at location l_3, a linear ranking function with minimum number of dimensions is $\langle k, 2i + j\rangle$. This ranking function can indeed be inferred by the approach discussed in [ADFG10]. However, for inferring a bound on the size of the co-domain of $\langle k, 2i + j\rangle$ in particular a bound on the value of j is needed. Note that any ranking function for l will include j. However, j can take values up to n^2, but $j \leq n^2$ is not expressible in the polyhedra domain (which only expresses arbitrary *linear* relations). [ADFG10] therefore fails to obtain a bound for the example in Figure 6.1 (a). In contrast, our approach obtains the precise bound n^2 for the loop at l_3.

[FH14] cannot infer the bound n^2 for the loop at l_3 of the program in Figure 6.1 (b): First of all, note that the loop at l_3 indeed can be executed up to n^2 times because its counter k can be incremented up to n^2 times in the loop at l_2. [FH14] infers that the loop at l_3 can be executed k_3 times, where k_3 is k at location l_3. [FH14], however, fails to infer that $k_3 < n^2$ because this relation is not expressible in the *polyhedra abstract domain*. [FH14] also fails to infer the *linear* bound of Example `xnu` in Figure 1.4.

CHAPTER 7

Proofs

We prove soundness of our basic bound algorithm for *DCP*s (Definition 27, Theorem 3) in Section 7.1.

In Section 7.2 we prove soundness of our reasoning on reset chains (Definition 31, Theorem 4).

Finally, we prove soundness of our path-sensitive reasoning (Definition 40, Theorem 7) in Section 7.3.

Throughout this chapter we assume a *well-defined* and *fan-in free DCP* $\Delta\mathcal{P}(L, E, l_b, l_e)$ over $\mathcal{A}$ to be given.

We first define some basic notions which we use to state our proofs precisely.

Definition 50 (Indices)**.** *Let $\pi = l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \dots$ be a path of $\Delta\mathcal{P}$. By $len(\pi)$ we denote the* length *of π, i.e., the total number of transitions on π (possibly ∞). Let $0 \leq i \leq j$. By $\pi_{[i,j]}$ we denote the sub-path of π that starts at l_i and ends at l_j. By $\pi(i) = l_i \xrightarrow{u_i} l_{i+1}$ we denote the $(i+1)$th transition on π.*
Let $\tau \in E$. We define $\Theta(\tau, \pi) = \{0 \leq i < len(\pi) \mid \pi(i) = \tau\}$.
Let $E' \subseteq E$. We define $\Theta(E', \pi) = \bigcup_{\tau \in E'} \Theta(\tau, \pi)$.
We write
$\Theta(\mathcal{R}(\mathtt{v}), \pi)$ to denote $\Theta(\{\tau \mid (\tau, \_, \_) \in \mathcal{R}(\mathtt{v})\}, \pi)$, and
$\Theta(\mathcal{I}(\mathtt{v}), \pi)$ to denote $\Theta(\{\tau \mid (\tau, \_) \in \mathcal{I}(\mathtt{v})\}, \pi)$.
We use the same notation for runs ρ of $\Delta\mathcal{P}$.

I.e., $\Theta(\tau, \pi)$ is the set of all indices of τ on π, $\Theta(\mathcal{R}(\mathtt{v}), \pi)$ is the set of indices of all transitions on π which reset $\mathtt{v}$ and $\Theta(\mathcal{I}(\mathtt{v}), \pi)$ is the set of indices of all transitions on π which increment $\mathtt{v}$.

On a run of $\Delta\mathcal{P}$ a variable $\mathtt{v}$ may take arbitrary values at locations at which $\mathtt{v}$ is not defined, i.e., at locations l with $\mathtt{v} \notin \mathtt{def}(l)$. In a well-defined *DCP* the value of a variable at a location where it is not defined can, however, not affect the programs behaviour. This observation motivates the notion of a *normalized run*: a normalized run is a run on which a variable takes value "0" at locations where it is not defined.

Definition 51 (Normalized Run)**.** *Let* $\rho = (l_0, \sigma_0) \xrightarrow{u_0} (l_1, \sigma_1) \xrightarrow{u_1} \cdots$ *be a run of* $\Delta\mathcal{P}$*. Let*

$$\sigma'_i(\mathtt{a}) = \begin{cases} 0 & \text{if } \mathtt{a} \in \mathcal{V} \text{ and } \mathtt{a} \notin \mathtt{def}(l_i) \\ \sigma_i(\mathtt{a}) & \text{else} \end{cases} \quad \text{for all } 0 \leq i \leq len(\rho) \text{ and all } \mathtt{a} \in \mathcal{A}.$$

We call $\lfloor \rho \rfloor = (l_0, \sigma'_0) \xrightarrow{u_0} (l_1, \sigma'_1) \xrightarrow{u_1} \cdots$ *a* normalized run.

Let Ξ *be a set of runs of* $\Delta\mathcal{P}$*. We say that* Ξ *is closed under* normalization *if* $\rho \in \Xi$ *implies that* $\lfloor \rho \rfloor \in \Xi$.

Lemma 1 states that the set of all runs of $\Delta\mathcal{P}$ is closed under normalization.

Lemma 1. *Let* ρ *be a run of* $\Delta\mathcal{P}$*. Then* $\lfloor \rho \rfloor$ *is a run of* $\Delta\mathcal{P}$*.*

Proof. Follows directly from Definition 6 (well-definedness) and Definition 51. □

7.1 Soundness of Basic Bound Algorithm

In Lemma 2 and Lemma 3 we formulate the two key insights on which our algorithm is based.

Lemma 2 formalizes the intuition given in Section 3.12: Let $\mathtt{v}$ be a local transition bound for τ. The question how often τ can appear on a run ρ is translated to the question how often the transitions which increase the value of $\mathtt{v}$ (i.e., $(t, \_) \in \mathcal{I}(\mathtt{v})$ and $(t, \_, \_) \in \mathcal{R}(\mathtt{v})$) can appear on ρ.

Lemma 2. *Let* ρ *be a run of* $\Delta\mathcal{P}$*. Let* $\tau \in E$*. Let* $\mathtt{v} \in \mathcal{V}$ *be a* local transition bound *for* τ *on* $\lfloor \rho \rfloor$*. Let* $vb : \mathcal{A} \to \mathbb{Z}$ *be s.t.* $vb(\mathtt{a})$ *is a* variable bound *for* $\mathtt{a}$ *on* ρ *for all* $(\_, \mathtt{a}, \_) \in \mathcal{R}(\mathtt{v})$*. Then*

$$\left(\sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} \sharp(t, \rho) \times \mathtt{c} \right) + \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(t, \rho) \times (vb(\mathtt{a}) + \mathtt{c})$$

is a transition bound *for* τ *on* ρ*.*

Proof. We first show that it is sufficient to consider the case $\lfloor \rho \rfloor = \rho$:
1. Let $\mathtt{expr}$ be a *transition bound* for τ on $\lfloor \rho \rfloor$. Then $\mathtt{expr}$ is also a *transition bound* for τ on ρ (follows directly from Definition 51).
2. By assumption $vb(\mathtt{a})$ is a *variable bound* for $\mathtt{a}$ on ρ. By Definition 51 we have that $vb(\mathtt{a})$ is also a *variable bound* for $\mathtt{a}$ on $\lfloor \rho \rfloor$. We thus assume that $\lfloor \rho \rfloor = \rho$.

We have to show that
$$\sharp(\tau,\rho) \leq \left(\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(t,\rho)\times \mathtt{c}\right) + \sum_{(t,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})} \sharp(t,\rho)\times(vb(\mathtt{a})+\mathtt{c}).$$

A) We first show that

$$\sharp(\tau,\rho) \leq \left(\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(t,\rho)\times \mathtt{c}\right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

We have

$$\begin{aligned}
\sharp(\tau,\rho) &\overset{(1)}{\leq} \sharp(\tau,\rho) + \sum_{i=0}^{len(\rho)-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \\
&\overset{(2a)}{=} \sharp(\tau,\rho) + \sum_{i=0}^{len(\rho)-1} \max(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0) + \sum_{i=0}^{len(\rho)-1} \min(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0) \\
&\overset{(2)}{\leq} \sum_{i=0}^{len(\rho)-1} \max(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0) \\
&\overset{(3a)}{=} \left(\sum_{i\in\Theta(\mathcal{I}(\mathtt{v}),\rho)} \max(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0)\right) + \sum_{i\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \max(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0) \\
&\overset{(3)}{\leq} \left(\sum_{i\in\Theta(\mathcal{I}(\mathtt{v}),\rho)} \max(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0)\right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v}) \\
&\overset{(4)}{\leq} \left(\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sum_{0\leq i<len(\rho) \text{ s.t. } \rho(i)=t} \mathtt{c}\right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v}) \\
&\overset{(5)}{=} \left(\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(t,\rho)\times \mathtt{c}\right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})
\end{aligned}$$

(1) We have $\sum_{i=0}^{len(\rho)-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) = \sigma_{len(\rho)}(\mathtt{v}) - \sigma_0(\mathtt{v}) = \sigma_{len(\rho)}(\mathtt{v})$
because $\sigma_0(\mathtt{v}) = 0$ with i) $\rho = \lfloor\rho\rfloor$ and ii) $\mathtt{v} \notin \mathtt{def}(l_b)$ (Definition 6).
Trivially $\sigma_{len(\rho)}(\mathtt{v}) \geq 0$. Therefore $\sum_{i=0}^{len(\rho)-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \geq 0$.

(2a) Case Distinction

(2) We have $\sharp(\tau,\rho) \leq \downarrow(\mathtt{v},\rho)$ (Definition 16).
Further $\downarrow(\mathtt{v},\tau) \leq \left(\sum_{i=0}^{len(\rho)-1} \min(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0)\right) \times -1$.
Thus $\sharp(\tau,\rho) + \sum_{i=0}^{len(\rho)-1} \min(\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}),0) \leq 0$.

(3a) $\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) > 0$ implies in particular that $\sigma_{i+1}(\mathtt{v}) > 0$. Thus $\mathtt{v} \in \mathtt{def}(l_{i+1})$ because $\rho = \lfloor \rho \rfloor$ by assumption. With $\sigma_{i+1}(\mathtt{v}) > \sigma_i(\mathtt{v})$ we have that either:
Case 1) $(\rho(i), \_) \in \mathcal{I}(\mathtt{v})$, i.e., $i \in \Theta(\mathcal{I}(\mathtt{v}), \rho)$, or
Case 2) $(\rho(i), \_, \_) \in \mathcal{R}(\mathtt{v})$, i.e., $i \in \Theta(\mathcal{R}(\mathtt{v}), \rho)$.

(3) Since $\sigma_i(\mathtt{v}) \geq 0$ we have that $\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \leq \sigma_{i+1}(\mathtt{v})$.

(4) If $i \in \Theta(\mathcal{I}(\mathtt{v}), \rho)$ then there is $(t, \mathtt{c}) \in \mathtt{Incr}(\mathtt{v})$ s.t. $\rho(i) = t$ (Definition 50). Further $\sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \leq \mathtt{c}$ and $\mathtt{c} > 0$ (Definition 19).

(5) By definition of $\sharp(t, \rho)$ (Definition 15).

B) We show that $\sum_{j \in \Theta(\mathcal{R}(\mathtt{v}), \rho)} \sigma_{j+1}(\mathtt{v}) \leq \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(t, \rho) \times (vb(\mathtt{a}) + \mathtt{c})$:

$$\begin{aligned}
\sum_{j \in \Theta(\mathcal{R}(\mathtt{v}), \rho)} \sigma_{j+1}(\mathtt{v}) &\overset{(1)}{=} \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sum_{j \in \Theta(t,\rho)} \sigma_{j+1}(\mathtt{v}) \\
&\overset{(2)}{\leq} \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sum_{j \in \Theta(t,\rho)} \sigma_j(\mathtt{a}) + \mathtt{c} \\
&\overset{(3)}{\leq} \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sum_{j \in \Theta(t,\rho)} vb(\mathtt{a}) + \mathtt{c} \\
&\overset{(4)}{=} \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(t, \rho) \times (vb(\mathtt{a}) + \mathtt{c})
\end{aligned}$$

(1) By commutativity: Let $j \in \Theta(\mathcal{R}(\mathtt{v}), \rho)$. By the assumption that $\Delta\mathcal{P}$ is *fan-in free* there is only exactly one $\mathtt{a} \in \mathcal{A}$ and exactly one $\mathtt{c} \in \mathbb{Z}$ s.t. $(\rho(j), \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$.

(2) With $(\rho(j), \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ we have that $\sigma_{j+1}(\mathtt{v}) \leq \sigma_j(\mathtt{a}) + \mathtt{c}$ (Definition 19).

(3) Let $(t, \mathtt{a}, \_) \in \mathcal{R}(\mathtt{v})$. By assumption $vb(\mathtt{a})$ is a variable bound for $\mathtt{a}$ on ρ. Let $j \in \Theta(t, \rho)$. We have that $\mathtt{a} \in \mathtt{def}(l_j)$ by *well-definedness* of $\Delta\mathcal{P}$. Thus $\sigma_j(\mathtt{a}) \leq vb(\mathtt{a})$.

(4) Let $(t, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$. We have $\sum_{j \in \Theta(t,\rho)} vb(\mathtt{a}) + \mathtt{c} = |\Theta(t, \rho)| \times (vb(\mathtt{a}) + \mathtt{c})$.
Further $|\Theta(t, \rho)| = \sharp(t, \rho)$ (Definition 15).

With A) and B) we have

$$\sharp(\tau, \rho) \leq \left(\sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} \sharp(t, \rho) \times \mathtt{c} \right) + \sum_{(t,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(t, \rho) \times (vb(\mathtt{a}) + \mathtt{c}).$$

□

Lemma 3 states that the value of a variable $\mathtt{v} \in \mathcal{V}$ on a run ρ of $\Delta\mathcal{P}$ is limited by the maximum over all values to which $\mathtt{v}$ is reset on ρ plus the total amount by which $\mathtt{v}$ is incremented on ρ.

Lemma 3. *Let* $\mathtt{v} \in \mathcal{V}$. *Let* ρ *be a run of* $\Delta\mathcal{P}$. *Let* $vb : \mathcal{A} \to \mathbb{Z}$ *be s.t.* $vb(\mathtt{a})$ *is a* variable bound *for* $\mathtt{a}$ *on* ρ *for all* $(\_, \mathtt{a}, \_) \in \mathcal{R}(\mathtt{v})$. *Then*
$\max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(vb(\mathtt{a}) + \mathtt{c}) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c}$
is a variable bound *for* $\mathtt{v}$ *on* ρ.

Proof. We have to show that
$\sigma_i(\mathtt{v}) \leq \max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(vb(\mathtt{a}) + \mathtt{c}) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c}$
for all $0 \leq i \leq len(\rho)$ with $\mathtt{v} \in \mathtt{def}(l_i)$.

Let $0 \leq i \leq len(\rho)$ be s.t. $\mathtt{v} \in \mathtt{def}(l_i)$. By *well-definedness* of $\Delta\mathcal{P}$ there is a $0 \leq j < i$, a $b \in \mathcal{A}$ and a $c \in \mathbb{Z}$ s.t. $(\rho(j), b, c) \in \mathcal{R}(\mathtt{v})$ and $\mathtt{v}$ is not reset on $\rho_{[j+1,i]}$, i.e., for all $j < k < i$ $(\rho(k), \_, \_) \notin \mathcal{R}(\mathtt{v})$. In other words: there is a maximal index $j < i$ such that $\mathtt{v}$ is reset on $\rho(j)$. We have:

$$
\begin{aligned}
\sigma_i(\mathtt{v}) &\overset{(1)}{\leq} \sigma_{j+1}(\mathtt{v}) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho_{[j+1,i]}) \times \mathtt{c} \\
&\overset{(2)}{\leq} \sigma_{j+1}(\mathtt{v}) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c} \\
&\overset{(3)}{\leq} \sigma_j(b) + c + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c} \\
&\overset{(4)}{\leq} vb(b) + c + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c} \\
&\overset{(5)}{\leq} \max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(vb(\mathtt{a}) + \mathtt{c}) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho) \times \mathtt{c}
\end{aligned}
$$

(1) We have that $\mathtt{v}$ is not reset on $\rho_{[j+1,i]}$. If $\mathtt{v}$ is incremented on $\rho_{[j+1,i]}$ there are indices $j < k < i$ s.t. $(\rho(k), \_) \in \mathcal{I}(\mathtt{v})$. Let $(\tau, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$. An execution of τ can increase the value of $\mathtt{v}$ by at most $\mathtt{c}$ (Definition 19). Therefore the total number $\sharp(\tau, \rho_{[j+1,i]})$ of executions of τ on $\rho_{[j+1,i]}$ adds at most $\sharp(\tau, \rho_{[j+1,i]}) \times \mathtt{c}$ to $\mathtt{v}$. Thus in total $\mathtt{v}$ cannot be increased by more than $\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(\tau, \rho_{[j+1,i]}) \times \mathtt{c}$ on ρ.

(2) $\sharp(\tau, \rho_{[j+1,i]}) \leq \sharp(\tau, \rho)$. Further for all $(\_, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ $\mathtt{c} \geq 0$ (Definition 19).

(3) $\sigma_{j+1}(\mathtt{v}) \leq \sigma_j(b) + c$ (Definition 4).

(4) With $(\rho(j), b, c) \in \mathcal{R}(\mathtt{v})$ we have by assumption that $vb(b)$ is a variable bound for b on ρ. Further $b \in \mathtt{def}(l_j)$ by *well-definedness* of $\Delta\mathcal{P}$. Thus $\sigma_j(b) \leq vb(b)$.

(5) We have $(\rho(j), b, c) \in \mathcal{R}(\mathtt{v})$. Therefore $vb(b) + c \leq \max_{(\_,\mathtt{a},\mathtt{c})\in\mathcal{R}(\mathtt{v})}(vb(\mathtt{a}) + \mathtt{c})$.

□

7.1.1 Proof of Theorem 3

We show the more general claim formulated in Theorem 8.

Theorem 8. *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP over atoms $\mathcal{A}$. Let Ξ be a set of runs of $\Delta\mathcal{P}$ closed under* normalization. *Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for all $\rho \in \Xi$. Let $T\mathcal{B}$ and $V\mathcal{B}$ be defined as in Definition 27. Let $\mathtt{a} \in \mathcal{A}$ and $\tau \in E$. Let $\rho \in \Xi$. Let σ_0 be the initial state of ρ. We have: (I) $[\![T\mathcal{B}(\tau)]\!](\sigma_0)$ is a* transition bound *for τ on ρ. (II) $[\![V\mathcal{B}(\mathtt{a})]\!](\sigma_0)$ is a* variable bound *for $\mathtt{a}$ on ρ.*

Proof. Let $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \cdots \in \Xi$.

If $[\![T\mathcal{B}(\tau)]\!] = \infty$ (I) holds trivially. If $[\![V\mathcal{B}(\mathtt{a})]\!] = \infty$ (II) holds trivially.

Assume $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ and $[\![V\mathcal{B}(\mathtt{a})]\!] \neq \infty$. Then in particular the computations of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a})$ terminate. We proceed by induction over the call tree of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a})$.

Base Case:

(I) No function call is triggered when computing $V\mathcal{B}(\mathtt{a})$. This is the case iff $\mathtt{a} \in \mathcal{C}$ (Definition 27). Then $V\mathcal{B}(\mathtt{a}) = \mathtt{a}$ and the claim holds trivially with $\mathtt{a} \in \mathcal{C}$.
(II) No function call is triggered when computing $T\mathcal{B}(\tau)$. This is the case iff $\zeta(\tau) \notin \mathcal{V}$ (Definition 27). Then $[\![T\mathcal{B}(\tau)]\!](\sigma_0) = [\![\zeta(\tau)]\!](\sigma_0)$ is a *transition bound* for τ on ρ by Definition 18.

Step Case:

(I) $\mathtt{a} \notin \mathcal{C}$, thus $\mathtt{a} \in \mathcal{V}$. Let $\mathtt{v} = \mathtt{a}$. Let $0 \leq i \leq len(\rho)$ be s.t. $\mathtt{v} \in \mathtt{def}(l_i)$. We have:

$$
\begin{aligned}
\sigma_i(\mathtt{v}) &\overset{(1)}{\leq} \max_{(\_,b,\mathtt{c})\in\mathcal{R}(\mathtt{v})}([\![V\mathcal{B}(b)]\!](\sigma_0) + \mathtt{c}) + \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(2)}{\leq} \max_{(\_,b,\mathtt{c})\in\mathcal{R}(\mathtt{v})}([\![V\mathcal{B}(b)]\!](\sigma_0) + \mathtt{c}) + \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c} \\
&\overset{(3)}{=} [\![\max_{(\_,b,\mathtt{c})\in\mathcal{R}(\mathtt{v})}(V\mathcal{B}(b) + \mathtt{c})]\!](\sigma_0) + [\![\mathtt{Incr}(\mathtt{v})]\!](\sigma_0) \\
&\overset{(4)}{=} [\![V\mathcal{B}(\mathtt{v})]\!](\sigma_0)
\end{aligned}
$$

(1) By Lemma 3: Let $(\_, b, \_) \in \mathcal{R}(\mathtt{v})$. We have that $V\mathcal{B}(b)$ is recursively called when computing $V\mathcal{B}(\mathtt{v})$ (Definition 27). Note that with $[\![V\mathcal{B}(\mathtt{v})]\!] \neq \infty$ also $[\![V\mathcal{B}(b)]\!] \neq \infty$. By I.H. $[\![V\mathcal{B}(b)]\!](\sigma_0)$ is a *variable bound* for b on ρ.

(2) Let $(t, \_) \in \mathcal{I}(\mathtt{v})$. We have that $T\mathcal{B}(t)$ is called when computing $V\mathcal{B}(\mathtt{v})$ (Definition 27). Note that with $[\![V\mathcal{B}(\mathtt{v})]\!] \neq \infty$ also $[\![T\mathcal{B}(t)]\!] \neq \infty$. By I.H. $\sharp(t, \rho) \leq [\![T\mathcal{B}(t))]\!](\sigma_0)$. We thus get $\sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} \sharp(t, \rho) \times \mathtt{c} \leq \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c}$ because for all $(\_, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ we have $\mathtt{c} > 0$ (Definition 19).

(3) $[\![\mathtt{Incr}(\mathtt{v})]\!](\sigma_0) = \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c}$ (Definition 27 and Definition 17).

(4) Definition 27 and Definition 17.

(II) $\zeta(\tau) \in \mathcal{V}$. We have:

$$
\begin{aligned}
\sharp(\tau, \rho) &\overset{(1)}{\leq} \Big(\sum_{(t,\mathtt{c}) \in \mathcal{I}(\zeta(\tau))} \sharp(t, \rho) \times \mathtt{c} \Big) + \sum_{(t,b,c) \in \mathcal{R}(\zeta(\tau))} \sharp(t, \rho) \times [\![V\mathcal{B}(b)]\!](\sigma_0) + c \\
&\overset{(2)}{\leq} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\zeta(\tau))} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c} + \sum_{(t,b,c) \in \mathcal{R}(\zeta(\tau))} \sharp(t, \rho) \times [\![V\mathcal{B}(b)]\!](\sigma_0) + c \\
&\overset{(3)}{=} [\![\mathtt{Incr}(\zeta(\tau))]\!](\sigma_0) + \sum_{(t,b,c) \in \mathcal{R}(\zeta(\tau))} \sharp(t, \rho) \times [\![V\mathcal{B}(b)]\!](\sigma_0) + c \\
&\overset{(4)}{\leq} [\![\mathtt{Incr}(\zeta(\tau))]\!](\sigma_0) + \sum_{(t,b,c) \in \mathcal{R}(\zeta(\tau))} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \max([\![V\mathcal{B}(b)]\!](\sigma_0) + c, 0) \\
&\overset{(5)}{=} [\![T\mathcal{B}(\tau)]\!](\sigma_0)
\end{aligned}
$$

(1) By Lemma 2: Since Ξ is closed under *normalization* we have that $\zeta(\tau)$ is a *local transition bound* for τ on $\lfloor \rho \rfloor$. Further: Let $(\_, b, \_) \in \mathcal{R}(\zeta(\tau))$. We have that $V\mathcal{B}(b)$ is called during the computation of $T\mathcal{B}(\tau)$ (Definition 27). Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ also $[\![V\mathcal{B}(b)]\!] \neq \infty$. By I.H. $[\![V\mathcal{B}(b)]\!](\sigma_0)$ is a *variable bound* for b.

(2) Let $(t, \_) \in \mathcal{I}(\zeta(\tau))$. We have that there is a recursive call to $T\mathcal{B}(t)$ during the computation of $T\mathcal{B}(\tau)$ (Definition 27). Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ also $[\![T\mathcal{B}(t)]\!] \neq \infty$. By I.H. $\sharp(t, \rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$. Further for all $(\_, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ $\mathtt{c} \geq 0$ (Definition 19).

(3) Definition 27 and Definition 17.

(4) Let $(t, \_, \_) \in \mathcal{R}(\zeta(\tau))$. We have that $T\mathcal{B}(t)$ is recursively called during the computation of $T\mathcal{B}(\tau)$ (Definition 27). Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ also $[\![T\mathcal{B}(t)]\!] \neq \infty$. By I.H. $\sharp(t, \rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(5) Definition 27 and Definition 17.

□

7.2 Soundness of Reasoning on Reset Chains

Lemma 7 extends Lemma 2 by chained resets.

Lemma 4, Lemma 5 and Lemma 6 are helper lemmas needed for the proof of Lemma 7.

Definition 52 (Matching of a Reset Chain)**.** *Let* $\kappa = \mathtt{a}_n \xrightarrow{\tau_n,\mathtt{c}_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1},\mathtt{c}_{n-1}} \cdots \mathtt{a}_0$ *be a* reset chain *of* $\Delta\mathcal{P}$*. Let* ρ *be a* run *of* $\Delta\mathcal{P}$*. We call* $i_n, i_{n-1} \dots i_1 \in \mathbb{N}$ *with* $0 \leq i_n < i_{n-1} \cdots < i_1 < len(\rho)$ *a* matching of κ *on* ρ *iff* $\rho(i_j) = \tau_j$ *holds for all* $n \geq j \geq 1$*. We call* i_n *the first index and* i_1 *is the last index. A matching* $i_n, i_{n-1}, \dots, i_1$ *of* κ *on* ρ *is precise iff for all* $n > j \geq 1$ *it holds that* a_j *is* not reset *on* $\rho_{[i_{j+1}+1,i_j]}$*, i.e.,* $(\rho(k), \_, \_) \notin \mathcal{R}(\mathtt{a}_j)$ *for all* $i_{j+1} < k < i_j$*.*

Informally: There is a matching of $\kappa = \mathtt{a}_n \xrightarrow{\tau_n,\mathtt{c}_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1},\mathtt{c}_{n-1}} \cdots \mathtt{a}_0$ on a run ρ if ρ contains the transitions $\tau_n, \tau_{n-1}, \dots, \tau_1$ in that order. A matching $i_n, i_{n-1}, \dots i_1$ is precise if for all $n > j \geq 1$ it holds that a_j flows into a_{j-1} when executing $\rho(i_j)$ because a_j is not reset between the reset of a_j to a_{j+1} on $\rho(i_{j+1})$ and the reset of a_{j-1} to a_j on $\rho(i_j)$.

Definition 53 (First- and Last-Indices of Precise Matchings)**.** *Let* ρ *be a run of* $\Delta\mathcal{P}$*. Let* $\kappa = \mathtt{a}_n \xrightarrow{\tau_n,c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_n,c_n} \dots \xrightarrow{\tau_1,c_1} \mathtt{a}_0$ *be a reset chain. We define* $\alpha(\kappa, \rho)$ *to denote the set*
$\{(i_n, i_1) \mid \exists i_{n-1}, \dots, i_2 \text{ s.t. } i_n, i_{n-1}, i_{n-2}, \dots, i_2, i_1 \text{ is a precise matching of } \kappa \text{ on } \rho\}$*.*

I.e., $\alpha(\kappa, \rho)$ is the set of first- and last-indices of all precise matchings of κ on ρ. Note that in particular $i \leq j$ for all $(i, j) \in \alpha(\kappa, \rho)$, i.e., the interval $[i \dots j]$ is non-empty.

Given a reset chain κ from b to $\mathtt{v}$ and a precise matching of κ on a run ρ with first index i and last index j, Lemma 4 states that the value of $\mathtt{v}$ in state σ_j on ρ is bounded by the value of b in state σ_i on ρ and the increments of $\mathtt{a} \in atm(\kappa)$ between index i and index j on ρ.

Lemma 4. *Let* ρ *be a run of* $\Delta\mathcal{P}$*. Let* $b \in \mathcal{A}$ *and* $\mathtt{v} \in \mathcal{V}$*. Let* κ *be a reset chain from* b *to* $\mathtt{v}$*. Let* $(i, j) \in \alpha(\kappa, \rho)$*. Then*
$$\sigma_{j+1}(\mathtt{v}) \leq \sigma_i(b) + c(\kappa) + \sum_{\mathtt{a} \in atm(\kappa) \setminus \{\mathtt{v}\}} \sum_{(\tau,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i+1,j]}) \times \mathtt{c}.$$

Proof. We show the claim by induction on the length of κ.

Base Case: Let $\kappa = b \xrightarrow{\tau,\mathtt{c}} \mathtt{v}$. With $(i, j) \in \alpha(\kappa, \rho)$ we have that $i = j$ and $\rho(i) = \rho(j) = \tau$. Further we have that $(\tau, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ (Definition 28). Thus $\sigma_{j+1}(\mathtt{v}) = \sigma_{i+1}(\mathtt{v}) \leq \sigma_i(b) + \mathtt{c}$ (Definition 19). Note that $atm(\kappa) \setminus \{\mathtt{v}\} = \emptyset$ since $b \notin atm(\kappa)$ (Definition 28).

Step Case: Let $\kappa = \mathtt{a}_{n+1} \xrightarrow{\tau_{n+1},c_{n+1}} \mathtt{a}_n \xrightarrow{\tau_n,c_n} \dots \xrightarrow{\tau_1,c_1} \mathtt{v}$ with $\mathtt{a}_{n+1} = b$. Let $i_{n+1}, i_n, i_{n-1}, \dots, i_1$ be a precise matching of κ on ρ with $i_{n+1} = i$ and $i_1 = j$.

$$\sigma_{j+1}(\mathtt{v}) = \sigma_{i_1+1}(\mathtt{v}) \overset{(1)}{\leq} \sigma_{i_n}(\mathtt{a}_n) + c(\kappa_{[n,0]}) + \sum_{\mathtt{a} \in atm(\kappa_{[n,0]}) \setminus \{\mathtt{v}\}} \sum_{(\tau,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_n+1,i_1]}) \times \mathtt{c}$$

$$
\begin{aligned}
&\overset{(2)}{\leq} \sigma_{i_{n+1}+1}(\mathtt{a}_n) + \Big(\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a}_n)} \sharp(\tau, \rho_{[i_{n+1}+1,i_n]}) \times \mathtt{c}\Big) \\
&\qquad + \; c(\kappa_{[n,0]}) + \sum_{\mathtt{a}\in atm(\kappa_{[n,0]})\setminus\{\mathtt{v}\}} \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_n+1,i_1]}) \times \mathtt{c} \\
&\overset{(3)}{\leq} \sigma_{i_{n+1}}(\mathtt{a}_{n+1}) + c_{n+1} + \Big(\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a}_n)} \sharp(\tau, \rho_{[i_{n+1}+1,i_n]}) \times \mathtt{c}\Big) \\
&\qquad + \; c(\kappa_{[n,0]}) + \sum_{\mathtt{a}\in atm(\kappa_{[n,0]})\setminus\{\mathtt{v}\}} \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_n+1,i_1]}) \times \mathtt{c} \\
&\overset{(4)}{\leq} \sigma_{i_{n+1}}(\mathtt{a}_{n+1}) + c_{n+1} + \Big(\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a}_n)} \sharp(\tau, \rho_{[i_{n+1}+1,i_1]}) \times \mathtt{c}\Big) \\
&\qquad + \; c(\kappa_{[n,0]}) + \sum_{\mathtt{a}\in atm(\kappa_{[n,0]})\setminus\{\mathtt{v}\}} \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_{n+1}+1,i_1]}) \times \mathtt{c} \\
&\overset{(5)}{=} \sigma_{i_{n+1}}(\mathtt{a}_{n+1}) + c(\kappa) + \sum_{\mathtt{a}\in atm(\kappa)\setminus\{\mathtt{v}\}} \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_{n+1}+1,i_1]}) \times \mathtt{c}
\end{aligned}
$$

(1) By I.H.: We have that $\kappa_{[n,0]}$ is also a reset chain (Definition 28, note that $\kappa_{[n,0]}$ is non-empty by definition of κ) and since $i_{n+1}, i_n, \ldots, i_1$ is a precise matching of κ on ρ, $i_n, \ldots, i_1$ is a precise matching of $\kappa_{[n,0]}$ on ρ (Definition 52).

(2) We have that for all $i_{n+1} < j < i_n$ $(\rho(j), \_, \_) \notin \mathcal{R}(\mathtt{a}_n)$ (Definition 52), i.e., $\mathtt{a}_n$ is not reset on $\rho_{[i_{n+1}+1,i_n]}$. In the proof of Lemma 3 we show that $\sigma_{i_n}(\mathtt{a}_n) \leq \sigma_{i_{n+1}+1}(\mathtt{a}_n) + \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a}_n)} \sharp(\tau, \rho_{[i_{n+1}+1,i_n]}) \times \mathtt{c}$.

(3) $\sigma_{i_{n+1}+1}(\mathtt{a}_n) \leq \sigma_{i_n+1}(\mathtt{a}_{n+1}) + \mathtt{c}_{n+1}$ (Definition 28)

(4) Note that $[i_{n+1}+1 \ldots i_n]$ is a sub-interval of $[i_{n+1}+1 \ldots i_1]$. Therefore $\sharp(\tau, \rho_{[i_{n+1}+1,i_n]}) \leq \sharp(\tau, \rho_{[i_{n+1}+1,i_1]})$ for all $\tau \in E$. Accordingly $[i_n+1 \ldots i_1]$ is a sub-interval of $[i_{n+1}+1 \ldots i_1]$. Therefore $\sharp(\tau, \rho_{[i_n+1,i_1]}) \leq \sharp(\tau, \rho_{[i_{n+1}+1,i_1]})$ for all $\tau \in E$. We thus get
$\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_{n+1}+1,i_n]}) \times \mathtt{c} \leq \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_{n+1}+1,i_1]}) \times \mathtt{c}$ and
$\sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_n+1,i_1]}) \times \mathtt{c} \leq \sum_{(\tau,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(\tau, \rho_{[i_{n+1}+1,i_1]}) \times \mathtt{c}$
because we have that for all $\mathtt{v} \in \mathcal{V}$ and for all $(\_, \mathtt{c}) \in \mathcal{I}(\mathtt{v})$ it holds that $\mathtt{c} > 0$ (Definition 19).

(5) We have $c(\kappa) = c(\kappa_{[n,0]}) + \mathtt{c}_{n+1}$ and $atm(\kappa) = atm(\kappa_{[n,0]}) \cup \{\mathtt{a}_n\}$ (Definition 28).

□

Let $\mathtt{v} \in \mathcal{V}$. Lemma 5 states that for each index j on a run ρ s.t. $\mathtt{v}$ is reset on $\rho(j)$, there is a corresponding *optimal* reset chain κ and a precise matching of κ on ρ ending at j.

Lemma 5. *Let ρ be a run of $\Delta\mathcal{P}$. Let $\mathtt{v} \in \mathcal{V}$. Let $(\tau, \_, \_) \in \mathcal{R}(\mathtt{v})$. Let j be s.t. $\rho(j) = \tau$. There is a $\kappa \in \mathfrak{R}(\mathtt{v})$ and a $i \leq j$ s.t. $(i, j) \in \alpha(\kappa, \rho)$.*

Proof. Let $\mathtt{a} \in \mathcal{A}$ and $\mathtt{c} \in \mathbb{Z}$ be such that $(\tau, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ (note that by *determinism* of $\Delta\mathcal{P}$ there is exactly one such $\mathtt{a}$ and $\mathtt{c}$). We proof the claim by the following recursive reasoning:

[Start] We show that there is a *sound* reset chain κ that ends at $\mathtt{v}$ and a precise matching of κ on ρ that ends at j: Obviously $\mathtt{a} \xrightarrow{\tau,\mathtt{c}} \mathtt{v}$ is a reset chain. Further $\mathtt{a} \xrightarrow{\tau,\mathtt{c}} \mathtt{v}$ is trivially *sound* (Definition 28). We have that j is a *precise* matching for $\mathtt{a} \xrightarrow{\tau,\mathtt{c}} \mathtt{v}$ on ρ because by assumption $\rho(j) = \tau$ (Definition 52).
[Recursive Step] We thus have that there is a *sound* reset chain $\kappa = \mathtt{a}_n \xrightarrow{\tau_n,c_n} \mathtt{a}_{n-1} \xrightarrow{\tau_{n-1},c_{n-1}} \ldots \xrightarrow{\tau_1,c_1} \mathtt{v}$ and a precise matching $i_n, i_{n-1}, \ldots, i_1$ of κ on ρ with $i_1 = j$. If κ is *optimal* then $\kappa \in \mathfrak{R}(\mathtt{v})$ (Definition 28) and with $(i_n, j) \in \alpha(\kappa, \rho)$ the claim is proven. Assume κ is *not* optimal. Then $\mathtt{a}_n \in \mathcal{V}$ because κ is not maximal (Definition 28). By well-definedness of $\Delta\mathcal{P}$ $\mathtt{a}_n$ is *reset* on $\rho_{[0,i_n]}$, i.e., there is a $0 \leq k < i_n$ s.t. $(\rho(k), \_, \_) \in \mathcal{R}(\mathtt{a}_n)$. Let i_{n+1} denote the maximal such k. Let $\tau_{n+1} = \rho(i_{n+1})$. Let $\mathtt{a}_{n+1} \in \mathcal{A}$ and $\mathtt{c}_{n+1} \in \mathbb{Z}$ be s.t. $(\tau_{n+1}, \mathtt{a}_{n+1}, \mathtt{c}_{n+1}) \in \mathcal{R}(\mathtt{a}_n)$. Then $\varkappa = a_{n+1} \xrightarrow{\tau_{n+1},\mathtt{c}_{n+1}} \mathtt{a}_n \xrightarrow{\tau_n,\mathtt{c}_n} \mathtt{a}_{n-1} \ldots \mathtt{v}$ is a *reset chain* ending in $\mathtt{v}$ and $i_{n+1}, i_n, \ldots, i_1$ is a *precise matching* of $\varkappa$ on ρ (Definition 52). We show that $\varkappa$ is *sound*: First note that $\varkappa_{[n,0]} = \kappa$ and because κ is *sound* we have that for all $1 \leq i < n$ it holds that $\mathtt{a}_i$ *is reset on all paths from the target location of τ_1 to the source location of τ_i*. It remains to show that this also holds for $\mathtt{a}_n$. Since κ is not optimal there is a *sound* reset chain that extends κ (Definition 28). Now, because $\mathtt{a}_n$ is on that extended *sound* reset chain we have that also $\mathtt{a}_n$ *is reset on all paths from the target location of τ_1 to the source location of τ_n* (Definition 28). We conclude that $\varkappa$ is *sound*. We can thus recursively apply our reasoning on $\varkappa$.
[Termination] Since by assumption the reset graph is *acyclic* and its node set $\mathcal{A}$ is *finite*, a *optimal* reset chain $\kappa \in \mathfrak{R}(\mathtt{v})$ and a matching of κ that ends at j is constructed by iterating the stated reasoning *finitely* often. □

Note that with Lemma 4 and Lemma 5 we can bound the value to which $\mathtt{v}$ is reset at index j in terms of the value of $in(\kappa)$ at index i, where i is the start-index of the matching that ends at j.

Lemma 6 states that precise matchings of *optimal* reset chains that share a common suffix never overlap.

Lemma 6. *Let ρ be a run of $\Delta\mathcal{P}$. Let $\mathtt{v} \in \mathcal{V}$. Let $\kappa, \varkappa \in \mathfrak{R}(\mathtt{v})$ be s.t. κ and $\varkappa$ have a common suffix, i.e., there exists $l > 0$ s.t. $\kappa_{[l,0]} = \varkappa_{[l,0]}$. Let $(i_k, i_1) \in \alpha(\kappa, \rho)$ and $(j_n, j_1) \in \alpha(\varkappa, \rho)$. Either $\kappa = \varkappa$ and $[i_k \ldots i_1] = [j_n \ldots j_1]$ or the two intervals $[i_k \ldots i_1]$ and $[j_n \ldots j_1]$ are* disjoint*, i.e., $i_1 < j_n$ or $j_1 < i_k$.*

Proof. Let $\kappa = \mathtt{a}_k \xrightarrow{\tau_k,\mathtt{c}_k} \mathtt{a}_{k-1} \ldots \mathtt{a}_1 \xrightarrow{\tau_1,\mathtt{c}_1} \mathtt{v}$.
Let $\varkappa = b_n \xrightarrow{t_n,c_n} b_{n-1} \ldots b_1 \xrightarrow{t_1,c_1} \mathtt{v}$.

Let $i_k, i_{k-1}, \dots i_1$ be a *precise* matching of κ on ρ.
Let $j_n, j_{n-1}, \dots j_1$ be a *precise* matching of $\varkappa$ on ρ.

[A] We show that if $i_1 = j_1$ then $i_k = j_n$ and $\kappa = \varkappa$: W.l.o.g. assume $k \leq n$.

[A.1] We show that for all $k \leq l \leq 1$ $i_l = j_l$: By assumption $i_1 = i_1 = j_1 = j_1$. We conclude that $\mathtt{a}_1 = b_1$ because since $\Delta\mathcal{P}$ is *fan-in free* there is exactly one $\mathtt{a}_1$ s.t. $(\mathtt{a}_1, \_, \rho(i_1)) \in \mathcal{R}(\mathtt{v})$. Assume $i_2 \neq j_2$. Case $j_2 < i_2$: By Definition 52 $\mathtt{a}_1$ is *not reset* on $\rho_{[j_2+1,j_1]}$, i.e., $(\rho(k), \_\_\_) \notin \mathcal{R}(\mathtt{a}_1)$ for all $j_2 < k < j_1$. Note that $j_2 < i_2 < i_1 = j_1$. We have $(\rho(i_2), \_, \_) \in \mathcal{R}(b_1)$ (Definition 52 and Definition 28). With $\mathtt{a}_1 = b_1$ we have $(\rho(i_2), \_, \_) \in \mathcal{R}(\mathtt{a}_1)$. Contradiction. Case $i_2 < j_2$: Analogous. Thus $i_2 = j_2$. We apply the same reasoning for $i_3, i_4 \dots i_k$ consecutively.

[A.2] We show that $k = n$: By [A.1] we have that $\varkappa_{[k,1]} = \kappa$ (Definition 52). Thus κ is a *suffix* of $\varkappa$. But by assumption κ is *optimal*. Thus $\kappa = \varkappa$ (Definition 28).

[A] is proven with [A.1] and [A.2].

[B] We show that if $i_1 \neq j_1$ then $i_1 < j_n$ or $j_1 < i_k$, i.e., the intervals $[i_k \dots i_1]$ and $[j_n \dots j_1]$ are *disjoint*:

[B.1] We have $\rho(i_1) = \rho(j_1) = t_1$ because by assumption κ and $\varkappa$ have a common suffix.

[B.2] We show [B.2.i] that for all l with $j_n \leq l < j_1$ it holds that $\rho(l) \neq t_1$ and [B.2.ii] that for all l with $i_k \leq l < i_1$ it holds that $\rho(l) \neq t_1$.
[B.2.i] Assume there is some l with $j_n \leq l < j_1$ s.t. $\rho(l) = t_1$. Then there is some $n \geq r > 1$ s.t. $j_r \leq l < j_{r-1}$. Since $j_n, j_{n-1}, \dots j_1$ is a *precise* matching of $\varkappa$ we have that for all $j_r < s < j_{r-1}$ $(\rho(s), \_, \_) \notin \mathcal{R}(\mathtt{a}_{r-1})$ (Definition 52). But since $\varkappa$ is *sound* $\mathtt{a}_{r-1}$ *must be reset* on *all paths from the target location of* t_1 *to the source location of* t_{r-1}, i.e., in particular on $\rho_{[l+1,j_{r-1}]}$ because $\rho(l) = t_1$ and $\rho(j_{r-1}) = t_{r-1}$ (Definition 52). Thus there must be some s with $j_r \leq l < s < j_{r-1}$ s.t. $(\rho(s), \_, \_) \in \mathcal{R}(\mathtt{a}_{r-1})$. Contradiction.
[B.2.ii] Analogous.

[B.1] and [B.2] imply [B]: By assumption $i_1 \neq j_1$. W.l.o.g. let $i_1 < j_1$. With $i_k \leq i_1$ and $j_n \leq j_1$ we have $i_k < j_1$. We thus have to show that $i_1 < j_n$: Assume $j_n \leq i_1$: Then $j_n \leq i_1 < j_1$. But with [B.1] this contradicts [B.2]. Therefore $i_1 < j_n$.

With [A] and [B] the claim is proven.

□

Lemma 7 extends Lemma 2 by chained resets. Let $\mathtt{v}$ be a local transition bound for τ: The question how often a given transition τ may appear on a run ρ is translated to the question how often the transitions that increase the value of the local transition bound $\mathtt{v}$ are executed. But in contrast to Lemma 2 Lemma 7 takes the context under which these transitions may increase $\mathtt{v}$ into account. See Section 3.5 for more details.

Lemma 7. *Let* $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \dots$ *be a run of* $\Delta\mathcal{P}$. *Let* $\tau \in E$. *Let* $\mathtt{v} \in \mathcal{V}$ *be a* local bound *for* τ *on* $\lfloor\rho\rfloor$. *Let* $vb : \mathcal{A} \to \mathbb{Z}$ *be s.t.* $vb(\mathtt{a})$ *is a* variable bound *for* $\mathtt{a}$ *on* ρ

for all $\mathtt{a} \in \{in(\kappa) \mid \kappa \in \mathfrak{R}(\mathtt{v})\}$. *Then*

$$\left(\sum_{\mathtt{a}\in \bigcup_{\kappa\in\mathfrak{R}(\mathtt{v})} atm_1(\kappa)} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho)\times \mathtt{c}\right) + \sum_{\kappa\in\mathfrak{R}(\mathtt{v})} (\min_{t\in trn(\kappa)} \sharp(t,\rho)) \times \max(vb(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a}\in atm_2(\kappa)} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho)\times \mathtt{c}$$

is a transition bound *for* τ *on* ρ.

Proof. As argued in the proof of Lemma 2 it is sufficient to consider the case $\rho = \lfloor\rho\rfloor$.

A) As shown in the proof of Lemma 2 we have that

$$\sharp(\tau,\rho) \leq \left(\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{v})} \sharp(t,\rho)\times\mathtt{c}\right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

B) We show that

$$\sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v}) \leq \left(\sum_{\mathtt{a}\in\bigcup_{\kappa\in\mathfrak{R}(\mathtt{v})} atm_1(\kappa)\setminus\{\mathtt{v}\}} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho)\times\mathtt{c}\right) + \sum_{\kappa\in\mathfrak{R}(\mathtt{v})} (\min_{t\in trn(\kappa)} \sharp(t,\rho)) \times \max(vb(in(\kappa)) + c(\kappa), 0) + \sum_{\mathtt{a}\in atm_2(\kappa)} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho)\times\mathtt{c}$$

With Lemma 5 we have that for each $j \in \Theta(\mathcal{R}(\mathtt{v}),\rho)$ there is *at least* one $\kappa \in \mathfrak{R}(\mathtt{v})$ and one $i \leq j$ s.t. $(i,j) \in \alpha(\kappa,\rho)$.

Further: Let $\kappa \in \mathfrak{R}(\mathtt{v})$. Let $(i,j) \in \alpha(\kappa,\rho)$. With Lemma 4 we have that:

$$\sigma_{j+1}(\mathtt{v}) \leq \sigma_i(in(\kappa)) + c(\kappa) + \sum_{\mathtt{a}\in atm(\kappa)\setminus\{\mathtt{v}\}} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho_{[i+1,j]})\times\mathtt{c}$$

Therefore:

$$\sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v}) \leq \sum_{\kappa\in\mathfrak{R}(\mathtt{v})} \sum_{(i,j)\in\alpha(\kappa,\rho)} \sigma_i(in(\kappa)) + c(\kappa) + \sum_{\mathtt{a}\in atm(\kappa)\setminus\{\mathtt{v}\}} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho_{[i+1,j]})\times\mathtt{c}$$

$$\overset{(1a)}{=} \sum_{\kappa\in\mathfrak{R}(\mathtt{v})} \left(\sum_{(i,j)\in\alpha(\kappa,\rho)} \sigma_i(in(\kappa)) + c(\kappa)\right) + \left(\sum_{(i,j)\in\alpha(\kappa,\rho)} \sum_{\mathtt{a}\in atm(\kappa)\setminus\{\mathtt{v}\}} \sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})} \sharp(t,\rho_{[i+1,j]})\times\mathtt{c}\right)$$

$$\overset{(1b)}{=} \sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)\rho}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}$$

$$\overset{(1)}{=} \sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)\rho}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm_1(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}$$
$$+\sum_{\mathtt{a}\in atm_2(\kappa)}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}$$

$$\overset{(2)}{\leq} \sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm_1(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}$$
$$+\sum_{\mathtt{a}\in atm_2(\kappa)}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\sharp(t,\rho)\times\mathtt{c}$$

$$\overset{(3a)}{=} \left(\sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\sum_{\mathtt{a}\in atm_1(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}\right)$$
$$+\sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm_2(\kappa)}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\sharp(t,\rho)\times\mathtt{c}$$

$$\overset{(3b)}{=} \left(\sum_{\mathtt{a}\in\bigcup_{\kappa\in\mathfrak{R}(\mathtt{v})}atm_1(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\sum_{\kappa\in\mathfrak{R}(\mathtt{v})\text{ s.t. }\mathtt{a}\in atm_1(\kappa)}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}\right)$$
$$+\sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm_2(\kappa)}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\sharp(t,\rho)\times\mathtt{c}$$

$$\overset{(3c)}{=} \left(\sum_{\mathtt{a}\in\bigcup_{\kappa\in\mathfrak{R}(\mathtt{v})}atm_1(\kappa)\setminus\{\mathtt{v}\}}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\left(\sum_{\kappa\in\mathfrak{R}(\mathtt{v})\text{ s.t. }\mathtt{a}\in atm_1(\kappa)}\sum_{(i,j)\in\alpha(\kappa,\rho)}\sharp(t,\rho_{[i+1,j]})\right)\times\mathtt{c}\right)$$
$$+\sum_{\kappa\in\mathfrak{R}(\mathtt{v})}\left(\sum_{(i,j)\in\alpha(\kappa,\rho)}\sigma_i(in(\kappa))+c(\kappa)\right)+\sum_{\mathtt{a}\in atm_2(\kappa)}\sum_{(t,\mathtt{c})\in\mathcal{I}(\mathtt{a})}\sharp(t,\rho)\times\mathtt{c}$$

$$
\begin{aligned}
&\overset{(3)}{\leq} \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa) \setminus \{\mathtt{v}\}} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} \left(\sum_{(i,j) \in \alpha(\kappa,\rho)} \sigma_i(in(\kappa)) + c(\kappa) \right) + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(4)}{\leq} \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa) \setminus \{\mathtt{v}\}} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} \left(\sum_{(i,j) \in \alpha(\kappa,\rho)} vb(in(\kappa)) + c(\kappa) \right) + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(5a)}{=} \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa) \setminus \{\mathtt{v}\}} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} |\alpha(\kappa,\rho)| \times (vb(in(\kappa)) + c(\kappa)) \\
&\qquad + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(5)}{\leq} \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa) \setminus \{\mathtt{v}\}} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} (\min_{t \in trn(\kappa)} \sharp(t,\rho)) \times \max(vb(in(\kappa)) + c(\kappa), 0) \\
&\qquad + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t,\rho) \times \mathtt{c}
\end{aligned}
$$

(1a) Commutativity.

(1b) Distributivity.

(1) We have $atm(\kappa) = atm_1(\kappa) \cup atm_2(\kappa)$, $atm_1(\kappa) \cap atm_2(\kappa) = \emptyset$ and $\mathtt{v} \in atm_1(\kappa)$ (Definition 30).

(2) With Lemma 6 we have that all intervals in $\alpha(\kappa,\rho)$ are pairwise disjoint. Therefore $\sum_{(i,j) \in \alpha(\kappa,\rho)} \sharp(t,\rho_{[i+1,j]}) \leq \sharp(t,\rho)$. Further note that $\mathtt{c} > 0$ for $(\_,\mathtt{c}) \in \mathcal{I}(\mathtt{a})$.

(3a) Commutativity.

(3b) Commutativity.

(3c) Distributivity.

(3) Let $\kappa_1, \kappa_2 \in \mathfrak{R}(\mathtt{v})$. Assume $\mathtt{a} \in atm_1(\kappa_1) \cap atm_1(\kappa_2)$ and $\mathtt{a} \neq \mathtt{v}$. By Definition 30 there is exactly one path in the reset graph from $\mathtt{a}$ to $\mathtt{v}$. Thus κ_1 and κ_2 have a common suffix: they share the single path from $\mathtt{a}$ to $\mathtt{v}$ in the reset graph. We therefore have by Lemma 6 that all intervals in $\alpha(\kappa_1, \rho) \cup \alpha(\kappa_2, \rho)$ are pairwise disjoint. Therefore $\sum_{\kappa \in \mathfrak{R}(\mathtt{v}) \text{ s.t. } \mathtt{a} \in atm_1(\kappa)} \sum_{(i,j) \in \alpha(\kappa,\rho)} \sharp(t, \rho_{[i+1,j]}) \leq \sharp(t, \rho)$. Further note that $\mathtt{c} > 0$ for $(\_, \mathtt{c}) \in \mathcal{I}(\mathtt{a})$.

(4) Let $\kappa \in \mathfrak{R}(\mathtt{v})$. By assumption $vb(in(\kappa))$ denotes a variable bound for $in(\kappa)$ on ρ.

(5a) With $\sum_{(i,j) \in \alpha(\kappa,\rho)} vb(in(\kappa)) + c(\kappa) = |\alpha(\kappa, \rho)| \times (vb(in(\kappa)) + c(\kappa))$

(5) Let $\kappa \in \mathfrak{R}(\mathtt{v})$. Let $(i_1, j_1), (i_2, j_2) \in \alpha(\kappa, \rho)$. We have by Lemma 6 that all intervals in $\alpha(\kappa, \rho)$ are pairwise disjoint. Further each transition $t \in trn(\kappa)$ appears at least once on each sub-run $\rho_{[i,j]}$ with $(i, j) \in \alpha(\kappa, \rho)$. Therefore: $|\alpha(\kappa, \rho)| \leq \min_{t \in trn(\kappa)} \sharp(t, \rho)$.

C)

$$
\begin{aligned}
\sharp(\tau, \rho) &\overset{(1)}{\leq} \left(\sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{v})} \sharp(t, \rho) \times \mathtt{c} \right) + \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa) \setminus \{\mathtt{v}\}} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t, \rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} (\min_{t \in trn(\kappa)} \sharp(t, \rho)) \times \max(vb(in(\kappa)) + c(\kappa), 0) \\
&\quad + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t, \rho) \times \mathtt{c} \\
&\overset{(2)}{=} \left(\sum_{\mathtt{a} \in \bigcup_{\kappa \in \mathfrak{R}(\mathtt{v})} atm_1(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t, \rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\mathtt{v})} (\min_{t \in trn(\kappa)} \sharp(t, \rho)) \times \max(vb(in(\kappa)) + c(\kappa), 0) \\
&\quad + \sum_{\mathtt{a} \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(\mathtt{a})} \sharp(t, \rho) \times \mathtt{c}
\end{aligned}
$$

(1) With A) and B).

(2) We have $\mathcal{R}(\mathtt{v}) \neq \emptyset$ by well-definedness of $\Delta\mathcal{P}$ and therefore $\mathfrak{R}(\mathtt{v}) \neq \emptyset$. Further $\mathtt{v} \in atm_1(\kappa)$ for all $\kappa \in \mathfrak{R}(\mathtt{v})$. □

7.2.1 Proof of Theorem 4

We prove the more general claim formulated in Theorem 9.

Theorem 9 (Soundness of Bound Algorithm using Reset Chains)**.** *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP over atoms $\mathcal{A}$ with a reset dag. Let Ξ be a set of runs of $\Delta\mathcal{P}$ that is closed under* normalization*. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* local bound mapping *for all $\rho \in \Xi$. Let $T\mathcal{B}$ and $V\mathcal{B}$ be defined as in Definition 31. Let $\tau \in E$ and $\mathtt{a} \in \mathcal{A}$. Let $\rho \in \Xi$. Let σ_0 be the initial state of ρ. We have: (I) $[\![T\mathcal{B}(\tau)]\!](\sigma_0)$ is a* transition bound *for τ on ρ. (II) $[\![V\mathcal{B}(\mathtt{a})]\!](\sigma_0)$ is a* variable bound *for $\mathtt{a}$ on ρ.*

Proof. Let $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \cdots \in \Xi$.

If $[\![T\mathcal{B}(\tau)]\!] = \infty$ (I) holds trivially. If $[\![V\mathcal{B}(\mathtt{a})]\!] = \infty$ (II) holds trivially.

Assume $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ and $[\![V\mathcal{B}(\mathtt{a})]\!] \neq \infty$. Then in particular the computation of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a})$ terminate. We proceed by induction over the call tree of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a})$.

Base Case: As in the proof of Theorem 8 (Section 7.1.1).

Step Case:
I) As in the proof of Theorem 8 (Section 7.1.1).
II)

$$
\begin{aligned}
\sharp(\tau,\rho) &\overset{(1)}{\leq} \left(\sum_{b \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} \Big(\min_{t \in trn(\kappa)} \sharp(t,\rho) \Big) \times \max([\![V\mathcal{B}(in(\kappa))]\!](\sigma_0) + c(\kappa), 0) \\
&\quad\quad + \sum_{b \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(2)}{\leq} \left(\sum_{b \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} \Big(\min_{t \in trn(\kappa)} [\![T\mathcal{B}(t)]\!](\sigma_0) \Big) \times \max([\![V\mathcal{B}(in(\kappa))]\!](\sigma_0) + c(\kappa), 0) \\
&\quad\quad + \sum_{b \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c}
\end{aligned}
$$

$$
\begin{aligned}
&\overset{(3)}{\leq} \left(\sum_{b \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} [\![T\mathcal{B}(trn(\kappa))]\!](\sigma_0) \times \max([\![V\mathcal{B}(in(\kappa))]\!](\sigma_0) + c(\kappa), 0) \\
&\qquad + \sum_{b \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} \sharp(t,\rho) \times \mathtt{c} \\
&\overset{(4)}{\leq} \left(\sum_{b \in \bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c} \right) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} [\![T\mathcal{B}(trn(\kappa))]\!](\sigma_0) \times \max([\![V\mathcal{B}(in(\kappa))]\!](\sigma_0) + c(\kappa), 0) \\
&\qquad + \sum_{b \in atm_2(\kappa)} \sum_{(t,\mathtt{c}) \in \mathcal{I}(b)} [\![T\mathcal{B}(t)]\!](\sigma_0) \times \mathtt{c} \\
&\overset{(5)}{=} [\![\mathtt{Incr}(\bigcup_{\kappa \in \mathfrak{R}(\zeta(\tau))} atm_1(\kappa))]\!](\sigma_0) \\
&\quad + \sum_{\kappa \in \mathfrak{R}(\zeta(\tau))} [\![T\mathcal{B}(trn(\kappa))]\!](\sigma_0) \times \max([\![V\mathcal{B}(in(\kappa))]\!](\sigma_0) + c(\kappa), 0) \\
&\qquad + [\![\mathtt{Incr}(atm_2(\kappa))]\!](\sigma_0) \\
&\overset{(6)}{=} [\![T\mathcal{B}(\tau)]\!](\sigma_0)
\end{aligned}
$$

(1) By Lemma 7: Since Ξ is closed under *normalization* we have that $\zeta(\tau)$ is a *local bound* for τ on $\lfloor \rho \rfloor$. Further: Let $\kappa \in \mathfrak{R}(\zeta(\tau))$. We have that $V\mathcal{B}(in(\kappa))$ is called during the computation of $T\mathcal{B}(\tau)$ (Definition 31). Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ also $[\![V\mathcal{B}(in(\kappa))]\!] \neq \infty$. By I.H. $[\![V\mathcal{B}(in(\kappa))]\!](\sigma_0)$ is a *variable bound* for $in(\kappa)$.

(2) Let $\kappa \in \mathfrak{R}(\zeta(\tau))$. Let $t \in trn(\kappa)$. We have that $T\mathcal{B}(t)$ is called during the computation of $T\mathcal{B}(\tau)$. Thus for $t \in trn(\kappa)$ with $[\![T\mathcal{B}(t)]\!] \neq \infty$ we have that $[\![T\mathcal{B}(t)]\!](\sigma_0)$ is a *transition bound* for t on ρ by I.H.. Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ there is a $t \in trn(\kappa)$ s.t. $[\![T\mathcal{B}(t)]\!] \neq \infty$. Thus $\min_{t \in trn(\kappa)} \sharp(t,\rho) \leq \min_{t \in trn(\kappa)} [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(3) With $T\mathcal{B}(trn(\kappa)) = \min_{t \in trn(\kappa)} T\mathcal{B}(t)$ (Definition 31) and Definition 17.

(4) Let $\kappa \in \mathfrak{R}(\zeta(\tau))$. Let $b \in atm(\kappa)$. Let $(t, \_) \in \mathcal{I}(b)$. We have that $T\mathcal{B}(t)$ is called when computing $T\mathcal{B}(\tau)$ (Definition 31). Note that with $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ also $[\![T\mathcal{B}(t)]\!] \neq \infty$. By I.H. $\sharp(t,\rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(5) Definition 31 and Definition 17.

(6) Definition 31 and Definition 17.

□

7.3 Soundness of Path-Sensitive Reasoning

7.3.1 Preliminaries

A main idea of our soundness proof is as follows: We decompose a given run ρ of $\Delta\mathcal{P}$ into the *simple and cyclic paths* that are taken by ρ. Based on this decomposition our proof reasons about the increments and decrements that occurs on the obtained simple and cyclic paths.

Definition 55 to Definition 57 and Lemma 8 to Lemma 11 are concerned with the *decomposition* of a given run.

The essential lemmas for the proof of Theorem 7 are Lemma 12 and Lemma 13. These lemmas should be familiar from the proof of the basic bound algorithm (Section 7.1): Lemma 12 is a path-sensitive version of Lemma 2, Lemma 13 is a path-sensitive version of Lemma 3.

We now discuss the aforementioned decomposition of a run ρ into the simple and cyclic paths taken by ρ: As an example consider Figure 3.1 (c) (page 38). Consider a run of Figure 3.1 (c) that takes the path $\pi = l_b \xrightarrow{u_0} l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_3} l_1 \xrightarrow{\emptyset} l_e$. This path can be decomposed into the simple and *acyclic* path $\pi_0 = l_b \xrightarrow{u_0} l_1 \xrightarrow{\emptyset} l_e$, and the simple and *cyclic* paths $\pi_1 = l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_3} l_1$, $\pi_2 = l_2 \xrightarrow{u_2} l_2$ and $\pi_3 = l_2 \xrightarrow{u_2} l_2$.

Below, we formulate path decomposition in two steps: In a first step (Definition 56), the transitions on the path are reordered by *permuting* (Definition 55) the transition indices. In a second step (Definition 57), the obtained sequence of transitions is *split* into *simple* paths.

The result is a list of *simple* paths such that only the first element of this list forms an acyclic (or empty) path, while all other elements of the list form cyclic paths (Lemma 10).

The two-step-approach of first reordering the transitions on the paths and then splitting the resulting transition sequence into simple paths, facilitates the proofs of our essential lemmas: The reordering allows to exploit commutativity and associativity of the addition operation: Given a path π, a reordering $\varpi(\pi)$ of the transitions on π and a variable $\mathtt{v}$, we have that $\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) = \mathtt{Sum}\mathcal{ID}(\varpi(\pi))(\mathtt{v})$ ($\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$ is defined in Definition 35, page 67). Further, let $\mathtt{Splt}(\varpi(\pi))$ denote a *splitting* of the reordering $\varpi(\pi)$, i.e., the sequence of transitions $\varpi(\pi)$ is *split* into several sequences of transitions $\mathtt{Splt}(\varpi(\pi))$. Then obviously $\mathtt{Sum}\mathcal{ID}(\varpi(\pi))(\mathtt{v}) = \sum_{\mathtt{l} \in \mathtt{Splt}(\varpi(\pi))} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v})$.

Definition 54 (Lists)**.** *Let $\mathbb{D}$ be some* domain*. We call a finite sequence $\langle \mathtt{e}_0, \mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n \rangle$ with $\mathtt{e}_i \in \mathbb{D}$ a* list *over $\mathbb{D}$. By ϵ we denote the empty list. By $\circ$ we denote the* list concatenation *operator: E.g., $\langle 1, 2 \rangle \circ \langle 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$. We denote the set of all lists over*

$\mathbb{D}$ *by* $\mathcal{L}(\mathbb{D})$. *We say that a list* $L_1 = \langle \mathtt{l}_0, \mathtt{l}_1, \ldots, \mathtt{l}_n \rangle$ *over* $\mathcal{L}(\mathbb{D})$ *is a* splitting *of a list* L_2 *over* $\mathbb{D}$ *iff* $\mathtt{l}_0 \circ \mathtt{l}_1 \cdots \circ \mathtt{l}_n = L_2$. *A* list $L = \langle \mathtt{e}_0, \mathtt{e}_2, \ldots, \mathtt{e}_n \rangle$ *over* $\mathbb{D}$ *defines the function* $L : [0, n] \to \mathbb{D}$ *with* $L(i) = \mathtt{e}_i$ *for all* $0 \leq i \leq n$. *By* $\sharp(\mathtt{e}, L)$ *we denote the number of elements* $\mathtt{e}_i$ *of* L *s.t.* $\mathtt{e}_i = \mathtt{e}$. *By* $\mathtt{hd}(L) = \mathtt{e}_0$ *we denote the first element of the list, by* $\mathtt{tl}(L) = \langle \mathtt{e}_1, \ldots \mathtt{e}_n \rangle$ *we denote the list without its first element.*

Definition 55 (Path Index Permutation)**.** *Let* $\Delta\mathcal{P}(L, E, l_b, l_e)$ *be a DCP. Let* $\pi = l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \ldots$ *be a path of* $\Delta\mathcal{P}$. *We call a bijective function* $[0 \ldots (len(\pi)-1)] \to [0 \ldots (len(\pi)-1)]$ *an* index permutation *of* π. *We represent index permutations in form of lists: E.g., an index permutation* $\mathtt{idxp} = \langle 1, 2, 0 \rangle$ *defines the function* $\mathtt{idxp} : [0 \ldots 2] \to [0 \ldots 2]$ *with* $\mathtt{idxp}(0) = 1$, $\mathtt{idxp}(1) = 2$ *and* $\mathtt{idxp}(2) = 0$.
Recall, that by $\pi(i) = l_i \xrightarrow{u_i} l_{i+1}$ *we denote the* $i+1$*th transition on* π. *Let* idxp *be an index permutation of* π. *Depending on the context, we denote by* idxp *either the index permutation itself, or the transition sequence* $\pi(\mathtt{idxp}(0)), \pi(\mathtt{idxp}(1)) \ldots$ *or the path formed by these transitions (if such a path exists).*

Definition 56 (The Index Permutation ϖ)**.** *Let* $\Delta\mathcal{P}(L, E, l_b, l_e)$ *be a DCP. Let* π *be a path of* $\Delta\mathcal{P}$. *We define the* index permutation $\varpi(\pi)$ *by stating* $\varpi(\pi) = \varpi(\pi, 0, len(\pi))$, *where*

$$\varpi(\pi, i, j) = \begin{cases} \epsilon & \text{if } i = j \\ \left.\begin{array}{l} \text{Let } i < k \leq j \text{ be the smallest } k \text{ s.t. } l_k = l_i \\ \varpi(\pi, k, j) \circ \langle i \rangle \circ \varpi(\pi, i+1, k) \end{array}\right\} & \text{if } l_i \text{ appears on } \pi_{[i+1,j]} \\ \langle i \rangle \circ \varpi(\pi, i+1, j) & \text{else} \end{cases}.$$

Example: Consider Figure 3.1 (c) (page 38).
Consider the path $\pi = l_b \xrightarrow{u_0} l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_3} l_1 \xrightarrow{\emptyset} l_e$.
We have: $\varpi(\pi) = \varpi(\pi, 0, 5) = \left\langle l_b \xrightarrow{u_0} l_1, l_1 \xrightarrow{\emptyset} l_e, l_1 \xrightarrow{u_1} l_2, l_2 \xrightarrow{u_3} l_1, l_2 \xrightarrow{u_2} l_2, l_2 \xrightarrow{u_2} l_2 \right\rangle$
(for the sake of readability we denote an element of $\varpi(\pi)$ by the respective transition rather than the index on π).

Lemma 8. *Let* $\Delta\mathcal{P}(L, E, l_b, l_e)$ *be a DCP. Let* π *be a path of* $\Delta\mathcal{P}$. *Let* $0 \leq i < j < len(\pi)$. $\varpi(\pi, i, j)$ *is an index permutation of* $\pi_{[i,j]}$.

Proof. By Definition. □

Definition 57 (The Splitting $\mathtt{Splt}_\varpi$ of ϖ)**.** *Let* $\Delta\mathcal{P}(L, E, l_b, l_e)$ *be a DCP. Let* $\pi = l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \ldots l_n$ *be a path of* $\Delta\mathcal{P}$. *We define the function* $\mathtt{Splt}_\varpi(\pi)$ *that returns a* splitting *(Definition 54) of* ϖ *by stating* $\mathtt{Splt}_\varpi(\pi) = \mathtt{Splt}_\varpi(\pi, 0, len(\pi))$, *where*

$$\mathtt{Splt}_\varpi(\pi, i, j) = \begin{cases} \langle \epsilon \rangle & \text{if } i = j \\ \left.\begin{array}{l} \text{Let } i < k \leq j \text{ be the smallest } k \text{ s.t. } l_k = l_i \\ \mathtt{Splt}_\varpi(\pi, k, j) \circ \langle \langle i \rangle \star \mathtt{Splt}_\varpi(\pi, i+1, k) \rangle \end{array}\right\} & \text{if } l_i \text{ appears on } \pi_{[i+1,j]} \\ \langle i \rangle \star \mathtt{Splt}_\varpi(\pi, i+1, j) & \text{else} \end{cases}$$
.

Here, $\star$ denotes the operator s.t. $\mathtt{l}_1 \star L_2 = \langle \mathtt{l}_1 \circ \mathtt{hd}(L_2)\rangle \circ \mathtt{tl}(L_2)$.
E.g., $\mathtt{l}_1 \star \langle \mathtt{l}_2, \mathtt{l}_3 \ldots\rangle = \langle \mathtt{l}_1 \circ \mathtt{l}_2, \mathtt{l}_3, \ldots\rangle$.

Example: Consider Figure 3.1 (c).
Consider the path $\pi = l_b \xrightarrow{u_0} l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_2} l_2 \xrightarrow{u_3} l_1 \xrightarrow{\emptyset} l_e$.
We have $\mathtt{Splt}_{\varpi}(\pi) = \mathtt{Splt}_{\varpi}(\pi, 0, 5) = \left\langle l_b \xrightarrow{u_0} l_1 \xrightarrow{\emptyset} l_e, l_1 \xrightarrow{u_1} l_2 \xrightarrow{u_3} l_1, l_2 \xrightarrow{u_2} l_2, l_2 \xrightarrow{u_2} l_2 \right\rangle$
(for the sake of readability we denote an element of $\mathtt{Splt}_{\varpi}(\pi)$ by the path that is formed by the respective transitions rather than the list of indices).

Lemma 9. *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP. Let π be a path of $\Delta\mathcal{P}$. Let $0 \leq i < j < len(\pi)$. $\mathtt{Splt}_{\varpi}(\pi, i, j)$ is a splitting of $\varpi(\pi, i, j)$.*

Proof. By Definition. □

Lemma 10. *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP. Let $\pi = l_0 \xrightarrow{u_0} l_1 \xrightarrow{u_1} \ldots l_n$ be a path of $\Delta\mathcal{P}$ with $n > 0$. We have:*
1) If $l_0 \neq l_n$ then $\mathtt{hd}(\mathtt{Splt}_{\varpi}(\pi)) \in \mathcal{S}(l_0, l_n)$ else $\mathtt{hd}(\mathtt{Splt}_{\varpi}(\pi)) = \epsilon$.
2) $\mathtt{tl}(\mathtt{Splt}_{\varpi}(\pi)) \subseteq \bigcup_{l \in L} \mathcal{S}(l, l)$.

Proof. We prove the following more general claim:

Let $0 \leq i \leq j < n$. We write $\mathtt{Splt}_{\varpi}(i, j)$ as a shorthand for $\mathtt{Splt}_{\varpi}(\pi, i, j)$. If $i < j$ then
1) If $l_i \neq l_j$ then $\mathtt{Splt}_{\varpi}(i, j)(0) \in \mathcal{S}(l_i, l_j)$ else $\mathtt{Splt}_{\varpi}(i, j)(0) = \epsilon$
2) $\mathtt{Splt}_{\varpi}(i, j)(h) \in \bigcup_{l \in L} \mathcal{S}(l, l)$ for $h > 0$.

We proceed by induction over the call tree of $\mathtt{Splt}_{\varpi}(i, j)$.

Base Case) No recursive call to $\mathtt{Splt}_{\varpi}$ is triggered. This is the case if $i = j$. The claim thus holds trivially.

Step Case) We have that $i < j$. Case Distinction:
Case A) l_i appears on $\pi_{[i+1,j]}$. Let $i < k \leq j$ be the smallest k s.t. $l_k = l_i$. We have

$$\mathtt{Splt}_{\varpi}(i, j) = \mathtt{Splt}_{\varpi}(k, j) \circ \langle \langle i \rangle \star \mathtt{Splt}_{\varpi}(i + 1, k)\rangle \tag{7.1}$$

by Definition 57.
By I.H. the claim holds for $\mathtt{Splt}_{\varpi}(k, j)$ and for $\mathtt{Splt}_{\varpi}(i + 1, k)$. We therefore have:
i) If $l_k \neq l_j$ then $\mathtt{Splt}_{\varpi}(k, j)(0) \in \mathcal{S}(l_k, l_j)$ else $\mathtt{Splt}_{\varpi}(k, j)(0) = \epsilon$.
ii) $\mathtt{Splt}_{\varpi}(k, j)(h) \in \bigcup_{l \in L} \mathcal{S}(l, l)$ for $h > 0$.
iii) If $l_{i+1} \neq l_k$ then $\mathtt{Splt}_{\varpi}(i + 1, k)(0) \in \mathcal{S}(l_{i+1}, l_k)$ else $\mathtt{Splt}_{\varpi}(i + 1, k)(0) = \epsilon$.
iv) $\mathtt{Splt}_{\varpi}(i + 1, k)(h) \in \bigcup_{l \in L} \mathcal{S}(l, l)$ for $h > 0$.

A.1) We show "if $l_i \neq l_j$ then $\texttt{Splt}_\varpi(i,j)(0) \in \mathcal{S}(l_i, l_j)$ else $\texttt{Splt}_\varpi(i,j)(0) = \epsilon$":
We have that

$$\texttt{Splt}_\varpi(i,j)(0) = \texttt{Splt}_\varpi(k,j)(0) \tag{7.2}$$

with (7.1).
Case Distinction: Case $l_i \neq l_j$) With i) and $l_k = l_i \neq l_j$ and (7.2) it holds that $\texttt{Splt}_\varpi(i,j)(0) \in \mathcal{S}(l_i, l_j)$.
Case $l_i = l_j$) With i) and $l_k = l_i = l_j$ and (7.2) it holds that $\texttt{Splt}_\varpi(i,j)(0) = \epsilon$.

A.2) We show "$\texttt{Splt}_\varpi(i,j)(h) \in \bigcup_{l \in L} \mathcal{S}(l,l)$ for $h > 0$": We have that for $h > 0$

$$\texttt{Splt}_\varpi(i,j)(h) = \langle\langle i \rangle \star \texttt{Splt}_\varpi(i+1,k)\rangle\,(h-1) \tag{7.3}$$

with (7.1). By definition of the $\star$-operator we have that for $h > 0$

$$\langle\langle i \rangle \star \texttt{Splt}_\varpi(i+1,k)\rangle\,(h) = \texttt{Splt}_\varpi(i+1,k)(h)$$

and thus with iv) we have for $h > 0$ that

$$(\langle i \rangle \star \texttt{Splt}_\varpi(i+1,k))(h) \in \bigcup_{l \in L} \mathcal{S}(l,l)$$

and thus with (7.3) we have that for $h > 1$

$$\texttt{Splt}_\varpi(i,j)(h) \in \bigcup_{l \in L} \mathcal{S}(l,l)$$

.

It remains to show that $\texttt{Splt}_\varpi(i,j)(1) \in \bigcup_{l \in L} \mathcal{S}(l,l)$: We have that

$$\texttt{Splt}_\varpi(i,j)(1) \overset{(1)}{=} \langle\langle i \rangle \star \texttt{Splt}_\varpi(i+1,k)\rangle\,(0) \overset{(2)}{=} \langle i \rangle \circ \texttt{Splt}_\varpi(i+1,k)(0)$$

where (1) holds with (7.3) and (2) holds by definition the $\star$-operator (Definition 57).
Case Distinction: Case $l_k \neq l_{i+1}$) With iii) $\langle i \rangle \circ \texttt{Splt}_\varpi(i+1,k)(0) \in \mathcal{S}(l_i, l_k)$ because by assumption $l_k = l_i$ and thus the path $\langle i \rangle \circ \texttt{Splt}_\varpi(i+1,k)(0)$ remains *simple*. Thus with $l_i = l_k$ we have $\texttt{Splt}_\varpi(i,j)(1) \in \mathcal{S}(l_i, l_i)$.
Case $l_k = l_{i+1}$) With iii) $\langle i \rangle \circ \texttt{Splt}_\varpi(i+1,k)(0) = \langle i \rangle$. Thus with $\langle i \rangle \in \mathcal{S}(l_i, l_{i+1})$ and $l_i = l_k = l_{i+1}$ we have $\texttt{Splt}_\varpi(i,j)(1) \in \mathcal{S}(l_i, l_i)$.

Case B) l_i does not appear on $\pi_{[i+1,j]}$. We have

$$\texttt{Splt}_\varpi(i,j) = \langle i \rangle \star \texttt{Splt}_\varpi(i+1,j) \tag{7.4}$$

by Definition 57.
By I.H. the claim holds for $\texttt{Splt}_\varpi(i+1,j)$. We therefore have
i) If $l_{i+1} \neq l_j$ then $\texttt{Splt}_\varpi(i+1,j)(0) \in \mathcal{S}(l_{i+1}, l_j)$ else $\texttt{Splt}_\varpi(i+1,j)(0) = \epsilon$.

ii) $\texttt{Splt}_{\varpi}(i+1,j)(h) \in \bigcup_{l \in L} \mathcal{S}(l,l)$ for $h > 0$.

B.1) We show "if $l_i \neq l_j$ then $\texttt{Splt}_{\varpi}(i,j)(0) \in \mathcal{S}(l_i,l_j)$ else $\texttt{Splt}_{\varpi}(i,j)(0) = \epsilon$". Since by assumption l_i does not appear on $\pi_{[i+1,j]}$ we have that $l_i \neq l_j$. It remains to show that $\texttt{Splt}_{\varpi}(i,j)(0) \in \mathcal{S}(l_i,l_j)$: We have that

$$\texttt{Splt}_{\varpi}(i,j)(0) = \langle i \rangle \circ \texttt{Splt}_{\varpi}(i+1,j)(0) \tag{7.5}$$

with (7.4) and the definition of the $\star$-operator (Definition 57).
Case Distinction: Case $l_{i+1} \neq l_j$) By assumption l_i does not appear on $\pi_{[i+1,j]}$ and since $\varpi(i+1,j)$ is an *index permutation* of $\pi_{[i+1,j]}$ (Lemma 8), we have that l_i does also not appear on $\varpi(i+1,j)$ either. Since $\texttt{Splt}_{\varpi}(i+1,j)$ is a *splitting* of $\varpi(i+1,j)$ (Lemma 9) we have that l_i does not appear on $\texttt{Splt}_{\varpi}(i+1,j)(0)$ either and therefore we have with i) $\langle i \rangle \circ \texttt{Splt}_{\varpi}(i+1,j)(0) \in \mathcal{S}(l_i,l_j)$. Thus with (7.5) $\texttt{Splt}_{\varpi}(i,j)(0) \in \mathcal{S}(l_i,l_j)$.
Case $l_{i+1} = l_j$) With i) and (7.5) it holds that $\texttt{Splt}_{\varpi}(i,j)(0) = \langle i \rangle$ and therefore $\texttt{Splt}_{\varpi}(i,j)(0) \in \mathcal{S}(l_i,l_j)$ since $l_{i+1} = l_j$.

B.2) We show "$\texttt{Splt}_{\varpi}(i,j)(h) \in \bigcup_{l \in L} \mathcal{S}(l,l)$ for $h > 0$". We have that for $h > 0$

$$\texttt{Splt}_{\varpi}(i,j)(h) = \texttt{Splt}_{\varpi}(i+1,j)(h)$$

with (7.4) and the definition of the $\star$-operator (Definition 57). The claim thus follows with ii). □

Finally, Lemma 11 is a *helper lemma* that is used in the proofs of the main lemmas Lemma 12 and Lemma 13. The lemma states that a list $\texttt{L}$ of transitions cannot appear more often in a list of lists $\texttt{splt}(\texttt{idxp})$ of transitions, that resulted from permuting and splitting the path π, than the minimal number of times that any transition $\tau \in \texttt{L}$ appears on π.

Lemma 11. *Let $\Delta\mathcal{P}(L,E,l_b,l_e)$ be a DCP. Let π be a path of $\Delta\mathcal{P}$. Let* $\texttt{idxp}$ *be an* index permutation *of π. Let $\texttt{L} = \langle \tau_1, \tau_2, \dots \tau_k \rangle$ be a list of transitions (i.e., $\tau_i \in E$). Let* $\texttt{splt}(\texttt{idxp})$ *be a* splitting *of* $\texttt{idxp}$. *Then* $\sharp(\texttt{L}, \texttt{splt}(\texttt{idxp})) \leq \min_{\tau \in \texttt{L}} \sharp(\tau, \pi)$.

Proof. Let $\tau \in \texttt{L}$.

$$\begin{aligned}
\sharp(\texttt{L}, \texttt{splt}(\texttt{idxp})) &\overset{(1)}{\leq} \sum_{\texttt{l} \in \texttt{splt}(\texttt{idxp})} \sharp(\tau, \texttt{l}) \\
&\overset{(2)}{=} \sharp(\tau, \texttt{idxp}) \\
&\overset{(3)}{=} \sharp(\tau, \pi)
\end{aligned}$$

(1) Let $\texttt{l} \in \texttt{splt}(\texttt{idxp})$. If $\texttt{l} = \texttt{L}$ then $\sharp(\tau, \texttt{l}) > 0$ because $\tau \in \texttt{L}$ by assumption.

(2) Because $\texttt{splt}(\texttt{idxp})$ is a *splitting* of $\texttt{idxp}$.

(3) Because $\texttt{idxp}$ is an *index permutation* of π.

□

7.3.2 Essential Lemmas

Lemma 12 is a path-sensitive version of Lemma 2.

Lemma 12. *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a DCP over $\mathcal{A}$. Let $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \ldots$ be a* complete *run of $\Delta\mathcal{P}$. Let $\tau \in E$. Let $\texttt{v} \in \mathcal{V}$ be a* path sensitive local bound *for τ on $\lfloor\rho\rfloor$. Let $vb(\texttt{a}, l_1)$ denote a* upper bound invariant *for $\texttt{a}$ at l_1 on ρ where $\texttt{a} \in \mathcal{A}$ and $l_1 \in L$ s.t. $(l_1 \xrightarrow{u} l_2, \texttt{a}, \_) \in \mathcal{R}(\texttt{v})$ for some $l_2 \in L$. Then*

$$\left(\sum_{\pi \in \texttt{C}^+(\texttt{v})} \left(\min_{\tau \in trn(\pi)} \sharp(\tau, \rho)\right) \times \texttt{Sum}\mathcal{ID}(\pi)(\texttt{v})\right) + \sum_{(l_1 \xrightarrow{u} l_2, \texttt{a}, \texttt{c}) \in \mathcal{R}(\texttt{v})} \sharp(l_1 \xrightarrow{u} l_2, \rho) \times (vb(\texttt{a}, l_1) + \texttt{c})$$

is a transition bound *for τ on ρ.*

Proof. As argued in the proof of Lemma 2 it is sufficient to consider the case $\rho = \lfloor\rho\rfloor$.

A) We first show that

$$\sharp(\tau, \rho) \leq \left(\sum_{\pi \in \texttt{C}^+(\texttt{v})} \left(\min_{\tau \in trn(\pi)} \sharp(\tau, \rho)\right) \times \texttt{Sum}\mathcal{ID}(\pi)(\texttt{v})\right) + \sum_{j \in \Theta(\mathcal{R}(\texttt{v}), \rho)} \sigma_{j+1}(\texttt{v})$$

We write ϖ as shorthand for $\varpi(\rho)$ and $\texttt{Splt}_\varpi$ as a shorthand for $\texttt{Splt}_\varpi(\rho)$ (in this context ρ is interpreted as the path taken by ρ).

We have:

$$\begin{aligned}
\sharp(\tau, \rho) &\overset{(1)}{\leq} \sharp(\tau, \rho) + \sum_{i=0}^{len(\rho)-1} \sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \\
&\overset{(2a)}{=} \sharp(\tau, \rho) + \sum_{i=0}^{len(\rho)-1} \sigma_{\varpi(i)+1}(\texttt{v}) - \sigma_{\varpi(i)}(\texttt{v}) \\
&\overset{(2)}{=} \sharp(\tau, \rho) + \sum_{\texttt{l} \in \texttt{Splt}_\varpi} \sum_{i \in \texttt{l}} \sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \\
&\overset{(3a)}{\leq} \sharp(\tau, \rho) + \sum_{\texttt{l} \in \texttt{Splt}_\varpi} \sum_{i \in l \text{ with } (\rho(i), _) \in \mathcal{I}(\texttt{v}) \cup \mathcal{D}(\texttt{v})} \sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \\
&\qquad + \sum_{i \in l \text{ with } (\rho(i), _, _) \in \mathcal{R}(\texttt{v})} \sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v})
\end{aligned}$$

$$\overset{(3b)}{=} \sharp(\tau,\rho) + \left(\sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \sum_{i\in l \text{ with } (\rho(i),\_)\in\mathcal{I}(\mathtt{v})\cup\mathcal{D}(\mathtt{v})} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \right) + \sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \sum_{i\in l \text{ with } (\rho(i),\_,\_)\in\mathcal{R}(\mathtt{v})} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v})$$

$$\overset{(3c)}{\leq} \sharp(\tau,\rho) + \left(\sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \sum_{i\in l \text{ with } (\rho(i),\mathtt{c})\in\mathcal{I}(\mathtt{v})\cup\mathcal{D}(\mathtt{v})} \mathtt{c} \right) + \sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \sum_{i\in l \text{ with } (\rho(i),\_,\_)\in\mathcal{R}(\mathtt{v})} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v})$$

$$\overset{(3)}{=} \sharp(\tau,\rho) + \left(\sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v}) \right) + \sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \sum_{i\in l \text{ with } (\rho(i),\_,\_)\in\mathcal{R}(\mathtt{v})} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v})$$

$$\overset{(4a)}{\leq} \sharp(\tau,\rho) + \left(\sum_{\mathtt{l}\in\mathtt{Splt}_\varpi} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v}) \right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

$$\overset{(4b)}{=} \sharp(\tau,\rho) + \left(\sum_{\pi\in\mathcal{S}(l_b,l_e)\cup\bigcup_{l\in L}\mathcal{S}(l,l)} \sharp(\pi,\mathtt{Splt}_\varpi) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

$$\overset{(4c)}{=} \sharp(\tau,\rho) + \left(\sum_{\pi\in\mathtt{C}^-(\mathtt{v})} \sharp(\pi,\mathtt{Splt}_\varpi) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \left(\sum_{\pi\in\mathtt{C}^+(\mathtt{v})} \sharp(\pi,\mathtt{Splt}_\varpi) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

$$\overset{(4)}{\leq} \left(\sum_{\pi\in\mathtt{C}^+(\mathtt{v})} \sharp(\pi,\mathtt{Splt}_\varpi) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

$$\overset{(5)}{\leq} \left(\sum_{\pi\in\mathtt{C}^+(\mathtt{v})} (\min_{\tau\in trn(\pi)} \sharp(\tau,\rho)) \times \mathtt{Sum}\mathcal{ID}(\mathtt{v})(\pi) \right) + \sum_{j\in\Theta(\mathcal{R}(\mathtt{v}),\rho)} \sigma_{j+1}(\mathtt{v})$$

(1) In the proof of Lemma 2 we show that $\sum_{i=0}^{len(\rho)-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) > 0$.

(2a) By commutativity: With Lemma 8 ϖ is an *index permutation* of ρ.

(2) With Lemma 9 $\texttt{Splt}_\varpi$ is a *splitting* (Definition 54) of ϖ.

(3a) Let $\texttt{l} \in \texttt{Splt}_\varpi$ and $i \in \texttt{l}$. Assume $(\rho(i), \_, \_) \notin \mathcal{R}(\texttt{v})$, I.e., $\texttt{v}$ is not reset on $\rho(i)$. Then either i) $\texttt{v} \notin \texttt{def}(l_{i+1})$ or ii) $\texttt{v}' \leq \texttt{v} + \texttt{c}_i \in u_i$ for some $\texttt{c}_i \in \mathbb{Z}$.
Case i): By assumption $\rho = \lfloor \rho \rfloor$, therefore $\sigma_{i+1}(\texttt{v}) = 0$. Thus $\sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \leq 0$.
Case ii): If $\texttt{c}_i \neq 0$ then $(\rho(i), \_) \in \mathcal{I}(\texttt{v}) \cup \mathcal{D}(\texttt{v})$. Else $\sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \leq 0$. Soundness thus follows by *commutativity*.

(3b) Commutativity

(3c) For $(\rho(i), \texttt{c}) \in \mathcal{I}(\texttt{v}) \cup \mathcal{D}(\texttt{v})$ we have that $\sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v}) \leq \texttt{c}$ (Definition 19 and Definition 34).

(3) Let $\texttt{l} \in \texttt{Splt}_\varpi$. Let $\texttt{l} = \langle i_1, i_2, \dots i_k \rangle$. With Lemma 10 and $l_b \neq l_e$ (note that $\Delta\mathcal{P}$ is *well-defined* and ρ is *complete*) we have that the transitions $i_1(\rho), i_2(\rho) \dots, i_k(\rho)$ form a non-empty path of $\Delta\mathcal{P}$. Note that in the notation $\texttt{Sum}\mathcal{ID}(\texttt{l})(\texttt{v})$ the list $\texttt{l}$ is interpreted as representing this path. Soundness thus follows from Definition 35:
$$\sum_{i \in l \text{ with } (\rho(i), \texttt{c}) \in \mathcal{I}(\texttt{v}) \cup \mathcal{D}(\texttt{v})} \texttt{c} = \texttt{Sum}\mathcal{ID}(\texttt{l})(\texttt{v})$$

(4a) With $\sigma_i(\texttt{v}) \geq 0$ for all i we have that $\sigma_{i+1}(\texttt{v}) \geq \sigma_{i+1}(\texttt{v}) - \sigma_i(\texttt{v})$.

(4b) With Lemma 10 and $l_b \neq l_e$ (note that $\Delta\mathcal{P}$ is *well-defined* and ρ is *complete*) we have that $\texttt{Splt}_\varpi \subseteq \mathcal{S}(l_b, l_e) \cup \bigcup_{l \in L} \mathcal{S}(l, l)$. Recall that $\sharp(\pi, \texttt{Splt}_\varpi)$ counts how often π occurs in the list $\texttt{Splt}_\varpi$ (Definition 54). Here we interpret an indice list in $\texttt{Splt}_\varpi$ as the path formed by the respective transitions.

(4c) Let $\pi \in \mathcal{S}(l_b, l_e) \cup \bigcup_{l \in L} \mathcal{S}(l, l)$. If $\texttt{Sum}\mathcal{ID}(\pi)(\texttt{v}) \neq 0$ then $\texttt{Sum}\mathcal{ID}(\pi)(\texttt{v}) \in \texttt{C}^+(\texttt{v})$ or $\texttt{Sum}\mathcal{ID}(\pi)(\texttt{v}) \in \texttt{C}^-(\texttt{v})$ (Definition 35).

(4) a) We show that $\sharp(\tau, \rho) \leq \sum_{\pi \in \texttt{C}^-(\texttt{v})} \sharp(\pi, \texttt{Splt}_\varpi)$:

$$\begin{aligned}
\sharp(\tau, \rho) &\overset{(i)}{=} |\{0 \leq i < len(\rho) \mid \rho(i) = \tau\}| \\
&\overset{(ii)}{=} |\{0 \leq i < len(\rho) \mid \rho(\varpi(i)) = \tau\}| \\
&\overset{(iii)}{=} \sum_{\texttt{l} \in \texttt{Splt}_\varpi} \sharp(\tau, \texttt{l}) \\
&\overset{(iv)}{=} \sum_{\texttt{l} \in \texttt{Splt}_\varpi} ITE(\texttt{l} \in C(\tau), 1, 0) \\
&\overset{(v)}{=} \sum_{\pi \in C(\tau)} \sharp(\pi, \texttt{Splt}_\varpi) \\
&\overset{(vi)}{\leq} \sum_{\pi \in \texttt{C}^-(\texttt{v})} \sharp(\pi, \texttt{Splt}_\varpi)
\end{aligned}$$

(i) Definition 15.
(ii) ϖ is an *index permutation* of ρ.
(iii) $\mathtt{Splt}_{\varpi}$ is a *splitting* of ϖ.
(iv) Let $\mathtt{l} \in \mathtt{Splt}_{\varpi}$. By Lemma 10 and $l_b \neq l_e$ (note that $\Delta\mathcal{P}$ is *well-defined* and ρ is *complete*) we have that the indices in $\mathtt{l}$ form a non-empty *simple* path of $\Delta\mathcal{P}$. Thus $\sharp(\tau, \mathtt{l}) = 1$ iff $\mathtt{l} \in C(\tau)$ and $\sharp(\tau, \mathtt{l}) = 0$ else.
(v) Definition 54.
(vi) We have that $C(\tau) \subseteq \mathtt{C}^-(\mathtt{v})$ because by assumption $\mathtt{v}$ is a *path sensitive local bound* for τ (Definition 38).
Thus $\sharp(\tau, \rho) - \left(\sum_{\pi \in \mathtt{C}^-(\mathtt{v})} \sharp(\pi, \mathtt{Splt}_{\varpi}) \right) \leq 0$.
b) Soundness of (4) now follows from
$\sum_{\pi \in \mathtt{C}^-(\mathtt{v})} \sharp(\pi, \mathtt{Splt}_{\varpi}) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \leq - \sum_{\pi \in \mathtt{C}^-(\mathtt{v})} \sharp(\pi, \mathtt{Splt}_{\varpi})$
because $\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) < 0$ for $\pi \in \mathtt{C}^-(\mathtt{v})$ (Definition 35).

(5) Let π be a path of $\Delta\mathcal{P}$. By Lemma 11 we have that $\sharp(\pi, \mathtt{Splt}_{\varpi}) \leq \min_{\tau \in \pi} \sharp(\tau, \rho)$.

B) By reasoning analogous to step B in the proof of Lemma 2 we get:

$$\sum_{j \in \Theta(\mathcal{R}(\mathtt{v}), \rho)} \sigma_{j+1}(\mathtt{v}) \leq \sum_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(t, \rho) \times (vb(\mathtt{a}, l_1) + \mathtt{c})$$

With A) and B) we have

$$\sharp(\tau, \rho) \leq \left(\sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{t \in trn(\pi)} \sharp(t, \rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} \sharp(l_1 \xrightarrow{u} l_2, \rho) \times (vb(\mathtt{a}, l_1) + \mathtt{c})$$

□

Lemma 13 is a path-sensitive version of Lemma 3.

Lemma 13. *Let* $\mathtt{v} \in \mathcal{V}$. *Let* $l \in L$ *be s.t.* $\mathtt{v} \in \mathtt{def}(l)$. *Let* $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \ldots$ *be a run of* $\Delta\mathcal{P}$. *Let* $vb(\mathtt{a}, l_1)$ *denote an* upper bound invariant *for* $\mathtt{a}$ *at* l_1 *on* ρ *where* $\mathtt{a} \in \mathcal{A}$ *and* $l_1 \in L$ *s.t.* $(l_1 \xrightarrow{u} l_2, \mathtt{a}, \_) \in \mathcal{R}(\mathtt{v})$ *for some* $l_2 \in L$. *Then*

$$\left(\max_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} vb(\mathtt{a}, l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{\tau \in trn(\pi)} \sharp(\tau, \rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

is an upper bound invariant *for* $\mathtt{v}$ *at* l *on* ρ. *We assume* $\max_{\pi \in \mathcal{S}_a(l_2, l)} \cdots = 0$ *for* $\mathcal{S}_a(l_2, l) = \emptyset$.

Proof. We have to show that

$$\sigma_i(\mathtt{v}) \leq \left(\max_{(l_1 \xrightarrow{u} l_2, \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} vb(\mathtt{a}, l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} \left(\min_{\tau \in trn(\pi)} \sharp(\tau, \rho) \right) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

for all $0 \leq i \leq len(\rho)$ with $l_i = l$ (Definition 25).

Let $0 \leq m \leq len(\rho)$ be s.t. $l_m = l$. Since $\mathtt{v} \in \mathtt{def}(l)$ (by assumption) we have by *well-definedness* of $\Delta\mathcal{P}$ that there is a $0 \leq h < m$, a $\mathtt{a} \in \mathcal{A}$ and a $\mathtt{c} \in \mathbb{Z}$ s.t. $(\rho(h), \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$ and $\mathtt{v}$ is not reset on $\rho_{[h+1,m]}$, i.e., for all $h < i < m$ $(\rho(i), \_, \_) \notin \mathcal{R}(\mathtt{v})$. In other words: There is a maximal index $h < m$ such that $\mathtt{v}$ is reset on $\rho(h)$.

We write $\varpi(i, j)$ as shorthand for $\varpi(\rho, i, j)$ and $\mathtt{Splt}_\varpi(i, j)$ as a shorthand for $\mathtt{Splt}_\varpi(\rho, i, j)$ (in this context ρ is interpreted as the path taken by ρ).

We have:

$$\begin{aligned}
\sigma_m(\mathtt{v}) &\overset{(1a)}{=} \sigma_{h+1}(\mathtt{v}) + \sum_{i=h+1}^{m-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \\
&\overset{(1b)}{=} \sigma_{h+1}(\mathtt{v}) + \sum_{i=0}^{m-h-2} \sigma_{\varpi(h+1,m)(i)+1}(\mathtt{v}) - \sigma_{\varpi(h+1,m)(i)}(\mathtt{v}) \\
&\overset{(1c)}{=} \sigma_{h+1}(\mathtt{v}) + \sum_{\mathtt{l} \in \mathtt{Splt}_\varpi(h+1,m) \setminus \{\epsilon\}} \sum_{i \in \mathtt{l}} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \\
&\overset{(1)}{\leq} \sigma_{h+1}(\mathtt{v}) + \sum_{\mathtt{l} \in \mathtt{Splt}_\varpi(h+1,m) \setminus \{\epsilon\}} \sum_{i \in l \text{ with } (\rho(i), \mathtt{c}) \in \mathcal{I}(\mathtt{v}) \cup \mathcal{D}(\mathtt{v})} \mathtt{c} \\
&\overset{(2a)}{=} \sigma_{h+1}(\mathtt{v}) + \sum_{\mathtt{l} \in \mathtt{Splt}_\varpi(h+1,m) \setminus \{\epsilon\}} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v}) \\
&\overset{(2b)}{=} \sigma_{h+1}(\mathtt{v}) + \mathtt{Sum}\mathcal{ID}(\mathtt{hd}(\mathtt{Splt}_\varpi(h+1,m)))(\mathtt{v}) + \sum_{\mathtt{l} \in \mathtt{tl}(\mathtt{Splt}_\varpi(h+1,m))} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v}) \\
&\overset{(2)}{\leq} \sigma_{h+1}(\mathtt{v}) + \max_{\pi \in \mathcal{S}_a(l_{h+1}, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) + \sum_{\mathtt{l} \in \mathtt{tl}(\mathtt{Splt}_\varpi(h+1,m))} \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v}) \\
&\overset{(3a)}{=} \sigma_{h+1}(\mathtt{v}) + \max_{\pi \in \mathcal{S}_a(l_{h+1}, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\quad + \sum_{\pi \in \bigcup_{l \in L} \mathcal{S}(l,l)} \sharp(\pi, \mathtt{tl}(\mathtt{Splt}_\varpi(h+1,m))) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\overset{(3b)}{\leq} \sigma_{h+1}(\mathtt{v}) + \max_{\pi \in \mathcal{S}_a(l_{h+1}, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\quad + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} \sharp(\pi, \mathtt{tl}(\mathtt{Splt}_\varpi(h+1,m))) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})
\end{aligned}$$

$$\overset{(3)}{\leq} \sigma_{h+1}(\mathtt{v}) + \max_{\pi \in \mathcal{S}_a(l_{h+1},l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{\tau \in trn(\pi)} \sharp(\tau,\rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

$$\overset{(4a)}{\leq} \sigma_h(\mathtt{a}) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_{h+1},l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{\tau \in trn(\pi)} \sharp(\tau,\rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

$$\overset{(4b)}{\leq} vb(\mathtt{a},l_h) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_{h+1},l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{\tau \in trn(\pi)} \sharp(\tau,\rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

$$\overset{(4)}{\leq} \left(\max_{(l_1 \xrightarrow{u} l_2,\mathtt{a},\mathtt{c}) \in \mathcal{R}(\mathtt{v})} vb(\mathtt{a},l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2,l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} (\min_{\tau \in trn(\pi)} \sharp(\tau,\rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$$

(1a) We have $\sum_{i=h+1}^{m-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) = \sigma_m(\mathtt{v}) - \sigma_{h+1}(\mathtt{v})$.
Thus $\sigma_{h+1}(\mathtt{v}) + \sum_{i=h+1}^{m-1} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) = \sigma_m(\mathtt{v})$.

(1b) By commutativity: With Lemma 8 $\varpi(h+1,m)$ is an *index permutation* of $\rho_{[h+1,m]}$.

(1c) With Lemma 9 $\mathtt{Splt}_\varpi(h+1,m)$ is a *splitting* (Definition 54) of $\varpi(h+1,m)$.

(1) By assumption we have that for all $h < i < m$ $(\rho(i),\_,\_) \notin \mathcal{R}(\mathtt{v})$. By well-definedness of $\Delta\mathcal{P}$ we have that for all $h < i \leq m$ $\mathtt{v} \in \mathtt{def}(l_i)$. Thus for all $h < i < m$ there is a $\mathtt{c}_i \in \mathbb{Z}$ s.t. $\mathtt{v}' \leq \mathtt{v} + \mathtt{c}_i \in u_i$. Let $\mathtt{l} \in \mathtt{Splt}_\varpi(h+1,m)$. Since $\varpi(h+1,m)$ is an *index permutation* of $\rho_{[h+1,m]}$ and $\mathtt{Splt}_\varpi(h+1,m)$ is a splitting of $\varpi(h+1,m)$ we have that $\sum_{i \in \mathtt{l}} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \leq \sum_{i \in \mathtt{l}} \mathtt{c}_i$ (Definition 5). Observe that if $\mathtt{c}_i \neq 0$ then $(\rho(i),\mathtt{c}_i) \in \mathcal{I}(\mathtt{v})$ or $(\rho(i),\mathtt{c}_i) \in \mathcal{D}(\mathtt{v})$. Further note that since $\Delta\mathcal{P}$ is *fan-in free* there is exactly one such $\mathtt{c}_i$ for each $h < i < m$. Thus $\sum_{i \in \mathtt{l}} \sigma_{i+1}(\mathtt{v}) - \sigma_i(\mathtt{v}) \leq \sum_{i \in l \text{ with } (\rho(i),\mathtt{c}) \in \mathcal{I}(\mathtt{v}) \cup \mathcal{D}(\mathtt{v})} \mathtt{c}$

(2a) Let $\mathtt{l} \in \mathtt{Splt}_\varpi(h+1,m)$. Let $\mathtt{l} = \langle i_1, i_2, \ldots i_k \rangle \neq \epsilon$. With Lemma 10 we have that the transitions $i_1(\rho), i_2(\rho) \ldots, i_k(\rho)$ form a path of $\Delta\mathcal{P}$. Note that in the notation $\mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v})$ $\mathtt{l}$ is interpreted as representing this path. Soundness thus follows from Definition 35: $\sum_{i \in l \text{ with } (\rho(i),\mathtt{c}) \in \mathcal{I}(\mathtt{v}) \cup \mathcal{D}(\mathtt{v})} \mathtt{c} = \mathtt{Sum}\mathcal{ID}(\mathtt{l})(\mathtt{v})$

(2b) Let $\mathtt{L}$ be a *list*. Then $\langle \mathtt{hd}(\mathtt{L}) \rangle \circ \mathtt{tl}(\mathtt{L}) = \mathtt{L}$ (Definition 54). We assume that $\mathtt{Sum}\mathcal{ID}(\epsilon)(\mathtt{v}) = 0$.

(2) Case Distinction: Case $[l_{h+1} \neq l_m]$ With Lemma 10 we have that $\mathtt{hd}(\mathtt{Splt}_\varpi(h+1,m)) \in \mathcal{S}(l_{h+1},l_m)$. By assumption $l_m = l$. Thus $\mathtt{Sum}\mathcal{ID}(\mathtt{hd}(\mathtt{Splt}_\varpi(h+1,m)))(\mathtt{v}) \leq \max_{\pi \in \mathcal{S}_a(l_{h+1},l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})$.
Case $[l_{h+1} = l_m]$ With Lemma 10 we have that $\mathtt{hd}(\mathtt{Splt}_\varpi(h+1,m)) = \epsilon$. Thus

$\mathtt{Sum}\mathcal{ID}(\mathtt{hd}(\mathtt{Splt}_{\varpi}(h+1,m)))(\mathtt{v}) = 0$ (by assumption). With $l_{h+1} = l_m = l$ we have $\mathcal{S}_a(l_{h+1}, l) = \emptyset$ and thus $\max_{\pi \in \mathcal{S}_a(l_{h+1},l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) = 0$ (by assumption).

(3a) With Lemma 10 we have that $\mathtt{tl}(\mathtt{Splt}_{\varpi}(h+1,m)) \subseteq \bigcup_{l \in L} \mathcal{S}(l,l)$. Recall that $\sharp(\pi, \mathtt{tl}(\mathtt{Splt}_{\varpi}(h+1,m)))$ counts how often π occurs in the list $\mathtt{tl}(\mathtt{Splt}_{\varpi}(h+1,m))$ (Definition 54). Here we interpret an indice list in $\mathtt{Splt}_{\varpi}(h+1,m)$ as the path formed by the respective transitions.

(3b) Let $\pi \in \bigcup_{l \in L} \mathcal{S}(l,l)$. We have that $\pi \in \mathtt{C}^+(\mathtt{v})$ if $\mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) > 0$ (Definition 35).

(3) Let $\pi \in \mathtt{C}^+(\mathtt{v})$. We have that $\sharp(\pi, \mathtt{Splt}_{\varpi}(h+1,m)) \leq \min_{\tau \in \pi} \sharp(\tau, \rho_{[h+1,m]})$ by Lemma 11. Soundness follows from i) $\sharp(\tau, \rho_{[h+1,m]}) \leq \sharp(\tau, \rho)$ for $\tau \in E$ and ii) $\sharp(\pi, \mathtt{tl}(\mathtt{Splt}_{\varpi}(h+1,m))) \leq \sharp(\pi, \mathtt{Splt}_{\varpi}(h+1,m))$.

(4a) By assumption $(\rho(h), \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$. Thus $\sigma_{h+1}(\mathtt{v}) \leq \sigma_h(\mathtt{a}) + \mathtt{c}$ (Definition 19), note that $\Delta\mathcal{P}$ is *fan-in free* by assumption.

(4b) By assumption $vb(\mathtt{a}, l_h)$ is a *upper bound invariant* for $\mathtt{a}$ at l_h since $(l_h \xrightarrow{u_h} l_{h+1}, \mathtt{a}, \_) \in \mathcal{R}(\mathtt{v})$ by definition of h.

(4) By assumption $(\rho(h), \mathtt{a}, \mathtt{c}) \in \mathcal{R}(\mathtt{v})$. Soundness thus follows by semantics of 'max'.

□

7.3.3 Proof of Theorem 7

We show the more general claim formulated in Theorem 10. Note that by Lemma 1 in particular the set of *complete* runs of $\Delta\mathcal{P}$ is closed under *normalization*.

Theorem 10. *Let $\Delta\mathcal{P}(L, E, l_b, l_e)$ be a well-defined and fan-in free DCP over atoms $\mathcal{A}$. Let Ξ be a set of* complete *runs of $\Delta\mathcal{P}$ that is closed under* normalization*. Let $\zeta : E \mapsto Expr(\mathcal{A})$ be a* path sensitive local bound mapping *for all $\rho \in \Xi$. Let $\mathtt{a} \in \mathcal{A}$ and $\tau \in E$. Let $l \in L$ be s.t. $\mathtt{a} \in \mathtt{def}(l)$. Let $\rho \in \Xi$. Let σ_0 denote the initial state of ρ. Let $T\mathcal{B}(\tau)$ and $V\mathcal{B}(\mathtt{a}, l)$ be as defined in Definition 40. We have: (I) $[\![T\mathcal{B}(\tau)]\!](\sigma_0)$ is a* transition bound *for τ on ρ. (II) $[\![V\mathcal{B}(\mathtt{a}, l)]\!](\sigma_0)$ is an* upper bound invariant *for $\mathtt{a}$ at l on ρ.*

Proof. Let $\rho = (\sigma_0, l_0) \xrightarrow{u_0} (\sigma_1, l_1) \xrightarrow{u_1} \ldots (\sigma_n, l_e) \in \Xi$.

If $[\![T\mathcal{B}(\tau)]\!] = \infty$ (I) holds trivially. If $[\![V\mathcal{B}(\mathtt{a}, l)]\!] = \infty$ (II) holds trivially.

Assume $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ and $[\![V\mathcal{B}(\mathtt{a}, l)]\!] \neq \infty$. Then in particular the computations of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a}, l)$ terminate. We proceed by induction over the call tree of $T\mathcal{B}(\tau)$ resp. $V\mathcal{B}(\mathtt{a}, l)$.

Base Case:

(I) No function call is triggered when computing $V\mathcal{B}(\mathtt{a}, l)$. This is the case iff $\mathtt{a} \in \mathcal{C}$ (Definition 40). Then $V\mathcal{B}(\mathtt{a}, l) = \mathtt{a}$ and the claim holds trivially with $\mathtt{a} \in \mathcal{C}$.
(II) No function call is triggered when computing $T\mathcal{B}(\tau)$. This is the case iff $\zeta(\tau) \notin \mathcal{V}$ (Definition 40). Then $[\![T\mathcal{B}(\tau)]\!](\sigma_0) = [\![\zeta(\tau)]\!](\sigma_0)$ is a *transition bound* for τ on ρ by Definition 18.

Step Case:

(I) $\mathtt{a} \notin \mathcal{C}$, thus $\mathtt{a} \in \mathcal{V}$. Let $\mathtt{v} = \mathtt{a}$. By assumption $\mathtt{v} \in \mathtt{def}(l)$. Let $0 \leq i \leq \mathit{len}(\rho)$ be s.t. $l_i = l$. We have to show that $\sigma_i(\mathtt{v}) \leq [\![V\mathcal{B}(\mathtt{a}, l)]\!](\sigma_0)$ holds for all $0 \leq i \leq \mathit{len}(\rho)$ with $l_i = l$ (Definition 25).

We have:

$$
\begin{aligned}
\sigma_i(\mathtt{v}) &\overset{(1)}{\leq} \left(\max_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} [\![V\mathcal{B}(b, l_1)]\!](\sigma_0) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) \\
&\quad + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} \left(\min_{t \in trn(\pi)} \sharp(t, \rho) \right) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\overset{(2)}{\leq} \left(\max_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} [\![V\mathcal{B}(b, l_1)]\!](\sigma_0) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) \\
&\quad + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} \left(\min_{t \in trn(\pi)} [\![T\mathcal{B}(\tau)]\!](\sigma_0) \right) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\overset{(3a)}{=} \left(\max_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} [\![V\mathcal{B}(b, l_1)]\!](\sigma_0) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) \\
&\quad + \sum_{\pi \in \mathtt{C}^+(\mathtt{v})} [\![T\mathcal{B}(trn(\pi))]\!](\sigma_0) \times \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \\
&\overset{(3b)}{=} \left(\max_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} [\![V\mathcal{B}(b, l_1)]\!](\sigma_0) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v}) \right) + [\![\mathtt{Incr}(\mathtt{v})]\!](\sigma_0) \\
&\overset{(3c)}{=} [\![\mathtt{Incr}(\mathtt{v})]\!](\sigma_0) + [\![\max_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\mathtt{v})} V\mathcal{B}(b, l_1) + \mathtt{c} + \max_{\pi \in \mathcal{S}_a(l_2, l)} \mathtt{Sum}\mathcal{ID}(\pi)(\mathtt{v})]\!](\sigma_0) \\
&\overset{(3)}{=} [\![V\mathcal{B}(\mathtt{v})]\!](\sigma_0)
\end{aligned}
$$

(1) By Lemma 13: Let $(l_1 \xrightarrow{u} l_2, b, \_) \in \mathcal{R}(\mathtt{v})$. We have that $V\mathcal{B}(b, l_1)$ is recursively called when computing $V\mathcal{B}(\mathtt{v}, l)$ (Definition 40). With $[\![V\mathcal{B}(\mathtt{v}, l)]\!] \neq \infty$ also $[\![V\mathcal{B}(b, l_1)]\!] \neq \infty$. Since $b \in \mathtt{def}(l_1)$ by *well-definedness* of $\Delta\mathcal{P}$ we thus have by I.H. that $V\mathcal{B}(b, l_1)$ is an *upper bound invariant* for b at l_1 on ρ.

(2) Let $\pi \in \mathtt{C}^+(\mathtt{v})$. Let $t \in trn(\pi)$. We have that $T\mathcal{B}(t)$ is called when computing $V\mathcal{B}(\mathtt{v})$ (Definition 40). Let $t \in trn(\pi)$ be s.t. $[\![T\mathcal{B}(t)]\!] \neq \infty$. Since $[\![V\mathcal{B}(\mathtt{v})]\!] \neq \infty$ we have that such a $t \in trn(\pi)$ exists. By I.H. $\sharp(t,\rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$. Therefore $\min_{t \in trn(\pi)} \sharp(t,\rho) \leq \min_{t \in trn(\pi)} [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(3a) With Definition 40 and Definition 17 we have that $\min_{t \in trn(\pi)} [\![T\mathcal{B}(\tau)]\!](\sigma_0) = [\![T\mathcal{B}(trn(\pi))]\!](\sigma_0)$.

(3b) Definition 40 and Definition 17

(3c) Commutativity and Definition 17

(3) Definition 40 and Definition 17

(II) $\zeta(\tau) \in \mathcal{V}$ (Definition 40).

We have:

$$
\begin{aligned}
\sharp(\tau,\rho) &\overset{(1)}{\leq} \left(\sum_{\pi \in \mathtt{C}^+(\zeta(\tau))} (\min_{t \in trn(\pi)} \sharp(t,\rho)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\zeta(\tau)) \right) \\
&\quad + \sum_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} \sharp(l_1 \xrightarrow{u} l_2, \rho) \times ([\![V\mathcal{B}(b,l_1)]\!](\sigma_0) + \mathtt{c}) \\
&\overset{(2)}{\leq} \left(\sum_{\pi \in \mathtt{C}^+(\zeta(\tau))} (\min_{t \in trn(\pi)} [\![T\mathcal{B}(t)]\!](\sigma_0)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\zeta(\tau)) \right) \\
&\quad + \sum_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} \sharp(l_1 \xrightarrow{u} l_2, \rho) \times ([\![V\mathcal{B}(b,l_1)]\!](\sigma_0) + \mathtt{c}) \\
&\overset{(3)}{\leq} \left(\sum_{\pi \in \mathtt{C}^+(\zeta(\tau))} (\min_{t \in trn(\pi)} [\![T\mathcal{B}(t)]\!](\sigma_0)) \times \mathtt{Sum}\mathcal{ID}(\pi)(\zeta(\tau)) \right) \\
&\quad + \sum_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} [\![T\mathcal{B}(l_1 \xrightarrow{u} l_2)]\!](\sigma_0) \times ([\![V\mathcal{B}(b,l_1)]\!](\sigma_0) + \mathtt{c}) \\
&\overset{(4a)}{=} \left(\sum_{\pi \in \mathtt{C}^+(\zeta(\tau))} [\![T\mathcal{B}(trn(\pi))]\!](\sigma_0) \times \mathtt{Sum}\mathcal{ID}(\pi)(\zeta(\tau)) \right) \\
&\quad + \sum_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} [\![T\mathcal{B}(l_1 \xrightarrow{u} l_2)]\!](\sigma_0) \times ([\![V\mathcal{B}(b,l_1)]\!](\sigma_0) + \mathtt{c}) \\
&\overset{(4b)}{=} [\![\mathtt{Incr}(\zeta(\tau))]\!](\sigma_0) + \sum_{(l_1 \xrightarrow{u} l_2, b, \mathtt{c}) \in \mathcal{R}(\zeta(\tau))} [\![T\mathcal{B}(l_1 \xrightarrow{u} l_2)]\!](\sigma_0) \times ([\![V\mathcal{B}(b,l_1)]\!](\sigma_0) + \mathtt{c}) \\
&\overset{(4)}{=} [\![T\mathcal{B}(\tau)]\!](\sigma_0)
\end{aligned}
$$

(1) By Lemma 12: Since Ξ is closed under *normalization* we have that $\zeta(\tau)$ is a *local bound* for τ on $\lfloor\rho\rfloor$ as well. Let $(l_1 \xrightarrow{u} l_2, b, \_) \in \mathcal{R}(\zeta(\tau))$. We have that $V\mathcal{B}(b, l_1)$ is called during the computation of $T\mathcal{B}(\tau)$ (Definition 40). With $T\mathcal{B}(\tau) \neq \infty$ also $V\mathcal{B}(b, l_1) \neq \infty$. Since $b \in \texttt{def}(l_1)$ by well-definedness of $\Delta\mathcal{P}$ we thus have by I.H. that $V\mathcal{B}(b)$ is an *upper bound invariant* for b at l_1 on ρ.

(2) Let $\pi \in \texttt{C}^+(\zeta(\tau))$. Let $t \in trn(\pi)$. We have that $T\mathcal{B}(t)$ is called when computing $T\mathcal{B}(\tau)$ (Definition 40). Let $t \in trn(\pi)$ be s.t. $[\![T\mathcal{B}(t)]\!] \neq \infty$. Since $[\![T\mathcal{B}(\tau)]\!] \neq \infty$ we have that such a $t \in trn(\pi)$ exists. By I.H. $\sharp(t, \rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$. Therefore $\min_{t \in trn(\pi)} \sharp(t, \rho) \leq \min_{t \in trn(\pi)} [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(3) Let $(t, \_, \_) \in \mathcal{R}(\zeta(\tau))$. We have that $T\mathcal{B}(t)$ is recursively called during the computation of $T\mathcal{B}(\tau)$ (Definition 40). With $T\mathcal{B}(\tau) \neq \infty$ also $T\mathcal{B}(t) \neq \infty$. By I.H. $\sharp(t, \rho) \leq [\![T\mathcal{B}(t)]\!](\sigma_0)$.

(4a) With Definition 40 and Definition 17 we have that $\min_{t \in trn(\pi)} [\![T\mathcal{B}(\tau)]\!](\sigma_0) = [\![T\mathcal{B}(trn(\pi))]\!](\sigma_0)$.

(4b) Definition 40 and Definition 17

(4) Definition 40 and Definition 17

□

List of Figures

List of Tables

List of Definitions

Bibliography

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.

[ABG12] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *SAS*, pages 405–421, 2012.

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133, 2010.

[AGM13] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Trans. Comput. Log.*, 14(3):22, 2013.

[BAL07] Amir M Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):5, 2007.

[BEF+16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Juergen Giesl. Analyzing runtime and size complexity of integer programs. ACM, 2016.

[Ben08] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3), 2008.

[BHHK10] Régis Blanc, Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: algebraic bound computation for loops. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 103–118. Springer, 2010.

[BM99] Ahmed Bouajjani and Richard Mayr. Model checking lossy vector addition systems. In *STACS 99*, pages 323–333. Springer, 1999.

[BMPZ12] Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215:47–67, 2012.

[BMS05] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

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

[CDF12] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI*, pages 89–98, 2012.

[CDZ14] Thomas Colcombet, Laure Daviaud, and Florian Zuleger. Size-change abstraction and max-plus automata. In *MFCS*, pages 208–219, 2014.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[ČHK+15] Pavol Černỳ, Thomas A Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment abstraction for worst-case execution time analysis. In *European Symposium on Programming Languages and Systems*, pages 105–131. Springer, 2015.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. PLDI, 2015.

[CLRS01] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[DU15] Vijay D'Silva and Caterina Urban. Conflict-driven conditional termination. In *International Conference on Computer Aided Verification*, pages 271–286. Springer, 2015.

[FH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *APLAS*, pages 275–295, 2014.

[GG08] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384, 2008.

[GJ09] Sumit Gulwani and Sudeep Juvekar. Bound analysis using backward symbolic execution. Technical Report MSR-TR-2004-95, Microsoft Research, 2009.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[GLAS09] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[KKZ11] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 227–242. Springer, 2011.

[KNP+10] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 17–26. ACM, 2010.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

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

[looa] `http://forsyte.at/software/loopus/`.

[loob] `http://forsyte.at/static/people/sinn/loopusJAR/`.

[looc] `http://forsyte.at/static/people/sinn/loopusPhD/`.

[Mét88] Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

[MV06] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *CAV*, pages 401–414, 2006.

[NJT13] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2013.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *FPCA*, pages 144–156, 1989.

[SGS14] Helmut Seidl, Thomas Martin Gawlitza, and Martin Schwarz. Parametric strategy iteration. In Temur Kutsia and Andrei Voronkov, editors, *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, volume 30 of *EPiC Series in Computing*, pages 62–76. EasyChair, 2014.

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In *FOSSACS*, pages 288–302, 2009.

[spe] `https://www.spec.org/cpu2006/`.

[SZV14a] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, pages 745–761. Springer, 2014.

[SZV14b] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. *arXiv preprint arXiv:1401.5842*, 2014.

[SZV15] Moritz Sinn, Florian Zuleger, and Helmut Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *FMCAD*, pages 144–151, 2015.

[SZV16] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning*, page to appear, 2016.

[Tar85] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985.

[Tur36] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[ZAH11] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102. ACM, 2011.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.

[ZH12] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76, 2012.

[ZZ89] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. 1989.