Resource Analysis of Imperative Programs

dissertation

by

Michael Schaper

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

advisor: Assoc. Prof. Dr. Georg Moser

Innsbruck, 16 September 2019

universität
innsbruck

dissertation

Resource Analysis of Imperative Programs

Michael Schaper (0715638)
michael.schaper@student.uibk.ac.at

16 September 2019

advisor: Assoc. Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

This thesis is devoted to resource analysis of imperative programs. Resource analysis falls within the wide spectrum of static program analysis, which is concerned with automatic methods for inferring reliable approximate informations about the dynamic behaviour of computer programs. In resource analysis, we are concerned with approximations of quantitative properties of program executions, such as the maximal number of execution steps or memory needed. This topic is an active area of research and several resource analysis tools have been established in recent years. In this work, we provide the following contributions.

First, we are concerned with the resource analysis of imperative programs in which states are formed over a finite set of integer-valued variables. We consider a standard abstract model of computation, so-called constraint transition systems. When syntactically restricting the precise notion of the program representation, one can study and implement dedicated techniques for resource analysis. In this work, we give an overview of theoretical properties and practical aspects of relevant abstract program representations that are known from the literature and that are used in modern resource analysis tools.

Second, we are concerned with the resource analysis of imperative programs with heap allocated data structures. Inspired by recent developments on automating runtime analysis for term rewriting, we present a term abstraction of programs with heap allocated data. This abstraction safely approximates the runtime of the target program such that we can make use of existing tools for the resource analysis. In this work, we outline the term abstraction and compare it to known approaches.

Third, we are concerned with the resource analysis of probabilistic programs. We consider a standard imperative programming language that is endowed with probabilistic primitives for sampling and probabilistic choice. In this work, we present a novel modular approach to automate the resource analysis of probabilistic programs.

Finally, we present a framework for automation. We provide a Haskell library that is dedicated to automate resource analysis. In this work, we outline the software architecture of this library and demonstrate several case studies in which the framework has been applied successfully.

Acknowledgments

At the beginning of this thesis I want to express my gratitude to all people who have influenced my work and supported me on this journey.

Foremost, I would like to express my special appreciation and thanks to my supervisor Georg Moser for his guidance throughout my studies. He sparked my interest in theoretical computer science and taught me scientific working. This dissertation would not have been possible without his support and encouraging words.

I am especially grateful to Martin Avanzini. I have learned a lot from our numerous discussions and working together.

I would like to thank Aart Middeldorp and all members of the Computational Logic research group, both past and present, for providing a pleasant and fruitful work environment over the past years. My sincere thanks go to René Thieman for being my second supervisor. I explicitly mention Thibault Gauthier, Stéphane Gimenez, Alexander Maringele, Kenji Miyamoto, Julian Nagele, David Obwaller, Thomas Powell, Maria Schett, Thomas Sternagel, and Sarah Winkler. I appreciate the numerous social events and joyful activities we have engaged after work.

I am grateful to all of my friends who silently endure my occasional rants. Finally, I would like to express my gratitude to my parents Ingrid and Knut for their continuous support and encouragement.

Contents

Chapter 1

Introduction

In this day and age, pocket sized computers accompany us in our daily lives, robots explore distant planets, and autonomous cars cross our ways on the streets. The technology in this digital age is controlled by computer programs. Unpredicted behaviour of programs and software errors can lead to fatal consequences. *Static program analysis* is a research area that is concerned with determining safe and computable approximations on the dynamic behaviour of programs without executing them. It is well-known that interesting program properties, such as, whether a program *terminates* on any input, can not be decided in general. Nevertheless, program analysis is one of the major tools that we have in our repertoire to exclude unwanted behaviour and establish reliable software.

Resource analysis (sometimes also referred to as cost analysis or complexity analysis) is a part of program analysis that focuses on *quantitative* properties of program executions. Traditional properties of interest include the number of execution steps, the memory used or the number of function calls. Resource analysis of programs is an active area of research. In recent years, several approaches have been investigated and implemented, to mention a few, Absynth [124], AProVe [82], C4B [53], CiaoPP [123, 143], CoFloCo [72], COSTA [1, 4], HoCA [16], HoSA [12], jat [119] KoAT [51], Loopus [145, 146, 158], paicc [139], Pastis [54], PUBS [3, 5], RAJA [97, 100], RAML [95], Rank [9], RESA [10, 69], SPEED [84, 87], T$_{\mathsf{C}}$T [18] and TiML [153]. Static approximation of quantitative properties has a variety of concrete applications, such as, software quality [7], certification and security [61, 62], worst-case execution time analysis [156], estimation of energy consumption [83], and approximation of gas consumption for smart contracts [8].

This work is devoted to resource analysis of imperative programs. In what follows, we narrow down the scope of the content.

Worst-Case Upper Bounds. Resource analysis reasons about quantitative properties of program executions. Besides the resource of interest, one usually distinguishes between (i) *best-*, *average-* and *worst-case* analysis, and (ii) *lower* and *upper bounds*. Item (i) specifies the resource usage in the presence of *non-determinism*, e.g. best-case analysis minimises the resource usage over all possible outcomes. Non-determinism can arise due to dedicated support in the programming language or stratification of the input, e.g. length of input list. Obtaining precise results is not always possible. Item (ii) indicates how the analysis approximates the actual resource usage.

In this work, we are concerned with *safety* properties. We focus on *upper bounds* on the *worst-case* resource usage of programs. That is, we maximise the resource of interest

over all possible outcomes. The resources of interest are runtime and size (or value). Informally, the *worst-case runtime* is the maximal number of execution steps with respect to the input, and the *worst-case size* is the maximal valuation of an optimisation function of all reachable states with respect to the input.

Automated Analysis. In this work, we are concerned with *fully automated* approaches to resource analysis, much in the spirit of a *push-button* technology. The main advantage of automated approaches is that they can be used with little to no prior knowledge and integrated in bigger tool chains, though, there are unique challenges when automating approaches in program analysis. In particular, usually the search space of the properties of interest is huge. Modern automated resource analysis tools often rely on incomplete heuristics to restrict the search space.

Imperative Programs. Imperative programming is a prominent programming paradigm that underlies popular languages, such as Fortran, C, Java, etc. In recent years, optimisation and program analysis of low-level intermediate representations, such as LLVM or Java bytecode, got a lot of attention. This is also the case for resource analysis.

In this work, we focus on the resource analysis of imperative programming languages with integer-valued variables, much like a restricted version of intermediate representations. In addition, we explore programs with support for heap allocated data structures and probabilistic primitives. To reason formally about resource properties, we use *abstract reduction systems*, i.e. binary relations over program states, as abstract model of computation.

Methods to Automated Resource Analysis

In what follows, we list several methods and concepts hat are used in tools for the automated resource analysis of imperative programs. This list is far from complete, but exemplifies the variety of different approaches and research areas related to this topic. We remark that tools often combine several approaches together and individual methods are often conceptually and theoretically related.

Recurrence Relations. The seminal work of Wegbreit [154, 155] is considered to be the first method that is concerned with *automating* resource analysis. The work investigates the runtime behaviour of Lisp programs. In doing so, programs are transformed into *recurrence relations* that capture the execution time. Closed-form expressions of the recurrences can be obtained by dedicated solvers such as PURRS (Bagnara et al. [24]).

Inspired by earlier work on automated complexity analysis of logic programs (Debray et al. [64, 65]), Navas et al. [123] present resource analysis of Java bytecode programs with *user-definable* resource applications. Here, class and method annotations are used to track the resource of interest. The analysis generates *size relations* in form of recurrence equations to represent the input-output relationship of the tracked resources. The approach is implemented in the CiaoPP (Hermenegildo et al. [91]) framework.

Albert et al. [2] propose *cost relation systems* as language independent representation for static resource analysis. Cost relations are a variation of recurrence relations that have a specific form and support *non-determinism*. These systems form the central problem representation in the resource analyser COSTA [1, 4], and related work. Dedicated solvers, like PUBS (Albert et al. [3, 5]) and CoFloCo (Flores-Montoya [72]), infer closed-form expressions of cost relation systems.

Kincaid et al. [106] combine abstract interpretation and symbolic analysis to generate recurrence relations that overapproximate the behaviour of loops and exemplify its application to resource analysis.

Termination Analysis. The worst-case runtime of a program captures the maximal number of execution steps with respect to the input and is a common resource of interest. It can be conceived as a quantitative variant of *termination analysis*, which shows that the computation halts within a limited number of steps and this number only depends on the input. This observation motivates to adapt techniques from automated termination analysis for runtime analysis.

A well-known approach to proof termination of (imperative) programs, which dates back to Floyd's seminal work on program analysis [75], is based on *ranking functions*. A ranking function is a *monotone mapping* $f\colon S \to D$ from program states into a *well-founded ordered set*. Here, monotone means that each evaluation step $s \to s'$ implies a decrease in the ranking measure $f(s) > f(s')$. The program terminates, because an infinite evaluation $s_0 \to s_1 \to \cdots$ would induce an infinite chain $f(s_0) > f(s_1) > \cdots$, which is not possible as the target domain is well-founded. When suitable restricting the construction of ranking functions, upper bounds on the number of executions steps can be obtained. A sufficient criterion is to fix the target domain to the set of *natural numbers* $\mathbb{N}$ with the standard ordering. This domain is of particular interest for automation, since necessary properties of ranking functions can be expressed as an *optimisation* problem. For instance, the synthesis of *affine linear* ranking functions is amenable to *linear programming* [25, 132, 141].

The application of ranking functions for runtime analysis has been investigated and employed for instance in the complexity tools Rank (Alias et al. [9]) and KoAT (Brockschmidt et al. [51]). Both approaches are based on the synthesis and combination of (affine linear) ranking functions, which under additional restrictions are used to obtain possible non-linear upper bounds on the worst-case runtime.

Numeric Invariant Analysis. A conceptual simple approach to resource analysis is based on *counter instrumentation* and *numeric invariant analysis*. The main idea is to instrument the target program with a *counter variable* (or *tick* instructions) that represents the consumption of the resource of interest during evaluation (cf. Rosendahl [137]). For instance, we can instrument the program with a global counter variable that is incremented by one in the body of all loops to represent the total number of loop iterations for a program run. This reduces the problem of resource analysis to the problem of inferring *numeric invariants* on the counter variable and advocates the application of

standard numeric invariant domains such as *octagon* [117] and *polyhedra* [60].

Gulwani et al. [87] identify additional challenges for the resource analysis of imperative programs. Imperative and in particular low-level programs often admit complex *disjunctive* control flow and *non-linear* resource usage. This excludes the straight-forward integration of many existing tools that focus on the inference of numeric linear invariants that are valid for all paths. The tool SPEED [87] promotes the application of *multiple counter variables* that can be *incremented* and *resetted*. The main idea is that resets imply *multiplicative* dependencies between counter variables, similar to the usage of variable indices that can be found in nested `for` loops. This way linear invariants can be used to obtain non-linear bounds on the resource usage.

Program Verification. Nielson [126] extends Hoare logic [93] for total correctness to prove properties about the execution time of programs. Morally, a triple $\{P\}C\{b \Downarrow Q\}$ states that if the evaluation of program C starts from an initial state s_0 satisfying P, then it terminates in a state satisfying Q after at most $b(s_0)$ steps, in which b is a bound expression that is evaluated in the initial state s_0.

A comparable notion of *quantitative* Hoare triples is introduced by Carbonneaux et al. [53, 54], which forms the central notion of the resource analysis tools C4B and Pastis. The proposed approach is based on Tarjan's and Sleator's *potential method* for *amortised complexity* analysis [148, 151]. Here, a potential is a *measure* that maps program states to non-negative numbers and indicates how many resources are left for consumption in the rest of the computation. The initial potential indicates the bound on the resource consumption for the whole program. The rules of quantitative Hoare logic are used to generate *verification conditions* that capture the change in the potential. A solution to the generated constraints provide a concrete bound expression for the initial potential. The method presented in Carbonneaux et al. [53, 54] is amenable to linear programming. Ngo et al. [124] extend this approach to *probabilistic* programs in the tool Absynth.

Atkey [10] is concerned with static resource analysis of imperative programs with *heap allocated data structures*. The central idea is to extend *separation logic* [135] with a logic of *resource consumption*. Fenacci and MacKenzie [69] provide an implementation for the resource analysis of Java bytecode programs.

Kaminski et al. [105] present a calculus based on *weakest precondition transformers* for the *expected runtime* of *probabilistic* programs. The proposed method is shown to be equivalent to Nielson [126] in the case of deterministic programs, i.e. programs without probabilistic and non-deterministic behaviour. Based on this weakest precondition transformer, Avanzini, Schaper, and Moser [20] present a fully automated and modular approach to the *expected cost* analysis of probabilistic *While* programs.

Implicit Computational Complexity. The research field of *implicit computational complexity* is concerned with *machine independent* characterisations of complexity classes. The key idea is to syntactically restrict programs to control the resources needed for computation. The seminal work of Bellantoni and Cook [26] for instance, provides a *recursion theoretic* characterisation of the complexity class FP, i.e. the class of com-

putable functions in polynomial time. The proposed *predicative* recursion scheme is akin to *primitive recursion* but syntactically restricts the precise notion of composition and recursion. The main idea is that the arguments of a function $f(x_1, \ldots, x_n \,;\, y_1, \ldots, x_n)$ are partitioned into *normal* (on the left) and *safe* (on the right) arguments. Morally, normal arguments are used to control the recursion, while safe arguments are used for the computation of the value, and by restricting the growth of normal arguments the depth of recursion is controlled. This mechanism is also referred to as *tiering*. Conceptually related ideas are used in automated resource analysis.

Hainry and Péchoux [90] present a type system with *tiering* to inspect the runtime of object-oriented programs. The tiering mechanism is used to control the growth of the valuation of variables that control recursion and loop iteration. A *well-typed* terminating program runs in polynomial time. The type system can be automated, and a more precise polynomial bound is obtained by inspecting the maximal nesting depth of loops.

Type Systems. Hofmann and Jost [96] present a *type system* for analysing *amortised heap space* usage of first-order functional programs. The central idea of this method are *type-based potentitals* that govern the resource usage. This system has been adapted by the same authors to *object-oriented* Java-like programs [97]. Hofmann and Rodriguez [99, 100] automate this approach for object-oriented programs in RAJA, but the implementation is restricted to *linear* bounds. Hoffmann et al. [94, 95] adapt this approach to polynomial resource bounds for ML programs in the resource analysis tool RAML.

Abstract Program Representations. Ben-Amram et al. [39] and related work is concerned with *decidable growth-rate* properties for imperative programs. The growth-rate of a variable relates its output valuation with the input valuation of all arguments. The main idea of [39] and related work is to restrict programs *syntactically* in such a way that a certificate on the growth-rate of a variable can be effectively inferred. Here, a certificate represents a class of functions, for instance linear functions or polynomials. This growth-rate analysis forms the basis of the complexity tool paicc, which is developed by the author of this thesis [139].

Zuleger et al. [158] and Sinn et al. [145, 146] investigate different *abstract program representations* for the static resource analysis of LLVM programs. The restriction of the program syntax promotes dedicated and efficient methods for the bound inference mechanism. Standard techniques for invariant analysis are used to obtain an abstract representation from the target program. Several approaches based on different representations are implemented in the tool Loopus.

Overview of the Contribution

In what follows, we outline the central contributions of this thesis. In Chapter 2 we introduce common definitions and notations. Chapters 3 to 6 constitute the main content of this work. Each of these chapters can be studied independently. Finally, we conclude this work in Chapter 7.

Imperative Programs. In Chapter 3 we are concerned with the resource analysis of *imperative programs*. We restrict our attention on the *worst-case* resource usage of a standard imperative programming fragment with *integer-valued* variables. As formal program representation we consider so-called *constraint transition systems*, which are labelled transitions systems in which edges are decorated with Boolean expressions over integer-valued variables. These systems are eligible to represent programs with *unstructured control flow*, which can be found in modern intermediate representations, such as LLVM or Java bytecode. Our main focus of interest, are *abstract program representations*. By abstract programs, we mean programs with a non-standard semantics in which details are abstracted by *non-determinism*. In syntactically restricting the precise notion of constraints, one can distinguish between different abstract representations. Notably, the representations may differ in *theoretical properties*, and are amenable to dedicated approaches to resource analysis in *practice*. Theoretical properties of interest include *termination* and *polynomial runtime* of programs.

In this work, we provide an overview of theoretical properties and resource analysis tools which are related to constraint transition systems. In doing so, we recall abstract program representations that are known from the literature and provide an overview of the theoretical properties of interest. Moreover, we discuss and compare recent resource analysis tools and investigate how different abstract program representations are used in practice. The material presented in this chapter is based on Schaper and Moser [140]:

On Abstract Program Representations for Automated Resource Analysis.

Imperative Programs With Heap. In Chapter 4 we are concerned with the resource analysis of *imperative programs with heap allocated data*. We consider a standard imperative programming language with support for allocation and manipulation of *records*. For the resource analysis, we consider *complexity reflecting transformations*, that is, programs are transformed into *abstract programs* such that the resource of interest, like runtime, is overapproximated in the abstraction. The main motivation is to reuse existing tools for resource analysis.

In this work, we revisit two known transformations from the literature which differ in the abstract program representations. We demonstrate a *size* abstraction to *constraint transition systems* and a *term* abstraction to *constraint term rewrite systems*. We provide a *uniform* presentation and give additional insights on the representations used. The material presented in this chapter is based on Moser and Schaper [119]:

From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation.

Imperative Probabilistic Programs. In Chapter 5 we are concerned with the resource analysis of *imperative probabilistic programs*. We study the resource usage analysis of an imperative programming language that is endowed with probabilistic primitives for *sampling* and *probabilistic choice*. Probabilistic program analysis is interested to reason about all *probabilistic branches*. This gives rise to new theoretical and practical challenges. For the resource analysis, we inspect the *expected resource consumption*, that is, we *average* the resource consumption over all *probabilistic branches*.

In this work, we present a *fully automated* and *modular* resource analysis of probabilistic programs. Here, modular means that programs are decomposed into smaller subprograms that are analysed separately. More specific, we provide sound conditions for a modular analysis of the expected resource consumption of sequential and nested loops. Moreover, we give insights about automation of the main result and implementation of the prototype. The material presented in this chapter is based on Avanzini, Schaper, and Moser [20]:

Modular Runtime Complexity Analysis of Probabilistic While Programs.

Framework for Automation. In Chapter 6 we present a *framework for automation*. The *Tyrolean Complexity Tool* TcT is a framework for automated resource analysis. It is based on a theoretic *combination* framework for complexity, which advocates a *transformational* approach. In doing, so we make use of different *abstract program representations* that are known from the literature. The framework encourages *rapid prototyping* and features a rich tactic-like *proof search* to facilitate automation. At its core is the seamless *integration and combination* of different abstract program representations and resource analysis thereof.

In this work, we provide an overview of TcT. We present the *software architecture* and give insights about its *implementation*. To demonstrate the viability of the framework, we illustrate several *case studies*, which include the resource analysis of constraint transition systems, term rewrite systems, higher-order functional programs, and object-oriented bytecode programs. The material presented in this chapter is based on Avanzini, Moser, and Schaper [18]:

TcT: Tyrolean Complexity Tool.

Chapter 2

Preliminaries

In this chapter we provide common definitions and notions that are used throughout this work. We use *abstract reduction systems* to give language independent definitions of termination, worst-case runtime complexity and worst-case size complexity. In the course of this work, we introduce several variations of these definitions that express more refined notions of complexity. For details on abstract reductions systems, we refer to Baader and Nipkow [22].

Definition 2.1 (Partial Order)**.** A *partial order* $\sqsupseteq \subseteq A \times A$ on a set A is a reflexive, transitive and anti-symmetric binary relation. Let $a, b \in A$. We write $a \sqsupset b$ for $a \sqsupseteq b$ and $a \neq b$.

Definition 2.2 (Weak Monotonicity)**.** Let $(A, \sqsupseteq)$ be a set A equipped with a partial order $\sqsupseteq \subseteq A \times A$. We say that the function $f\colon A^n \to A$ is *weakly monotone in its i-th argument* with respect to $\sqsupseteq$, if $a_i \sqsupseteq b$ implies

$$f(a_1, \ldots, a_i, \ldots, a_n) \sqsupseteq f(a_1, \ldots, b, \ldots, a_n)$$

for all $a_1, \ldots, a_n, b \in A$. It is said to be *weakly monotone*, if it is weakly monotone in all its arguments.

Definition 2.3 (Upper Bound)**.** Let A be a set and let $(B, \sqsupseteq)$ be a set B equipped with a partial order $\sqsupseteq \subseteq B \times B$. We say that $f\colon A \to B$ is an *upper bound* of $g\colon A \to B$, in notation $f \succcurlyeq g$, if $f(a) \sqsupseteq g(a)$ for all $a \in A$.

Given two relations $R \subseteq A \times B$ and $S \subseteq B \times C$. The *composition* of R and S is defined by

$$R \cdot S \triangleq \{(a, c) \in A \times C \mid \exists b \in B.\ (a, b) \in A \text{ and } (b, c) \in B\}\ .$$

Definition 2.4 (Abstract Reduction System)**.** An *abstract reduction system* (ARS for short) is a pair $\mathcal{A} = (A, \to)$ consisting of a set A and a binary relation $\to$ on A. An element $(a, b) \in \to$ is called a *reduction step* from a to b. We write $a \to_{\mathcal{A}} b$ instead of $(a, b) \in \to$. Moreover, if $\mathcal{A}$ is clear from the context we just write $a \to b$.

Let $\mathcal{A} = (A, \to)$ and $\mathcal{B} = (B, \to)$ be two ARSs such that $B \subseteq A$. We derive the following binary relations:

$$
\begin{aligned}
\to^0 &\triangleq \{(a, a) \mid a \in A\} && \text{(identity)} \\
\to^{n+1} &\triangleq \to^n \cdot \to && ((n+1)\text{-fold composition)} \\
\to^= &\triangleq \to \cup \to^0 && \text{(reflexive closure)} \\
\to^+ &\triangleq \textstyle\bigcup_{n>0} \to^n && \text{(transitive closure)} \\
\to^* &\triangleq \to^+ \cup \to^0 && \text{(reflexive transitive closure)} \\
\to_{\mathcal{A}/\mathcal{B}} &\triangleq \to_{\mathcal{B}}^* \cdot \to_{\mathcal{A}} \cdot \to_{\mathcal{B}}^* && \mathcal{A} \text{ relative to } \mathcal{B}
\end{aligned}
$$

We say that there exists a *reduction sequence* from a to b (of length n), if $a \to^* b$ ($a \to^n b$).

Definition 2.5 (Termination)**.** Let $\mathcal{A} = (A, \to)$ be an ARS. An element $a \in A$ is called *terminating* if there are no infinite reduction sequences starting from a. The ARS $\mathcal{A}$ is called *terminating* on $S \subseteq A$ if all $a \in S$ are terminating.

In this work, $\mathcal{P}(A)$ denotes the *powerset* of a set A. For the domains $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ we use superscript ∞ to denote its extension with *infinity* and subscript $\geqslant 0$ to denote its restriction to the *non-negative* domain. For a function $f\colon A \to B$, we use $\mathsf{dom}(f)$ and $\mathsf{rng}(f)$ to denote the *domain* and *range* of f. Central to our discussion on the worst-case runtime of a program is the notion of derivation height.

Definition 2.6 (Derivation Height, Bounded)**.** Let $\mathcal{A} = (A, \to)$ be an ARS. The *derivation height* $\mathsf{dh}_{\mathcal{A}}\colon A \to \mathbb{N}^\infty$ of $a \in A$ with respect to $\mathcal{A}$ is defined by

$$\mathsf{dh}_{\mathcal{A}}(a) \triangleq \mathsf{sup}\,\{n \mid \exists b.\ a \to^n b\}\ .$$

The ARS $\mathcal{A}$ is called *bounded* on $S \subseteq A$, if for every $a \in S$, there exists $m \in \mathbb{N}$ such that $a \to^n b$ implies $n \leqslant m$. This is equivalent to saying that $\mathsf{dh}_{\mathcal{A}}(a) < \infty$ for every $a \in S$.

The problem whether $\mathcal{A}$ is bounded (on $S \subseteq A$) is called the *bounded termination problem (on S)* (cf. Ben-Amram and Vainer [38]).

Definition 2.7 (Canonical Runtime Complexity, Canonical Size Complexity)**.** Let $\mathcal{A} = (A, \to)$ be an ARS. The *canonical worst-case runtime complexity* $\mathsf{rc}_{\mathcal{A}} : \mathcal{P}(A) \to \mathbb{N}^\infty$ of $\mathcal{A}$ on $S \subseteq A$ is defined by

$$\mathsf{rc}_{\mathcal{A}}(S) \triangleq \mathsf{sup}\,\{\mathsf{dh}_{\mathcal{A}}(a) \mid a \in S\}\ .$$

Let $f\colon A \to \mathbb{N}$. The *canonical worst-case size complexity* $\mathsf{sc}^f_{\mathcal{A}}\colon \mathcal{P}(A) \to \mathbb{N}^\infty$ of $\mathcal{A}$ on $S \subseteq A$ with respect to f is defined by

$$\mathsf{sc}^f_{\mathcal{A}}(S) \triangleq \mathsf{sup}\,\{f(b) \mid \exists b.\ a \to^* b \text{ and } a \in S\}\ .$$

Chapter 3

Imperative Programs

In this chapter we are concerned with automated resource analysis of *imperative programs*. As model of computation we consider constraint transition systems with integer-valued variables. We discuss three modern tools, namely KoAT, Loopus and paicc. In doing so, we pay special attention to the program representations that are used in the implementations for automating the analyses. We outline key concepts of the different tools and recall known theoretical results on termination and complexity on related representations.

This chapter is based on Schaper and Moser [140]. Section 3.1 provides an informal discussion to resource analysis of integer programs. In Section 3.2 we outline the contribution of this chapter. Section 3.3 introduces constraint transition systems and common notations that are used throughout this chapter. In Section 3.4 we recall theoretical properties of related program representations that are known from the literature. We provide an overview of the resource analysis tools KoAT, Loopus and paicc in Section 3.5, Section 3.6 and Section 3.7, respectively. A summary of the key concepts of the individual tools is then given in Section 3.8, while we report on different case studies in Section 3.9. Finally, we conclude in Section 3.10.

3.1 Introduction

Automated resource analysis of imperative programs is an active area of research. In recent years several approaches have been investigated and implemented, to mention a few, AProVe [76], C4B [53], CiaoPP [123], CoFloCo [72], COSTA [1, 4], jat [119] KoAT [51], Loopus [145, 146, 158], paicc [139], Pastis [54], PUBS [3, 5], RAJA [97, 100], Rank [9], RESA [10, 69], and SPEED [84, 87]. Various results have been established in implicit computational complexity, which are related to the resource analysis of imperative programming languages, based on syntax [108], data-flow [39, 103, 127], graphs [121], interpretations [115], and types [90, 97, 114]. On the other hand, well-known approaches in program analysis are exploited to inspect the resource behaviour of imperative programs, which include, program abstraction [119, 146], ranking functions [3, 9, 46] and invariant analysis [87]. In this chapter we investigate the state-of-the art in automated resource analysis of imperative programming languages. We link theory and practice of related approaches emerging from the *implicit computational complexity* and the *program analysis* community.

We focus on a restricted set of (abstract) program representations that are known from the literature. To be more specific, we investigate the automated resource analysis of

integer constraint transition systems. These systems are expressed via control flow graphs over a finite set of integer-valued program variables. Edges in the control flow graphs are associated with constraints over arithmetic expressions that induce the one-step reduction relation of the program. Based on the exact form of the constraints one can syntactically distinguish between different abstract program representations. These representations vary in expressiveness and computational power. Most relevant, we are interested how different representations are exploited in automated resource analysis. The resources of interest include *worst-case runtime*, that is, the maximal length of a program trace, and *worst-case size*, that is, the maximal valuation of a program state with respect to a chosen function or norm.

Before introducing any notion of programs and resources formally, we present an illustrative example that provides an informal attempt to reason about the worst-case runtime of a program. We imagine this scenario from the view point of a developer who reasons about the runtime of the program using experience, some observations and basic mathematics.

Example 3.1 (Motivating Example)**.** Consider the following motivating program. All variables range over the domain of integers, and the scope of the loop body is indicated by indentation.

```
main(x, y)
while(x > 0)
  x = x − 1;  y = y + x;  z = y
  while(z > 0)
    z = z − 1
```

The program has two loops, an outer loop and an inner loop. First, we inspect the outer loop. The guard of the loop is $x > 0$. The variable x is bounded from below by zero, it is decreasing by one in the loop body, and it is unaffected by the inner loop. It follows that the body of the outer loop can be executed at most x times. Next, we inspect the inner loop, its guard is $z > 0$. The variable z is bounded from below by zero and it is decreasing by one in the loop body. Thus, the inner loop alone can be iterated at most z times, however, z is modified in the outer loop. To express the runtime of the inner loop with respect to the whole program, we are interested in (i) how often the loop can be accessed, and (ii) how big z can be in terms of the initial input. We already know that (i) is the iteration bound of the outer loop, viz x. Considering item (ii), the variable z is not increasing in the inner loop. Thus, we inspect the outer loop. Variable z depends on variable y, which on the other hand depends on itself and variable x. Roughly, the input value of y is incremented by x at most x times. Thus, an approximation for input z for the inner loop is $y + x^2$. To obtain the total runtime of the program we add the runtime of the outer loop x and the runtime of the inner loop $x \cdot (y + x^2)$. If we inspect the expression carefully, then it is not a runtime bound if one of the input parameters is negative. We argue that a valid upper bound on the runtime is $\mathsf{max}(0, x) + \mathsf{max}(0, x) \cdot (\mathsf{max}(0, y) + x^2)$. This bound is not precise because the input for the inner loop z is overapproximated by its maximal valuation.

The informal reasoning is based upon different observations of program properties. It relies on measuring progress, approximating values, and inferring dependencies between properties. It reasons about observations for the whole program, and for smaller components. Program analysis tools for resource estimation make this observations concrete. They reason formally about resources of the original program based upon different abstract program representations. These abstract program representations have theoretical and practical impact on the analysis itself.

In the course of this chapter, we investigate different tools and abstract program representations that are used in practice. We are interested in relating theory and practice of tools and the program representations used. Next, we provide an overview of the contribution.

3.2 Overview of the Contribution

In this section we provide an overview of the contribution and outline the rest of this chapter. For reference consider Figure 3.1.

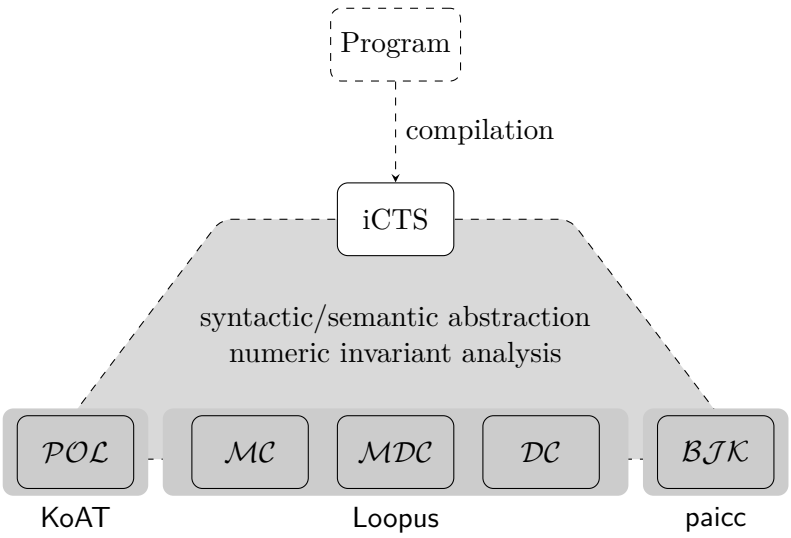


Figure 3.1: Overview of Discussed Approaches.

We are interested in the *automated inference* of *upper bounds* on the *worst-case runtime complexity* for imperative programs. The worst-case runtime of an input is the maximal number of evaluation steps of a program run, or analogously the maximal length of a program trace. As starting point for our discussion, we consider *integer constraint transition systems* (iCTSs for short) that serve as formal representation of programs in terms of *control flow graphs* with constraints over a finite set of integer-valued variables.

We focus on theoretical and practical properties of different *abstract program representations* that are obtained by syntactically restricting the exact notion of constraints. Abstract program representations constitute (non-standard) programs that are obtained

by lossy abstractions of the target program, and provide a language independent representation of the problem. In doing so, we are going to inspect the complexity analysis tools KoAT (Brockschmidt et al. [51]), Loopus (Zuleger et al. [158], Sinn et al. [145, 146]), and paicc (Schaper [139]). The individual approaches to resource analysis are based on different abstract program representations. The tool paicc is developed and maintained by the author of this thesis.

When studying abstract representations, it is not immediate how they are obtained from standard programs. In the general case, the transformation from programs to an abstract representation is lossy and relies on heuristics. Here, lossy indicates that the program executions from the original target program are overapproximated by the program executions of the abstraction. In this work, we group used approaches in *syntactic and semantic abstraction* and *numeric invariant analysis*.

Following the above discussion, we comment on Figure 3.1. Without loss of generality, we assume that iCTS programs are obtained from real-world programs by compilation. This representation is suitable to formally represent *pure integer* programs with *unstructured* control flow that are obtained from (a restricted version of) intermediate representations and low-level bytecode languages such as LLVM and Java bytecode. Such programs can be also derived via numerical abstractions from heap manipulating programs (see for example [76, 113]).

The tool KoAT processes transition systems, where constraints are inequalities of polynomial expressions ($\mathcal{POL}$). This representation is very expressive and captures the semantics of standard operations precisely.

Different abstract representations from the literature have been investigated within the scope of the development of Loopus. Among them, representations based on *monotonicity constraints* ($\mathcal{MC}$) [29, 30, 38], *monotone difference constraints* ($\mathcal{MDC}$) [101], and *difference constraints* ($\mathcal{DC}$) [27] have been taken into consideration and applied in practice.

Decidable properties of abstract but still expressive programming fragments have been investigated in implicit computational complexity. We focus on the development on decidable *growth-rate* properties for Core programs [28, 33, 34, 36, 37, 39]. The approach has been implemented together with a suitable program abstraction to *Ben-Amram - Jones - Kristiansen* ($\mathcal{BJK}$) constraints in the prototype paicc [139].

Outline. In Section 3.3 we provide basic terminology and recall techniques from the program analysis literature that are relevant to discuss the approaches to runtime analysis of the individual tools. Then, in Section 3.4 we provide an overview of theoretical properties of the abstract program representations of interest. In particular, we consider the properties (bounded) termination and (polynomial) runtime complexity. In Section 3.5, Section 3.6 and Section 3.7, we discuss the tools KoAT, Loopus and paicc, respectively. Section 3.8 provides an overview of the studied tools. Then, in Section 3.9 we inspect several case studies to practically relate different approaches. Finally, we conclude this chapter in Section 3.10. The theoretical overview in Section 3.4 and each tool in Sections 3.5 to 3.7 can be studied independently.

3.3 Preliminary Discussion

In this section we fix the notation of relevant standard definitions before introducing constraint transitions systems formally. Afterwards, we recall program analysis techniques that are known from the literature and which are helpful to discuss the subsequent approaches to resource analysis.

3.3.1 Directed Graph

Throughout this chapter we use standard properties on directed graphs to syntactically restrict the control flow in constraint transition systems.

Definition 3.2 (Directed Graph)**.** A *directed graph (with edge labels)* $G = (N, E)$ over the set L of *labels* consists of a finite set N of *nodes* and a set $E \subseteq N \times L \times N$ of *edges*. We usually write $u \xrightarrow{l} v \in G$ instead of $(u, l, v) \in E$. If the label is not relevant we just write $u \to v$.

Definition 3.3 (Paths)**.** Let $G = (N, E)$ be a directed graph. A *path* from *source* u to *target* v is a sequence of edges such that $u \to^* v$. A path is called *simple*, if all its nodes are distinct. The path $u \to^+ u$ is called *cyclic* at u. A cyclic path $u \to u$ is a *self-loop*. If $u \to^* v$ is a simple path, then $u \to^* v \to u$ is called *simple cyclic* at u.

Definition 3.4 ((Strongly) Connected Components)**.** Two nodes u and v are *connected*, whenever there is a path from u to v or a path from v to u. A *connected component* is a set $CC \subseteq E$ of edges such that all its nodes are connected. Two nodes u and v are *strongly connected*, whenever there is a path from u to v and a path from v to u. A *strongly connected component* is a set $SCC \subseteq E$ of edges such that all its nodes are strongly connected. A (strongly) connected component is said to be *trivial* if it does not contain any edges, otherwise it is *non-trivial*.

Definition 3.5 (Subgraph)**.** Let $G = (N, E)$ be a directed graph and $G' = (N', E')$ with $N' \subseteq N$ and $E' \subseteq E$ be a subgraph. Usually we just write $G' \subseteq G$. Suppose $v \in N \setminus N'$ and $v' \in N'$. The set of *incoming edges* of G' is defined by all edges $v \xrightarrow{l} v' \in E \setminus E'$, and the set of *outgoing edges* of G' is defined by all edges $v' \xrightarrow{l} v \in E \setminus E'$. The set of *entry nodes* of G' is defined by all target nodes of incoming edges, and the set of *exit nodes* of G' is defined by all source nodes of outgoing edges.

Definition 3.6 (Reducible Graph, Loop Header, Loop Path)**.** Let $G = (N, E)$ be a directed graph with a unique initial node, i.e. a node with no incoming edges. A node u *dominates* a node v if all paths from the initial node to v must go through u. An edge $u \xrightarrow{l} v$ is a *back-edge* if v dominates u. G is *reducible* if G becomes acyclic after removing all back-edges. If G is reducible then each SCC of G has a unique entry point, that is, a node which dominates all nodes in the SCC. We call this node the *loop header* of the SCC. Suppose l is a loop header. A *loop path* is a simple cyclic path $l \to^* l$ starting at the loop header.

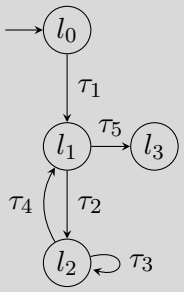
3.3.2 Constraint Transition Systems

As model of computation we consider *constraint transition systems* (CTSs for short). Let Var (Var') denote a countable set of variables (primed variables) and Loc denote a finite set of *(program) locations*. We denote by BExp the set of *Boolean expression* over Var and Var'. We keep the domain of the valuation of variables and the precise notion of BExp abstract for now, however, we are going to inspect different domains within the course of this chapter.

Definition 3.7 (Constraint Transition System)**.** A *constraint transition system* (CTS for short) is a directed graph $\mathcal{T} = (N, E)$ such that $N \subseteq \mathsf{Loc}$ and $E \subseteq \mathsf{Loc} \times \mathsf{BExp} \times \mathsf{Loc}$. Let $\mathcal{T}$ be a constraint transition system. An edge $l \xrightarrow{\phi} l'$ is called a *transition* with *source location* l, *target location* l' and *constraint* ϕ.

By convention, for a given constraint ϕ, variables in Var indicate the valuation of variables at source locations and primed variables in Var' indicate the valuation of variables at target locations. When depicting programs, we use different representations. Usually we provide a set of transitions $\langle l, l', \phi \rangle$, which sometimes is accompanied by a control flow graph. Sometimes we use syntax for `while` or `loop` programs and assume a straightforward translation to transition systems. For brevity, we often drop the canonical exit location when depicting CTS programs. We illustrate non-deterministic control flow and unconstrained assignments with $(*)$.

Example 3.8 (Cont'd from Example 3.1)**.** We provide an alternative representation for the motivating program in Example 3.1.



$$
\begin{aligned}
\tau_1 &\colon \langle l_0, l_1, \quad x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle \\
\tau_2 &\colon \langle l_1, l_2, x > 0 \wedge x' = x - 1 \wedge y' = y + x \wedge z' = x + y \rangle \\
\tau_3 &\colon \langle l_2, l_2, z > 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z - 1 \rangle \\
\tau_4 &\colon \langle l_2, l_1, z \leqslant 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle \\
\tau_5 &\colon \langle l_1, l_3, x \leqslant 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle
\end{aligned}
$$

Let D denote the target domain for variables, usually the integer or natural numbers. The set of *stores* is given by $\Sigma \triangleq \mathsf{Var} \to D$ and associates variables to the target domain. The set of *configurations* is given by $\mathsf{Conf} \triangleq \mathsf{Loc} \times \Sigma$. We indicate that the constraint ϕ holds for source store (or valuation) σ and target store σ' with $\sigma, \sigma' \models \phi$, i.e. the constraint ϕ evaluates to true when substituting variables $x \in \mathsf{Var}$ with $\sigma(x)$ and variables $x' \in \mathsf{Var}'$ with $\sigma'(x')$ in ϕ. The *one-step reduction* relation $(l, \sigma) \to (l', \sigma')$ for transition $\langle l, l', \phi \rangle$ is defined by $\sigma, \sigma' \models \phi$.

Let $\mathcal{T}$ be a CTS. Then, the one-step reduction relation of $\mathcal{T}$ induces the ARS $(\mathsf{Conf}, \to_{\mathcal{T}})$ over program configurations. Throughout this chapter, we occasionally make use of relevant notions on ARSs if no confusion can arise.

3.3.3 Complexity Functions and Complexity Bounds

In the course of this chapter we are going to discuss different tools for the automated runtime analysis. Given a program, a tool provides feedback to the user in form of *complexity bounds*. We address some challenges that arise in defining the (worst-case runtime) complexity of a program and expressing (upper) bounds on the complexity in automation.

As illustrating case, we consider *merge sort* (cf. Cormen et al. [58]). We say that *merge sort* has worst-case (runtime) complexity $\mathcal{O}(n\ log\ n)$, in which n denotes the input size, i.e. the number of elements to sort. We point out some observations. First, the complexity is expressed in terms of the *input size*. In this case the input size is the number of elements to sort or analogously the length of the list. Second, the complexity bound is *weakly monotone* in its argument. Here, weak monotonicity reflects the intuition that the complexity depends on the input size and that the complexity increases when the input size increases. Third, the complexity bound is *comprehensible* in the sense that it is expressed using standard notation and it is easy to understand. This is also important when we are interested in comparing different algorithms or the output of different tools.

These observations provide some challenges in automation. When inspecting the complexity of programs by hand one can precisely state what is measured and how it is measured. This option is restricted in the automated setting. Consider for instance programs where the input arguments are integer. It is not obvious what the input size should be. We recall two standard approaches how to formally define complexity functions.

First, one fixes the notion of input size and defines the complexity function in terms of the input size. Formally, the input size is usually represented in terms of a norm or a set of norms. Here, a *norm* is a mapping $\Sigma \to \mathbb{N}$. For instance, one may define the runtime complexity $rc\colon \mathbb{N}^n \to \mathbb{N}^\infty$ as a function in the absolute values of the program variables

$$rc(n_1, \ldots, n_n) \triangleq \mathsf{sup}\{\mathsf{dh}((l_0, \sigma_0)) \mid \mathsf{abs}(\sigma_0(x_i)) \leqslant n_i \text{ for all } 1 \leqslant i \leqslant n\}\ .$$

This definition does reflect most of the previous observations. The complexity is defined in the input size of all arguments and by definition it is weakly monotone in all arguments. However, the main disadvantage is that the input size is fixed and may not necessarily reflect the program property of interest, which is usually not known. Moreover, the definition itself already introduces imprecision. It is not necessarily the case that a program has the same complexity for negative and positive values. The problem becomes more apparent in a modular setting, when bounds on the complexity functions are combined from subprograms, then the imprecision may accumulate.

Second, one defines the complexity function in terms of the initial valuation or configuration. For instance, one may fix the runtime complexity $rc\colon \Sigma \to \mathbb{N}^\infty$ to

$$rc(\sigma_0) \triangleq \mathsf{sup}\{\mathsf{dh}((l_0, \sigma_0))\}\ .$$

While this definition is precise, it does not comply with our previous observations. In particular, it is not obvious what the input size is and unclear how a change in the

valuation changes the complexity. This observation becomes again relevant in a modular setting.

Throughout this chapter we opt for the second alternative. In what follows, we formally introduce the *worst-case runtime complexity* and *worst-case size complexity* for constraint transition systems. To mitigate some of the discussed problems we introduce *bound expressions*. Bound expressions provide a comprehensible representation of complexity bounds making use of the fact that we can naturally lift functions from the integer domain to the natural domain by standard operations.

Runtime Complexity and Size Complexity

Throughout this work we focus on *upper bound* analysis of the *worst-case* behaviour of programs. Informally, the worst-case runtime of a program corresponds to the maximal number of evaluation steps or analogously the maximal length of a program trace, while the worst-case size maximises a function over all reachable valuations. We are going to inspect different variations that are of interest when discussing modular approaches to the automated complexity analysis. If no confusion can arise, we sometimes drop the term upper or worst-case.

We define the runtime complexity and size complexity as functions in the initial valuation.

Definition 3.9 (Runtime Complexity, Size Complexity)**.** Let $\mathcal{T}$ be a transition system and $I \subseteq \mathsf{Loc}$ denote a set of initial locations. The *worst-case runtime complexity* $\mathsf{rc}^I_{\mathcal{T}} \colon \Sigma \to \mathbb{N}^\infty$ of $\mathcal{T}$ on I is defined by

$$\mathsf{rc}^I_{\mathcal{T}}(\sigma_0) \triangleq \mathsf{sup}\{\mathsf{dh}_{\mathcal{T}}((l_0, \sigma_0)) \mid l_0 \in I\} \ .$$

Let $f \colon \Sigma \to \mathbb{N}^\infty$. The *worst-case size complexity* $\mathsf{sc}^I_{\mathcal{T}} \colon \Sigma \to \mathbb{N}^\infty$ of $\mathcal{T}$ on I with respect to f is defined by

$$\mathsf{sc}^I_{\mathcal{T}}(\sigma_0) \triangleq \mathsf{sup}\{f(\sigma) | (l_0, \sigma_0) \to^*_{\mathcal{T}} (l, \sigma) \text{ and } l_0 \in I\} \ .$$

Complexity Bounds

We fix the notion of *bound expressions* (cf. Albert et al. [6]).

Definition 3.10 (Bound Expression)**.** In the following a, b denote arithmetic expressions over the variables, n denotes a norm, and c, d denote bound expressions.

$$\begin{aligned} a, b &::= a + b \mid a * b \mid i \in \mathbb{Z} \mid x \in \mathsf{Var} \\ n &::= \mathsf{max}(a, 0) \mid \mathsf{abs}(a) \mid k \in \mathbb{N} \\ c, d &::= n \mid \mathsf{max}(c, d) \mid c + d \mid c \cdot d \mid 2^c \end{aligned}$$

For two bound expressions c, d and variable x we denote by $c[d/x]$ the *substitution* of x by d in c. We write $c[d_i/x_i]$ to indicate the parallel substitution of x_i by d_i in c for $1 \leqslant i \leqslant n$.

The function $f\colon \Sigma \to \mathbb{N}^\infty$ is an *upper bound* on $g\colon \Sigma \to \mathbb{N}^\infty$, in notation $f \succcurlyeq g$, if $f(\sigma) \geqslant g(\sigma)$ for all $\sigma \in \Sigma$ (see also Definition 2.3). For a bound expression b we indicate the interpretation with respect to a valuation by $\underline{b}\colon \Sigma \to \mathbb{N}$. Consequently, we say that a bound expression rb is an upper bound on the worst-case runtime if $\underline{\mathsf{rb}} \succcurlyeq \mathsf{rc}^I_{\mathcal{T}}$.

We remark that for any valuation the interpretation of a norm n evaluates to $\mathbb{N}$ and by construction bound expressions are weakly monotone in all arguments.

3.3.4 A Mundane Approach to Modular Runtime Analysis

In what follows we discuss common observations to the *modular runtime* analysis. Here, we provide a conceptual overview rather than details. More details are given when discussing the individual approaches in Sections 3.5 to 3.7. We discuss two interesting cases. The first case inspects the sequential application of two distinct programs, and the second case considers the alternating application of two nested programs. For the sake of the argument, we assume that the individual programs have dedicated unique entry and exit nodes such that the programs can be plugged together.

Assume that we have two distinct programs $\mathcal{T}_1$ and $\mathcal{T}_2$ with runtime upper bounds $\lambda\sigma.f_1(\sigma)$ and $\lambda\sigma'.f_2(\sigma')$, respectively. We discuss how known bounds can be used to express the runtime on $\mathcal{T}_1$ followed by $\mathcal{T}_2$. Figure 3.2 illustrates the main idea.

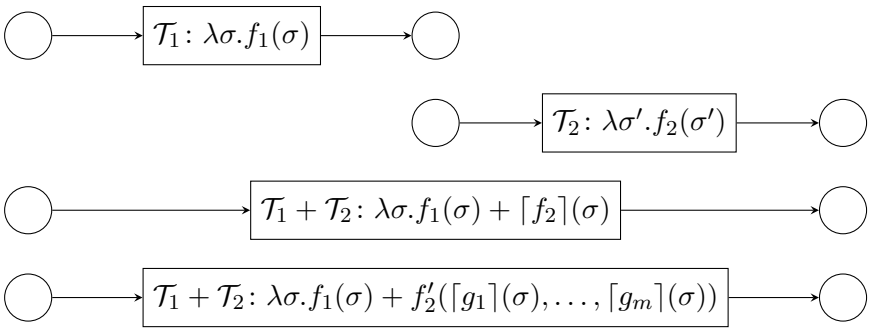


Figure 3.2: Modular Analysis of Sequential CTS Programs.

The first observation is that appending $\mathcal{T}_2$ to $\mathcal{T}_1$ does not affect the runtime of $\mathcal{T}_1$. On the other hand, we have to factor in the evaluations of $\mathcal{T}_1$ before executing $\mathcal{T}_2$. We want to express the upper bound $f_2(\sigma')$ in terms of the initial input of $\mathcal{T}_1$. Informally, we want to assess the input size for the evaluations in $\mathcal{T}_2$, or analogously, we want to asses the output size for the evaluations in $\mathcal{T}_1$. However, since we have not fixed the notion of input size we have to choose an appropriate one. The trivial candidate is to consider f_2 itself, i.e. we infer a size bound of $\mathcal{T}_1$ with respect to f_2. The size complexity of $\mathcal{T}_1$ with respect to f_2 is defined by the maximal valuation of f_2 applied to any store σ' that is reachable from the input store σ. In particular this includes all final stores of $\mathcal{T}_1$ and thus all input stores of $\mathcal{T}_2$. Now consider that f_2 can be a complex expression, e.g. a non-linear polynomial. To simplify the problem we inspect f_2. Suppose that $f_2(\sigma') = \lambda\sigma.f_2'(g_1(\sigma'), \ldots, g_m(\sigma'))$ such that f_2' is weakly monotone in its arguments, then it is usually enough to find upper bounds $\lceil g_1 \rceil, \ldots, \lceil g_m \rceil$ on the size complexity of $\mathcal{T}_1$ with respect to $g_1, \ldots, g_m$.

By restricting the shape of bound expressions, like in Definition 3.10, it is easy to control the decomposition. In particular bound expressions are always weakly monotone in all arguments.

Above we have addressed the issue of sequentially applying two programs. Figure 3.3 illustrates the case of nested programs $\mathcal{T}_2 \subseteq \mathcal{T}_1$. Such programs are typically derived from programs with nested `while` loops.

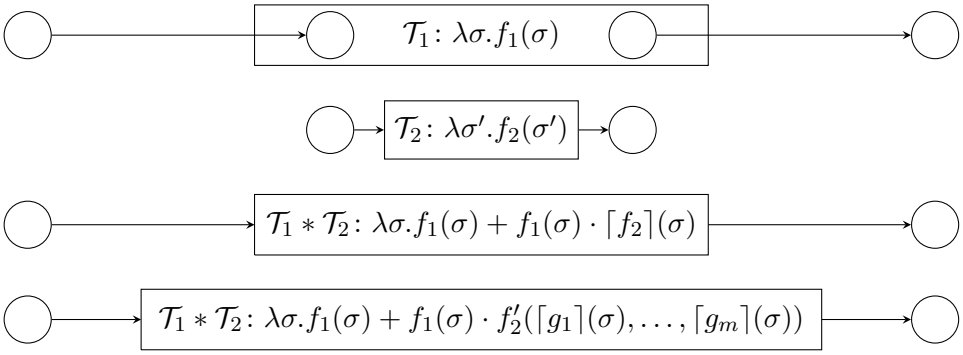
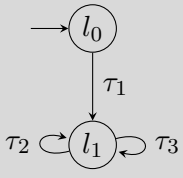


Figure 3.3: Modular Analysis of Nested CTS Programs.

Here $\lambda\sigma.f_1(\sigma)$ is an upper bound on the runtime of $\mathcal{T}_1$ relative to $\mathcal{T}_2$, i.e. we suppose $\mathcal{T}_2$ does not affect the runtime of $\mathcal{T}_1$, and $\lambda\sigma'.f_2(\sigma')$ is an upper bound on the runtime of $\mathcal{T}_2$. In contrast to the sequential application, the subprogram $\mathcal{T}_2$ may be evaluated more than once. Informally, we bound the maximal number of evaluating $\mathcal{T}_2$ by the runtime of $\mathcal{T}_1$. Furthermore, we express the upper bound on the subprogram in terms of the input of $\mathcal{T}_1$. Akin to the sequential case, we investigate the upper bound on the size complexity of $\mathcal{T}_1$ with respect to f_2. The expression $\lceil f_2 \rceil$ denotes an upper bound on the runtime for evaluating $\mathcal{T}_2$ once in terms of the input of $\mathcal{T}_1$. Putting it all together, we take the upper bound of $\mathcal{T}_1$ plus the upper bound of $\mathcal{T}_2$ for a single evaluation times the maximal number $\mathcal{T}_2$ can be evaluated, $f_1(\sigma) + f_1(\sigma) \cdot \lceil f_2 \rceil(\sigma)$. Again, we can simplify the analysis by inspecting the shape of f_2.

While the previous discussion provides insights how bounds can be composed, it does not address how to decompose programs with unstructured control flow. We provide additional insights about decomposing CTSs programs when presenting lexicographic ranking functions in Section 3.3.5.

Example 3.11 (Non-Determinism)**.** Consider the following variation of Example 3.8. Here, we replace the nesting of loops by non-deterministically choosing either one. This does not affect the worst-case runtime, but complicates our informal reasoning that relied on the structure of the loops.



$$\tau_1 \colon \langle l_0, l_1, \quad x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle$$
$$\tau_2 \colon \langle l_1, l_1, x > 0 \wedge x' = x - 1 \wedge y' = y + x \wedge z' = x + y \rangle$$
$$\tau_3 \colon \langle l_1, l_1, z > 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z - 1 \rangle$$

3.3.5 Ranking Functions

Ranking functions (RFs for short) map program states into a well-founded ordered set such that an evaluation step implies a decrease in the order. This is a well-researched topic in (automated) termination analysis as ranking functions provide a certificate for the absence of non-terminating sequences. When suitable restricted, ranking functions of termination proofs can be used for runtime analysis (see for example [3, 9, 14, 51]). The main idea is to estimate the cardinality of the co-domain of the ranking function in terms of the input.

Example 3.12 (Linear Ranking Function)**.** Consider the set of natural numbers equipped with the standard order $(\mathbb{N}, \geqslant)$. The expression $\mathsf{max}(0, n)$ is a ranking function for the program $\texttt{while}(n > 0)\{\ n = n - 1\ \}$. For each iteration step, i.e. if $n > 0$ holds, we have $\mathsf{max}(0, n) > \mathsf{max}(0, n - 1)$. For all inputs n, the co-domain of the ranking function is given by $\mathsf{max}(0, n)$.

Different notions of ranking functions have been investigated for termination analysis, for instance, lexicographic [44], disjunctive [57], eventual linear [23], and multiphase [111] ranking functions. These variations are often too expressive to obtain runtime bounds directly.

Example 3.13 (Lexicographic Ranking Function)**.** Consider the product domain of natural numbers equipped with the *lexicographic* order $(\mathbb{N}^2, \geqslant_{\mathsf{lex}})$, in which $(x, y) >_{\mathsf{lex}} (x', y')$ iff $x > x' \vee (x = x' \wedge y > y')$. The example below is a motivational example for lexicographic ranking functions. The assignment $y = *$ is unconstrained, i.e. the variable y can take any integer value. The program can be shown to be terminating via the lexicographic ranking function $(\mathsf{max}(0, x), \mathsf{max}(0, y))$. However, since y can take any value the program is not *bounded* on inputs with $x > 0$, i.e. there is no function $f\colon \mathbb{N}^2 \to \mathbb{N}$ in the input arguments that bounds the number of iterations. In particular, the cardinality of the co-domain cannot be expressed as a bound in the input arguments.

$$
\begin{array}{l}
\texttt{while}(x > 0 \ \wedge \ y > 0) \\
\quad \texttt{if}(x > 0) \ \{\ x = x - 1;\ y = *\ \} \\
\quad \texttt{else} \qquad \{\ y = y - 1 \qquad\ \}
\end{array}
$$

In the following we comment on two common applications for ranking functions.

(i) We discuss *linear (and polynomial) ranking functions* to estimate runtime bounds.

(ii) We present *lexicographic combinations of ranking functions* as a way to decompose programs.

We recall the notion of *(non-trivial) quasi-ranking functions* of (cf. Ben-Amram and Genaim [31]). We do not fix the shape of the ranking functions, but fix the target domain to the non-negative rational numbers equipped with the standard order, $(\mathbb{Q}_{\geqslant 0}, \geqslant)$.

Definition 3.14 (Quasi-Ranking Function)**.** Consider a mapping $\eta\colon \mathsf{Loc} \times \Sigma \to \mathbb{Q}_{\geqslant 0}$. Let $\mathcal{T}$ be a set of transitions with $\tau\colon \langle l, l', \phi\rangle \in \mathcal{T}$. We define the following properties:

$$\forall \sigma, \sigma' \in \Sigma.\ \sigma, \sigma' \models \phi \quad \Longrightarrow \quad \eta(l;\sigma) \geqslant 0 \qquad \text{(bounded)}$$
$$\forall \sigma, \sigma' \in \Sigma.\ \sigma, \sigma' \models \phi \quad \Longrightarrow \quad \eta(l;\sigma) - \eta(l';\sigma') \geqslant 1 \qquad \text{(decreasing)}$$
$$\forall \sigma, \sigma' \in \Sigma.\ \sigma, \sigma' \models \phi \quad \Longrightarrow \quad \eta(l;\sigma) - \eta(l';\sigma') \geqslant 0 \qquad \text{(non-increasing)}$$

We say that η

(i) is a *quasi-RF* for $\mathcal{T}$ if all transitions satisfy *non-increasing*,

(ii) is a *non-trivial quasi-RF* for $\mathcal{T}$ if all transitions satisfy *non-increasing* and at least one transition satisfies *decreasing* and *bounded*, and

(iii) is a *RF* for $\mathcal{T}$ if all transitions satisfy *decreasing* and *bounded*.

Synthesis of Linear Ranking Functions

A special case that is often considered in implementations are (affine) linear RFs when constraints conform to convex polyhedra. We provide the general idea how the individual properties for *synthesising* RFs can be encoded as a linear programming (LP) problem (cf. [9, 23, 25, 111]). The synthesis approach is based on the application of the affine version of Farka's Lemma (see Schrijver [141]), which states that if the polyhedron $P = \{\vec{x} \in \mathbb{R}^n \mid p_i(\vec{x})\}$ is non-empty, then every affine linear function $p(x)$ that is non-negative on P can be written as $p(\vec{x}) = \lambda_0 + \sum_{i=1}^{m} \lambda_i p_i(\vec{x})$ with $\lambda_0, \lambda_i \geqslant 0$. Now consider the following statement, where $\mathsf{f}\colon \mathbb{Z}^n \to \mathbb{Q}$ is unknown:

$$\forall \vec{x} \in \mathbb{Z}^n.\, \textstyle\bigwedge_i p_i(\vec{x}) \geqslant 0 \quad \Longrightarrow \quad \mathsf{f}(\vec{x}) \geqslant 0$$

We associate a *template* expression with f. Let $\mathsf{f}(\vec{x}) = \mathsf{a}_0 + \sum_{j=1}^{n} \mathsf{a}_j x_j$ with unknown coefficients $\mathsf{a}_0, \mathsf{a}_j \in \mathbb{Q}$. Then, by applying Farka's Lemma we obtain:

$$\mathsf{a}_0 + \textstyle\sum_{j=1}^{n} \mathsf{a}_j x_j = \lambda_0 + \sum_{i=1}^{m} \lambda_i p_i(\vec{x})$$

Finally, we encode this equality to an equality over the coefficients as LP problem. When encoding properties for the synthesis of RFs, we associate to each location an affine linear template $\eta(l;\sigma) = \mathsf{a}_{l_0} + \sum_{j=1}^{n} \mathsf{a}_{l_j} \sigma(x_j)$ and apply the previous encoding.

We remark that this procedure is sound but not complete for the general case. Completeness of synthesising RFs has been studied independently for different programs and domains. As an illustrating example, consider the work by Ben-Amram and Genaim [31] on the inference of affine linear RFs for *multipath linear-constraint loops*. A multipath loop program corresponds to a single `while` loop with alternative paths in the loop body that can be chosen non-deterministically. The loop guard and variable updates are restricted to affine linear expressions. For such programs, the inference of affine linear RFs is complete when the variables range over the integer or rational domain. More specific, the problem is decidable in polynomial time for the rational domain, whereas it is decidable in exponential time for the integer domain. The inference algorithm has been implemented in the tool iRankFinder[1].

[1] `http://www.loopkiller.com/irankfinder/interfaces/web/`

Inference of Upper Bounds

We have indicated before that ranking functions are used in automated runtime analysis. In what follows we make this observation concrete for CTS programs. We focus on upper bounds inferred from linear (and polynomial) RFs. For a more formal discussion consider for instance Avanzini and Moser [14], Brockschmidt et al. [51].

Let $\mathcal{T}$ be a constraint transition system. W.l.o.g. let l_0 be the only initial location of $\mathcal{T}$, i.e. there is no transition with target location l_0. Suppose that η is a polynomial RF for $\mathcal{T}$. Then $(l, \sigma) \to_{\mathcal{T}} (l', \sigma')$ implies $\eta(l; \sigma) \geqslant 1 + \eta(l'; \sigma')$ and $\eta(l; \sigma) \geqslant 0$. It follows directly that for all non-empty traces $(l_0, \sigma_0) \to_{\mathcal{T}}^{n+1} (l', \sigma')$ starting with (l_0, σ_0), we have $\eta(l_0; \sigma_0) \geqslant n + 1$. What is left is to consider the case in which the initial configuration (l_0, σ_0) cannot be reduced. In this case $\eta(l_0; \sigma_0)$ may actually be negative. However, for all σ_0, we have $\mathsf{max}(0, \eta(l_0; \sigma_0)) \geqslant \mathsf{rc}_{\mathcal{T}}^{l_0}(\sigma_0)$. If there are multiple initial locations it is enough to take the maximum.

The synthesis approach, discussed above, is appealing since it can be expressed as a linear programming problem. However, complexity bounds often are *non-linear* and *disjunctive*, and polynomial ranking functions (without max) in particular do not cope well with sign-changes. Compare with the informal analysis of Example 3.1.

However, the synthesis approach is useful in a modular setting and is used in tools such as PUBS [3] and KoAT [51]. Consider for instance the problem of inferring an upper bound on the number of occurrences of a single transition (or the number of iterations of a loop), which can be investigated with the synthesis of non-trivial quasi-RFs. Formally, this can be expressed by inspecting the runtime complexity of the relation $\to_{\tau/\mathcal{T}} = \to_{\mathcal{T}}^* \cdot \to_r \cdot \to_{\mathcal{T}}^*$.

In the course of this chapter we are going to discuss this idea and similar ones in more detail.

Example 3.15 (Cont'd from Example 3.11)**.** We define the non-trivial quasi-RF $\eta(l_0; \sigma) = \eta(l_1; \sigma) = \sigma(x)$ that is decreasing and bounded for τ_2 and non-increasing for τ_3. In any evaluation starting from (l_0, σ_0) the transition τ_2 occurs at most $\mathsf{max}(0, \sigma_0(x))$ times.

Lexicographic Ranking Function

Next, we present *lexicographic ranking functions* (LexRFs for short). Most of the discussed approaches in sequent sections rely explicitly or implicitly on LexRFs. Formally there are different notions of LexRFs. For an overview we refer to Ben-Amram and Genaim [32]. Here, we consider *mult-dimensional* ranking functions as presented in Alias et al. [9], that is, a combination of ranking components $\langle \eta_1, \ldots, \eta_d \rangle$, with lexicographic descent.

Definition 3.16 (Lexicographic Ranking Function)**.** Consider the well-founded order $(\mathbb{Q}_{\geqslant 0}^d, \geqslant_{\mathsf{lex}})$, in which

$$(x_1, \ldots, x_d) >_{\mathsf{lex}} (x'_1, \ldots, x'_d) \text{ iff}$$
$$x_i \geqslant 1 + x'_i \text{ for any } 1 \leqslant i \leqslant d \text{ and } x_j \geqslant x'_j \text{ for all } 1 \leqslant j < i \,.$$

We write $x \geqslant_{\mathsf{lex}} x'$ if $x = x'$ or $x >_{\mathsf{lex}} x'$. We say that $\langle \eta_1, \ldots, \eta_d \rangle$ is a *lexicographic combination of ranking functions* (or just *lexicographic ranking function*) for $\mathcal{T}$ if for all transitions $\langle l, l', \phi \rangle \in \mathcal{T}$

$$\forall \sigma, \sigma' \in \Sigma.\ \sigma, \sigma' \models \phi \implies (\eta_1(l;\sigma), \ldots, \eta_d(l;\sigma)) \geqslant_{\mathsf{lex}} (0_1, \ldots, 0_d)$$
$$\forall \sigma, \sigma' \in \Sigma.\ \sigma, \sigma' \models \phi \implies (\eta_1(l;\sigma), \ldots, \eta_d(l;\sigma)) >_{\mathsf{lex}} (\eta_1(l';\sigma'), \ldots, \eta_d(l';\sigma'))$$

We recall three applications of LexRFs.

Termination. Sometimes resource analysis is coupled with an external termination argument. For instance, Hainry and Péchoux [90] propose a type system to control resources for programs with heap allocated data. While the system ensures that the domain space does not grow uncontrollable, it does not guarantee termination. LexRFs provide a powerful mechanism for (automated) termination analysis.

Runtime Analysis. As illustrated above in Example 3.13, it is not possible to infer upper bounds on the runtime from LexRFs in the general case. In particular, programs which can be shown to terminate using a lexicographic RF may not be bounded (see Definition 2.6 on page 10). However, in restricted cases we can still inspect the cardinality of the co-domain. Consider a LexRF $\langle \eta_1, \ldots, \eta_d \rangle$, where the co-domain of the norms range from 0 to some upper bound M. The upper bound may be derived from size bound analysis or numeric invariants. Then the cardinality of the co-domain of the LexRF is M^d. Therefore, any trace terminates in at most M^d steps. This idea is used, for instance, in the complexity analyser Rank [9].

Program Decomposition. The most relevant application that we consider in this chapter is program decomposition. The ranking properties bounded, decreasing and non-increasing provide dependencies on the growth of variables or expressions. This can be used in connection with syntactic based decomposition techniques to construct a nested hierarchy of subprograms. In the following, let SCC denote a strongly connected component of a transition system. Consider some program $\mathcal{T}$. We set $\mathcal{T}' = \mathcal{T}$.

(i) Compute all SCC_i of (sub)program $\mathcal{T}'$.

(ii) For each SCC_i let η_i be a RF of SCC_i that is non-increasing for all transitions of SCC_i and bounded and decreasing for at least one transition, w.l.o.g. the transitions $\{\tau_{i_1}, \ldots, \tau_{i_n}\} \subseteq SCC_i$ satisfy bounded and decreasing.

(iii) Take $SCC_i \setminus \{\tau_{i_1}, \ldots, \tau_{i_n}\}$ and apply recursively (i).

We obtain a hierarchy of subprograms that reflects the ranking properties on expressions η_i. In a modular approach we make use of this hierarchy by analysing transition bounds and size bounds on norms locally and combining it conforming to specified rules to obtain global bounds.

We provide this observation here, since all discussed approaches in Sections 3.5 to 3.7 either explicitly or implicitly rely on lexicographic RFs. The above presented decomposition should be considered as a helpful guideline. The exact details vary for the individual approaches and will be outlined in the subsequent sections.

3.3.6 Applications for Numeric Invariants

Next, we recall applications for standard numeric invariant generation that are related to resource analysis.

Inference of Upper Bounds

Numeric invariants can be used directly to infer upper bounds on the resource of interest or indirectly within a modular approach.

Counter Instrumentation. A conceptual simple approach to worst-case runtime analysis is based on *counter instrumentation* (cf. Rosendahl [137]). Briefly, one instruments the program with a *counter variable* to count the number of loop iterations. Then, upper bounds on the maximal valuation of the counter variable imply an upper bound on the number of loop iterations.

Such upper bounds can in principle be inferred by off-the-shelf numeric invariant generation tools. However, two challenges that arise in bound analysis is that complexity bounds often are *non-linear* and *disjunctive*. Compare with the informal analysis of Example 3.1. This excludes the direct application of common numeric domains such as *octagon* [117] and *polyhedra* [60].

A more modular approach based on *multiple counter variables* has been proposed by Gulwani et al. [87] and implemented in the tool SPEED. On the other hand, Cadek et al. [52] investigate new approaches to the inference of non-linear and disjunctive invariants based on recent developments in bound analysis.

Modular Runtime Analysis. In Section 3.3.4 we discuss two approaches to modular resource analysis based on size bounds. Numeric invariant generation can be used to obtain size bounds on variables or expressions. Most notably, in a modular setting the problem can often be decomposed into a smaller program. For instance to assess an upper bound on the expression $\mathsf{max}(0, x) \cdot \mathsf{max}(0, y)$ it is enough to investigate the size bound on $\mathsf{max}(0, x)$ and $\mathsf{max}(0, y)$ separately. This makes standard numeric invariant generation more viable.

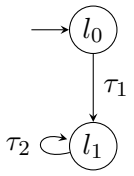
A concrete application of this idea is exploited by Alias et al. [9] in Rank. Here size bounds are used to estimate the co-domain of individual components of a multi-dimensional ranking functions.

Supporting Invariants

Next, we provide a trivial example where the synthesis approach for linear ranking functions presented in Section 3.3.5 fails.

Example 3.17 (Supporting Invariants)**.** Consider the following program. This example has no affine linear RF with respect to Definition 3.14. The definition crudely approximates the set of reachable states by the constraints itself and is not able to infer the loop invariant $(y > 0)$ from the assumption which is necessary to generate a ranking function.

```
assume(y > 0)
while(x > 0)
  x = x - y
```



$$\tau_1 \colon \langle l_0, l_1, y > 0 \wedge x' = x \qquad \wedge y' = y \rangle$$
$$\tau_2 \colon \langle l_1, l_2, x > 0 \wedge x' = x - y \wedge y' = y \rangle$$

A common approach to make the generation of ranking functions for programs more viable in practice is to provide *supporting invariants* [44, 111], which make the approximation of reachable states more precise.

We recall two known approaches. First, invariants that are inferred by encoding *inductive invariants* [41, 56]. This approach interacts well with SAT/SMT based approaches to generate ranking functions. Second, one makes use of the wealth of research in data-flow analysis and abstract numerical domains. In particular well-known abstractions like octagon and polyhedra combine well with the synthesis approach of linear ranking functions.

3.3.7 Program Abstraction

Next, we discuss a simple approach to *lossy* program abstractions using *transition invariants*. Here, lossy means that any trace of the original program is also a trace of the abstraction. Therefore, the abstraction is sound for termination and upper bound analysis.

Definition 3.18 (Transition Invariant)**.** Let $\mathcal{T}$ be a transition system. A constraint ψ is called a *transition invariant* for $\langle l, l', \phi \rangle \in \mathcal{T}$, if for all valuations $\sigma, \sigma' \in \Sigma$, $\sigma, \sigma' \models \phi$ implies $\sigma, \sigma' \models \psi$.

As a direct consequence we obtain the following statement. Suppose that ψ_i for $1 \leqslant i \leqslant n$ are transition invariants for some transition $\tau \in \mathcal{T}$. Then, $\bigwedge_{i=1}^{n} \psi_i$ is also a transition invariant for τ.

Now consider an abstraction $\mathcal{T}'$ of $\mathcal{T}$ which is obtained by replacing each constraint by a set of transition invariants. The next observation follows directly. Suppose there exists a trace (not necessarily starting from an initial configuration) $(l, \sigma) \to_{\mathcal{T}}^{n} (l', \sigma')$. Then, there exists a trace $(l, \sigma) \to_{\mathcal{T}'}^{n} (l', \sigma')$.

3.4 Overview of Abstract Program Representations

In this section we provide an overview of theoretical properties for *abstract program representations* that are known from the literature. We focus on a selected few representations that are related to the tools which are discussed in this chapter and are interested in the decision problems *termination*, *bounded termination* and *(polynomial) worst-case runtime complexity*. For decidable problems we provide the complexity of the decision procedure, if available.

We use standard notations P, PSPACE and EXPTIME to denote the complexity classes polynomial time, polynomial space and exponential time, respectively. With PRIMREC we denote the class of primitive recursive functions. We conclude this section with a table that provides all referenced results.

3.4.1 Loop Programs

Meyer and Ritchie [116] present Loop programs which consists of a small set of commands that characterise PRIMREC.

```
C,D ::= LOOP X {C} | C;D | X := 0 | X := X+1 | X := Y
```

Most commands are straightforward. The valuation of the program variables range over the natural numbers. The command `LOOP X {C}` executes the command `C` exactly `X` times and the variable `X` cannot be modified within `C`.

Termination. By definition, the number of iterations of a loop is bounded by the valuation of the loop variable. Hence, all Loop programs are terminating.

Bounded Termination. The valuation of variables in Loop programs are bounded by primitive recursive functions. The runtime of Loop programs can be expressed via counter instrumentation, and therefore, the runtime complexity of Loop programs is also bounded by a primitive recursive function.

Runtime Complexity. If we are interested in polynomial runtime bounds, then the decision problem is undecidable. This follows from a reduction of the halting problem for Turing machines (see Ben-Amram and Kristiansen [34]).

3.4.2 Core Programs

Inspired by earlier work on the computational complexity of imperative programming languages (see for example [103, 108, 127]), Ben-Amram et al. [39] present a *core language*. Here, we refer to programs that conform to the core language as Core programs. Core programs are a variant of Loop programs with *weak semantics*, that is, the abstraction of conditional control flow with non-deterministic flow.

```
C,D ::= loop X {C} | choice {C} {D} | C;D | skip |
        X := Y |  X := Y+Z | X := X*Z
```

The command `loop X {C}` executes the command `C` $0, 1 \ldots$ at most `X` times, and the command `choice {C} {D}` indicates non-deterministic choice. In contrast to more standard programming languages, Core does not support numeric constants. Most relevant for our discussion, the *polynomial growth-rate* of variables, i.e. whether the valuation of a variable after executing a program is bounded by a polynomial in the input, is decidable for Core programs [39]. The growth-rate analysis discussed in [39] is presented as an application of *abstract interpretation* [59]. It is a compositional *bottom-up* analysis making use of the structural definition of the Core programming language. We provide additional details about the growth-rate analysis in Section 3.7 when discussing the tool paicc.

Termination & Bounded Termination. Like Loop programs, Core programs are terminating, and the valuation of all program variables are bounded by primitive recursive functions.

Runtime Complexity. The worst-case runtime complexity of Core programs can be defined via counter instrumentation. Hence, the decision problem, whether the runtime is bounded by a polynomial in the input, is decidable. Moreover, it is decidable in P.

Various extensions of the polynomial-growth rate problem for Core have been studied.

Weak Assignment. Assignments in Core can be interpreted as *weak* (or *lossy*) assignments [34, 39], that is, `X <= Y` instead of `X := Y`. This is relevant if one considers Core as a target abstract representation for termination and complexity analysis.

Support for Reset with Zero. The polynomial growth-rate problem is decidable when adding support for *reset with zero* `X := 0` to the Core language [28, 36]. In particular, the problem is PSPACE-complete. The key idea is to associate locations with a context that indicates which variables are assigned to zero. This allows to refine the control flow and in turn, the growth-rate analysis with respect to the context. This can be formally expressed as an application of *transition partitioning* [136] or *elaboration* [38].

Definite Loops. Extending Core with definite loops, i.e. loops with exact iteration count, makes the polynomial growth-rate problem undecidable again [34]. This is also the case when weak assignments are considered.

Support for Increment. Decidable growth-rates for Core programs supporting constants or more specific increments is an open problem [28, 34]. Consider the following Core program with support for *reset* `X := 0` and *increment* `X := Y+1`.

```
U := 0; V := 0; Y := Z
loop Y
  Z := U; U := V+1; V := Z
```

The precise growth-rate of Z can be expressed as $z' \leqslant \lfloor \frac{1}{2} z \rfloor$, where z' corresponds to the final value of Z. One key observation for (standard) Core programs is that the growth-rate of variables is not decreasing, in the sense that the final valuation of a variable is not less than the initial valuation of any variable. This fact is exploited in the compositionality of the growth-rate analysis. The expression $z' \leqslant \lfloor \frac{1}{2} z \rfloor$ however contradicts this observation and its growth-rate is not monotone under iteration. Consider wrapping the previous program within another loop, then the valuation of Z decreases with increasing iteration count.

Tight Bounds. At the time of writing Ben-Amram and Hamilton [33] present the inference of *tight polynomial bounds* on the growth-rate of variables for Core programs. The proposed analysis is conceptually similar to the original one. However, it makes use of a more refined abstract domain. A *multi-polynomial* is a sequence of (abstract) polynomial expression with undetermined coefficients, and represents simultaneous growth-rate bounds on all program variables. The growth-rate analysis infers bounds in form of a set of multi-polynomials. For instance, consider a program with variables X,Y and Z. The set $\{\langle x, y, z\rangle, \langle x, x^2 + y, x^2 + z\rangle\}$ indicates that for each run the growth-rate of X,Y,Z is bounded by $\langle x, y, z\rangle$ or $\langle x, x^2 + y, x^2 + z\rangle$.

The main result states that the inference is tight, that is, for each run there exists a multi-polynomial in the inferred set such that the final valuation of all variables is bounded up to a constant factor, and for each multi-polynomial in the inferred set there exists a run such that the final valuation of all variables corresponds to a multi-polynomial up to a constant factor.

The proposed algorithm in [33] runs in EXPTIME. However, the precise complexity of the problem has not been inspected yet. The runtime can be inspected via counter instrumentation. In the overview we write $\Theta(N^k)$, where $k \in \mathbb{N}$ denotes the maximal degree of all polynomial expressions that bound the counter and N denotes the maximal value of the input arguments.

Flowchart Programs. Ben-Amram and Pineles [36, 37] introduce *loop annotated flowchart* programs where edges of the flowchart programs are associated with assignments that conform to Core expressions. Informally this corresponds to an unstructured version of Core programs. To mimic the behaviour of `loop` commands, flowchart programs are associated with a loop structure. A *loop structure* is a nesting hierarchy of subprograms, in which each subprogram is associated with a local iteration bound that indicates the maximum length of a path that can be taken within the subprogram. This variant is a strict generalisation in the sense that all Core programs can be represented as flowchart variant but not the converse.

When discussing the worst-case runtime analysis of imperative programs with the tool paicc in Section 3.7, we represent such programs as CTSs with $\mathcal{BJK}$ constraints. This is a purely syntactical variant that allows for a more uniform representation of programs. A $\mathcal{BJK}$ *order constraint* is an inequality constraint $x' \leqslant p(x_1, \ldots, x_n)$ where $p(x_1, \ldots, x_n)$ is a polynomial expression (or composed Core expression) with variables in

Var and coefficients in $\mathbb{N}$. A $\mathcal{BJK}$ program is a CTS where edges are associated with conjunctions of $\mathcal{BJK}$ order constraints, in which variables range over the natural numbers. We restrict to fan-in free programs. A *fan-in free* program implies that for each variable $x' \in \mathsf{Var}'$ there is at most one order constraint $x' \leqslant p(x_1, \ldots, x_n)$ associated to each edge. Otherwise, $\mathcal{BJK}$ programs would support minimisation of expressions. Furthermore, we always assume that $\mathcal{BJK}$ programs are associated with a loop structure. Most relevant, the polynomial growth-rate problem is also decidable in P for $\mathcal{BJK}$ programs.

3.4.3 Size-Change Constraints Programs

A *size-change order constraint* is an inequality $x > y'$ or $x \geqslant y'$ with $x \in \mathsf{Var}$ and $y \in \mathsf{Var}'$ in which variables range over a well-founded domain. A *size-change program* $\mathcal{SC}$ is a CTS where edges are associated with a conjunction of size-change order constraints.

Termination. Lee et al. [110] introduce the size-change principle for termination. The *size-change termination* (SCT for short) criterion is a language independent method for proving termination of programs. The main idea of SCT is that a program terminates if every infinite computation implies an infinite descent in a well-founded domain. The SCT problem is decidable, more specifically it is PSPACE-complete [110]. Ben-Amram and Lee [35] investigate applications of the SCT criterion and provide a polynomial time algorithm that is incomplete in the general but complete for specific subclasses that restrict non-determinism in the size-change argument.

The termination problem for $\mathcal{SC}$ programs where the variables range over $\mathbb{N}$ is completely characterised by the SCT criterion, i.e. a $(\mathcal{SC}, \mathbb{N})$ program is terminating if and only if it is size-change terminating. The result is derived as a special case of monotonicity constraints programs (Ben-Amram [29]), which we are going to discuss below.

Bounded Termination. The bounded termination problem for $\mathcal{SC}$ programs with domain $\mathbb{N}$ is decidable in PSPACE. In particular bounded termination implies that the worst-case runtime is in $\mathcal{O}(N^{|V|})$. Here $N \in \mathbb{N}$ specifies the maximal input argument and $|V|$ denotes the number of variables. We derive this information as a special case of monotonicity constraints programs (Ben-Amram and Vainer [38]), which we are going to discuss below.

Runtime Complexity. Colcombet et al. [55] study the asymptotic worst-case runtime complexity of size-change programs in terms of *max-plus* automata. The complexity for a terminating $\mathcal{SC}$ program is a polynomial $\Theta(N^k)$ with rational exponent k.

The special case of *fan-out free* $\mathcal{SC}$ programs has been studied by Zuleger [157]. In fan-out free programs, all transitions are associated with at most one constraint $x > y'$ (or $x \geqslant y'$) for all $x \in \mathsf{Var}$. Informally, fan-out free programs are not duplicating. The complexity is a polynomial $\Theta(N^k)$ with natural exponent $k \in \mathbb{N}$. The exponent can be inferred by a PSPACE algorithm.

3.4.4 Monotonicity Constraints Programs

Monotonicity constraints form a generalisation of size-change constraints. A *monotonicity order constraint* is an inequality constraint $x > y$ or $x \geqslant y$ for $x, y \in \mathsf{Var} \cup \mathsf{Var}'$. A *monotonicity constraints program* $\mathcal{MC}$ is a CTS where edges are associated with a conjunction of monotonicity constraints. An approach for runtime analysis in Loopus based on $\mathcal{MC}$ programs in which variables range over $\mathbb{Z}$ is discussed in Section 3.6.1.

Termination. Ben-Amram [29] presents a sound and complete method to the termination problem of $\mathcal{MC}$ programs over a well-founded domain in terms of the size-change termination principle. The termination problem of $\mathcal{MC}$ programs over a well-founded domain is PSPACE-complete. This algorithm has been generalised to the integer domain by Ben-Amram [30].

Bounded Termination. Ben-Amram and Vainer [38] show that the bounded termination problem for $\mathcal{MC}$ programs over $\mathbb{Z}$ is PSPACE-complete. $\mathcal{MC}$ programs that are bounded terminating are implicitly bounded by a polynomial in the initial values, and the exponent is at most the dimension of the state, i.e. the number of variables.

Runtime Complexity. It is an open problem whether precise (asymptotic) bounds can be obtained for $\mathcal{MC}$ programs.

3.4.5 Vector Addition Systems with States

Vector addition systems with states ($\mathcal{VASS}$) are programs where states are defined as n-tuples over the natural numbers and an evaluation step conforms to an addition with a n-tuple of integer (cf. Hopcroft and Pansiot [101]). We can represent $\mathcal{VASS}$ programs as CTSs with equality constraints $x' = x + k$ for each $x \in \mathsf{Var}$ with $k \in \mathbb{Z}$ and where the variables range over $\mathbb{N}$. In Section 3.6.2 we discuss the worst-case runtime analysis of monotone difference constraint programs $\mathcal{MDC}$ in Loopus. $\mathcal{MDC}$ programs are similar to $\mathcal{VASS}$ with weakened constraints of the form $x' \leqslant x + k$.

Termination. The termination problem for $\mathcal{VASS}$ programs is decidable in P (Brázdil et al. [45]). We provide some insights below when discussing the runtime complexity.

Bounded Termination. The class WCPN denotes the *weakly computable functions by Petri Nets* (and equivalently $\mathcal{VASS}$ programs). Morally, a total function $f(x)$ is *weakly computable*, if there exists a $\mathcal{VASS}$ program such that for all $x \in \mathbb{N}$ there exists a terminating program run $(l_{init}, x) \to^* (l_{final}, x')$ with $x' = f(x)$ and for all terminating program runs $(l_{init}, x) \to^* (l_{final}, x')$ we have $x' \leqslant f(x)$. Weakly computable functions are primitive recursive, i.e. WCPN $\subseteq$ PRIMREC (Leroux and Schnoebelen [112]). Given a $\mathcal{VASS}$ program. Suppose we instrument the program with a counter variable to represent the runtime. Then, if the program is terminating the final valuation of the counter variable is bounded by a weakly computable function.

Runtime Complexity. The construction of precise asymptotic bounds for $\mathcal{VASS}$ programs is discussed by Brázdil et al. [45]. In particular, it provides (i) a decision procedure to the termination problem, (ii) a characterisation of linear bounded $\mathcal{VASS}$ based on the existence of a linear ranking function, and (iii) a characterisation of a restricted class of programs with precise asymptotic bounds that are obtained based on the existence of a set of linear quasi-ranking functions and the nesting depth of the lexicographic decomposition.

Briefly, the termination problem is decidable in P. The linear bound problem, i.e. whether the runtime of a terminating program is in $\Theta(N)$ is also decidable in P. Here N corresponds to a chosen vector norm. Item (iii) captures the case of terminating non-linear programs. However, the decision procedure captures only a restricted class of problems. This restriction is not syntactical but given by a decidable property for quasi-ranking functions. That is why we mark the result with ‡ in the overview. The asymptotic complexity bounds that are obtained for the last case correspond to $\Theta(N^k)$ with $1 < k \leqslant d, k \in \mathbb{N}$ and d being the dimension of the vector (or the number of variables).

3.4.6 Difference Constraints Programs

A *difference order constraint* is an inequality $x' \leqslant y + k$, where $x' \in \mathsf{Var}', y \in \mathsf{Var}$ and $k \in \mathbb{Z}$, and a *difference constraint* is a conjunction of difference order constraints. A *difference constraints program* $\mathcal{DC}$ is a CTS with difference constraints and in which the valuation of variables range over $\mathbb{N}$. In Section 3.6.3 we discuss the worst-case runtime analysis of *fan-in free* $\mathcal{DC}$ programs in Loopus. A fan-in free program has constraints in which there is at most one order constraint $x' \leqslant y + k$ for each variable $x' \in \mathsf{Var}'$.

Termination. Ben-Amram [27] investigates termination of $\mathcal{DC}$ programs based on an extension of the size-change termination criterion. The termination problem is PSPACE-complete for *fan-in free* $\mathcal{DC}$ programs. In the general case the termination problem is undecidable. As a direct consequence the bounded termination problem and the runtime complexity problem are undecidable for $\mathcal{DC}$ programs.

3.4.7 Polynomial Constraints Programs

The most expressive program representation that we consider are based on polynomial order constraints. A *polynomial order constraint* is an (in-)equality of polynomial expressions over Var and Var' with integer coefficients. A *polynomial constraints program* $\mathcal{POL}$ is a CTS where constraints are conjunctions of polynomial order constraints and the valuation of variables range over $\mathbb{Z}$. The properties of interest, termination, bounded termination, and runtime complexity are undecidable for $\mathcal{POL}$ programs. In Section 3.5 we discuss the worst-case runtime analysis of $\mathcal{POL}$ programs in the tool KoAT.

Overview

Although, the program representations are very similar they are incomparable for the most part. In Figure 3.4 we compare the expressiveness of the discussed representations. We say that representation A is *as expressive as* representation B, if all B programs have an equivalent A program. Here, equivalent means that for any initial configuration the set of program traces that are obtained from both programs are identical. We use a solid arrow from B to A to indicate that A generalises B syntactically, and a dashed arrow to indicate that A generalises B with some additional assumptions.

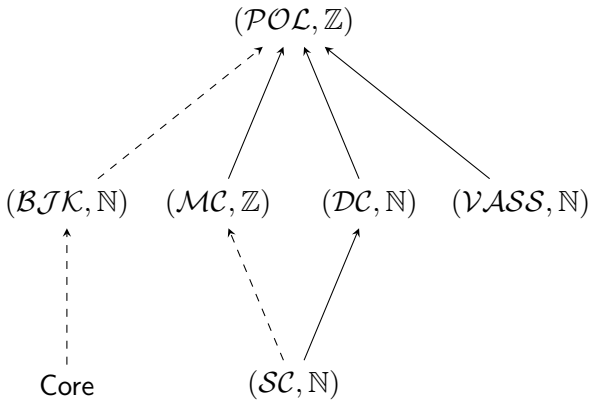


Figure 3.4: Hierarchy of Program Representations.

The $\mathcal{POL}$ order constraints syntactically subsume the order constraints of all other representations. It is easy to restrict the domain of variables to $\mathbb{N}$ with extra constraints $x \geqslant 0$ for all variables.

To express a $\mathcal{BJK}$ program as $\mathcal{POL}$ program, it is necessary to emulate local iteration bounds on subprograms that are induced by its loop structure. Morally, we replace bounded iteration `loop X{...}` with conditional flow $\texttt{while}(x > 0 \wedge *)\{x = x - 1;...\}$. This can also be done for loop structures of $\mathcal{BJK}$ programs and allows to control the maximal length of traces within subprograms. Core programs can be easily transformed to $\mathcal{BJK}$ programs. The nesting hierarchy of loops of a Core program determines the loop structure of the $\mathcal{BJK}$ program. To emulate the $\mathcal{BJK}$ order constraints $x' \leqslant x + x$ and $x' \leqslant x * y$ in other representations besides $\mathcal{POL}$, additional loops within the CFG would be necessary.

Aside from $\mathcal{POL}$ programs, only variables of $\mathcal{MC}$ programs range over the integers. The known theoretical results for $\mathcal{MC}$ programs also hold for the natural domain. If the domain is restricted to $\mathbb{N}$, then $\mathcal{MC}$ order constraints subsume $\mathcal{SC}$ order constraints.

$\mathcal{DC}$ order constraints syntactically subsume $\mathcal{SC}$ order constraints.

The $\mathcal{DC}$ and $\mathcal{VASS}$ program representation allow order constraints $x' \leqslant x + 1$ and $x = x + 1$, respectively, which otherwise can only be represented in $\mathcal{POL}$. The assignment $x' = x + 1$ can be represented in $\mathcal{VASS}$ but not in $\mathcal{DC}$. On the other hand, to emulate a weak assignment $x' \leqslant x + 1$ in $\mathcal{VASS}$ an additional loop is necessary.

Table 3.1 provides all referenced results. Undecidable problems are marked with ✗ and open problems are marked with **?**. Here **1** denotes that it is an implicit property of the abstract program representation. We indicate with † that the result is derived from a more general result. With ‡ in $\mathcal{VASS}$ we indicate that the property does not hold for all $\mathcal{VASS}$ programs but a restricted subset, which depends on a decidable criterion on quasi-ranking functions.

Abstract Program		Termination		Bounded Termination		Polynomial Runtime Complexity	
Loop		✓	**1**	PRIMREC	**1**	✗	✗
Core		✓	**1**	PRIMREC	**1**	polynomially bounded [39]	P
Core	tight bounds	✓	**1**	PRIMREC	**1**	$\Theta(N^k), k \in \mathbb{N}$ [33]	EXPTIME
Core	weak assignment	✓	**1**	PRIMREC	**1**	polynomially bounded [39]	PSPACE
Core	`X := 0`	✓	**1**	PRIMREC	**1**	polynomially bounded [28]	PSPACE
Core	`X := Y+1`	✓	**1**	PRIMREC	**1**	**?** [34]	**?**
Core	definite loop	✓	**1**	PRIMREC	**1**	✗ [34]	✗
$(\mathcal{BJK}, \mathbb{N})$	fan-in free	✓	**1**	PRIMREC	**1**	polynomially bounded [37]	P
$(\mathcal{SC}, \mathbb{N})$	fan-out free	✓	PSPACE [110]†	$\mathcal{O}(N^{\lvert V\rvert})$ [38]†	PSPACE	$\Theta(N^k), k \in \mathbb{N}$ [157]	PSPACE
$(\mathcal{SC}, \mathbb{N})$		✓	PSPACE [110]	$\mathcal{O}(N^{\lvert V\rvert})$ [38]†	PSPACE	$\Theta(N^k), k \in \mathbb{Q}$ [55]	**?**
$(\mathcal{MC}, \mathbb{Z})$		✓	PSPACE [29]	$\mathcal{O}(N^{\lvert V\rvert})$ [38]	PSPACE	**?**	**?**
$(\mathcal{VASS}, \mathbb{N})$		✓	P [45]	WCPN [112]	P	$\Theta(N^k), k \in \mathbb{N}$ [45]‡	P
$(\mathcal{DC}, \mathbb{N})$	fan-in free	✓	PSPACE [27]	**?**	**?**	**?**	**?**
$(\mathcal{DC}, \mathbb{N})$		✗	✗	✗	✗	✗	✗
$(\mathcal{POL}, \mathbb{Z})$		✗	✗	✗	✗	✗	✗

Table 3.1: Overview of Decidable Properties of Abstract Program Representations.

3.5 Automated Resource Analysis with KoAT

Brockschmidt et al. [50, 51] investigate automated runtime and size complexity analysis of integer programs. The approach is implemented in the tool KoAT[2] and focuses on modularity via *alternating runtime and size complexity* analysis. KoAT is used for the resource analysis of C, or more precisely LLVM programs, Java bytecode programs, and term rewrite systems. The tool llvm2KITTeL[3] provides a lossy abstraction of LLVM programs to constraint transition systems (Falke et al. [68]), which is sound for termination, runtime and size analysis. This abstraction captures the semantics of bytecode programs closely, suitably abstracting unsupported operations like array access and pointer arithmetic. The AProVe[4] verifier (Giesl et al. [82]) uses KoAT as back-end for the resource analysis of Java bytecode programs (Frohn and Giesl [76]). The proposed abstraction for Java bytecode programs incorporates numerical abstractions for heap allocated data. The tool is also used as back-end for the automated runtime complexity analysis of term rewrite systems (Naaf et al. [122]).

In what follows, we mostly restrict to the core approach that is presented in [50]. We summarise the extensions of [51] at the end of this section.

3.5.1 Polynomial Constraints Programs

We comment on the program representation processed by KoAT.

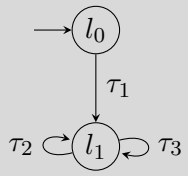
Program Representation

Programs in Brockschmidt et al. [51] are called *integer transition systems* and are analogues to constraint transition systems with integer-valued variables. The implementation processes a more restricted format in which constraints are conjunctions of (in)equalities of polynomial expressions with integer coefficients. Disjunctive control flow is represented using multiple transitions. We call such programs *polynomial constraints programs* (or $\mathcal{POL}$ programs).

$\mathcal{POL}$ programs generalise textbook Goto programs (or any equivalent model) where assignments and conditions are restricted to polynomial expressions. This often allows a straightforward translation from unstructured program representations, such as LLVM or Java bytecode, to $\mathcal{POL}$ programs. For termination and upper bound resource analysis, unsupported operations are usually approximated with unconstrained assignments (or unbounded non-determinism). The representation is expressive enough to capture common domains for numeric invariant generation, such as the octagon and polyhedra domain. Hence, $\mathcal{POL}$ programs can be naturally augmented with assumptions and invariants that are given by the user or inferred with external tools.

[2] https://github.com/s-falke/kittel-koat/
[3] https://github.com/s-falke/llvm2kittel/
[4] http://aprove.informatik.rwth-aachen.de/

Example 3.19 (Motivating Example)**.** We recall the motivating example from the preliminary discussion (see Example 3.11). It is easy to see, that the program is indeed a $\mathcal{POL}$ program.



$$\tau_1 \colon \langle l_0, l_1, \quad x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle$$
$$\tau_2 \colon \langle l_1, l_1, x > 0 \wedge x' = x - 1 \wedge y' = y + x \wedge z' = x + y \rangle$$
$$\tau_3 \colon \langle l_1, l_1, z > 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z - 1 \rangle$$

Motivation. The program representation is very expressive and precise, in fact, it is a well-researched representation that allows to make use of well-known established results from the literature. The upper bound inference relies on the synthesis of *polynomial ranking functions*. However, to make the approach viable in practice *modularity* is essential.

Program Abstraction

With regard to the overview given in Figure 3.1, we assume that $\mathcal{POL}$ programs are obtained from iCTS programs by *syntactically* restricting to polynomial order constraints and decorating the programs with additional *supporting invariants* that are obtained from standard numeric invariant generation. The tool KoAT infers additional invariants based on the octagonal domain using the APRON[5] library [102].

Bound Analysis

In this section we present the key aspects of the runtime analysis of KoAT. We conclude with an illustrative example.

Norms and Bound Expressions

The approach expresses the runtime and size complexity in terms of a fixed set of norms, that is, the absolute value of all program variables. This is a design choice that restricts the problem space for the size bounds of interest to the absolute value of all variables and promotes the application of dedicated methods. Complexity bounds are expressions composed of non-negative constants, the absolute value of variables, maximum, addition, multiplication and exponentiation. We observe that complexity bounds are weakly monotone with respect to the chosen norm, such that an increase in the absolute value of a variable implies an increase in the complexity. When depicting bound expressions we often omit the valuation if it is clear from the context. Further, we write $|x|$ instead of $\mathsf{abs}(\sigma(x))$ to denote the absolute value of x.

[5] `http://apron.cri.ensmp.fr/library/`

Example 3.20 (Bound Expression)**.** Consider the following two programs.

$$\begin{array}{ll} \texttt{while}(x > 0) & \texttt{while}(x < 0) \\ \quad x = x - 1 & \quad x = x + 1 \\ \quad y = y + 1 & \quad y = y - 1 \end{array}$$

The complexity bound on the runtime complexity that is inferred by KoAT for both programs is $|x|$, i.e. the number of iterations is bounded by the absolute value of the initial value $\sigma_0(x)$. The complexity bound on the size complexity of the norm $|y|$ that is inferred by KoAT is $|y| + |x|$, i.e. the final valuation of y is in the range of $-(|y| + |x|)$ and $|y| + |x|$ with respect to the initial values $\sigma_0(x)$ and $\sigma_0(y)$.

In what follows we provide an overview of the worst-case runtime and size analysis presented in Brockschmidt et al. [50]. First, we introduce the notion of global and local transition bounds, as well as global and local size bounds. The algorithm does not rely on an explicit notion of decomposition but expresses the runtime bound in a *top-down* fashion with mutual recursive dependencies of local and global bounds. Afterwards, we provide an example inference.

In the rest of this section let $\mathcal{T}$ be a constraint transitions system such that there is a single initial location $l_0 \in \mathsf{Loc}$, i.e. there is no transition in $\mathcal{T}$ with target location l_0, and all locations in $\mathcal{T}$ are reachable from l_0. Moreover, any proper subprogram $\mathcal{T}' \subset \mathcal{T}$ that we consider is connected and does not contain l_0.

Global Transition Bounds

A global transition bound for $\tau \in \mathcal{T}$ bounds the maximal number of occurrences of the transition τ in any program run of $\mathcal{T}$ in terms of the initial valuation (or absolute value thereof). More formally, a *global transition bound* for transition $\tau \in \mathcal{T}$ is a complexity bound $\mathsf{tb}_{\mathcal{T}}(\tau)$ such that

$$\underline{\mathsf{tb}}_{\mathcal{T}}(\tau) \succcurlyeq \mathsf{rc}^{l_0}_{\tau/\mathcal{T}} \,.$$

We recall that $\to_{\tau/\mathcal{T}} = \to^*_{\mathcal{T}} \cdot \to_\tau \cdot \to^*_{\mathcal{T}}$. The runtime complexity problem of $\mathcal{T}$ can then be expressed as the sum of all global transition bounds for $\tau \in \mathcal{T}$. We present the method to infer global transition bounds after introducing some additional notions.

Local Transition Bounds

Local transition bounds are the specialisation of global transition bounds for subprograms $\mathcal{T}' \subseteq \mathcal{T}$. A *local transition bound* for transition $\tau \in \mathcal{T}'$ is a complexity bound $\mathsf{tb}^l_{\mathcal{T}'}(\tau)$ such that

$$\underline{\mathsf{tb}}^l_{\mathcal{T}'}(\tau) \succcurlyeq \mathsf{rc}^l_{\tau/\mathcal{T}'} \,.$$

The local transition bound is specialised for a specific location. Consider for instance a subprogram that forms a SCC in the CFG with multiple entry locations. Then, the local transition bounds can be inspected individually with respect to all entry locations. One approach that is implemented in KoAT to infer local transition bounds is by generating non-trivial quasi-polynomial ranking functions (see Section 3.3.5 on page 23 for details).

Local Size Bounds

The local size bound (or local growth) captures the change of a norm, i.e. the growth of the absolute value of a variable with respect to a single transition. More formally, a *local size bound* on variable x with respect to transition $\tau\colon \langle l, l', \phi\rangle$ is a complexity bound $\mathsf{sb}_\tau(x)$ such that

$$\underline{\mathsf{sb}}_\tau(x) \succcurlyeq \lambda\sigma.\ \mathsf{sup}\{|\sigma'(x)| \mid \sigma, \sigma' \models \phi\}\ .$$

This is a specialisation of *transition invariants* for the chosen norm (see Section 3.3.7 on page 26). If τ is fixed or not important we present local size bounds in form of inequalities, e.g. $|x'| \leqslant |y|$. KoAT uses the following *syntactical* classification of local size bounds.

- *identity:* assignment of a variable or constant, e.g. $|x'| \leqslant |y|$, $|x'| \leqslant 0$
- *increment:* assignment of a variable plus a constant, e.g. $|x'| \leqslant |y| + 1$
- *additive:* assignment of a sum of variables plus a constant, e.g. $|x'| \leqslant |x| + |z| + 1$
- *multiplicative:* assignment of a weighted sum of variables plus a constant, e.g. $|x'| \leqslant 2(|y| + |z| + 1)$

The classification is used to infer closed-form expressions of global size bounds. We provide more details below.

Local size bounds can be inferred for all transitions and variables individually and imply *flow dependencies* of variables along transitions. For example, let $|x'| \leqslant |y| + |z|$ be a local size bound, then the norm $|x'|$ at the target location depends on y and z. The local size bounds are inferred in KoAT using syntactical pattern matching and constraint solving. For the latter, consider that we can adapt the synthesis approach of ranking functions to find lower and upper bounds on x' w.r.t. constraint ϕ. Lower and upper bounds can then be combined to a local size bound taking the maximum of the absolute value of its coefficients. For example, suppose $\sigma, \sigma' \models 2y + (-4) \geqslant x'$ and $\sigma, \sigma' \models y + (-z) \leqslant x'$. Then $\sigma, \sigma' \models 2|y| + \mathsf{abs}(-4) \geqslant x'$ and $\sigma, \sigma' \models |y| + |z| \geqslant (-x')$ We obtain the local size bound $|x'| \leqslant 2|y| + |z| + 4$.

Global Size Bounds

A global size bound expresses a bound on the maximum of a norm at a target location of some transition. More formally, a *global size bound* on variable x with respect to transition $\tau \in \mathcal{T}$ is a complexity bound $\mathsf{sb}_{\tau/\mathcal{T}}(x)$ such that

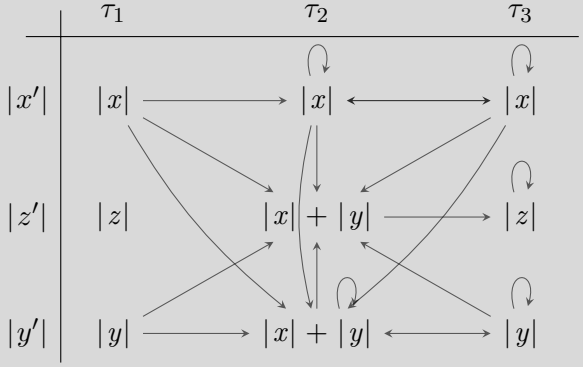
$$\underline{\mathsf{sb}}_{\tau/\mathcal{T}}(x) \succcurlyeq \lambda\sigma_0.\ \mathsf{sup}\{|\sigma'(x)| \mid (l_0, \sigma_0) \to_{\mathcal{T}}^* \cdot \to_\tau (l', \sigma')\}\ .$$

The global size bounds of a program are obtained by inspecting the flow dependencies between variables and obtaining closed-form bound expressions from local size bounds.

The flow dependencies that are induced by local size bounds can be represented as a graph, termed *result variable graph* in [50, 51]. The result variable graph of a program has nodes $(|x'|, \tau) \in \mathsf{Var} \times \mathcal{T}$. Each node is labelled by its local size bound $\mathsf{sb}_\tau(x)$. There

is an edge from $(|x'|_i, \tau_i)$ to $(|x'|_j, \tau_j)$, if (i) there exists a location $l \in \mathsf{Loc}$ in the CFG such that l is the target location of τ_i and the source location of τ_j, i.e. τ_j succeeds τ_i, and (ii) there is a flow-dependency from $|x|_i$ to $|x'|_j$, i.e. $|x|_i$ occurs in the local bound at $(|x'|_j, \tau_j)$.

Example 3.21 (Result Variable Graph of Example 3.19)**.** Below we depict the result variable graph for our running example. If we inspect the CFG of the program, then one of the interesting cases to consider is the size bound on $|z|$ for transition τ_3, which is also the local transition bound for τ_3. The result variable graph shows that it depends only on itself and $|x| + |y|$, which on the other hand depends on all transitions.



The analysis makes use of its strongly connected components (SCCs). We say that a SCC is trivial if it does not contain any edges, otherwise it is non-trivial. The global size bound of a variable that conforms to a trivial SCC in this graph can be expressed as the composition of its local size bound and global size bounds of its predecessors. Non-trivial SCCs denote cyclic dependencies of variables that grow under repetition. To obtain closed-form expressions for variables, global transition bounds, i.e. bounds on the repetition, and growth classes are considered. We illustrate the main idea of obtaining closed-form bound expressions with the next example.

Example 3.22 (Closed-Form Size Bounds)**.** Suppose that the global transition bound inferred for the loop is $|i|$. The closed-form expressions for all norms are given on the right side.

```
while( i > 0)
  i = i − 1 // identity          |i'| ⩽ |i|
  w = w + 2 // increment         |w'| ⩽ |w| + 2|i|
  x = x + k // additive          |x'| ⩽ |x| + |i|·|k|
  y = y + w // additive          |y'| ⩽ |y| + |i|·(|w| + 2|i|)
  z = z + z // multiplicative    |z'| ⩽ 2^(|i|·|z|)
```

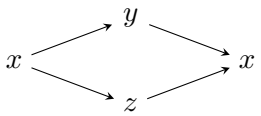
The valuation of i' is restricted by $|i|$ during evaluation. To obtain a bound on $|w'|$ we take the initial bound $|w|$ plus $|i|$ times the constant 2. The bound for $|x'|$ is obtained similarly, here k is a constant variable expression. There is a flow dependency from w to y but not vice-versa. The case for $|y'|$ is similar to the case $|x'|$ but depends on the closed-form of $|w'|$. The last case $|z'|$ is duplicating and implies an exponential growth.

In the previous example we capture the main idea of obtaining closed-form expressions for the individual classes. We restrict to a simple case where flow-dependencies induce only trivial SCCs in the result variable graph and non-trivial SCCs with one node, hence we do not obtain mutual recursive flow dependencies. We omit the technical details that are necessary to obtain closed-form expressions in the general case and refer to Brockschmidt et al. [50, 51]. Briefly, an upper bound can be obtained by inspecting all transitions of the SCC separately, like above, and taking the sum of all obtained bound expressions.

Additionally, the result variable graph is used to safely infer *duplicating flow*, which implies exponential growth under iteration.

Example 3.23 (Duplicating Flow)**.** Consider the following example. Duplicating flow manifests in *diamond* shaped flow patterns.

```
while( i > 0)
  i = i − 1
  y = x
  z = x
  x = y + z // duplicating
```



Inference of Global Transition Bounds

We present the two main methods to infer global transition bounds.

First, consider the subprogram $\mathcal{T}' = \mathcal{T}$. Then any local transition bound inferred on $\tau \in \mathcal{T}'$ is also a global transition bound on $\tau \in \mathcal{T}$. Local transition bounds are inferred via the synthesis of polynomial quasi-ranking functions.

Second, for a subprogram $\mathcal{T}' \subset \mathcal{T}$ a modular approach based on runtime and size complexity is applied. The proposed approach is akin to the informal discussion in Section 3.3.4 on page 19 of the modular runtime analysis of nested programs. An important detail of the analysis in KoAT is that it is *path-sensitive*. Here, path-sensitive means that runtime and size bounds are associated to transitions (rather than locations) and that transitions that flow into the subprogram are considered separately.

W.l.o.g. assume that there is only one incoming transition $\tau\colon \langle l, l', \phi\rangle \in \mathcal{T} \setminus \mathcal{T}'$ of the subprogram, i.e. the source location l is defined in $\mathcal{T} \setminus \mathcal{T}'$ and the target location l' is defined in $\mathcal{T}'$.

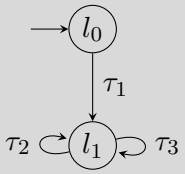
- A global transition bound for $\tau \in \mathcal{T}$ bounds the maximal number the subprogram $\mathcal{T}'$ can be entered via τ.
- A local transition bound for $\tau' \in \mathcal{T}'$ w.r.t. the subprogram $\mathcal{T}'$ bounds the number of occurrences of τ' in any evaluation of $\mathcal{T}'$. Crucially the bound is defined in terms of the input valuation at location l when entering $\mathcal{T}'$ via τ.
- A global transition bound for $\tau' \in \mathcal{T}$ for a single execution of the subprogram $\mathcal{T}'$ is obtained by providing global size bounds on the norms w.r.t. τ. We obtain a global transition bound for $\tau' \in \mathcal{T}$ for all executions of the subprogram $\mathcal{T}'$ by multiplying it with the global transition bound of τ.

In the case that there is more than one incoming transition $\tau \in \mathcal{T} \setminus \mathcal{T}'$ we inspect the individual cases separately and take the sum of the inferred bounds.

Strategy. We have presented the individual components of the complexity analysis in KoAT. Now, we combine those components to a strategy for the inference of complexity bounds on the runtime complexity. Local size and local transition bounds (for $\mathcal{T}' = \mathcal{T}$) only depend on $\mathcal{T}$ and can be inferred on-demand. Global size and transition bounds are propagated *top-down*, that is, SCCs in $\mathcal{T}$ and the variable flow are resolved in topological order alternating between the inference of global transition and global size bounds.

Lexicographic Ranking Function. Although, it is not immediate the inference is conceptually close to the decompositional approach based on lexicographic ranking functions (cf. Brockschmidt et al. [51] and Section 3.3.5). The main idea being, that each local bound, which is inferred for a subprogram, amounts to a component in the ranking function. As the construction is hidden in the individual steps we refer to it as being *implicit* in the overview of the tools in Section 3.8.

Example 3.24 (Runtime Inference)**.** We recall the running example and conclude with the upper bound inference on its worst-case runtime complexity.



$$\begin{aligned}
\tau_1 &\colon \langle l_0, l_1, \quad x' = x \quad \wedge y' = y \quad \wedge z' = z \rangle \\
\tau_2 &\colon \langle l_1, l_1, x > 0 \wedge x' = x - 1 \wedge y' = y + x \wedge z' = x + y \rangle \\
\tau_3 &\colon \langle l_1, l_1, z > 0 \wedge x' = x \quad \wedge y' = y \quad \wedge z' = z - 1 \rangle
\end{aligned}$$

Step 1: We assume $\mathcal{T}' = \mathcal{T}$ and try to infer local (and therefore global) transition bounds via the synthesis of polynomial quasi-ranking functions. We iterate this process until the synthesis fails. We obtain linear global transition bounds for τ_1 and τ_2.

- global transition bound $\mathsf{tb}_{\mathcal{T}}(\tau_1) = 1$ via RF $\eta(l_0; \sigma) = 1$ $\eta(l_1; \sigma) = 0$
- global transition bound $\mathsf{tb}_{\mathcal{T}}(\tau_2) = |x|$ via RF $\eta(l_0; \sigma) = \eta(l_1; \sigma) = \sigma(x)$

What is left to obtain an upper bound on the worst-case runtime complexity of $\mathcal{T}$, is to infer a global transition bound for τ_3. We set $\mathcal{T}' = \{\tau_3\}$.

Step 2: We infer the following local size bounds by inspecting each transition and norm separately.

- local size bounds for τ_1, τ_2 and τ_3

$$\begin{array}{lll}
|x'| \leqslant |x| & |x'| \leqslant |x| & |x'| \leqslant |x| \\
|y'| \leqslant |y| & |y'| \leqslant |x| + |y| & |y'| \leqslant |y| \\
|z'| \leqslant |z| & |z'| \leqslant |x| + |y| & |z'| \leqslant |z|
\end{array}$$

Step 3: We use the obtained local size bounds and global transition bounds to infer global size bounds.

– global size bounds for τ_1 and τ_2

$$\begin{array}{ll} |x'| \leqslant |x| & |x'| \leqslant |x| \\ |y'| \leqslant |y| & |y'| \leqslant |x|^2 + \ |x| + |y| \\ |z'| \leqslant |z| & |z'| \leqslant |x|^2 + 2|x| + |y| \end{array}$$

The indicated expressions bound the size of the norms at the target location.

Step 4: We synthesise a local transition bound for $\tau \in \mathcal{T}'$ w.r.t. $\mathcal{T}'$.

– local transition bound $\mathsf{tb}^{l_1}_{\mathcal{T}'}(\tau_3) = |z|$ via RF $\eta(l_1;\sigma) = \sigma(z)$.

Step 5: The local transition bound of τ_3 w.r.t. $\mathcal{T}'$ and location l_1 is $|z|$. The transitions τ_1 and τ_2 have target location l_1. The global transition bound of τ_3 is obtained by considering how often τ_1 and τ_2 can occur in an evaluation (their global transition bounds) and upper bounds on $|z|$ (their global sizebounds).

– global transition bound

$$\begin{aligned} \mathsf{tb}_{\mathcal{T}}(\tau_3) &= \mathsf{tb}_{\mathcal{T}}(\tau_1) \cdot \mathsf{tb}^{l_1}_{\mathcal{T}'}(\tau_3)[\mathsf{sb}_{\tau_1/\mathcal{T}}(x_i)/x_i] + \mathsf{tb}_{\mathcal{T}}(\tau_2) \cdot \mathsf{tb}^{l_1}_{\mathcal{T}'}(\tau_3)[\mathsf{sb}_{\tau_2/\mathcal{T}}(x_i)/x_i] \\ &= 1 \cdot |z|[\mathsf{sb}_{\tau_1/\mathcal{T}}(x_i)/x_i] + |x| \cdot |z|[\mathsf{sb}_{\tau_2/\mathcal{T}}(x_i)/x_i] \\ &= 1 \cdot |z|[|z|/z] + |x| \cdot |z|[|x|^2 + 2|x| + |y|/z] \\ &= |z| + |x| \cdot (|x|^2 + 2|x| + |y|) \end{aligned}$$

We have obtained global transition bounds for all transitions and therefore obtain the following upper bound on the worst-case runtime complexity.

$$\mathsf{tb}_{\mathcal{T}}(\tau_1) + \mathsf{tb}_{\mathcal{T}}(\tau_2) + \mathsf{tb}_{\mathcal{T}}(\tau_3) = 1 + |x| + |x|^3 + 2|x|^2 + |x| \cdot |y| + |z| \ .$$

Extensions

In this section we comment on some extension that are presented in [51].

Recursive Programs. KoAT supports multiple recursive calls by collecting calls on the right-hand sides of transitions. The notion of transitions is extended to $\langle l, \{l'_i\}, \phi\rangle$, i.e. the target is a set of locations. The proposed approach is conceptually similar to the original one, but requires a stronger notion of ranking functions.

Cost Models. KoAT supports *weighted* integer transition systems, that is, transitions are associated with a bound expression. Then, the runtime corresponds to a specific cost model in which all transitions are weighted with bound expression constant one. This enables some interesting use cases. For instance, by adapting the program abstraction suitably, KoAT can analyse different cost models, such as the number of function calls or

heap allocations. Additionally, it allows for a *bottom-up* strategy, by isolating subprograms of the original problem which can be analysed separately and incorporated again as weight. Morally, subprograms are replaced by its weight (or cost) of evaluating it. The tool AProVe makes use of this strategy to analyse (non-recursive) function calls for Java programs individually (Frohn and Giesl [76]).

Exponential Growth Rates. Whereas the initial version focuses on polynomial bounds, [51] provides a more refined approach for analysing exponential bounds. This includes a variation of ranking functions for exponential runtime and more precise size analysis.

3.6 Automated Resource Analysis with Loopus

Zuleger et al. [158] and Sinn et al. [145, 146] investigate different abstract program representations that are known from the literature for the inference of (symbolic) bounds on loops for C and LLVM programs. The central idea is to use the abstract domains to reflect the change in a norm and specialise the inference of upper bound resource analysis. In what follows, we outline the central concepts that have been implemented in the tool Loopus[6].

Overview. At first, in Section 3.6.1 we discuss reachability bound analysis of integer programs using an abstract program representations based on *monotonicity constraints* [158]. Then, in Section 3.6.2 we provide an overview of the resource analysis using constraint transition systems with *monotone difference constraints* [145]. Finally, in Section 3.6.3 we recall the bound analysis with *difference constraints* [146], which forms the basis for the most recent version of the tool Loopus at the time of writing.

3.6.1 Monotonicity Constraints Programs

Zuleger et al. [158] investigate bound analysis of integer programs using constraint transition systems with *monotonicity constraints* over the integer domain. The program representation is inspired by earlier work on the *size-change termination* problem by Lee et al. [110]. We have discussed theoretical properties of monotonicity constraints programs before in Section 3.4.4. Next, we recall its definition.

Program Representation

A *monotonicity order constraint* is an inequality $x > y$ or $x \geqslant y$ for $x, y \in \mathsf{Var} \cup \mathsf{Var}'$, and a *monotonicity constraint* is a conjunction of monotonicity order constraints. A *monotonicity constraints program* $\mathcal{MC}$ is a CTS where transitions are associated with a monotonicity constraint and the valuation of the variables range over the integers. The tool Loopus restricts to $\mathcal{MC}$ programs that have a *reducible* control flow graphs with a unique initial location l_0 (see Definition 3.6 on page 15).

Motivation. The analysis based on $\mathcal{MC}$ constraints is motivated by the idea that loops often admit state based behaviour with different loop phases. Loop invariants that are inferred from standard numeric invariant domains such as the octagon and polyhedra domain reason about all paths. In contrast, the $\mathcal{MC}$ domain is naturally disjunctive. Given a finite set of variables, the domain of $\mathcal{MC}$ constraints is finite and so is its powerset construction. These properties motivate to use $\mathcal{MC}$ constraints to infer *disjunctive invariants*. Notably, $\mathcal{MC}$ constraints syntactically capture the ranking function properties *bounded* ($x \geqslant 0$), *non-increasing* ($x \geqslant x'$) and *decreasing* ($x > x'$) (see Definition 3.14 on page 21) .

[6] `http://forsyte.at/software/loopus/`

Example 3.25 (Motivating Example)**.** The following program illustrates one of the motivating examples in Zuleger et al. [158].

```
     main(n)
      i = 0
l1 :  while( i < n)
        i = i + 1
        j = 0
l2 :    while( i < n ∧ ∗)
          i = i + 1
          j = j + 1
        if( j > 0)
          i = i −1
```

$$
\begin{aligned}
\tau_0 &: \langle l_0, l_1, \quad \wedge i' = 0 \quad \wedge j' = j \quad \wedge n' = n \rangle \\
\tau_1 &: \langle l_1, l_2, i < n \wedge i' = i + 1 \wedge j' = 0 \quad \wedge n' = n \rangle \\
\tau_2 &: \langle l_2, l_2, i < n \wedge i' = i + 1 \wedge j' = j + 1 \wedge n' = n \rangle \\
\tau_3 &: \langle l_2, l_1, j > 0 \wedge i' = i - 1 \wedge j' = j \quad \wedge n' = n \rangle \\
\tau_4 &: \langle l_2, l_1, j \leqslant 0 \wedge i' = i \quad \wedge j' = j \quad \wedge n' = n \rangle
\end{aligned}
$$

Program Abstraction

The proposed abstraction and bound analysis is based on the *reachability bound problem* for individual loop headers (cf. Gulwani and Zuleger [86]). Formally, suppose l is a location and $\mathcal{T}' = \{\tau \in \mathcal{T} \mid \tau \text{ has target location } l\}$. Then, the *reachability bound* for location l is a complexity bound $\mathsf{rb}_{\mathcal{T}}(l)$ such that

$$\underline{\mathsf{rb}}_{\mathcal{T}}(l) \succcurlyeq \mathsf{rc}^{l_0}_{\mathcal{T}'/\mathcal{T}} \,.$$

The reachability bound problem inspects the maximal number of visits to a specified location. The proposed approach restricts to reducible CFGs. In a reducible CFG all SCCs have a unique entry location, its *loop header*. A bound on the worst-case runtime for the whole program can be obtained by analysing each loop header individually.

The proposed program abstraction is akin to the *transition predicate abstraction* presented by Podelski and Rybalchenko [133]. For the viability of the approach it is essential that variables of the $\mathcal{MC}$ domain represent symbolic expressions. Those symbolic expressions are usually called *norms* and are inferred with heuristics. A standard heuristic is to collect norms from conditions that reflect the ranking properties bounded and decreasing. For instance, expressions $x - 1$ and $y - z$ are inferred for $\texttt{while}(x > 0 \wedge y > z)\{\ldots\}$. Here, x and $y - z$ are bounded by 0 at the entry of the loop. For the loop to terminate, we expect that one of them is decreasing, which amounts to infer the relation $x > x'$ or $y - z > y' - z'$ for the body of the loop.

The abstraction performs a *pathwise* abstraction of simple cyclic paths from the loop header to itself. In the process nested loops are summarised bottom-up constructing disjunctive invariants that represent the *reflexive and transitive transition invariant* of the loop. Consider a transition system with a self-loop $\tau\colon \langle l, l, \phi \rangle$. Here, the reflexive and transitive transition invariant is a disjunction of monotonicity constraints ϕ' such that $\sigma, \sigma' \models \phi'$ for all traces $(l, \sigma) \to^*_\tau (l, \sigma')$. The result of the abstraction is a set of transitions with a single location, the loop header.

Additional precision is gained by *contextualisation*, that is, the refinement of the control flow graph based on the feasibility of sequentially executing transitions within a specified

context. Conceptually, contextualisation corresponds to the inference of loop phases. Example 3.28 on page 48 illustrates the application of contextualisation.

Example 3.26 (Cont'd from Example 3.25)**.** We inspect the reachability bound of l_1. In doing so, the inner loop is summarised by its reflexive and transitive transition invariant. This invariant is obtained by repeatedly (i) abstracting transition constraints to $\mathcal{MC}$ constraints, and (ii) composing transitions, until a fixed-point is reached. The following loop summary is obtained for the inner loop at program location l_2. We display the disjunctive constraint using multiple transitions. Here, τ_{2a} represents the case where the inner loop is not iterated once and τ_{2b} represents the case where the inner loop is iterated arbitrarily often, but at least once.

$$\tau_{2a}\colon \langle l_2, l_2, \quad n' - i' = n - i \land j' = j \rangle$$
$$\tau_{2b}\colon \langle l_2, l_2, n - i > 0 \land n' - i' < n - i \land j' > j \rangle$$

The proposed pathwise abstraction inspects the composition of all simple cyclic paths from l_1 to l_1, which are $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{\tau_3} l_1$ and $l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{\tau_4} l_1$. In doing so, the inner loop at location l_2 is replaced by its loop summary. The following transitions are obtained from the $\mathcal{MC}$ abstraction.

$$l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{\tau_3} l_1\colon \langle l_1, l_1, n - i > 0 \land n - i > n' - i' \land j' > 0 \rangle$$
$$l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{\tau_4} l_1\colon \langle l_1, l_1, n - i > 0 \land n - i > n' - i' \land j' \geqslant 0 \rangle$$

Bound Analysis

The bound analysis is akin to the *lexicographic decomposition* presented in Section 3.3.5 on page 24. Observe that the ranking properties bounded, decreasing, and non-increasing can be expressed as $\mathcal{MC}$ constraints, $n \geqslant 0, n > n'$, and $n \geqslant n'$, respectively. At this stage the decomposition is purely syntactical and can be computed effectively. The bound analysis symbolically unfolds the additive and multiplicative dependencies of the decomposition. Morally, sequential components are summed up and nested components are multiplied. Closed-forms are obtained by incorporating global size invariants on variables (or norms) w.r.t. the input. These are obtained by standard numeric invariant generation.

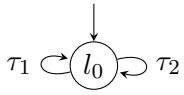
Example 3.27 (Cont'd from Example 3.26)**.** The expression $n - i$ is bounded and decreasing. An upper bound $n - i$ in terms of the input is n since $i = 0$ before reaching l_1. The reachability bound $\mathsf{rb}_{\mathcal{T}}(l_1) = \mathsf{max}(0, n)$ is obtained for l_1.

We remark that the proposed abstraction and bound analysis do not capture the growth of norms. Although, a relation $x' \leqslant x + 1$ can be expressed in this domain (here $x + 1$ is a norm), there is no means to obtain closed-form bound expressions for repeated application of this relation. Here, the connection to the size-change abstraction presented in Lee et al. [110] becomes apparent, in which expressions that grow are not reflected within the abstraction.

Next, we exemplify the application of *contextualisation*. Morally, we associate constraints to locations and check the feasibility of sequentially applying two transitions. This technique is independent of the constraint domain but motivated to refine the CFG of disjunctive invariants.

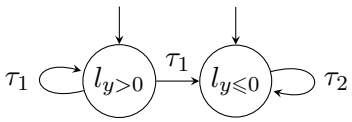
Example 3.28 (Contextualisation)**.** The following example illustrates an application of contextualisation.

```
    main(x, y, n)
l0 : while(x > 0)
       if(y > 0)
         x = n
         y = y − 1
       else
         x = x − 1
```



$$\tau_1 \colon \langle l_0, l_0, x > 0 \wedge y > 0 \geqslant \wedge\, x = n \;\wedge y > y' \wedge n' = n \rangle$$
$$\tau_2 \colon \langle l_0, l_0, x > 0 \qquad\qquad \wedge\, x > x' \wedge y \geqslant y' \wedge n' = n \rangle$$

It is easy to see that transition τ_1 cannot succeed τ_2 in any program trace. The system below provides an equivalent program with a refined CFG. We restrict to inspecting the case whether the relation $y > 0$ holds.



$$\tau_1 \colon x > 0 \wedge y > 0 \geqslant \wedge\, x' = n \wedge y > y' \wedge n' = n$$
$$\tau_2 \colon x > 0 \qquad\qquad \wedge\, x > x' \wedge y \geqslant y' \wedge n' = n$$

Most relevant, the bound that is inferred by the algorithm is better for the refined program. This is due to the fact that the CFG of the original program is conceived as a single SCC with nested loops and the CFG of the refined program is conceived as two sequential SCCs. The bound inference algorithm multiplies the bounds of nested loops, while adding the bounds of sequential loops. See also the discussion on modular bound analysis in Section 3.3.4 on page 19. With the size invariant $\mathsf{max}(x, n)$ on x we obtain the bound $\mathsf{max}(0, \mathsf{max}(x, n)) \cdot \mathsf{max}(0, y)$ for the original program and $\mathsf{max}(0, \mathsf{max}(x, n)) + \mathsf{max}(0, y)$ for the refined program.

3.6.2 Monotone Difference Constraints Programs

Sinn et al. [145] propose *(parameterised lossy) vector addition systems with states (parameterised lossy $\mathcal{VASS}$)* as suitable abstract program representation for bound analysis. $\mathcal{VASS}$ programs have been introduced by Hopcroft and Pansiot [101] and the subclass *lossy $\mathcal{VASS}$* by Bouajjani and Mayr [42]. Here, lossy indicates that assignments are interpreted weakly, e.g. $x' \leqslant x+1$ instead of $x' = x+1$. The term *parameterised* indicates the support of symbolic constant expressions k in assignments $x' \leqslant x+k$. In what follows, we use the equivalent notion of *monotone difference constraints* programs $\mathcal{MDC}$ which have been introduced in Sinn [144]. We have discussed known theoretical properties of $\mathcal{VASS}$ and $\mathcal{MDC}$ programs in Section 3.4.5.

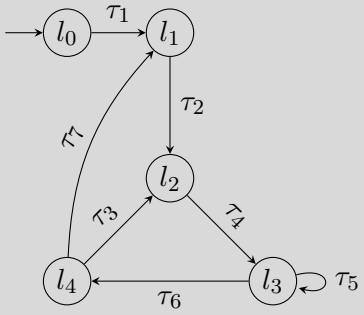
Program Representation

A *monotone difference order constraint* is an inequality $x' \leqslant x + k$ with $x' \in \mathsf{Var}'$, $x \in \mathsf{Var}$ and k is a (symbolic) constant expression over $\mathbb{Z}$. The variables range over $\mathbb{N}$. We say that a constraint $x' \leqslant x + k$ is an *increment* if $k \geqslant 0$ and a *decrement* if $k < 0$. A *monotone difference constraints program* $\mathcal{MDC}$ is a CTS with monotone difference order constraints over the natural numbers. The resource analysis in the tool Loopus restricts to $\mathcal{MDC}$ programs that have a *reducible* control flow graph (see Definition 3.6 on page 15).

Motivation. $\mathcal{MDC}$ constraints are syntactically restricted in comparison to standard programs. More specific, assignments of variables and conditional flow ($x' \leqslant y \wedge y \neq x$) cannot be expressed precisely. Adding support for symbolic constants allows incorporating external knowledge in terms of size bounds. For example, $x' \leqslant x + \mathsf{sb}(y)$ where $\mathsf{sb}(y)$ denotes a bound on the size complexity of y. Most noteworthy, the syntactic restriction of $\mathcal{MDC}$ constraints allows for a efficient and controlled resource analysis.

Example 3.29 (Motivating Example)**.** Consider one of the motivating examples of [145]. It is easy to see that all assignment can be abstracted to $\mathcal{MDC}$ constraints when interpreting assignments weakly, i.e. with not greater than.

```
     main(m)
      i = m;  n = 0
l1 :  while( i > 0)
        i = i − 1
l2 :    if(∗)
          n = n + 1
        else
l3 :      while(n > 0 ∧ ∗)
            n = n − 1
l4 :    skip
```



Program Abstraction

$\mathcal{MDC}$ constraints represent increment and decrement operations on a tuple of norms. To analyse C or iCTS programs a suitable abstraction is necessary. The authors propose an abstraction based on inferring transition invariants and symbolic execution. The program abstraction constitutes of three steps: (i) guessing norms, (ii) abstraction of transitions, and (iii) abstraction of the control flow.

By the definition of $\mathcal{MDC}$ constraints, a candidate norm x satisfies the following properties. The predicate $x' \geqslant 0$ is invariant for all locations and for every transition the predicate $x' \leqslant x + k$ is invariant for some constant expression k. We emphasize that k is not restricted to a number but is considered to be a constant expression, hence it is possible to obtain k as a global invariant.

Central to the proposed analysis is the inspection of individual loop paths. We recall, if the CFG is reducible then each SCC of the CFG has a unique entry node, its loop

header, which dominates all nodes of the SCC. A loop path is a simple cyclic path $l \to^* l$ starting at the loop header. When inferring upper bounds, the proposed approach uses norms as ranking arguments to bound the number of iterations of a loop path. Candidate expressions for norms are obtained by inspecting loop paths. An expression should satisfy the properties *bounded* and *decreasing* for at least one loop path (see Definition 3.14 on page 21). In what follows, given a (loop) path of the CFG $l_0 \xrightarrow{\tau_1} l_1 \xrightarrow{\tau_2} l_2 \cdots$ we write $\tau_1 \cdot \tau_2 \cdots$.

Example 3.30 (Cont'd from Example 3.29)**.** Location l_1 and l_3 denote labels for loop header. The paths $\tau_2 \cdot \tau_3 \cdot \tau_7$ and $\tau_2 \cdot \tau_4 \cdot \tau_6 \cdot \tau_7$ are loop paths for l_1, and the path τ_5 is a loop path for l_3. The path $\tau_6 \cdot \tau_7 \cdot \tau_2 \cdot \tau_4$ is not a loop path for l_3 since l_3 is dominated by l_1. The norms $\mathsf{max}(0, i)$ and $\mathsf{max}(0, n)$ are obtained from the loop conditions. The expression $\mathsf{max}(0, i)$ is decreasing for all loop paths of l_1 and $\mathsf{max}(0, n)$ is decreasing for all loop paths of l_3.

To simplify the bound analysis, $\mathcal{MDC}$ programs are further abstracted to transition systems with only one location. The proposed *control flow abstraction* extracts for each loop header all loop paths and fixes source and target location to a single location. We write $\pi = \tau_1 \cdot \tau_2 \cdots$ for the transition that is obtained by the composition of all transitions along the path. The composition of transitions is straightforward for $\mathcal{MDC}$ constraints. For instance, the sequential application of $x' \leqslant x + k_1$ and $x' \leqslant x + k_2$ is modelled by $x' \leqslant x + k_1 + k_2$.

Example 3.31 (Cont'd from Example 3.30)**.** We illustrate the loop paths that are obtained from the control flow abstraction as single transitions.

$$\begin{aligned} \tau_2 \cdot \tau_3 \cdot \tau_7 = \pi_1 \colon \langle \bullet, \bullet, n' \leqslant n + 1 \wedge i' \leqslant i - 1 \rangle \\ \tau_2 \cdot \tau_4 \cdot \tau_6 \cdot \tau_7 = \pi_2 \colon \langle \bullet, \bullet, n' \leqslant n \quad \wedge i' \leqslant i - 1 \rangle \\ \tau_5 = \pi_3 \colon \langle \bullet, \bullet, n' \leqslant n - 1 \wedge i' \leqslant i \rangle \end{aligned}$$

Bound Analysis

The proposed bound analysis inspects the maximal number a loop path π can occur in any trace of $\mathcal{T}$. Let π be the transition that is obtained from the path $\tau_1 \cdots \tau_n$. Then the *path bound* of π is a complexity bound $\mathsf{pb}_{\mathcal{T}}(\pi)$ such that

$$\underline{\mathsf{pb}}_{\mathcal{T}}(\pi) \succcurlyeq \mathsf{rc}^{l_0}_{\pi/\mathcal{T}} \,.$$

The worst-case runtime of $\mathcal{T}$ is then given by the sum of the path bounds of all loop paths (plus some constant for all transitions that do not appear in any loop path).

In the following, let $\mathcal{T}$ denote the transition system that is obtained from the control flow abstraction of a $\mathcal{MDC}$ program. We summarise the worst-case runtime analysis of $\mathcal{T}$. First, the proposed algorithm attempts to construct a *lexicographic combination of ranking functions* (see Section 3.3.5 on page 23). For a program $\mathcal{T}$ with n transitions

the constructions returns a LexRF $\langle x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \rangle$ with n components, if successful. By definition, $\mathcal{MDC}$ constraints are bounded from below by 0. The ranking properties decreasing and non-increasing can be checked syntactically. This allows for a simple and efficient construction of the LexRF.

(i) Choose $\tau \in \mathcal{T}$ such that there is a norm x that is decreasing for τ and non-increasing for all transitions in $\mathcal{T}$. Append $x \colon \tau$ as the next component of the LexRF.

(ii) Repeat for $\mathcal{T} \setminus \{\tau\}$.

The construction may fail if item (i) cannot be fulfilled. If the inference of a ranking function is successful, the path bound can be inferred for all components. The inference relies on the following observation for *incremental* growth. Consider the transition $\langle \tau \colon \bullet, \bullet, x' \leqslant x - 1, y' \leqslant y + k \rangle$ for $k \geqslant 0$.

$$
\begin{array}{l}
\texttt{while}(x > 0) \\
\quad \texttt{assume}(x \geqslant 0 \ \wedge \ y \geqslant 0 \ \wedge \ k > 0) \\
\quad x = x - 1 \\
\quad y = y + k
\end{array}
$$

Then the final value of y is bounded by $y + x \cdot k$. In particular, the growth of y is a (parameterised) linear expression. We emphasise that constraints do not express resets (e.g. $x' \leqslant N$) and that the decomposition that is implied by the LexRF is not multiplicative but additive, that is, the iteration bounds of the components are summed up rather than multiplied. Intuitively, for a LexRF $\langle x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \rangle$ that is obtained from a $\mathcal{MDC}$ program the worst-case runtime can be expressed by successive loops, therefore maximising the valuation of all variables.

$$
\begin{array}{l}
\texttt{while}(x_1 > 0) \\
\quad \tau_1 \\
\ldots \\
\texttt{while}(x_n > 0) \\
\quad \tau_2
\end{array}
$$

Here, x_i is decreasing for τ_i and the valuation of x_j for $i < j$ may increase, though it can always be approximated using the previous observation.

Example 3.32 (Cont'd from Example 3.31)**.** This program has a three component LexRF $\langle i : \pi_1, i : \pi_2, n : \pi_3 \rangle$. We obtain the bounds $\mathsf{max}(0, i)$ for π_1, $\mathsf{max}(0, i)$ for π_2, and $\mathsf{max}(0, n) + \mathsf{max}(0, i) \cdot 1$ for π_3. Taking the initial assignments $i = m$ and $n = 0$ into account, we obtain the path bounds $\mathsf{pb}(\pi_1) = \mathsf{pb}(\pi_2) = \mathsf{pb}(\pi_3) = \mathsf{max}(0, m)$. Thus, $\mathsf{max}(0, 3m)$ is an upper bound on the worst-case runtime.

The proposed construction of the LexRF returns a ranking function with n components, in which n is the number of transitions. Sinn et al. [145] propose a preprocessing step to *merge* transitions, thus reducing the number of transitions and improving precision. Two transitions τ_1 and τ_2 can be merged if all decrements decrease by the same amount. Transition τ_3 is obtained from τ_1 and τ_2 by taking the maximum k for each constraint.

Example 3.33 (Cont'd from Example 3.32)**.** Transition π_1 and transition π_2 can be merged into

$$\pi_{12} \colon \langle \bullet, \bullet, n' \leqslant \mathsf{max}(n, n+1) \wedge i' \leqslant i-1 \rangle \ .$$

Then, we obtain a LexRF $\langle i \colon \pi_{12}, n \colon \pi_3 \rangle$ and an improved upper bound $\mathsf{max}(0, 2m)$.

We conclude with the runtime analysis of a variation of the motivating example from the overview.

Example 3.34 (MDC analysis of Example 3.11)**.** The referenced program is not a $\mathcal{MDC}$ program. We consider a variation (which is not equivalent). The assignment $z' = y + x$ is replaced with $z' \leqslant z + \mathsf{sb}(x)$, where $\mathsf{sb}(x)$ is a symbolic constant that represents the size bound of x. For the norms $\mathsf{max}(0, x)$ and $\mathsf{max}(0, z)$ the following transitions are obtained from the control flow abstraction.

$$\begin{aligned} \tau_2 \colon \langle \bullet, \bullet, x' \leqslant x - 1 \wedge z' \leqslant z + \mathsf{sb}(x) \rangle \\ \tau_3 \colon \langle \bullet, \bullet, x' \leqslant x \quad\ \wedge z' \leqslant z - 1 \rangle \end{aligned}$$

We construct a two component LexRF $\langle x \colon \tau_2, z \colon \tau_3 \rangle$. The inferred upper bounds are $\mathsf{pb}(\tau_2) = \mathsf{max}(0, x)$ and $\mathsf{pb}(\tau_3) = \mathsf{max}(0, z) + \mathsf{max}(0, x) \cdot \mathsf{max}(0, \mathsf{sb}(x))$. Suppose that the invariant $\mathsf{max}(0, \sigma_0(x)) \geqslant \mathsf{sb}(x)$ holds, that is, the valuation of x is bounded by the initial valuation of x. Then, the upper bound is $\mathsf{max}(0, x) + \mathsf{max}(0, z) + \mathsf{max}(0, x) \cdot \mathsf{max}(0, x)$.

3.6.3 Difference Constraints Programs

Sinn [144] and Sinn et al. [146] investigate *difference constraints* as suitable abstract representation to resource analysis. At the time of writing this representation forms the basis of the latest iteration of the Loopus tool. We have discussed theoretical properties of difference constraints programs before in Section 3.4.4. Next, we recall its definition.

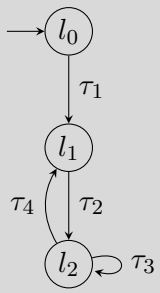
Program Representation

A *difference order constraint* is an inequality $x' \leqslant y + k$, where $x' \in \mathsf{Var}', y \in \mathsf{Var}$ and $k \in \mathbb{Z}$, and a *difference constraint* is a conjunction of difference order constraints. A *difference constraints program* $\mathcal{DC}$ is a CTS with difference constraints and where the valuation of variables range over $\mathbb{N}$. A difference constraint is called *fan-in free* if for all $x' \in \mathsf{Var}'$ there exists at most one order constraint $x' \leqslant y + k$. A $\mathcal{DC}$ program is called *fan-in free* if all difference constraints are fan-in free. It is helpful to think of fan-in free programs as programs with parallel updates. The tool Loopus restricts to fan-in free programs with a unique initial location l_0.

Motivation. While $\mathcal{MDC}$ constraints focus on *increments* and *decrements* of the form $x' \leqslant x + k$, $\mathcal{DC}$ constraints also support *resets* of the form $x' \leqslant y + k$ with $x \neq y$. The bound inference is designed to exploit properties of increments and resets.

Example 3.35 (Motivating Example)**.** It is easy to verify that this program is a $\mathcal{DC}$ program if we interpret assignments *weakly*, e.g. $i \leqslant i - 1$ instead of $i = i$.

```
l0 : main(n,m)
     i = n
     k = m
l1 : while( i > 0)
       i = i − 1
       k = k + 1
       j = k
l2 :   while(j > 0)
         j = j −1
```



Program Abstraction

The proposed program abstraction recursively infers a set of *transition invariants DC*. A difference order constraint $x' \leqslant y + k$ is called a transition invariant for $\langle l, l', \phi\rangle \in \mathcal{T}$ if for all $\sigma, \sigma' \in \Sigma$, $\sigma, \sigma' \models \phi$ implies $\sigma, \sigma' \models \sigma'(x') \leqslant \sigma(y) + k$ (see Definition 3.18 on page 26). The approach suggests deriving an initial set of variables (or norms) from the conditions of the loop header and individual loop paths. For instance, the norm $\mathsf{max}(0, a - b)$ is derived from a condition $a > b$. Starting from an initial set of norms, transition invariants and additional norms are recursively inferred. For each norm n_1 and transition $\langle l, l', \phi\rangle$ the suggested method checks if there already exists an order constraint $n_1' \leqslant n_2 + k$ in DC, in which n_1' is obtained from n_1 by symbolic execution. If not, the method tries to infer a new order constraint $n_1' \leqslant n_2 + k$, in which n_2 is either taken from the existing set of norms or heuristically obtained from n_1. In the latter case, n_2 is added to the set of norms. Termination is enforced by limiting the maximum number of recursive steps.

Bound Analysis

At the core of the analysis is the definition of local bounds and the inspection of its domain space. Each transition is associated with a local bound. A *local bound* is a variable (or norm) that satisfies the following property. Let $(l_0, \sigma_0) \to^* (l, \sigma) \to_\tau \cdot \rightsquigarrow (l', \sigma')$ denote a terminating program run starting with initial configuration (l_0, σ_0) over some intermediate configuration (l, σ) to a terminal configuration (l', σ'). Then the valuation of the local bound associated to τ bounds the number of its occurrences in $(l, \sigma) \rightsquigarrow (l', \sigma')$, i.e. the bound for a transition during a program run is given by the valuation of its local bound. A natural candidate for local bounds of transitions is given by *decrements*.

The bound inference is based on mutual recursive definitions of transition bounds and variable bounds. A *transition bound* for transition $\tau \in \mathcal{T}$ is a complexity bound $\mathsf{tb}_\mathcal{T}(\tau)$ such that

$$\underline{\mathsf{tb}}_\mathcal{T}(\tau) \succcurlyeq \mathsf{rc}^{l_0}_{\tau/\mathcal{T}} \,.$$

A *variable bound* for $x \in \mathsf{Var}$ is a complexity bound $\mathsf{vb}_\mathcal{T}(x)$ such that

$$\underline{\mathsf{vb}}_\mathcal{T}(x) \succcurlyeq \lambda\sigma_0.\ \mathsf{sup}\{\sigma'(x) \mid (l_0, \sigma_0) \to^*_\mathcal{T} (l', \sigma')\} \,.$$

We syntactically differentiate between two kinds of $\mathcal{DC}$ constraints, namely *increments* ($x' \leqslant x + k \wedge k > 0$) and *resets* ($x' \leqslant y + k \wedge y \neq x$). We omit some details but use the following definition to illustrate the main idea (cf. [144, Section 3.3]). We write $x' \leqslant y + k \in \tau$ to indicate whether $x' \leqslant x + k$ is an order constraint of the transition τ and assume that the local bound associated to transition τ is x. Then $x' \leqslant x + k \in \tau' \wedge k > 0$ is an increment of variable x for some transition τ' and $x' \leqslant y + k \in \tau' \wedge x \neq y$ is a reset of variable x for some transition τ'. Moreover, for the sake of simplicity we assume that the input arguments are constants and therefore $\mathsf{vb}(x) = \mathsf{max}(0, x)$ for all input arguments.

$$\mathsf{tb}(\tau) = \sum_{x' \leqslant x+k \in \tau' \wedge k > 0} \mathsf{tb}(\tau') \cdot k + \sum_{x' \leqslant y+k \in \tau' \wedge x \neq y} \mathsf{tb}(\tau') \cdot \mathsf{max}(0, \mathsf{vb}(y) + k)$$

$$\mathsf{vb}(x) = \sum_{x' \leqslant x+k \in \tau' \wedge k > 0} \mathsf{tb}(\tau') \cdot k + \max_{x' \leqslant y+k \in \tau' \wedge x \neq y} (0, \mathsf{vb}(y) + k)$$

The definition of variable and transition bounds are based on all increments and resets of the program. The variable bound is obtained by inspecting the growth induced by the number of increments together with the maximum of its resets. The transition bound is related to the variable bound of its local bound but additionally incorporates the induced domain space of the number of resets.

The bound inference is defined recursively and may not terminate. In particular, in the presence of cyclic dependencies induced by increments. There is a close connection to lexicographic ranking functions. If the bound inference terminates, then a lexicographic ranking function can be obtained from the local bounds, moreover, if there exists a lexicographic ranking function then the bound inference terminates (cf. [144]). The tool Loopus infers a lexicographic ranking functions prior to the bound analysis to ensure its termination.

Example 3.36 (Cont'd from Example 3.35)**.** Unrolling the definition of transition bound and variable bound, we obtain the following equations:

$$\begin{aligned}
\mathsf{vb}(m) &= \mathsf{max}(0, m) & \mathsf{vb}(n) &= \mathsf{max}(0, n) \\
\mathsf{vb}(i) &= \mathsf{max}(0, \mathsf{vb}(n)) & \mathsf{vb}(j) &= \mathsf{max}(0, \mathsf{vb}(k)) \\
\mathsf{vb}(k) &= \mathsf{tb}(\tau_2) \cdot 1 + \mathsf{max}(0, \mathsf{vb}(m)) \\
\mathsf{tb}(\tau_1) &= 1 & \mathsf{tb}(\tau_4) &= \mathsf{tb}(\tau_2) \\
\mathsf{tb}(\tau_2) &= 0 + \mathsf{tb}(\tau_1) \cdot \mathsf{max}(0, \mathsf{vb}(n)) & \mathsf{tb}(\tau_3) &= 0 + \mathsf{tb}(\tau_2) \cdot \mathsf{max}(0, \mathsf{vb}(k))
\end{aligned}$$

We apply some simplifications. The transition bound for τ_1 is constant 1, and for transition τ_4 it depends only on transition τ_2. Variables n and m are constant. The associated local bound for τ_2 and τ_3 is i and j, respectively. We obtain $\mathsf{tb}(\tau_4) = \mathsf{tb}(\tau_2) = \mathsf{max}(0, n)$ and $\mathsf{tb}(\tau_3) = \mathsf{max}(0, n) \cdot \mathsf{max}(0, \mathsf{max}(0, n) + \mathsf{max}(0, m))$.

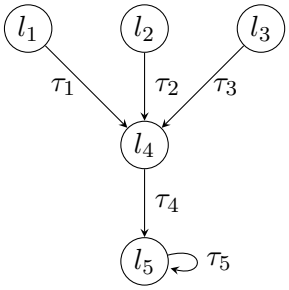
Extensions

Several improvements based upon the main idea have been investigated and presented in Sinn [144]. We only present a selected few.

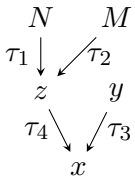
Symbolic Increments. Symbolic increments extend $\mathcal{DC}$ constraints to $x' \leqslant y + k$ where k is a symbolic constant evaluating to an integer. The main application is to support constraints of the form $x' \leqslant x + z$ with $z \in \mathsf{Var}$, which are not conform to $\mathcal{DC}$, using $x' \leqslant x + \mathsf{vb}(z)$. By supporting symbolic increments $\mathcal{DC}$ programs generalise (fan-in free) $\mathcal{MDC}$ programs (cf. Section 3.6.2).

Contextualisation via Reset Chains. In this section we discuss an extension based on reset chains to improve the precision of the analysis (cf. [144, Section 3.5]). In the definition of tb, resets are considered separately. The main idea of this refinement is to consider resets not separately but in terms of reset chains. A *reset chain* is a sequence $x_1 \overset{\tau_1}{\to} \cdots \overset{\tau_{n-1}}{\to} x_n$ that forms a chain of resets $x_1 \leqslant x_2 + c_1 \cdots x_n \leqslant x_{n-1} + c_n$. The extension refines tb based on reset chains. We present the main idea with the following example.

Example 3.37 (Reset Chains)**.** As illustrating example consider the following transitions with local bound x for τ_5. All transitions except τ_5 are resets. The most right figure displays the *reset graph*. There is an edge $x \overset{\tau}{\to} y$ for any reset $x' \leqslant y + c$ with $y \neq x$ defined for τ. A path in the reset graph is a reset chain.



$$\begin{aligned}
&\tau_1 \colon z' \leqslant N + 1\\
&\tau_2 \colon z' \leqslant M + 2\\
&\tau_3 \colon x' \leqslant y\\
&\tau_4 \colon x' \leqslant z + 3\\
&\tau_5 \colon x' \leqslant x - 1
\end{aligned}$$



According to the basic definition the transition bound of τ_5 depends on variable bounds y and z. When unrolling the definitions, we obtain:

$$\begin{aligned}
\mathsf{vb}(z) &= \mathsf{max}(\mathsf{vb}(N) + 1, \mathsf{vb}(M) + 2)\\
\mathsf{tb}(\tau_5) &= \mathsf{tb}(\tau_3) \cdot \mathsf{max}(0, \mathsf{vb}(y)) + \mathsf{tb}(\tau_4) \cdot \mathsf{max}(0, \mathsf{vb}(z) + 3)
\end{aligned}$$

For the sake of the example, we assume that $\mathsf{tb}(\tau_4) = \mathsf{tb}(\tau_1) + \mathsf{tb}(\tau_2) + \mathsf{tb}(\tau_3)$. It is not difficult to extend the program in such a way by adding additional transitions with decrements on a unique variable from l_4 to l_1, l_2 and l_3.

We identify two sources of imprecision in the above formula for $\mathsf{tb}(\tau_5)$. First, the bound depends on $\mathsf{vb}(y)$. It is easy to see that the bound should only depend on the valuation of z and its valuation is not affected by y. Second, the bound depends on $\mathsf{tb}(\tau_4)$, which

by our assumption corresponds to the sum of the transition bounds of the preceding transitions.

Based on the reset graph, the proposed analysis identifies a set of sound and optimal reset chains. Informally, a sound reset chain is one that occurs in any program run, and a optimal reset chain is a maximal reset chain.

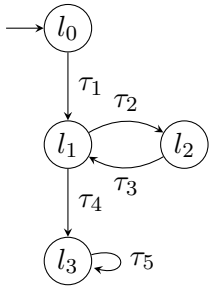
Here τ_3 is not a sound reset chain. The reset chains $N \xrightarrow{\tau_1} z \xrightarrow{\tau_4} x$ and $M \xrightarrow{\tau_2} z \xrightarrow{\tau_4} x$ are maximal and sound. A refined bound is obtained by inspecting all reset chains separately. We obtain the following transition bound.

$$\begin{aligned}\mathsf{tb}(\tau_5) = {} & \mathsf{min}_{\tau' \in \{\tau_1, \tau_4\}}\, \mathsf{tb}(\tau') \cdot \mathsf{max}(0, \mathsf{vb}(N) + 1 + 3) \\ & + \mathsf{min}_{\tau' \in \{\tau_2, \tau_4\}}\, \mathsf{tb}(\tau') \cdot \mathsf{max}(0, \mathsf{vb}(M) + 2 + 3)\end{aligned}$$

The transition bound of a reset chain is approximated by the minimum (taking into account that the transitions are evaluated in sequence) of the individual transition bounds. Constant expressions are summed up along the chain.

Path-Sensitive Analysis. In this section we discuss *path-sensitive* analysis (cf. [144, Section 3.8]). While the proposed approach supports increments and resets we focus only on increments. The main idea is to compose increments and decrements along a path. We illustrate the main idea with the following example.

Example 3.38 (Path-Sensitive Analysis)**.** Consider the following program with local transition bound x for τ_5.



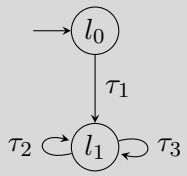
$$\begin{aligned}
&\tau_1 \colon \langle l_0, l_1, x' \leqslant x_0 \quad \wedge y' \leqslant y_0 \rangle \\
&\tau_2 \colon \langle l_1, l_2, x' \leqslant x - 3 \ \wedge y' \leqslant y \rangle \\
&\tau_3 \colon \langle l_2, l_1, x' \leqslant x + 2 \ \wedge y' \leqslant y - 1 \rangle \\
&\tau_4 \colon \langle l_1, l_3, x' \leqslant x \rangle \\
&\tau_5 \colon \langle l_3, l_3, x' \leqslant x - 1 \rangle
\end{aligned}$$

There is an increment $x' \leqslant x + 2$ defined for τ_3. Hence, $\mathsf{tb}(\tau_5) = \mathsf{tb}(\tau_3) \cdot 2 + \mathsf{max}(0, x_0)$. It is easy to see that we can improve upon this bound considering the following observation. The transition τ_2 is always succeeded by τ_3. Moreover, we have $x' \leqslant x - 1$ for its composition $\tau_2 \cdot \tau_3$. This eliminates the increment and $\mathsf{tb}(\tau_5) = \mathsf{max}(0, x_0)$.

The proposed extension refines the original definition of tb composing increments and decrements along simple cyclic paths instead of single transitions. We omit the formal definition here.

We conclude with the analysis of a variation of the motivating example from the overview.

Example 3.39 (DC analysis of Example 3.11)**.** The assignments $y' = y+x$ and $z' = y+x$ are not difference order constraints. We make use of the symbolic increments extension and write $y' \leqslant y + \mathsf{vb}(x)$ and $z' \leqslant y$. Furthermore, we use the constant expressions x_0, y_0 and z_0 to indicate the initial arguments.



$$\begin{aligned}
&\tau_1 \colon \langle l_0, l_1, && x' \leqslant x_0 && \wedge\, y' \leqslant y_0 && \wedge\, z' \leqslant z_0 \rangle \\
&\tau_2 \colon \langle l_1, l_1, && x > 0 \wedge x' \leqslant x - 1 && \wedge\, y' \leqslant y + \mathsf{vb}(x) && \wedge\, z' = y \rangle \\
&\tau_3 \colon \langle l_1, l_1, && z > 0 \wedge x' \leqslant x && \wedge\, y' \leqslant y && \wedge\, z' \leqslant z - 1 \rangle
\end{aligned}$$

Suppose that the local bound of τ_2 and τ_3 is x and z, respectively, and the transition bound of τ_1 is set to constant 1. Then, unrolling the definition we obtain the following constraints.

$$\begin{aligned}
&\mathsf{tb}(\tau_1) = 1 && \mathsf{vb}(x) = \mathsf{max}(0, \mathsf{vb}(x_0)) \\
&\mathsf{tb}(\tau_2) = \mathsf{tb}(\tau_1) \cdot \mathsf{max}(0, \mathsf{vb}(x_0)) && \mathsf{vb}(y) = \mathsf{tb}(\tau_2) \cdot \mathsf{vb}(x) + \mathsf{max}(0, \mathsf{vb}(y_0)) \\
&\mathsf{tb}(\tau_3) = \mathsf{tb}(\tau_1) \cdot \mathsf{max}(0, \mathsf{vb}(z_0)) && \mathsf{vb}(z) = \mathsf{max}(0, \mathsf{vb}(z_0)) + \mathsf{max}(0, \mathsf{vb}(y)) \\
&\qquad + \mathsf{tb}(\tau_2) \cdot \mathsf{max}(0, \mathsf{vb}(y)) &&
\end{aligned}$$

We obtain $\mathsf{tb}(\tau_2) = \mathsf{max}(0, x_0)$ and $\mathsf{vb}(y) = \mathsf{max}(0, x_0) \cdot \mathsf{max}(0, x_0) + \mathsf{max}(0, y_0)$. Finally, we have $\mathsf{tb}(\tau_3) = \mathsf{max}(0, \mathsf{vb}(z_0)) + \mathsf{max}(0, x_0) \cdot (\mathsf{max}(0, x_0) \cdot \mathsf{max}(0, x_0) + \mathsf{max}(0, y_0))$. The reader may want compare the result with the informal discussion of the introductory program in Example 3.1.

3.7 Automated Resource Analysis with Paicc

In the following we comment on the ongoing work on decidable growth-rate properties for loop programs [28, 33, 34, 36, 37, 39]. Ben-Amram et al. [39] present a *core programming language* in which the *polynomial growth-rate* problem of variables, i.e. the problem whether the final valuation of a variable is bounded by a polynomial in the input valuation, is decidable.

Here, we refer to this core language as Core programs. Core programs are a variant of Meyer and Ritchie's loop programs [116] with *weak semantics*, that is, the abstraction of conditional control flow with non-deterministic control flow. The *worst-case runtime* of a Core program is given via *counter instrumentation* and growth-rate analysis thereof. Ben-Amram and Pineles [36, 37] have extended the main result to *flowchart* programs in which edges are associated with core statements. Schaper [139] presents an abstraction from integer-valued CTSs to flowchart programs with core statements. The abstraction together with the growth-rate analysis have been implemented in the prototype paicc (program analysis and implicit computational complexity), which is available online at

`http://cbr.uibk.ac.at/tools/paicc/` .

In the most recent related work, Ben-Amram and Hamilton [33] improve upon the original result and investigate *tight* worst-case bounds for Core programs.

Overview. In Section 3.7.1 we recall Core programs and the key concepts of the growth-rate analysis presented in [39]. Afterwards, in Section 3.7.2 we introduce constraint transition systems with *Ben-Amram - Jones - Kristiansen* constraints ($\mathcal{BJK}$ programs). The program representation is derived from flowchart programs with core expressions [37]. Furthermore, we provide insights about the program abstraction and automation in the tool paicc.

3.7.1 Loop Programs

We have discussed theoretical properties of Core and $\mathcal{BJK}$ programs in Section 3.4.2. In order to present the analysis implemented in paicc, we recall the central notions.

Program Representation

Program states in Core are n-tuples over the natural numbers, where n is the number of (global) variables of the program. Expressions are terms over variables and binary operations, addition and multiplication. In contrast to more standard programming languages, Core does not support numeric constants. Statements consist of (weak) assignments, skip, non-deterministic choice and indefinite loops. A *weak assignment* is an assignment that can be interpreted as *not greater than*. For example, the evaluation of $x = x + y$ is interpreted as the value of x after the assignment is not greater than $x + y$. An *indefinite loop* is a loop that may exit any time before completion of the iteration count. For a loop statement `loop` x {`S`} we call x the bound variable and `S` the body of

the loop. The bound variable may not be modified within the loop body, and the body of the loop is executed $0, 1, \ldots$ up to x times.

The usage of non-determinism in Core has multiple motivations. First, we are interested in a program fragment with decidable growth-rate properties. Standard loop programs are known to represent the class of primitive recursive functions [116], and the polynomial growth-rate problem is undecidable for such programs. This follows immediately by a reduction from the equivalence problem of primitive recursive functions (cf. Kahrs [104]). Second, we consider Core programs from the view point of program and data-flow analysis. Due to non-determinism Core programs represent relations rather than functions. We can motivate Core programs as abstract representation of the *collecting semantics* of more expressive programs (cf. Nielson et al. [125]). We make use of this observation when presenting the abstraction from integer programs.

Example 3.40 (Illustrating Example)**.** Consider this variation of one of the motivating examples from [39]. By the nature of the Core programming language, all programs are terminating. The example is challenging for automated resource analysis as it exhibits complex interdependent data-flow between variables. The proposed growth-rate analysis infers that the final valuation of all variables is bounded by a polynomial in the input valuation. In particular, this holds for the variable x_1 whose value increases in the first loop and is the bound variable for the second loop. The central observation, which is inferred by the growth-rate analysis, is that x_1 is not doubled when iterating the body of the first loop. We obtain a polynomial bound on the worst-case runtime via counter instrumentation.

```
x5 = x1 * x2
loop x5
  choice
    { x3 = x1 + x2; x4 = x2     }
    { x3 = x2     ; x4 = x1 + x2 }
  x1 = x3 + x4
loop x1
  skip
```

Main Result. We recall the main result of Ben-Amram et al. [39]. Let $\vec{x}$ denote a tuple of variables (or a program state) and $\vec{x} \rightsquigarrow \vec{x}'$ denote the evaluation of input state $\vec{x}$ to output state $\vec{x}'$. The *growth-rate* of a variable x_i is *polynomially bounded* if there exists a polynomial in $\vec{x}$, in notation $p(\vec{x})$, such that ${x_i}' \leqslant p(\vec{x})$ for all evaluations $\vec{x} \rightsquigarrow \vec{x}'$. Otherwise, the growth-rate is *super-polynomial*. The question whether the growth-rate of a variable is polynomially bounded is decidable for Core programs, furthermore it is decidable in P.

The analysis can be specialised to inspect *linear* growth-rates. The growth-rate analysis does not provide a precise bound, but a certificate for the existence of a polynomial that bounds the growth-rate.

Bound Analysis

We summarise the main ideas of the growth-rate analysis. Our goal is to provide some intuition, for details we refer to [39]. The analysis is presented as an application of *abstract interpretation* (Cousot and Cousot [59]). It is a *bottom-up* fixed-point construction that makes use of the structural definition of the Core programming language. There are three key concepts that govern the analysis, (i) *flow-abstraction* of assignments, (ii) *composition* of flow-abstraction, and (iii) *fixed-point* computation and *loop correction*.

Central to the growth-rate analysis is the definition of *unary* and *binary* variable flow. The unary flow represents growth-rate dependencies between program variables, whereas the binary flow represents flow-dependencies between pairs of variables.

Assignments can be categorised into different classes of unary flows.

- $x = y$ implies an *identity* flow from y to x
- $x = y + z$ implies an *additive* flow from y and z to x
- $x = y + y$ implies a *multiplicative* flow from y to x
- $x = y * z$ implies a *multiplicative* flow from y and z to x
- other assignments are categorised as *exponential* flow

Unary flows can be composed the intended way, that is, the flow abstraction of the composition of statements equals to the composition of the flow abstraction of statements.

During the construction each loop is replaced with its *loop summary* (or invariant). Briefly, one may infer a fixed-point of the flow abstraction by repeated composition. When analysing loops, we pay special attention to cyclic (or self-referential) flow. A *cyclic* flow is a unary flow from a variable to itself. Under iteration, cyclic flows imply growth of the valuation of the variables. This observation is incorporated in the *loop correction* phase which provides a closed-form abstraction which depends on the loop bound variable. Precise handling of the unary flow is crucial for programs with nested loops and non-deterministic choice.

Example 3.41 (Loop Correction)**.** We illustrate the effect of loops on cyclic flow with the following example. Under iteration, cyclic additive flows turn into multiplicative flows and cyclic multiplicative flows turn into exponential flows.

```
loop z          // Cyclic Flow     –  Loop Correction
  x1 = x1       // identity        –  no flow from z → x1
  x2 = x2 + y   // additive        –  multiplicative flow z → x2
  x3 = x3 + x3  // multiplicative  –  exponential flow z → x3
  x4 = x4 * y   // multiplicative  –  exponential flow z → x4
```

The binary flow keeps track of *duplication*. For some variable x we are interested whether there exists a sequence of statements that duplicates it, i.e. a sequence of statements that corresponds to an update satisfying $x' \leqslant 2x$.

Example 3.42 (Duplicating Flow)**.** Consider the following example. The illustrated binary relation keeps track of combination of variables and correctly identifies duplicating flow. Repeated duplication is super-polynomial. Crucially this inference is complete, a duplicating flow is detected if and only if there exists one.

```
loop w
  y = x                                  x→y . y→x
  z = x                                  x→z   z→x
  x = y + z  // duplicating
```

3.7.2 Ben-Amram - Jones - Kristiansen Constraints Programs

Ben-Amram and Pineles [36, 37] provide an extension of the growth-rate analysis of Core programs to *flowchart* programs with *loop structures*. We refer to them as FC-Core programs. This is a strict extension in the sense that all Core programs can be represented as FC-Core programs, but not the other way around. Most relevant, FC-Core programs are unstructured (albeit bounded) in the sense that the control flow is given by `goto` statements instead of `loop` statements. This allows for a more direct comparison to other representations used.

Program Representation

The domain and expressions in FC-Core are equal to Core. Here we present the flowchart programs of [36, 37] as CTSs with *Ben-Amram - Jones - Kristiansen* order constraints over the natural numbers. We define $\mathcal{BJK}$ order constraints as weak assignments of composed Core expressions. To simplify the upcoming abstraction we treat constants as symbolic variable k. More formally, a $\mathcal{BJK}$ *order constraint* is an inequality of the form $x'_i \leqslant p(x_1, \ldots, x_n, k)$, that is, x'_i is not greater than a polynomial (with natural coefficients).

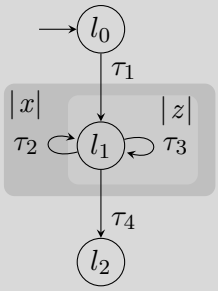
We restrict to *fan-in free* programs, that is, for each transition and variable $x' \in \mathsf{Var}'$ there is at most one constraint $x' \leqslant p(x_1, \ldots, x_n, k)$. The restriction to fan-in free programs is natural if we think of constraints as (parallel) weak assignments.

Order constraints do not determine control flow and without further restrictions such programs are non-terminating. To remedy this fact, *loop structures* are introduced in [37]. Loop structures are akin to call-graphs and form a rooted tree that represents a nesting hierarchy of (locally) bounded subprograms, so called *loops*. Loops are associated with a polynomial expression, its *loop bound*, which indicates how often edges within a loop can be traversed. Alternatively we can construe loop structures as an explicit representation of program decomposition. For Core programs the loop structure is implied by the scope of the loops. Assume L is a loop and L_i its descendants in the loop structure. The semantics of a loop L is interpreted as follows. Suppose the loop L is entered with configuration (l, σ). Then the bound expression associated to L evaluated under the current store σ is a bound on the maximum number of occurrences of edges (or transitions) in $L \setminus L_i$ before leaving the loop (or subprogram) L.

In the rest of this work we assume that $\mathcal{BIK}$ programs are associated with a loop structure. Moreover, we assume that the CFG of $\mathcal{BIK}$ program has at least one (possible more) initial and final locations. This requirement is due to the nature of the growth-rate analysis in [37] which expresses the grow-rate as an invariant between all initial and final locations.

Example 3.43 (Motivating Example)**.** The following $\mathcal{BIK}$ program is obtained from Example 3.11 on page 20 by the upcoming abstraction. It illustrates how the original conditional control flow of the program is replaced by a notion of local repetition, and constraints only reflect the growth of the variables.

By convention the root consists of all transitions, its bound is constant. The outer loop consists of the transitions τ_2 and, τ_3, and its bound is x. The inner loop consists of the transition τ_3, and its bound is z. In any trace τ_1 and τ_4 can occur only once, τ_2 up to x times, and τ_3 can occur up to z times before leaving the inner loop. The inner loop can be re-entered again, in total up to x times. The iteration count for τ_3 depends on the valuation of z when entering the inner loop.



$$\tau_1 \colon \langle l_0, l_1, x' \leqslant x \wedge y' \leqslant y \quad \wedge z' \leqslant z \rangle$$
$$\tau_2 \colon \langle l_1, l_1, x' \leqslant x \wedge y' \leqslant x + y \wedge z' \leqslant x + y \rangle$$
$$\tau_3 \colon \langle l_1, l_1, x' \leqslant x \wedge y' \leqslant y \quad \wedge z' \leqslant z \rangle$$
$$\tau_4 \colon \langle l_1, l_2, x' \leqslant x \wedge y' \leqslant y \quad \wedge z' \leqslant z \rangle$$

Program Abstraction

Next, we present an abstraction from integer-valued CTS programs to $\mathcal{BIK}$ programs. This abstraction together with the $\mathcal{BIK}$ growth-rate analysis is implemented in the prototype paicc [139], which is developed and maintained by the author of this thesis. The abstraction was initially inspired by the synthesis of lexicographic ranking functions in Rank [9] and the local size bound approximation in KoAT [51].

The abstraction is an overapproximation of the reachable *set-of-states* with respect to the application of a chosen norm. We fix the set of norms to be the absolute value of variables. We write, $|x|$ instead of $\mathsf{abs}(\sigma(x))$, if the store σ is not important. Additionally, we abstract constants via a symbolic variable k, which represents an arbitrary natural number. The approach is motivated by the idea that $\mathcal{BIK}$ is a suitable program representation to reason about the growth-rate of norms, here, the absolute value of each variable.

In the following let $\mathcal{T}$ be a integer-valued CTS. The abstraction of $\mathcal{T}$ to $\mathcal{BIK}$ consists of two steps: (i) the inference of the loop structure, and (ii) the transformation of transitions to $\mathcal{BIK}$ constraints.

Inferring loop structures

$\mathcal{BJK}$ programs use an explicit notion of decomposition in form of *loop structures*. The loop structure of a program $\mathcal{T}$ is not unique and a semantic rather than syntactic property.

We have discussed before a strategy for deomposing programs based on *lexicographic combinations of ranking functions* (Section 3.3.5 on page 24). For the construction, we synthesise *non-trivial linear quasi-ranking functions* (see Section 3.3.5 on page 21). That is, given a set of transitions we encode the following properties: (i) all transitions are *non-increasing*, (ii) some transitions are *decreasing* and *bounded*. In notation, $(>, \geqslant)$

The following algorithm synthesises a lexicographic combination of linear RFs and constructs a loop structure for the given program $\mathcal{T}$. The algorithm fails if no ranking function for the considered (sub)program can be established.

(i) The root node L of the loop structure consists of all transitions of $\mathcal{T}$.

(ii) For the considered (sub)program $\mathcal{T}$, add all transitions of each SCC_i in the CFG of $\mathcal{T}$ as child node L_i to the current node.

(iii) For each leaf L_i synthesise a RF $(>_i, \geqslant_i)$ of L_i such that $>_i$ is non-empty. Let $\mathcal{T}_i$ denote the transitions of L_i. The RF is a linear polynomial, $c_1x_1 + \cdots + c_nx_n + c_kk$. This is a local bound on the maximum number of occurrences of transitions in $>_i$ w.r.t. $\mathcal{T}_i$. The associated loop bound of L_i is $|c_1|x_1 + \cdots + |c_n|x_n + |c_k|k$.

(iv) For $\mathcal{T}_i \setminus >_i$ apply recursively (ii).

Inferring Local Size Bounds

Next, we describe the approximation of *local size bounds* (or growth-rates). The main objective is to capture the change in norms for a single evaluation step, by transforming transitions conforming to iCTS constraints to transitions conforming to $\mathcal{BJK}$ constraints taking the application of the norm into account. This is a specialisation of the inference of *transition invariants* (see Section 3.3.7 on page 26). Let $\tau\colon \langle l, l', \phi\rangle$ be a transition in $\mathcal{T}$. A *local size bound* for variable $x \in \mathsf{Var}$ is a complexity bound $\mathsf{sb}_\tau(x)$ such that

$$\underline{\mathsf{sb}}_\tau(x) \succcurlyeq \lambda\sigma.\ \mathsf{sup}\{|\sigma'(x)| \mid \sigma, \sigma' \models \phi\}\ .$$

For programs in which constraints correspond to standard assignments $x = y + z$ and $x = y * z$, one can make use of the following observations to analyse programs over the integer domain: $|x*y| \leqslant |x| * |y|$ and $|x+y| \leqslant |x| + |y|$. However, paicc supports a more flexible transformation via the synthesis of polynomials. Let $\tau\colon \langle l, l', \phi\rangle$ be a transition in $\mathcal{T}$. For each variable $x_i \in \mathsf{Var}$ two polynomials are synthesised that represent the lower and upper bound, respectively.

(i) $\phi \models x_i' \geqslant c_1x_1 + \cdots + c_nx_n + c_kk$, and

(ii) $\phi \models x_i' \leqslant d_1x_1 + \cdots + d_nx_n + d_kk$,

Then, the local size bound of x_i is a $\mathcal{BJK}$ constraint $x_i' \leqslant \mathsf{max}(|c_1|, |d_1|)x_1 + \cdots + \mathsf{max}(|c_n|, |d_n|)x_n + \mathsf{max}(|c_k|, |d_k|)k$. The implementation restricts to affine linear updates and uses the synthesis approach described in Section 3.3.5 on page 22. The algorithm may fail to provide a bound.

Example 3.44 (Inference of the Loop Structure of Example 3.43)**.** The loop structure illustrated in Example 3.43 is obtained from Example 3.11 using the approach presented above. Let $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ in which $\tau_4 \colon \langle l_1, l_2, x' \leqslant x \wedge y' \leqslant y \wedge z' \leqslant z \rangle$ is added to the original program.

Step 1 By construction, the root node L of the loop structure consists of all transitions of $\mathcal{T}$. $L = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. The associated loop bound of L is constant.

Step 2 The single SCC consisting of the transitions $\mathcal{T}_1 = \{\tau_2, \tau_3\}$ forms the loop L_1.

Step 3 The linear RF $\eta(l_1; \sigma) = \sigma(x)$ is synthesised, which is decreasing and bounded for τ_2, i.e. $\tau_2 \in >_1$. The loop bound associated to L_1 is $|x|$.

Step 4 We recursively apply the inference for $\mathcal{T}' = \{\tau_3\}$.

Step 5 The single SCC $\{\tau_3\}$ forms the loop L_{1_1}.

Step 6 The ranking function $\eta(l_1; z) = \sigma(z)$ is decreasing and bounded for τ_3. The loop bound associated for L_{1_1} is $|z|$.

The algorithm terminates since no (non-trivial) subprograms are left.

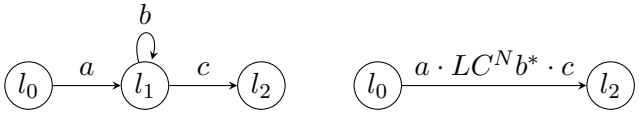
Bound Analysis

The growth-rate analysis of FC-Core programs presented in [37] is conceptually similar to the original growth-rate analysis of Core programs presented in [39]. We provide some additional insights how loop structures are processed. The approach is inspired by the *ripping* algorithm for the conversion of non-deterministic finite automata to regular expressions (cf. Sipser [147]), suitably adapting the *loop correction* of [39] for the loops within the loop structure.

Let $\mathcal{T}$ be a $\mathcal{BJK}$ program. At first, each constraint in $\mathcal{T}$ is mapped to its flow-abstraction. Akin to the analysis of Core programs the loop structure is processed bottom-up. A loop in the loop structure can have multiple entry and exit nodes that are connected with the outer loop within the loop structure. The algorithm repeatedly transforms each loop that forms a leaf of the loop structure to its *loop summary*, which is a set of transitions that represents the unary and binary flow abstraction between entry and exit nodes. Loop summaries are obtained by *ripping* (or eliminating) non-entry and non-exit nodes.

We illustrate the interesting case of a self-loop. Suppose that the loop bound associated to the enclosing loop structure is N. We rip location l_1. In doing so, we replace l_1 with the loop correction of the fixed-point that is obtained by repeatedly composing

the flow-abstraction b of the self-loop. This operation is indicated below by $LC^N b^*$. Afterwards, the abstraction is composed with the incoming and outgoing flow.



Example 3.45 (Cont'd from Example 3.43)**.** The tool paicc performs the bound analysis bottom-up, in doing so, the program is instrumented with a dedicated counter variable. The inner loop $l_1 \xrightarrow{\tau_3} l_1$ is processed first. It has an additive dependency from z to the counter variable. All program variables are not modified and have a reflexive identity flow. The outer loop is processed next. It implies a multiplicative flow from x and y to the counter variable. Therefore, the worst-case runtime is polynomially bounded.

Tight Bounds for Core Programs

At the time of writing Ben-Amram and Hamilton [33] present the inference of *tight polynomial bounds* on the growth-rate of variables for Core programs. The proposed analysis is conceptually similar to the original one. However, it crucially differs in the underlying abstract domain used.

The abstract domain is defined over sets of (abstract) *multi-polynomials* (see also Section 3.4.2 on page 27). Consider a program with n program variables. A multi-polynomial is a n-tuple of polynomial expressions that represents a polynomial bound on each variable. On the other hand, a set of multi-polynomials is interpreted as *disjunctive simultaneous bounds*, i.e. the output value of all variables is bounded by (at least) one multi-polynomial. For instance, consider a program with variables x, y and z. The set $\{\langle x, y, z\rangle, \langle x, x^2 + y, x^2 + z\rangle\}$ indicates that for each run the growth-rate of x, y, z is bounded by $\langle x, y, z\rangle$ or $\langle x, x^2 + y, x^2 + z\rangle$.

The abstract domain is precise for programs without loops. Straight line code translates into composition of polynomials and non-deterministic choice translates into union of sets. During the closure algorithm of loops, coefficients in polynomials are abstracted. In doing so (under scrutiny) *loop summaries* in form of sets of (abstract) multi-polynomials are obtained.

Example 3.46 (Tight Bounds)**.** Consider the following example of Ben-Amram and Hamilton [33]. The set of multi-polynomials $\{\langle x_1, x_2, x_3, x_1\rangle, \langle x_1, x_1^2 + x_2, x_1^2 + x_2, x1\rangle\}$ is returned by the proposed analysis. The first multi-polynomial represents the case in which the loop is never executed. The second multi-polynomial represents the case in which the loop is executed (up to) x_4 times.

$x_4 = x_1$
`loop` x_4
  $x_2 = x_1 + x_2$
  $x_3 = x_2$

We have yet to investigate the application of this result in paicc. We are confident that the abstract domain and fixed-point computation could also be applied for flowchart programs instead of core programs. One argument to support this claim is that the proposed analysis reduces the problem of analysing programs without loops to disjunctive paths. However, in doing so it is not obvious if the bounds that are obtained are still tight for flowchart programs.

3.8 Overview of Tools

Based on Sections 3.5 to 3.7 we provide an overview of a selected few key concepts. We conclude this section with Table 3.2.

KoAT

The tool KoAT processes programs with polynomial order constraints $\mathcal{POL}$ and relies on known approaches to solve constraints over (linear) inequalities. At its core is the *synthesis of (polynomial) ranking functions*. The tool focuses on the inference of *transition bounds and size bounds*. The approach uses a fixed set of norms, namely the *absolute value* of all program variables. Modularity is obtained by an interdependent *alternating runtime and size bound* analysis. To make constraint solving more viable *supporting invariants* are inferred by standard numeric invariant generation. The decomposition in KoAT is not explicit but the constructed complexity proof is analogous to lexicographic combinations of ranking functions. We say that the application of LexRFs is *implicit*.

Loopus

The tool Loopus uses different abstract program representations. As preprocessing step it identifies norms via heuristics and abstracts the target program such that it is conform with the abstract program representation. Norms are usually combinations of (but not restricted to) *max* and *linear* expressions.

The $(\mathcal{MC}, \mathbb{Z})$ domain is used as *disjunctive domain* for program abstraction. The runtime for monotonicity constraints programs is expressed in form of *reachability bounds*. Upper bounds are inferred via *lexicographic decomposition* incorporating *size bounds* on expressions, which are obtained from standard numeric invariant generation.

The $(\mathcal{MDC}, \mathbb{N})$ domain expresses *incremental growth* on subprograms. The runtime for monotone difference constraints programs is expressed in form of *path bounds*. Upper bounds are obtained from *lexicographic ranking functions* taking the *incremental growth* for subcomponents into account. Standard numeric invariant generation is used to obtain *size bounds* on expressions for program abstraction.

The $(\mathcal{DC}, \mathbb{N})$ domain is used to express the domain space of local bounds via *increments* and *resets*. The runtime problem for difference constraints programs is expressed in form of interdependent constraints on *transition and variable (or size) bounds*. The existence of a lexicographic ranking function ensures that the bound analysis *terminates*.

paicc

The tool paicc is based on $(\mathcal{BJK}, \mathbb{N})$ constraints which provide a *compositional growth-rate* analysis to certify *polynomial growth-rate*. The runtime bound is expressed via *counter instrumentation*. Lexicographic ranking functions are used to generate an explicit notion of decomposition in form of *loop structures*. The set of norms is fixed to the *absolute value* of variables. Standard numeric invariant generation is used to infer *supporting invariants*.

	KoAT	Loopus			paicc
Domain	$(\mathcal{POL}, \mathbb{Z})$	$(\mathcal{MC}, \mathbb{Z})$	$(\mathcal{MDC}, \mathbb{N})$	$(\mathcal{DC}, \mathbb{N})$	$(\mathcal{BJK}, \mathbb{N})$
Motivation	synthesis of polynomial RF	predicate abstraction	bound analysis via increments	bound analysis via increments and resets	growth-rate
Runtime	transition bound & size bound	reachability bound	path bound	transition bound & size bound	counter instrumentation
Norm	absolute value	max & linear	max & linear	max & linear	absolute value
Invariant	support	size bounds	size bounds	-	support
LexRF	implicit	explicit	explicit	termination	loop structure
Modularity	alternating runtime and size analysis	LexRF & size bounds	LexRF & incremental growth	recursive constraints on transition and size bounds	loop structure & compositional growth-rate

Table 3.2: Overview of Key Concepts in Complexity Tools.

3.9 Comparing Tools and Abstract Program Representations

In this section we provide additional insights between different tools and abstract program representations.

KoAT & paicc

The development of paicc was motivated by the ongoing work of KoAT and the developments of decidable growth-rates for FC-Core programs. We compare both approaches.

Representation of Complexity Bounds

The tool KoAT uses a more representative notion of complexity bounds in form of expressions that are constructed from the absolute value of variables, addition, multiplication and exponentiation. This is of practical relevance if one is interested in tight upper (e.g. quadratic, cubic, ...) complexity bounds. In contrast, FC-Core and paicc provide only a certificate for polynomial (or linear) growth-rate.

Norms and Local Sizebounds

The tool KoAT processes $\mathcal{POL}$ programs. The proposed analysis computes local size bounds that represent the growth of a norm with respect to a single reduction step. The representation of local size bounds (e.g. $|x'| \leqslant |y|+1$) in KoAT resembles $\mathcal{BJK}$ constraints. Both tools fix the set of norms to the set of absolute value of variables. There is a small but significant difference, which we are going to discuss below, in the support of numeric constants. Constants in $\mathcal{BJK}$ are abstracted to constant expressions that represent any natural number. Keeping in mind that we can represent the worst-case runtime by the size (or growth-rate) of a dedicated counter variable, we focus on comparing the inference of size bounds and the modularity thereof.

Global Sizebounds

Based on the previous observation we investigate the inference of size bounds. We focus on the difference in finding closed-form expressions of variable growth under repetition.

It is known that the analysis of KoAT is not complete w.r.t. polynomial growth-rate. The following motivating example is taken from the growth-rate analysis of FC-Core programs [36, 37].

```
loop m
  choice
    { x = z;  y = n }
    { x = n;  y = z }
    { z = x + y }
```

The difference in the analysis is in the detection of duplicating flow (see Section 3.5.1 on page 41 and Section 3.7.1 on page 60), which implies super polynomial growth. The analysis of KoAT is based on inspecting cyclic flow dependencies, here $z \to x$, $z \to y$

and $y \to z$, $x \to z$, which is interpreted as duplicating flow. The binary flow analysis of FC-Core correctly detects the correlation between sums of variables, here $x + y = z + w$ and infers polynomial growth-rate.

On the other hand, the FC-Core analysis does not support constants. It is an open problem whether Core extended with constants, or more specifically increments, has decidable growth-rate [28, 34].

```
loop x
  z = z + 1
```

In this example the value 1 is treated as a (constant) variable expression and a multiplicative growth-rate for z that depends on x and 1 is obtained. In paicc numeric constants are replaced with a fresh variable k, which represents some arbitrarily chosen but fixed natural number. The multiplicative growth-rate is optimal for the abstraction used, consider the instantiation of k with a constant greater than one. The tool KoAT on the other hand, infers the precise linear bound $z' \leqslant z + x$ (under optimistic assumption for the local size and transition bounds inferred).

Decomposition

In paicc an explicit notion of program decomposition is used in form of loop structures. The loop structure for the program under consideration is obtained via the inference of a lexicographic combination of polynomial ranking functions. The runtime bound analysis in KoAT, on the other hand, makes use of a dedicated strategy to select candidate subprograms for the modular bound inference. Albeit its construction is hidden in the complexity proof, the proposed approach is also based on lexicographic combinations of ranking functions.

We remark that the algorithm for the inference of loop structures that is implemented in paicc (see Section 3.7.2 on page 63) is a natural candidate for the strategy used in KoAT. The precise theoretical and practical connections, however, are unclear. In particular, it is an open question whether the proposed algorithm for constructing the loop structure is a complete strategy in the sense that all possible program decompositions that are amenable to the bound analysis in KoAT are also conform to the loop structure construction.

Norms and Complexity Bounds

KoAT and paicc define norms as the absolute value of a variable. This fixes the domain space for the size analysis of norms and allows providing dedicated strategies to infer upper bounds.

Furthermore, fixing the domain space provides modularity in a bigger scope by analysing methods of a larger program individually. Within the *CAGE*[7] (Complexity Analysis-Based Guaranteed Execution) project, safety properties based on resource analysis have been investigated. The tools AProVe and KoAT use method summaries as an abstraction

[7] https://www.draper.com/news-releases/drapers-cage-could-spot-code-vulnerable-denial-service-attacks

of method calls (see Frohn and Giesl [76]). Method summaries collect several data-facts such as heap-shape information and numeric invariants. Moreover, method summaries incorporate resource bounds. By fixing the domain space of method summaries on call-sites, resource bounds can be integrated in a modular analysis.

On the other hand, it can be a source of imprecision. Consider the following program.

```
while(x > 0)
  x = x − 1; y = y + z; z = z + z
while(y > z)
  y = y − 1
```

For the sake of the argument we assume that the runtime of both loops are analysed separately. The expressions $\mathsf{max}(0, x)$ and $\mathsf{max}(0, y - z)$ bound the number of iterations of the first and second loop, respectively, and the program variables y and z grow exponentially within the first loop. The tools KoAT and paicc inspect bounds on the absolute values $|y|$ and $|z|$ of the variables. The iteration bound for the second loop is approximated by $|y| + |z|$, and therefore, the runtime for the second loop is super-polynomial.

However, the expression $\mathsf{max}(0, y - z)$ is constant during the iteration of the first loop, and thus, the runtime of the program is linear. This imprecision is due to the choice of abstraction in KoAT and paicc. While extending the approaches to incorporate more sophisticated norms is non-trivial, it would be interesting to instrument programs beforehand with an additional transformation step.

Loopus & KoAT

In the following we remark on some observations for the runtime analysis of $\mathcal{MDC}$ programs in Loopus and KoAT.

Amortised Resource Analysis

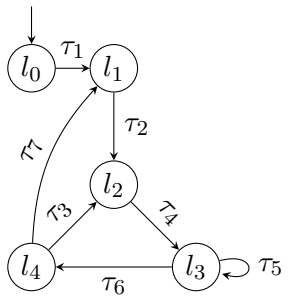
Amortised resource analysis [148, 151] aims to inspect the worst-case cost of sequences of operations. In doing so it *averages* the cost (or runtime) of a sequence of operations over the whole program execution. The approach has been established by Tarjan and Sleator in the context of self-balancing data structures.

In what follows we address the discussion about amortisation for $\mathcal{MDC}$ programs in Sinn et al. [145]. Below, we recall the program of Example 3.29 on page 49. The program models Tarjan's *stack* example [151]. In each iteration of the outer loop, one of the following sequence of operations is performed. *Push one* element onto the stack or *pop many* elements from the stack. We associate to a single push or pop operation the cost (or runtime) one. Since the operations mimic a stack, it is easy to see that *pop many* cannot pop more elements than are available on the stack. In total, m elements are pushed onto the stack and therefore at most m elements are popped. We obtain the runtime bound $2m$.

Additionally, we illustrate an approximation using Core syntax. The approximation abstracts the conditional control flow induced by `while` with bounded control flow induced

by `loop`. The analysis of Core programs conceptually aligns with our initial discussion about modular resource analysis in Section 3.3.4 on page 19. We inspect the iteration bound of the outer loop, the iteration bound of the inner loop, and the size bound of the loop bound variables. We obtain the runtime bound m^2.

```
     main(m)
      i = m;  n = 0
l1 :  while( i > 0)
        i = i − 1
l2 :    if(∗)
          n = n + 1
        else
l3 :      while(n > 0 ∧ ∗)
            n = n − 1
l4 :    skip
```




```
main(m)
i = m;  n = 0
loop i
  skip
  choice
    { n = n + 1 }
    {
    loop n
      skip }
skip
```

As discussed in Section 3.6.1, the worst-case runtime bound that is obtained by the path bound analysis of $\mathcal{MDC}$ programs is $2m$. The bound is derived from the LexRF $\langle i : \pi_1, i : \pi_2, n : \pi_3 \rangle$ (after some additional simplifications). The discussion in Sinn et al. [145] provides an informal argument for the support of amortised bound analysis. In particular, the lexicographic ranking function is interpreted as a *multidimensionial potential function*. Consider the LexRF $\langle i : \pi_1, i : \pi_2, n : \pi_3 \rangle$. Then, one can imagine that the potential of π_3 can be increased by the operations of π_2 and π_1.

The authors provide ample experimental evidence to show that their approach supports amortisation, in the sense that the obtained bounds are often asymptotic smaller than one would expect from the loop-nesting depth, and claim that all other considered tools in the experiment, which includes KoAT, fail to infer precise bounds on these examples. However, we obtain a *linear* runtime bound with KoAT on the running example, given that we augment the program with a supporting invariant $i \geqslant 0$ for transition τ_5. We were also able to infer a linear bound using paicc, given that we restrict the domain of the valuation to the natural numbers. In the following we provide some additional insights.

Based on the application of LexRFs, one may expect a non-linear bound. We recall that $\mathsf{pb}(\pi)$ denotes the bound on the maximal number of occurrences of the path π and $\mathsf{sb}(x)$ denotes the bound on the maximal valuation of the variable x. Take the co-domain of the ranking function $(\mathsf{pb}(\pi_1), \mathsf{pb}(\pi_2), \mathsf{pb}(\pi_3))$. Since each loop path π_i is associated to a variable, we may consider $(\mathsf{sb}(i), \mathsf{sb}(i), \mathsf{sb}(n))$ instead. Let $\geqslant_{\mathsf{lex}}$ denote the lexicographic order over $\mathbb{N}^d$, i.e. $(x_1, \ldots, x_d) >_{\mathsf{lex}} (x'_1, \ldots, x'_d)$ if $x_i > x'_i$ for any $1 \leqslant i \leqslant d$ and $x_j \geqslant x'_j$ for all $1 \leqslant j < i$. Then, the height of the order relation, i.e. the maximum length of a strictly descending chain $n_0 >_{\mathsf{lex}} n_1 >_{\mathsf{lex}} \cdots >_{\mathsf{lex}} n_n$, is $\mathsf{sb}(i) \cdot \mathsf{sb}(i) \cdot \mathsf{sb}(n)$. However, it is actually enough to consider a disjunctive relation for $\mathcal{MDC}$ programs, i.e. $(x_1, \ldots, x_d) >_{\mathsf{dis}} (x'_1, \ldots, x'_d)$ if $x_i \geqslant x'_i$ for all $1 \leqslant i \leqslant n$ and $x_i > x'_i$ for any $1 \leqslant i \leqslant n$. Alternatively, consider the sum $\sum_{i=1}^{d} x_i$ with the standard order on natural numbers.

This construction is analogues to non-trivial quasi-ranking functions. Indeed, the linear runtime bound returned by KoAT is obtained via the synthesis of linear quasi-RFs, which bound the maximal number of occurrences of the individual transitions. What is left, is to consider when the synthesis of RFs succeeds. Consider a $\mathcal{MDC}$ program where

constraints are restricted to $x' \leqslant x + k$ with $k \in \mathbb{Z}$, in particular, k does not represent a symbolic constant. Suppose that $\langle x_1 : \pi_1, \ldots, x_n : \pi_n \rangle$ is a LexRF. Then, the path bounds on π_i are all linear and amenable to the synthesis approach of RFs discussed in the preliminaries.

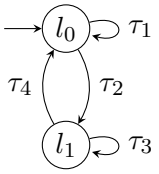
We emphasise that the modular approach to bound inference in KoAT, which is based on inspecting subprograms separately, is not used for the running example. This is also the case for paicc, which infers a loop structure with a single non-root loop. In particular, instead of nested loops, like above, the abstraction is a program with a single loop and therefore effectively limiting the multiplicative composition in the growth-rate analysis.

Most relevant, the runtime bounds that are inferred by KoAT and paicc are obtained solely via the inference of polynomial RFs that inspect the whole program. The precise connection between amortised resource analysis and polynomial interpretations (and polynomial ranking functions) is discussed in Hofmann and Moser [98].

Above, we restrict the discussion to constraints $x' \leqslant x + k$ with $k \in \mathbb{Z}$. However, Loopus processes the more general case with k being symbolic constant expressions that evaluate to $\mathbb{Z}$. This does not complicate the bound inference algorithm in Loopus, however, it allows for the construction of non-linear bounds. In contrast, in KoAT and paicc non-linear bounds are primarily inferred via a modular analysis. These approaches inspect and combine worst-case runtime and size bounds of subprograms rather than averaging the cost of evaluating the subprogram.

Case Study: Quadratic $\mathcal{VASS}$

$\mathcal{MDC}$ programs are closely related to $\mathcal{VASS}$ programs. Consider the following example conforming to standard $\mathcal{VASS}$ (see Section 3.4.5 on page 31).



$$\begin{aligned}
\tau_1 &\colon \langle l_0, l_0, x' = x - 1 \wedge y' = y + 1 \rangle \\
\tau_2 &\colon \langle l_0, l_1, x' = x \wedge y' = y \rangle \\
\tau_3 &\colon \langle l_1, l_1, x' = x + 1 \wedge y' = y - 1 \rangle \\
\tau_4 &\colon \langle l_1, l_0, x' = x - 1 \wedge y' = y \rangle
\end{aligned}$$

The program has a LexRF $\langle x+y \colon \tau_2, \tau_4, x \colon \tau_1, y \colon \tau_2 \rangle$. A decidable procedure for precise asymptotic bounds for $\mathcal{VASS}$ is presented by Brázdil et al. [45]. The proposed algorithm bears resemblance with the lexicographic decomposition discussed in Section 3.3.5. The asymptotic bound is obtained by 1 plus the maximal nesting depth of the decomposition. For the given example, a quadratic runtime bound is inferred. As informal argument consider that the maximal valuation of x and y is $x + y$. This provides a bound on $\tau_2 \cdot \tau_4$. Therefore, both loops can be entered at most $x + y$ times. The loops can then be inspected individually.

Next, we inspect the analysis in Loopus based on $\mathcal{MDC}$ constraints (see Section 3.6.2 on page 48). The proposed control flow abstraction collects all loop paths. We obtain

the following system.

$$\begin{aligned} \tau_1 = \pi_1 \colon \langle \bullet, \bullet, x' \leqslant x - 1 \wedge y' \leqslant y + 1 \rangle \\ \tau_3 = \pi_2 \colon \langle \bullet, \bullet, x' \leqslant x + 1 \wedge y' \leqslant y - 1 \rangle \\ \tau_2 \cdot \tau_4 = \pi_3 \colon \langle \bullet, \bullet, x' \leqslant x - 1 \wedge y' \leqslant y \rangle \end{aligned}$$

The system is non-terminating. Consider the run obtained by applying $\pi_1 \cdot \pi_2$ consecutively. It is easy to see that no can be obtained. The control flow abstraction is the choice of abstraction in Loopus. This example shows that it is a *lossy* abstraction.

We also have not been able to infer a polynomial bound on the runtime complexity with KoAT and paicc. With the given lexicographic decomposition both tools fail to control the size bound on the variables.

3.10 Concluding Remarks

In this chapter we have been concerned with practical and theoretical aspects of automated resource analysis of integer-valued imperative programs. We have studied three tools that have been developed in recent years, namely KoAT, Loopus and paicc, and outlined the key concepts of the individual approaches to resource analysis. Our discussion covers practical aspects as well as theoretical properties of the abstract program representations that are used within the tools. Furthermore, we have given an overview of known relevant theoretical results on different related abstract program representations that are known from the literature.

Chapter 4

Imperative Programs with Heap

In this chapter we are concerned with automated resource analysis of *imperative programs with heap allocated data*. We consider a Goto programming language with primitives for allocating and manipulating (typed) *records*. Adding support for heap allocated data raises new challenges for the resource analysis, among them, *composed data structures*, *data sharing* and *side effects*. To automate the analysis we focus on a *transformational* approach that makes use of existing *abstract program representations*. Here, a transformation from program $\mathcal{P}$ to $\mathcal{P}'$ is called *complexity reflecting* if the (worst-case runtime) complexity of P is bounded by the (worst-case runtime) complexity of $\mathcal{P}'$. Consequently, we can assess the complexity of the target $\mathcal{P}$ via its abstraction $\mathcal{P}'$. In the course of this chapter we present two complexity reflecting transformations for our programming language that are known from the literature:

(i) A *size abstraction* to *constraint transition systems* (CTSs for short), and

(ii) a *term abstraction* to *constraint term rewrite systems* (cTRSs for short).

Moser and Schaper [119] present the technical details for the term abstraction of *object-oriented bytecode programs*. In this chapter, however, we focus on the central ideas of both transformations. We provide a uniform presentation and give additional insights about conceptual challenges that arise from the resource analysis of programs with heap allocated data.

Section 4.1 illustrates a motivating example and provides an informal discussion about its worst-case runtime behaviour. In Section 4.2 we recall constraint term rewrite systems. The programming language under consideration is presented in Section 4.3, while the size and term abstraction are presented in Section 4.4. Section 4.5 provides additional insights on the term abstraction presented in Moser and Schaper [119]. In Section 4.6 we present related work. Finally, we conclude this chapter in Section 4.7.

4.1 Introduction

In this section we provide some initial insights for the automated resource analysis of programs with heap allocated data structures. When considering programs with heap, it is not immediate how to define resource analysis at first. We recall three alternatives.

First, consider *whole-program* analysis. In whole-program analysis we assume that the heap of the initial configuration is empty. This is a very natural definition and

enables a straightforward extension of the notion of complexity from programs without heap. However, by its nature this approach is not modular and thus excludes common interesting use cases. Consider for instance, a library that exposes a data structure `List` together with several methods, e.g. `insert`, `member`, `sort` etc. A motivating use case for static resource analysis is to infer and guarantee the resource behaviour of all individual methods of the library. However, in whole-program analysis it is not obvious how to represent all possible instances of a data structure.

Second, consider the case where the procedure of interest is separated into an *initialisation* and *computation* phase. For example, `main`(){*init*; *comp*} (cf. Hainry and Péchoux [90]). Here, we are interested in analysing the subprogram *comp* with respect to all output states that are generated by the subprogram *init*. By construction, this approach guarantees that all input states for the computation phase are well-formed program states. This approach is not restricted to a single program execution. Imagine that data is generated during the initialisation phase using built-in non-determinism. Then, it is essential to *stratify* the input states of the computation phase. Typically, this depends on the exact nature of the analysis. Since the heap of the input states may contain data, we require a finite representation of the heap, viz its *input size*. As an illustrating example, consider that input states are specified in terms of the valuation of integer variables together with the number of allocated data cells in the heap.

Third, consider a variation of the second case using *data-flow* information (cf. Albert et al. [4]). Instead of an initialisation phase, we use data-flow information as *assumptions* to restrict the set of input states. In the context of data-flow analysis these assumptions can often be made precise. For example, consider the iteration over a singly-linked list `iterate`(*l*:`List`). An analysis may distinct the cases where the argument l references an acyclic list and where l references a cyclic list. For the latter case the number of iterations is unbounded.

In Section 4.4 we discuss two abstractions of programs with heap allocated data, which make use of additional data-flow information that are obtained by external analyser. Thus, we are going to present the abstractions in terms of the last alternative. As motivating example, we consider the traversal over a binary tree.

Example 4.1 (Binary Tree Traversal)**.** Figure 4.1 depicts a program in Java syntax that declares a tree and stack data structure together with a few standard methods. We investigate the worst-case runtime of the method `void traverse`(`Tree` *t*). The method takes as argument a variable with type `Tree` which references an instance of the class `Tree` allocated on the heap. We are going to see that the runtime does not only depend on the *input size* but also on the *input shape*.

For the sake of the discussion, we omit any formal definitions for now. We inspect the worst-case runtime in terms of the maximal derivation height of all input configurations. The heap of the input is possible non-empty. A finite representation of the heap is obtained by mapping the data to its size. Imagine that the heap can be represented as a labelled directed graph in which nodes represent locations in the memory. We fix the input size of the argument by the number of reachable locations.

```
class Tree{ int val; Tree left; Tree right; }

class Stack{
  Tree elem; Stack next;

  Tree pop(){
    Tree t    = this.next.elem;
    this.next = this.next.next;
    return t;
  }

  void push(Tree t){
    Stack st = new Stack();
    st.next = this.next;
    st.elem = t;
    this.next = st;
  }

  boolean isEmpty(){
    return this.next == null;
  }
}
```

```
public class Main {

  void traverse(Tree t){
    Stack st = new Stack();
    st.push(t);
    while(!st.isEmpty()){
      t = st.pop()
      //visit(t)
      if(t != null){
        st.push(t.right);
        st.push(t.left)
      }
    }
  }

}
```

Figure 4.1: Binary Tree Traversal in Java.

Next, we discuss the expected result. The intention of the method `traverse` is to visit each node once. However, the algorithm actually depends on the number of paths to each node. We obtain three interesting cases.

(i) If the argument is *cyclic*, that is, there is a cycle in the graph representation of the heap, then the number of visits to its nodes is unbounded. In fact, the program is non-terminating.

(ii) If the argument is *acyclic*, that is, the graph representation of the heap is a directed acyclic graph, then the program is terminating. Consider a *fully-shared binary tree* in which the left and right child of a parent node always reference the same node. Then the number of paths starting from the root is exponential in the number of nodes (or height) of the tree. The expected runtime is exponential.

(iii) If the argument is *tree-shaped*, that is, the graph representation of the heap conforms to a binary tree, then the program is terminating. Consider a *full binary tree* in which the left and right child never reference the same node. Then we obtain a linear bound on the maximal derivation height in terms of the input size.

The last case is what one would expect when formally reasoning about *binary trees*. However, in an object-oriented language like Java there is no guarantee that the referenced structure `Tree` is actually a binary tree.

4.2 Preliminaries

In this section we recall *constraint term rewrite systems*, see Moser and Schaper [119] for details. For an overview for (standard) term rewrite systems see Baader and Nipkow [22].

We denote by $\mathcal{V}$ a countably infinite set of variables and by $\mathcal{F}$ a signature. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is written as $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A rewrite relation $\to$ is a binary relation on terms closed under contexts and stable under substitution.

Let $\mathcal{C}$ be a (not necessarily finite) sorted signature, and let $\mathcal{V}'$ denote a countably infinite set of sorted variables. Furthermore, let T denote a theory over $\mathcal{C}$. Quantifier-free formulas over $\mathcal{C}$ are called *constraints*. Suppose $\mathcal{F}$ is a sorted signature that extends $\mathcal{C}$ and let $\mathcal{V} \supseteq \mathcal{V}'$ denote an extension of the variables in $\mathcal{V}'$. Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the set of *(sorted) terms* over the signature $\mathcal{F}$ and $\mathcal{V}$. Note that the sorted signature is necessary to distinguish between *theory* variables that are to be interpreted over the theory T and *term* variables whose interpretation is free. A *constraint rewrite rule*, denoted as $l \to r \,[\![C]\!]$, is a triple consisting of terms l and r, together with a constraint C. We assert that $l \notin \mathcal{V}$, but do *not* require that $\mathsf{Var}(l) \supseteq \mathsf{Var}(r) \cup \mathsf{Var}(C)$, where $\mathsf{Var}(t)$ ($\mathsf{Var}(C)$) denotes the variables occurring in the term t (constraint C). A *constraint term rewrite system* (cTRS for short) is a finite set of constraint rewrite rules.

Let $\mathcal{R}$ denote a cTRS. A *context* D is a term with exactly one occurrence of a *hole* $\Box$, and $D[t]$ denotes the term obtained by replacing the hole $\Box$ in D by the term t. A *substitution* σ is a function that maps variables to terms, and $t\sigma$ denotes the homomorphic extension of this function to terms. We define the rewrite relation $\to_{\mathcal{R}}$ as follows. For terms s and t, $s \to_{\mathcal{R}} t$ holds, if there exists a context D, a substitution σ and a constraint rule $l \to r \,[\![C]\!] \in \mathcal{R}$ such that $s =_T D[l\sigma]$ and $t = D[r\sigma]$ with $T \vdash C\sigma$. Here $=_T$ denotes unification modulo T. For extra variables x, possibly occurring in t, we demand that $\sigma(x)$ is in normal-form.

We often drop the reference to the cTRS $\mathcal{R}$, if no confusion can arise from this. A function symbol in $\mathcal{F}$ is called *defined* if f occurs as the root symbol of l, where $l \to r \,[\![C]\!] \in \mathcal{R}$. Function symbols in $\mathcal{F} \setminus \mathcal{C}$ that are not defined are called *constructor* symbols, and the symbols in $\mathcal{C}$ are called *theory* symbols. A term $t = f(t_1, \ldots, t_k)$ is a *basic term* if f is a defined symbol and the terms t_i are only built over constructor, theory symbols, and variables.

In the course of this chapter we introduce *integer* cTRSs, i.e. rewrite systems in which the theory T is fixed to support standard arithmetic operations over the integers. Most relevant, integer cTRSs generalise integer CTSs which have been formally introduced in Chapter 3 on page 16.

4.3 Goto Programs with Records

In the following we present the imperative programming language `Goto`R with support for arithmetic expressions over the integers and custom data types in form of *records*.

Each program is associated with a finite set of variable declarations and record declarations. In the following Var denotes a finite set of variable identifiers, RName

denotes a finite set of record identifiers and Field denotes a finite set of field identifiers. The set of types (Type) of a program is given by the integer type Int together with RName. In the following let $x \in \mathsf{Var}$, $fld \in \mathsf{Field}$ and $T \in \mathsf{Type}$. A *variable declaration* is of the form $x\colon T$. A *record declaration* is of the form $T\{fld_1\colon T_1, \ldots, fld_k\colon T_k\}$, where all field identifiers are distinct.

A program *value* $v \in \mathsf{Val}$ is either an integer, an address or the null value `null`. We do not support pointer arithmetic and therefore keep the concrete type of addresses $p \in \mathsf{Addr}$ abstract. A *store* (or environment) $\sigma \in \Sigma$ is a mapping from variables to values. A *heap* $\mu \in \mathsf{Heap}$ is a partial mapping from addresses to records. We decorate records with its type. A *record* provides a mapping from its defined fields (or selectors) to values. Let $\mathsf{Loc} \triangleq \mathbb{N}$ define the set of program locations. A configuration $(l, \sigma, \mu) \in \mathsf{Conf}$ is a triple consisting of program location l, store σ and heap μ.

$$\begin{aligned} \mathsf{Type} &\triangleq \mathsf{RName} \uplus \{\mathsf{Int}\} & \Sigma &\triangleq \mathsf{Var} \to \mathsf{Val} \\ \mathsf{Val} &\triangleq \mathsf{Addr} \uplus \mathbb{Z} \uplus \{\mathtt{null}\} & \mathsf{Heap} &\triangleq \mathsf{Addr} \to \mathsf{Record}_T \\ \mathsf{Record}_T &\triangleq \mathsf{Field} \to \mathsf{Val} & \mathsf{Conf} &\triangleq \mathsf{Loc} \times \Sigma \times \mathsf{Heap} \end{aligned}$$

In the following let $n \in \mathbb{Z}$, $x, y \in \mathsf{Var}$, $fld \in \mathsf{Field}$, $l \in \mathsf{Loc}$, and $T \in \mathsf{Type}$. The statements of a program are given in normal form that simplify the subsequent presentation. The syntax of `Goto`R is given by the following grammar:

$$\begin{aligned} a \in \mathsf{AExp} &::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \\ \mathsf{Stmt} &::= x = y && \textit{variable assignment} \\ &\mid x = a && \textit{assignment of expression} \\ &\mid x = \mathtt{null} && \textit{assignment of null} \\ &\mid x = \mathtt{new}\ T(y_1, \ldots, y_k) && \textit{record allocation} \\ &\mid x = y.fld && \textit{field selection} \\ &\mid x.fld = y && \textit{field update} \\ &\mid \mathtt{if}\ a_1 \geqslant a_2\ \mathtt{goto}\ l && \textit{conditional jump} \\ &\mid \mathtt{ifnull}\ x\ \mathtt{goto}\ l && \textit{conditional jump.} \end{aligned}$$

A `Goto`R program has a unique *method declaration* of the form

$$\mathtt{method}(x_1\colon T_1, \ldots, x_j\colon T_j)\ \mathtt{with}\ x_{j+1}\colon T_{j+1} \cdots x_k\colon T_k\ .$$

We assert that `Goto`R programs are *well-typed*, in doing so `null` is implicitly decorated with a type. For the sake of simplicity we assume that the language is *garbage-collected*. However, since we focus on runtime complexity and the upcoming transformations formally restricts to reachable addresses within the heap, the transformations are also sound for a language that is not garbage-collected. We note that integer values can either be directly manipulated via the store or indirectly via records. Consequently, programs without heap can be directly represented as constraint transition systems.

Example 4.2 (Binary Tree Traversal in Goto$R$)**.** The following program is a variation of tree traversal of Example 4.1 written in Goto$R$.

```
data Tree  {val : Int,   left : Tree, right : Tree}
data Stack {elem : Tree, next : Stack}

   traverse(t : Tree) with st : Stack, tmp : Tree
0   st = null
1   st = new Stack(t, st)
2   ifnull st goto B
3    t  = st.elem
4    st = st.next
5    ifnull t goto 2
6     tmp = t.right
7     st  = new Stack(tmp, st)
8     tmp = t.left
9     st  = new Stack(tmp, st)
A     if 1 > 0 goto 2
```

The *one-step reduction relation* of Goto$R$ is illustrated in Figure 4.2. With $[\![\cdot]\!] : \mathsf{AExp} \to \Sigma \to \mathbb{Z}$ we denote the evaluation function of arithmetic expressions, i.e. $[\![a]\!](\sigma)$ gives the result of evaluating a with store σ. Most rules are standard.

case: $x = y$	$(l, \sigma, \mu) \to (l+1, \sigma[x \mapsto \sigma(y)], \mu)$
case: $x = a$	$(l, \sigma, \mu) \to (l+1, \sigma[x \mapsto [\![a]\!](\sigma)], \mu)$
case: $x =$ `null`	$(l, \sigma, \mu) \to (l+1, \sigma[x \mapsto \texttt{null}], \mu)$
case: $x =$ `new` $T(y_1, \ldots, y_k)$	$(l, \sigma, \mu) \to (l+1, \sigma, \mu \uplus [q.fld_i \mapsto \sigma(y_i)]_{1 \leqslant i \leqslant k})$
case: $x = y.fld$	
$\sigma(y) \in \mathsf{dom}(\mu)$	$(l, \sigma, \mu) \to (l+1, \sigma[x \mapsto \mu(\sigma(y)).fld], \mu)$
case: $x.fld = y$	
$\sigma(x) \in \mathsf{dom}(\mu)$	$(l, \sigma, \mu) \to (l+1, \sigma, \mu[\sigma(x).fld \mapsto \sigma(x)])$
case: `ifnull` x `goto` l'	
$\sigma(x) =$ `null`	$(l, \sigma, \mu) \to (l', \sigma, \mu)$
otherwise	$(l, \sigma, \mu) \to (l+1, \sigma, \mu)$
case: `if` $a_1 \geqslant a_2$ `goto` l'	
$[\![a_1]\!](\sigma) \geqslant [\![a_2]\!](\sigma)$	$(l, \sigma, \mu) \to (l', \sigma, \mu)$
otherwise	$(l, \sigma, \mu) \to (l+1, \sigma, \mu)$

Figure 4.2: One-Step Reduction Relation of Goto$R$.

In the case $x = a$ we update the store σ with x being equal to the result of evaluating the arithmetic expression a. In the case $x =$ `new` $T(y_1, \ldots, y_k)$ we expect the arguments to conform to the record declaration of $T \in \mathsf{RName}$. The address q is fresh. For clarity,

we write $q.fld$ instead of $q(fld)$ when referencing or updating record fields. In the case of field selection $x = y.fld$ and field update $x.fld = y$ we test for `null`. Dereferencing `null` halts the computation in the current configuration. The conditional cases are straightforward.

Next, we fix the notion of *worst-case runtime complexity* of Goto$R$ programs. The runtime of an initial configuration is the maximal derivation height of all possible program traces starting from it (see Definition 2.6 on page 10). As discussed above, the runtime may depends on properties on the shape of the data allocated on the heap. Therefore, we define the runtime complexity in terms of a set of *initial configurations* $I \subseteq \mathsf{Conf}$ and maximal *input size* $m \in \mathbb{N}$. We use a univariate notion of the complexity function to simplify the subsequent presentation and remark that resource analysis tools often provide a more refined notion of bounds. The notion of input size used here, is similar to the one used by Hainry and Péchoux [90], which provide a type based characterisation of polynomial time computable functions of an object-oriented programming language.

The input size of a configuration is defined by collecting all its values. In doing so, we associate the size of `null` with zero, the size of an address with one, and the size of an integer value with its absolute value.

Definition 4.3 (Input Size)**.** Let $v \in \mathsf{Val}$. The *value size* of v is

$$\mathsf{size}(v) \triangleq \begin{cases} \mathsf{abs}(v) & \text{if } v \in \mathbb{Z} \,, \\ 0 & \text{otherwise} \,. \end{cases}$$

The *input size* of a configuration $c = (l, \sigma, \mu)$ is defined as

$$\mathsf{size}(c) \triangleq \textstyle\sum_{x \in \mathsf{Var}} \mathsf{size}(x) + \sum_{p \in \mathsf{dom}(\mu)} \left(1 + \sum_{v \in \mathsf{rng}(\mu(p))} \mathsf{size}(v)\right) \,.$$

Definition 4.4 (Runtime Complexity)**.** Let $\mathcal{P}$ be a Goto$R$ program and $I \subseteq \mathsf{Conf}$ denote a set of initial configurations. Furthermore, let $m \in \mathbb{N}$ denote the maximal input size. The *worst-case runtime complexity* $\mathsf{rc} \colon \mathbb{N} \to \mathbb{N}^\infty$ of $\mathcal{P}$ on I is defined by

$$\mathsf{rc}^I_{\mathcal{P}}(m) \triangleq \{\mathsf{dh}_{\mathcal{P}}(c) \mid c \in I \text{ and } \mathsf{size}(c) \leqslant m\} \,.$$

4.4 Complexity Reflecting Program Abstraction

In this section we present a *size abstraction* to CTSs and a *term abstraction* to cTRSs of the Goto$R$ programming language. Both abstractions are *complexity reflecting*, i.e. an upper bound on the worst-case runtime complexity of the abstract program implies an upper bound on the worst-case runtime complexity of the target program.

To make the subsequent program abstractions more viable in practice, we make use of different heap shape properties. In particular, we employ the *acyclicity domain* by Rossignoli and Spoto [138] and the *pair sharing domain* by Secci and Spoto [142], which have been developed for the static program analyser Julia [149]. The referenced abstract domains have been defined within the *abstract interpretation* framework (Cousot and Cousot [59]). While we provide the definition for the individual domains, we skip the details of the abstract semantics.

Definition 4.5 (Reachable Addresses)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration. The set of *reachable addresses* of value $v \in \mathsf{Val}$ is defined by:

$$\begin{aligned}
\mathsf{addresses}^0(v) &\triangleq \{v\} \cap \mathsf{Addr} \\
\mathsf{addresses}^{i+1}(v) &\triangleq \bigcup\{\mathsf{rng}(\mu(p)) \cap \mathsf{Addr} \mid p \in \mathsf{addresses}^i(v)\} \\
\mathsf{addresses}^+(v) &\triangleq \textstyle\bigcup_{i \geqslant 1} \mathsf{addresses}^i(v) \\
\mathsf{addresses}^*(v) &\triangleq \mathsf{addresses}^0(v) \cup \mathsf{addresses}^+(v)
\end{aligned}$$

Let $x \in \mathsf{Var}$, then $\mathsf{addresses}(x) \triangleq \mathsf{addresses}^*(\sigma(x))$.

We motivate the acyclicity domain. The main application of the domain is to approximate data on the heap that is acyclic. This enables to define different behaviour for data access, in particular *field selection*. Consider for example, iterating over a singly-linked list `while`$(x$!= `null`$)\{\ x = x.next\ \}$. If the list is cyclic, then the referenced heap address of x may change in each iteration but the data accessible from x not. Alternatively, consider that we are interested in establishing a ranking argument for termination based on the length of the list from x to `null`. The length of the list is not defined (or unbounded) if the list cyclic.

Definition 4.6 (Acyclic)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration. A value $v \in \mathsf{Val}$ is called *cyclic* at program location l if there exists $v' \in \mathsf{addresses}(v)$ such that $v' \in \mathsf{addresses}^+(v')$, otherwise v is *acyclic*. Similarly, a variable $x \in \mathsf{Var}$ is called *cyclic* at program location l if $\sigma(x)$ is cyclic, otherwise x is *acyclic*.

Following the presentation in Rossignoli and Spoto [138] we define the abstract domain in terms of an *abstraction* and *concretisation* function (cf. Nielson et al. [125]).

Definition 4.7 (Acyclicity Domain)**.** We define the *acyclicity domain* as $\mathsf{AC} \triangleq \mathcal{P}(\mathsf{Var})$ together with the *abstraction* function $\alpha_l^{\mathsf{AC}} \colon \mathsf{AC} \to \mathcal{P}(\mathsf{Conf})$ and the *concretisation* function $\gamma_l^{\mathsf{AC}} \colon \mathcal{P}(\mathsf{Conf}) \to \mathsf{AC}$:

$$\begin{aligned}
\alpha_l^{\mathsf{AC}}(CS) &\triangleq \{x \in \mathsf{Var} \mid \text{for all } (l, \sigma, \mu) \in CS, x \text{ is acyclic at } l\ \} \\
\gamma_l^{\mathsf{AC}}(AC) &\triangleq \{(l, \sigma, \mu) \in \mathsf{Conf} \mid \text{for all } x \in AC, x \text{ is acyclic at } l\}
\end{aligned}$$

In the subsequent program transformations we employ different abstractions for heap allocated data. These abstractions cannot represent operations that manipulate data precisely. To accommodate for possible *side effects* we employ the pair sharing domain. Consider a field update $x.fld = y$ associated with program location l. The pair sharing domain overapproximates the set of variables that share data with x at program location l and which would be indirectly affected by the field update.

Definition 4.8 (Variable Sharing)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration and $x, y \in \mathsf{Var}$. We say that x and y *share* at program location l if $\mathsf{addresses}(x) \cap \mathsf{addresses}(y) \neq \varnothing$.

Definition 4.9 (Pair Sharing Domain)**.** We define the *pair sharing domain* by $\mathsf{SH} \triangleq \mathcal{P}(\mathsf{Var} \times \mathsf{Var})$ together with the *abstraction* function $\alpha_l^{\mathsf{SH}} \colon \mathcal{P}(\mathsf{Conf}) \to \mathsf{SH}$ and the *concretisation* function $\gamma_l^{\mathsf{SH}} \colon \mathsf{SH} \to \mathcal{P}(\mathsf{Conf})$:

$$\alpha_l^{\mathsf{SH}}(CS) \triangleq \{(x,y) \in \mathsf{Var} \times \mathsf{Var} \mid \text{there exists } (l,\sigma,\mu) \in CS \text{ st. } x \text{ and } y \text{ share at } l\}$$
$$\gamma_l^{\mathsf{SH}}(SH) \triangleq \{(l,\sigma,\mu) \in \mathsf{Conf} \mid \text{for all } x,y \in \mathsf{Var}, \text{ if } x \text{ and } y \text{ share at } l \text{ then } (v,w) \in SH\}$$

In the remainder of this chapter we make use of the following observation. Let $\mathcal{P}$ be a program, and $I \subseteq \mathsf{Conf}$ denote a set of initial configurations with program location $l = 0$. Within the abstract interpretation framework we obtain an abstraction of $\mathcal{P}$ with respect to I from $I^\sharp \triangleq \alpha_0(I)$. Crucially, this abstraction overapproximates the one-step reduction relation of $\mathcal{P}$. Suppose that $c^\sharp$ and $d^\sharp$ are the abstractions obtained for the two program locations l and l', and $(l_0,\sigma_0,\mu_0) \in \gamma_0(I^\sharp)$. Then, $(l_0,\sigma_0,\mu_0) \to_{\mathcal{P}}^* (l,\sigma,\mu) \to_{\mathcal{P}} (l',\sigma',\mu') \subseteq \gamma_0(I^\sharp) \to_{\mathcal{P}}^* \gamma_l(c^\sharp) \to_{\mathcal{P}} \gamma_{l'}(d^\sharp)$. In other words, the evaluation does not get stuck with respect to the abstract domains.

Alternatively, we fix an initial abstract element $I^\sharp$. Then we obtain an abstraction of $\mathcal{P}$ with respect to the set of initial configurations $\gamma_0(I^\sharp)$. The latter variant is relevant for implementing the analysis. For instance, consider a method `iterate(`l`:List)`. If we are interested in restricting the analysis to acyclic lists, then we infer the data-flow information of the abstract domain from $I^\sharp = (\{l\}, \{(l,l)\}) \in \mathsf{AC} \times \mathsf{SH}$.

In the following we assume that programs are decorated with the acyclicity domain and the pair sharing domain. Let $\mathsf{AC}_l \in \mathcal{P}(\mathsf{Var})$ denote the acyclicity abstraction associated with program location l. Then x is acyclic at program location l if $x \in \mathsf{AC}_l$, otherwise x is maybe-cyclic. Let $\mathsf{SH}_l \in \mathcal{P}(\mathsf{Var} \times \mathsf{Var})$ denote the pair sharing abstraction associated with program location l, and let $\mathsf{SH}_l^x = \{z \mid z \in \mathsf{Var} \text{ and } (x,z) \in \mathsf{SH}_l\}$. Then x and y *may-share* at program location l if $y \in \mathsf{SH}_l^x$, otherwise x and y do not share.

The next definition guarantees that the acyclicity and pair sharing domain which is associated with the program do not contradict with the program traces that start from the considered set of initial configurations.

Definition 4.10 (Compatibility)**.** Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ denote a set of initial configurations. The set of initial configurations I is *compatible* with $\mathcal{P}$, if $I \subseteq \gamma_0^{\mathsf{AC}}(\mathsf{AC}_0) \cap \gamma_0^{\mathsf{SH}}(\mathsf{SH}_0)$.

4.4.1 Size Abstraction

Next, we discuss numeric abstractions of programs with heap allocated data. The main idea is to associate programs with a *size function* (or *norm*) sz that maps configurations to a numeric value (or a combination of values), and to relate the size of the input and output of the program relation by arithmetic constraints. This constitutes of finding constraints $\phi_{l,l'} \models sz(\sigma,\mu) \wedge sz(\sigma',\mu')$ that model the one-step reduction relation $(l,\sigma,\mu) \to_{\mathcal{P}} (l',\sigma',\mu')$ induced by the semantics of program $\mathcal{P}$ (see also *transition invariants* of Section 3.3.7 on page 26). Typically, constraints relate individual components of the configuration such as the variables of the store. The assignment of an expression $x = w + 1$, for instance, can then be expressed by the constraint $\phi \models sz(x') = sz(w+1)$.

In the following we fix the notion of size to *path-length*, which is the choice of abstraction in the termination analyser Julia [150] and the termination and cost analyser COSTA [4].

Definition 4.11 (Path-Length)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration and $p \in \mathsf{Addr}$. The *path-length* of p is defined by:

$$\begin{aligned} \mathsf{plength}^0(p) &\triangleq 0 \\ \mathsf{plength}^{i+1}(p) &\triangleq 1 + \mathsf{max}\{\mathsf{plength}^i(p') \mid p' \in \mathsf{addresses}^1(p)\} \\ \mathsf{plength}(p) &\triangleq \mathsf{lim}_{i\to\infty}\, \mathsf{plength}^i(p) \end{aligned}$$

Let $x \in \mathsf{Var}$, then

$$\mathsf{plength}(x) \triangleq \begin{cases} \mathsf{plength}(p) & \text{if } \sigma(x) = p \text{ and } p \in \mathsf{Addr}\ , \\ 0 & \text{if } \sigma(x) = \texttt{null}\ , \\ n & \text{if } \sigma(x) = n \text{ and } n \in \mathbb{Z}\ . \end{cases}$$

Informally, the path-length of an address corresponds to the maximal path that can be constructed by following the fields of the record instances allocated on the heap. The path-length of cyclic data is infinite. We extend the definition to variables of the store. The path-length abstraction of a configuration is given by the path-length of all variables in the store. We note that the path-length abstraction ignores record fields of type integer, however the integer-valued variables of the store are taking into account.

Definition 4.12 (Path-Length Domain)**.** We denote the *path-length domain* with $\mathsf{PLength} \triangleq \mathsf{Loc} \times \Sigma$. The *representation* function $(\cdot)^\circ \colon \mathsf{Conf} \to \mathsf{PLength}$ and the *concretisation* function $\gamma^\circ \colon \mathsf{PLength} \to \mathcal{P}(\mathsf{Conf})$ are defined by:

$$\begin{aligned} (c)^\circ &\triangleq (l, [x \mapsto \mathsf{plength}(x)]_{x\in\mathsf{Var}}) \\ \gamma^\circ(d) &\triangleq \{c \in \mathsf{Conf} \mid (c)^\circ = d)\} \end{aligned}$$

In Figure 4.3 we present the path-length abstraction of Goto$R$ programs to CTSs. We recall that a transition is a triple $\langle l, l', \phi\rangle$ with source location $l \in \mathsf{Loc}$, target location $l' \in \mathsf{Loc}$ and constraint $\phi \in \mathsf{BExp}$ over variables $\mathsf{Var} \cup \mathsf{Var}'$. Primed variables Var' indicate the valuation at the target location. A set of transitions is called a CTS. For further details, see Section 3.3.2 on page 16.

We make use of the restricted syntax of Goto$R$ and provide the transformation rules by case distinction on the statement at program location l. Typically, most variables are not modified. Thus, in the following let $\psi_S = \bigwedge_{x\in\mathsf{Var}\setminus S} x' = x$. We write ψ_x instead of $\psi_{\{x\}}$. We comment on the transformation rules.

The individual cases for the assignment of variables $y = x$, arithmetic expressions $y = a$ and the null value $y = \texttt{null}$ are straightforward.

In the case of $y = \texttt{new}\ T(x_1, \ldots, x_k)$ we use V to collect all non-integer arguments. We approximate the path-length of the new record instance by one plus the sum of all variables in V. It would be correct to take the maximum instead of the sum, however we restrict to max free constraints.

case: $x = y$	$\langle l, l+1, \psi_x \wedge x' = y \rangle$
case: $x = a$	$\langle l, l+1, \psi_x \wedge x' = a \rangle$
case: $x = \texttt{null}$	$\langle l, l+1, \psi_x \wedge x' = 0 \rangle$
case: $x = \texttt{new}\ T(y_1, \ldots, y_k)$	
Let $V = \{y_i \mid y_i \colon T_i \text{ and } T_i \in \mathsf{RName} \text{ for } 1 \leqslant i \leqslant k\}$.	
	$\langle l, l+1, \psi_x \wedge x' \geqslant 0 \wedge x' \leqslant 1 + \sum_{z \in V} z \rangle$
case: $x = y.fld$	
$y.fld : \mathsf{Int}$	$\langle l, l+1, \psi_x \rangle$
$y \in \mathsf{AC}_l$	$\langle l, l+1, \psi_x \wedge x' \geqslant 0 \wedge x' < y \rangle$
$y \notin \mathsf{AC}_l$	$\langle l, l+1, \psi_x \wedge x' \geqslant 0 \wedge x' \leqslant y \rangle$
case: $x.fld = y$	
$x.fld : \mathsf{Int}$	$\langle l, l+1, \psi \rangle$
$y \notin \mathsf{SH}_l^x$	$\langle l, l+1, \psi_{\mathsf{SH}_l^x} \wedge \bigwedge_{z' \in \mathsf{SH}_l^x} z' \geqslant 0 \wedge z' \leqslant z + y \rangle$
$y \in \mathsf{SH}_l^x$	$\langle l, l+1, \psi_{\mathsf{SH}_l^x} \wedge \bigwedge_{z' \in \mathsf{SH}_l^x} z' \geqslant 0 \rangle$
case: $\texttt{ifnull}\ v\ \texttt{goto}\ l'$	$\langle l, l', \psi \wedge w = 0 \rangle$
	$\langle l, l+1, \psi \wedge w > 0 \rangle$
case: $\texttt{if}\ a_1 \geqslant a_2\ \texttt{goto}\ l'$	$\langle l, l', \psi \wedge a_1 \geqslant a_2 \rangle$
	$\langle l, l+1, \psi \wedge \neg(a_1 \geqslant a_2) \rangle$

Figure 4.3: Path-Length Abstraction of Goto$R$ Programs.

In the case of field selection $x = y.fld$ we consider three subcases. First, the path-length of integer fields is undefined. Second, if y is acyclic at program location l, i.e. $y \in \mathsf{AC}_l$, then the path-length of $y.fld$ is at least one less than the path-length of y. Third, if y is maybe-cyclic at program location l, i.e. $y \notin \mathsf{AC}_l$ then the path-length of $y.fld$ is at most the path-length of y.

The most interesting case is field assignment $x.fld = y$. The path-length of x is unchanged if an integer field is updated. Otherwise, we collect the set of variables that may-share with x at program location l in SH_l^x. The path-length of those variables is potentially affected by the update. By definition, if x is not always `null` at program location l, then $x \in \mathsf{SH}_l^x$. There are two subcases to consider. First, we consider the case that x and y do not share before the assignment, i.e. $y \notin \mathsf{SH}_l^x$. Then we add the path-length of y to the path-length of x and all variables that share with x. Second, we consider the case that x and y may-share before the assignment, i.e. $y \in \mathsf{SH}_l^x$. Then we assume the worst-case, that is, that x and all variables that share with x are potentially cyclic after the update. Therefore, the path-length of all variables in SH_l^x is unbounded after the assignment. The conditional cases are straightforward.

Example 4.13 (Path-Length Abstraction of Binary Tree Traversal)**.** The following CTS is obtained from Example 4.2 via the rules of Figure 4.3. For the abstraction we assume that the input argument is acyclic.

$$
\begin{array}{lll}
\langle l_0, l_1, \psi \rangle & & \\
\langle l_1, l_2, \psi_{st} & \wedge\, st' \geqslant 0 & \wedge\, st' \leqslant 1 + st + t \rangle \\
\langle l_2, l_B, \psi & \wedge\, t = 0 \rangle & \\
\langle l_2, l_3, \psi & \wedge\, t > 0 \rangle & \\
\langle l_3, l_4, \psi_t & \wedge\, t' \geqslant 0 & \wedge\, t' < st \rangle \\
\langle l_4, l_5, \psi_{st} & \wedge\, st' \geqslant 0 & \wedge\, st' < st \rangle \\
\langle l_5, l_2, \psi & \wedge\, t = 0 \rangle & \\
\langle l_5, l_6, \psi & \wedge\, t > 0 \rangle & \\
\langle l_6, l_7, \psi_{tmp} & \wedge\, tmp' \geqslant 0 & \wedge\, tmp' < t \rangle \\
\langle l_7, l_8, \psi_{st} & \wedge\, st' \geqslant 0 & \wedge\, st' < 1 + st + tmp \rangle \\
\langle l_8, l_9, \psi_{tmp} & \wedge\, tmp' \geqslant 0 & \wedge\, tmp' < t \rangle \\
\langle l_9, l_A, \psi_{st} & \wedge\, st' \geqslant 0 & \wedge\, st' < 1 + st + tmp \rangle \\
\langle l_A, l_2, \psi \rangle & &
\end{array}
$$

We obtain the following soundness result (compare with Albert et al. [4] for details).

Theorem 4.14 (Soundness of $\mathsf{PLength}$)**.** *Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ be a compatible set of initial configurations. Moreover, let $\mathcal{P}^\circ$ be the path-length abstraction of $\mathcal{P}$. Suppose that there exists a trace $c \to_{\mathcal{P}}^n d$ in $\mathcal{P}$ starting from $c \in I$. Then, there exists a trace $(c)^\circ \to_{\mathcal{P}^\circ}^n (d)^\circ$ in $\mathcal{P}^\circ$.*

To relate the abstraction to the runtime complexity of Definition 4.4 we provide a suitable notion of input size based on the path-length.

Definition 4.15 (Input Size of $\mathsf{PLength}$)**.** The *input size* of $c \in \mathsf{PLength}$ is defined by

$$\mathsf{size}^\circ(c) \triangleq \textstyle\sum_{x \in \mathsf{Var}} \mathsf{abs}(\mathsf{plength}(x)) \;.$$

We define the runtime in terms of the maximal derivation height of possible input configurations.

Definition 4.16 (Runtime Complexity of $\mathsf{PLength}$)**.** Let $\mathcal{P}^\circ$ be a CTS and $I^\circ \subseteq \mathsf{PLength}$ be a set of initial configurations. The *worst-case runtime complexity* $\mathsf{rc}^{I^\circ}_{\mathcal{P}^\circ}(m) \colon \mathbb{N} \to \mathbb{N}^\infty$ of $\mathcal{P}^\circ$ on I° is defined by

$$\mathsf{rc}^{I^\circ}_{\mathcal{P}^\circ}(m) \triangleq \{\mathsf{dh}_{\mathcal{P}^\circ}(c) \mid c \in I^\circ \text{ and } \mathsf{size}^\circ(c) \leqslant m\} \;.$$

Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ be a compatible set of initial configurations. Assume that $\mathcal{P}^\circ$ is obtained from $\mathcal{P}$ by the rules of Figure 4.3. As an immediate consequence

of Theorem 4.14 we obtain that $\mathsf{dh}_{\mathcal{P}}(c) \leqslant \mathsf{dh}_{\mathcal{P}^\circ}((c)^\circ)$ for all initial configurations $c \in I$. To relate the runtime complexity of $\mathcal{P}$ and $\mathcal{P}^\circ$ we have to inspect the set of input elements induced by the set of input configurations I and the maximal input size $m \in \mathbb{N}$.

We fix the initial elements of the path-length domain with $I^\circ = \{(c)^\circ \mid c \in I\}$. Let $c \in I$ be an initial configuration and $x \in \mathsf{Var}$. Then, it is easy to see that $\mathsf{plength}(x) \leqslant \mathsf{size}(c)$ whenever x is acyclic. However, if the heap allocated data in c overlaps, the abstraction may duplicate the path-length up to a constant factor. The constant only depends on the number of variables of the store k. Suppose that all variables in the configuration c are acyclic. Then, $\mathsf{size}(c) \leqslant m$ implies $\mathsf{size}^\circ((c)^\circ) \leqslant k \cdot m$.

Theorem 4.17 (Complexity Reflection of PLength)**.** *Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ denote a compatible set of initial configurations. Furthermore, assume that I is restricted to acyclic data. Suppose $\mathcal{P}^\circ$ is obtained from $\mathcal{P}$ by path-length abstraction and $I^\circ = \{(c)^\circ \mid c \in I\}$. Then,*

$$\mathsf{rc}^{I}_{\mathcal{P}}(m) \preccurlyeq \mathsf{rc}^{I^\circ}_{\mathcal{P}^\circ}(k \cdot m) \ .$$

Example 4.18 (Cont'd from Example 4.13)**.** The worst-case runtime complexity of the program obtained by the path-length abstraction is exponential in the input size. The bound becomes easy to see after simplifying the body of the loop. Consider the following snippet that is obtained by composing transitions.

$$\begin{aligned}&\langle l_1, l_2, st' = st \wedge t' \geqslant 0 \wedge t' < st\rangle\\ &\langle l_2, l_1, t' = t \quad \wedge 0 \leqslant l' < t \wedge 0 \leqslant r' < t \wedge 0 \leqslant st' < 1 + 1 + l' + r'\rangle\end{aligned}$$

Here, t denotes the path-length of the tree that is obtained from top of the stack in each iteration, and l and r denote the children of t that are pushed on top of the stack again. However, the maximal path-length of l and r is one less than t, thus (almost) duplicating the size of the stack.

The obtained result conforms to our expected result from the initial discussion of Example 4.1. In particular, the path-length abstraction is not able to distinguish between acyclic and tree-shaped input.

We remark that the path-length abstraction is not the only possible size abstraction, rather the *choice of abstraction* in Julia and COSTA. There is a trade-off between precision and efficiency, and more refined size abstractions rely on more refined heap shape analysis. We discuss some further options in Section 4.6.

4.4.2 Term Abstraction

In this section we present a term abstraction from GotoR programs to *constraint term rewrite systems* (cTRSs for short). Term rewriting forms a Turing complete abstract model of computation, which underlies much of declarative programming (cf. Baader and Nipkow [22]). Complexity analysis of (standard) TRSs has received significant attention in the last decade, for details we refer to Moser [118]. In our setup we consider TRSs as an abstract program representation that is amenable to automated complexity analysis.

The main motivation for a term based abstraction is the representation of composed data structures. Terms allow to construct and deconstruct data easily. Further, record instances $T(t_1, \ldots, t_k)$ can be conceived as terms. However, in GotoR data is modified via operations on the heap, which are conceptually close to manipulating a labelled graph with pointers, while term rewriting is close to first-order functional programming. In particular, term abstractions have to take sharing of data and side effects into account. Moser and Schaper [119] provide the details of a term abstraction for an object-oriented bytecode language. In what follows, we present a conceptually similar but simplified approach. We provide additional insights on [119] in Section 4.5.

In the following we are only interested in cTRSs over a specific theory T, namely (first-order) *integer arithmetic*. To represent instructions on arithmetic expressions we collect the following connectives in $\mathcal{C}$: $\wedge$, $\vee$, $\neg$ together with the following relations and operations: $=$, $\neq$, $\geqslant$, $+$, $-$, $*$. Furthermore, we add infinitely many constants to represent integers. We often write $l \to r$ instead of $l \to r \,[\![t]\!]$, if t holds trivially. As expected, T makes use of the sort int.

Let $\mathcal{P}$ be a GotoR program. We suppose that all variables $x \in \mathsf{Var}$ and $fld \in \mathsf{Field}$ are present in the set of variables $\mathcal{V}$. Program variables and field identifiers with type Int are assigned sort int and all other elements are assigned sort univ. Alternatively, we could have introduced additional sorts for types in RName. However, this complicates matters without gaining additional benefits. The remaining elements of the signature $\mathcal{F}$ will be defined in the course of this section. As the signature of these function symbols is easily read off from the translation given below, the sort information is left implicit for the rest of this chapter.

The main idea of the term domain is to represent configurations (l, σ, μ) as (ground) terms $\mathsf{f}_l(t_1, \ldots, t_n)$ over $\mathcal{T}(\mathcal{F}, \varnothing)$. Here f is an arbitrary chosen function symbol that serves as compound symbol for the term representation of the store and the heap. The subterms $t_1, \ldots, t_n$ indicate the data that is referenced by the variables of the store. In doing so, we assume a fixed order on the sequence of program variables, that is, $\mathsf{Var} = x_1, \ldots, x_n$.

To represent custom data we require constructor symbols for $\mathtt{null} \in \mathcal{F}$ and $\mathsf{RName} \subseteq \mathcal{F}$. The set of function symbols $\mathcal{F}$ typically includes f_l for all program locations l. Thus, t_i is either an integer, `null` or data that is allocated on the heap. Terms for custom data structures are obtained by *unfolding* the mapping of the heap to terms. A record instance of type T is represented by $T(t_1, \ldots, t_k)$, where T is a constructor symbol and $t_1, \ldots, t_k$ indicate the data that is referenced by the field selectors. To obtain a finite representation of cyclic data we introduce a dedicated constructor symbol $\top \in \mathcal{F}$. We indicate an erroneous program state with $\bot \in \mathcal{F}$.

We present cTRSs with rules $\mathsf{f}_l(s_1, \ldots, s_n) \to \mathsf{f}_{l'}(t_1, \ldots, t_n)$ and $\mathsf{f}_l(s_1, \ldots, s_n) \to \bot$, in which instances of the left-hand side represent configurations at program location l and instances of the right-hand side represent configurations at the succeeding program location l' or an erroneous state.

In what follows next, we provide the definition of *unfolding* and present the term domain Term.

Definition 4.19 (Unfolding)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration. The *unfolding* of a value $v \in \mathsf{Val}$ is defined as

$$\mathsf{unfold}(v) \triangleq \begin{cases} T(\mathsf{unfold}(\mu(v).fld_1), \ldots, \mathsf{unfold}(\mu(v).fld_k)) & \text{if } v \in \mathsf{Addr} \text{ and } v\colon T\ , \\ v, & \text{otherwise}\ . \end{cases}$$

Let $x \in \mathsf{Var}$, then

$$\mathsf{unfold}(x) \triangleq \begin{cases} \top & \text{if } x \text{ is cyclic}\ , \\ \mathsf{unfold}(\sigma(x)) & \text{otherwise}\ . \end{cases}$$

To represent a set of configurations we use terms $\mathsf{f}_l(t_1, \ldots, t_n)$ over $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

Definition 4.20 (Term Domain)**.** We denote the *term domain* with $\mathsf{Term} \triangleq \mathcal{T}(\mathcal{F}, \mathcal{V})$. The *representation* function $(\cdot)^\bullet\colon \mathsf{Conf} \to \mathsf{Term}$ along with the *concretisation* function $\gamma^\bullet\colon \mathsf{Term} \to \mathcal{P}(\mathsf{Conf})$ are defined by:

$$(c)^\bullet \triangleq \mathsf{f}_l(\mathsf{unfold}(x_1), \ldots, \mathsf{unfold}(x_n))$$
$$\gamma^\bullet(t) \triangleq \{c \in \mathsf{Conf} \mid \text{ there exists a substitution } \delta \text{ st. } t\delta = (c)^\bullet\}$$

Next, we present the transformation from programs to cTRSs. We remark that the cTRSs that are obtained from `Goto`R do not have nested defined symbols, moreover, for programs in which all variables of the store are of type integer we obtain CTSs in rule based notation. In contrast to standard term rewriting, fresh variables, which are indicated below with prime, are allowed on the right-hand side. However, the instantiation of fresh variables is restricted to ground normal forms (see Section 4.2). Fresh variables indicate unbounded non-determinism and are used when the exact term representation in the succeeding configuration is not known.

Figure 4.4 illustrates the *term abstraction* of `Goto`R programs to constraint term rewrite systems. The transformation follows by case distinction on program instructions at program location l and maps a single instruction to one or multiple rewrite rules. If necessary, we provide subcases that depend on the properties of acyclicity and sharing domain. For brevity, we represent rewrite rules by $(l, F) \to (l', F')$ possible equipped with a constraint $[\![\phi]\!]$. We write (l, F) to indicate the term $\mathsf{f}_l(x_1, \ldots, x_n)$ with $\mathsf{Var} = x_1, \ldots, x_n$, and $(l, F[t]_x)$ to indicate the term that is obtained by replacing x with t in F.

In the case of the assignment $x = y$ the variable x is substituted by y for the succeeding program location. This rule is independent of x being of type integer or some record type. Let a denote an arithmetic expression. In the case of the assignment $y = a$ we substitute y with a fresh variable y' which is constraint to be equal to the expression a. Next, we consider the allocation of a new record instance. The case $x = \texttt{new}\ T(y_1, \ldots, y_k)$ substitutes x with the term representation $T(y_1, \ldots, y_k)$ of a record instance. By construction the variables $y_1, \ldots, y_k$ occur on the left-hand side of the rule.

Whenever a variable x with type $T \in \mathsf{RName}$ is referenced, a case distinction on the possible content of x is performed. More specifically, we inspect the term representation of configurations at some program location l. The term obtained for some variable

case: $x = y$	(l, F)	$\to (l+1, F[y]_x)$
case: $x = a$	(l, F)	$\to (l+1, F[x']_x)$ $[\![x' = a]\!]$
case: $x = \texttt{null}$	(l, F)	$\to (l+1, F[\texttt{null}]_x)$
case: $x = \texttt{new}\ T(y_1, \ldots, y_k)$	(l, F)	$\to (l+1, F[T(y_1, \ldots, y_k)]_x)$
case: $x = y.fld_i$		
$y \in \mathsf{AC}_l$	$(l, F[T(fld_1, \ldots, fld_k)]_y)$	$\to (l+1, F[fld_i]_x)$
	$(l, F[\texttt{null}]_y)$	$\to \bot$
$y \notin \mathsf{AC}_l$	(l, F)	$\to (l+1, F[x']_x)$
case: $x.fld_i = y$		
$x, y \in \mathsf{AC}_l$ and $y \notin \mathsf{SH}_l^x$	$(l, F[T(fld_1, \ldots, fld_k)]_v)$	$\to (l+1, F^{\mathsf{SH}_l^x \setminus \{x\}}[T[y]_{fld_i}]_x)$
	$(l, F[\texttt{null}]_v)$	$\to \bot$
otherwise	(l, F)	$\to (l+1, F^{\mathsf{SH}_l^x})$
case: $\texttt{if}\ a_1 \geqslant a_2\ \texttt{goto}\ l'$	(l, F)	$\to (l', F)$ $[\![a_1 \geqslant a_2]\!]$
	(l, F)	$\to (l+1, F)$ $[\![\neg(a_1 \geqslant a_2)]\!]$
case: $\texttt{ifnull}\ v\ \texttt{goto}\ l'$		
$x \in \mathsf{AC}_l$	$(l, F[\texttt{null}]_v)$	$\to (l', F)$
	$(l, F[T(fld_1, \ldots, fld_k)]_v)$	$\to (l+1, F)$
$x \notin \mathsf{AC}_l$	(l, F)	$\to (l', F)$
	(l, F)	$\to (l+1, F)$

Figure 4.4: Term Abstraction of Goto$R$ Programs.

x with type $T \in \mathsf{RName}$ is defined by $\mathsf{unfold}(x)$, which is either (i) $\texttt{null}$, (ii) a term $T(fld_1, \ldots, fld_k)$ (for some subterms fld_i with $1 \leqslant i \leqslant k$), or (iii) $\top$ if x is cyclic at program location l. Thus, we perform pattern matching on the individual cases at program location l.

Consider the case $x = y.fld_i$. If y is acyclic, i.e. $y \in \mathsf{AC}_l$, then we match on $\texttt{null}$ and $T(fld_1, \ldots, fld_k)$. The subterms fld_i are distinct variables in $\mathcal{V}$ not occurring in Var. In the case that y equals $\texttt{null}$ we rewrite to an erroneous state, which is indicated with $\bot$. Otherwise, we substitute x with fld_i. If we cannot infer that y is definitely acyclic, i.e. $y \notin \mathsf{AC}_l$, we match on all terms including $\top$. For the latter case the information on $y.fld_i$ is imprecise, and we substitute x with a fresh variable x'. In fact, we could ignore the rule in which we match $\texttt{null}$. Then the reduction would get stuck, like in Goto$R$. However, we have chosen to match on all possible data representations.

Next, we consider the statement $x.fld_i = y$. When updating the content of a record instance we have to accommodate for possible side effects that we cannot capture precisely in the abstraction. With SH_l^x we indicate all variables that share with x at program location l. We write F^{SH}, for the term that is obtained from F by substituting all

variables $z \in SH$ with a distinct fresh variable z'. There are two subcases to consider. First, we consider the case in which the variables x and y are acyclic and do not share before the assignment. Then, we match the term representation of the record instance referenced by x. Most relevant, in this case x is also acyclic after the assignment. We reflect the field update via substituting fld_i with y. For all variables distinct from x that may-share with it fresh variables are introduced. Second, we consider the case in which x or y are maybe-cyclic or x and y share before the assignment. Then, we assume the worst-case, that is, that the data referenced by all variables that share with x including x are modified. However, the abstraction cannot express the succeeding configurations precisely, thus, fresh variables are introduced for all variables that share with x.

The case `if` $a_1 \geqslant a_2$ `goto` l' is straightforward. Consider the case that the instruction at program location l is `ifnull` v `goto` l'. If x is acyclic we match on x. After unrolling, we can evaluate the condition. If x is maybe-cyclic, the representation does not have enough information to evaluate the condition. We non-deterministically select the succeeding program location.

Example 4.21 (Term Abstraction of Binary Tree Traversal)**.** The following cTRS is obtained from Example 4.2 via the rules of Figure 4.4. For the abstraction we assume that the input is acyclic.

$$
\begin{array}{l}
(\mathsf{VAR}\ t\ \ t1\ \ st\ \ tmp\ \ l\ \ r) \\
(\mathsf{RULES} \\
\quad \mathsf{f}_0(t, st, tmp) \to \mathsf{f}_1(t, \mathsf{null}, tmp) \\
\quad \mathsf{f}_1(t, st, tmp) \to \mathsf{f}_2(t, \mathsf{Stack}(t, st), tmp) \\
\quad \mathsf{f}_2(t, \mathsf{null}, tmp) \to \mathsf{f}_D(t, \mathsf{null}, tmp) \\
\quad \mathsf{f}_2(t, \mathsf{Stack}(t, st), tmp) \to \mathsf{f}_3(t, \mathsf{Stack}(t, st), tmp) \\
\quad \mathsf{f}_3(t, \mathsf{null}, tmp) \to \bot \\
\quad \mathsf{f}_3(t, \mathsf{Stack}(t1, st), tmp) \to \mathsf{f}_4(t1, \mathsf{Stack}(t1, st), tmp) \\
\quad \mathsf{f}_4(t, \mathsf{null}, tmp) \to \bot \\
\quad \mathsf{f}_4(t, \mathsf{Stack}(t1, st), tmp) \to \mathsf{f}_5(t, st, tmp) \\
\quad \mathsf{f}_5(\mathsf{null}, st, tmp) \to \mathsf{f}_2(\mathsf{null}, st, tmp) \\
\quad \mathsf{f}_5(\mathsf{Tree}(v, l, r), st, tmp) \to \mathsf{f}_6(\mathsf{Tree}(v, l, r), st, tmp) \\
\quad \mathsf{f}_5(\mathsf{Tree}(v, l, r), st, tmp) \to \mathsf{f}_6(\mathsf{Tree}(v, l, r), st, tmp) \\
\quad \mathsf{f}_6(\mathsf{null}, st, tmp) \to \bot \\
\quad \mathsf{f}_6(\mathsf{Tree}(v, l, r), st, tmp) \to \mathsf{f}_7(\mathsf{Tree}(v, l, r), st, r) \\
\quad \mathsf{f}_7(t, st, tmp) \to \mathsf{f}_8(t, \mathsf{Stack}(tmp, st), tmp) \\
\quad \mathsf{f}_8(\mathsf{null}, st, tmp) \to \bot \\
\quad \mathsf{f}_8(\mathsf{Tree}(v, l, r), st, tmp) \to \mathsf{f}_9(\mathsf{Tree}(v, l, r), st, l) \\
\quad \mathsf{f}_9(t, st, tmp) \to \mathsf{f}_A(t, \mathsf{Stack}(tmp, st), tmp) \\
\quad \mathsf{f}_A(t, st, tmp) \to \mathsf{f}_2(t, st, tmp) \\
)
\end{array}
$$

We have the following soundness result (compare with Moser and Schaper [119]).

Theorem 4.22 (Soundness of Term Abstraction)**.** *Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ be a compatible set of initial configurations. Moreover, let $\mathcal{P}^\bullet$ be the term abstraction of $\mathcal{P}$. Suppose that there exists a trace $c \to_{\mathcal{P}}^n d$ in $\mathcal{P}$ starting from $c \in I$. Then, there exists a trace $(c)^\bullet \to_{\mathcal{P}^\bullet}^n (d)^\bullet$ in $\mathcal{P}^\bullet$.*

We adapt the runtime complexity with respect to standard TRSs suitable for cTRSs (see Hirokawa and Moser [92] for the standard definition).

Definition 4.23 (Input Size of Term)**.** The *input size* of $t \in \mathsf{Term}$ is defined by

$$\mathsf{size}^\bullet(t) \triangleq \begin{cases} 1 & \text{if } t \text{ is a variable ,} \\ \mathsf{abs}(t) & \text{if } t \text{ is an integer ,} \\ 1 + \sum_{i=1}^{n} \mathsf{size}^\bullet(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \text{ is not an integer .} \end{cases}$$

Definition 4.24 (Runtime Complexity of Term)**.** Let $\mathcal{P}^\bullet$ be a cTRS and $I^\bullet \subseteq \mathsf{Term}$ denote a set of initial terms. Furthermore, let $m \in \mathbb{N}$ denote the maximal input size. The *worst-case runtime complexity* $\mathsf{rc}^{I^\bullet}_{\mathcal{P}^\bullet} \colon \mathbb{N} \to \mathbb{N}^\infty$ of $\mathcal{P}^\bullet$ on $I^\bullet$ is defined by

$$\mathsf{rc}^{I^\bullet}_{\mathcal{P}^\bullet}(m) \triangleq \{\mathsf{dh}_{\mathcal{P}^\bullet}(c) \mid c \in I^\bullet \text{ and } \mathsf{size}^\bullet(c) \leqslant m\} \ .$$

Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ be a compatible set of initial configurations. Suppose that $\mathcal{P}^\bullet$ is obtained from $\mathcal{P}$ by the rules of Figure 4.3. Like before, to relate the runtime of $\mathcal{P}$ and $\mathcal{P}^\bullet$ we inspect the input elements with respect to the initial elements and maximal input size $m \in \mathbb{N}$. We fix the set of initial terms with $I^\bullet = \{(c)^\bullet \mid c \in I\}$. It is easy to see that the unfold operation may cause an exponential blow-up in size. Consider for instance a fully-shared binary tree with height (or input size) m. The input size of the term representation is exponential in m. Therefore, we restrict the input to *tree-shaped* data.

Definition 4.25 (Tree-Shaped)**.** Let $(l, \sigma, \mu) \in \mathsf{Conf}$ be a configuration. A value $v \in \mathsf{Val}$ is called *tree-shaped* at program location l, if for all $v_1, v_2 \in \mathsf{addresses}^1(v)$ with $v_1 \neq v_2$, $\mathsf{addresses}(v_1) \cap \mathsf{addresses}(v_2) = \varnothing$. Similarly, a variable $x \in \mathsf{Var}$ is called *tree-shaped* if $\sigma(x)$ is tree-shaped.

Akin to the path-length abstraction, the proposed term abstraction may duplicate data if it is shared in the input configuration. Suppose all variables in configuration $c \in I$ are tree-shaped. Then, $\mathsf{size}(c) \leqslant m$ implies $\mathsf{size}^\bullet((c)^\bullet) \leqslant k \cdot m$.

Theorem 4.26 (Complexity Reflection of Term)**.** *Let $\mathcal{P}$ be a program and $I \subseteq \mathsf{Conf}$ be a compatible set of initial configurations. Furthermore, assume that I is restricted to tree-shaped data. Suppose $\mathcal{P}^\bullet$ is obtained from $\mathcal{P}$ by term abstraction and $I^\bullet = \{(c)^\bullet \mid c \in I\}$. Then,*

$$\mathsf{rc}^{I}_{\mathcal{P}} \preccurlyeq \mathsf{rc}^{I^\bullet}_{\mathcal{P}^\bullet}(k \cdot m) \ .$$

Example 4.27 (Cont'd from Example 4.21)**.** The complexity analysis tool T$\subset$T (Avanzini, Moser, and Schaper [18]) infers a linear bound on the term abstraction of tree traversal. Due to the complexity of the proof, it is omitted here. In Example 4.29 we depict a simplified program that is obtained automatically from applying the abstraction in Moser and Schaper [119] together with a proof for the linear runtime bound based on polynomial interpretations.

We emphasize that the term abstraction does not require tree-shaped data for the abstraction itself, but only to prevent a blow-up in the input size when relating the runtime of the program representations. In contrast to the path-length abstraction, the approximation of side effects of the term abstraction is more coarse. While the path-length of all affected variables increases at most by the path-length of y for a field update $x.field = y$, there is no bounded term representation for the affected variables.

4.5 Term Abstraction of Object-Oriented Bytecode Programs

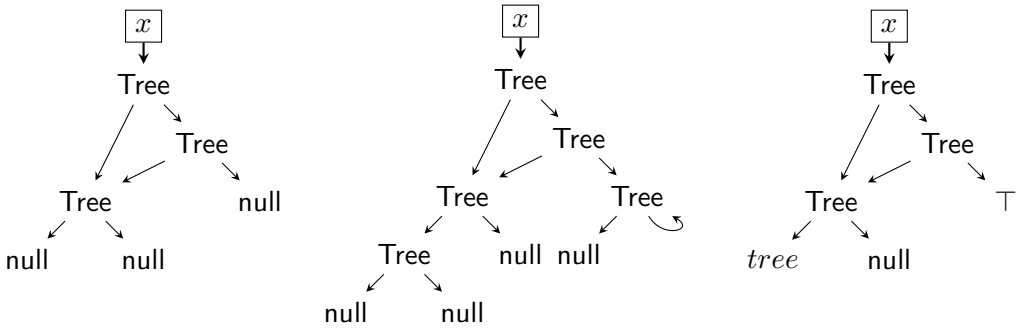
In Moser and Schaper [119] we have worked out the details of the term abstraction for the Jinja programming language. Jinja (Jinja Is Not JAva) is a Java like language that exhibits its core features. The language has a formal semantics which is formalised and machine checked in the theorem prover Isabelle/HOL [107].

The proposed transformation in [119] is analogues to the transformation presented above. However, it does make use of a more detailed intermediate abstract representation to generate *shape invariants* on heap allocated data. This often allows for a more precise term representation than the type based pattern matching used above. Our work was initially inspired by the *non-termination preserving* abstraction of Java bytecode programs to (integer) term rewrite systems presented by Otto et al. [129] (and follow-up work). Our approach differs in the intermediate abstraction used and provides the necessary details for runtime analysis.

We comment on the intermediate abstraction. The abstract domain is based on key concepts in *(acyclic) term graph rewriting* (cf. Avanzini and Moser [15]). We make use of a graph based representation of states (or configurations) and abstract states, dubbed *stategraphs*. The nodes for abstract stategraphs may be labelled with variables. We generalise the *matching* definition for acyclic term graphs, which is based on *(rooted) graph morphsims*, by a subtyping mechanism, and provide an abstraction and concretisation function much like Definition 4.20. To make the approach more viable we make use of existing heap shape properties, in particular we rely on *acyclicity* (Rossignoli and Spoto [138]), *sharing* (Secci and Spoto [142]) and *reachability* (Genaim and Zanardini [78]). Stategraphs can be *unfolded* to a term representation.

Example 4.28 (Graph Based Abstraction)**.** Consider the illustrations in Figure 4.5. Figures 4.5a and 4.5b depict stategraphs A and B with a single variable x that references a tree structure allocated on the heap. Figure 4.5c shows an abstract state graph. It is easy to see that A and B are concretisations of C. Consider the substitution of variable *tree* to either null or an instance of Tree (recall that $\top$ represents all arbitrary well-formed instances). Figure 4.5d provides the term representation obtained by unfolding the abstract stategraph C.

The abstract representation of the program is obtained by *symbolic evaluation*. Informally this corresponds to the abstract semantics presented in Figure 4.4 but specialised for stategraphs. To obtain a finite abstraction we equip the abstract domain with a *join* operator that provides an upper bound on two stategraphs (at the same program



(a) Stategraph A.

(b) Stategraph B.

(c) Abstraction C.

$$\mathsf{f}(\mathsf{Tree}\ (\mathsf{Tree}(\mathit{tree}, \mathsf{null})),\ (\mathsf{Tree}(\mathsf{Tree}(\mathit{tree}, \mathsf{null})), \top))$$

(d) Term Abstraction.

Figure 4.5: Stategraph Abstraction.

location). Via a standard *fixed-point* construction using symbolic evaluation and repeated application of join we obtain finitely many abstract stategraphs that overapproximate the reachable set of states. We have worked out the details of the abstraction within the abstract interpretation framework (Cousot and Cousot [59]). From the abstraction we construct a cTRS by unfolding the obtained abstract stategraphs and taking the instructions at the corresponding program location into account.

The approach has been implemented in the tool jat (*Jinja Analysation Tool*), available online at

`http://cbr.uibk.ac.at/tools/jat/` ,

and integrated within the complexity analysis framework T$_\complement$T. We present the framework in Chapter 6. The implementation supports *class inheritance* and *dynamic dispatch*. Our fixed-point computation of the abstraction is restricted to *non-recursive* method calls. The integer type is considered to be unbounded. Floating point arithmetic and arrays are not supported, as well as exceptional control flow. Further, we apply various program simplifications such as *inlining* and *slicing*.

Example 4.29 (Term Abstraction of Binary Tree Traversal)**.** The following (standard) TRS is obtained by T$_\complement$T from the Jinja bytecode of the motivating example after removing some redundant arguments. From the given system, we obtain fully-automatically a *strongly linear polynomial interpretation* $\mathcal{I}$ which implies a linear upper bound on the worst-case runtime complexity (see Moser [118] for details).

```
(VAR t l r st)                                          p(::) = x1 + x2
(RULES                                                p(Tree) = 10 + x1 + x2
 f0(t,nil)              → f1(::(t,nil))                 p(f0) = 15 + x1
 f1(::(null,st))        → f1(st)                        p(f1) = 2 + x1
 f1(::(Tree(l,r),st))   → f1(::(l,::(r,st)))            p(f2) = 10
 f1(nil)                → f2(nil)                      p(nil) = 12
)                                                     p(null) = 8
```

The most interesting rule is $\mathsf{f}_1(::(\mathsf{Tree}(l,r),st)) \to \mathsf{f}_1(::(l,::(r,st)))$, which pops the tree on top of the stack and pushes its children on top of the stack again. Applying the polynomial interpretation $\mathcal{I}$ results in the following inequality, which is true for all $l, r, st \in \mathbb{N}$.

$$[\![\mathsf{f}_1(::(\mathsf{Tree}(l,r),st))]\!]_{\mathcal{I}} = 2 + 10 + l + r + st > 2 + l + r + st = [\![\mathsf{f}_1(::(l,::(r,st)))]\!]_{\mathcal{I}}$$

In contrast to the path-length abstraction, the polynomial interpretation represents a tree as a linear combination $\mathsf{Tree}(l,r) = 10 + l + r$. In the above rule we decompose the tree on the left-hand side and reuse the children on the right-hand side without duplicating it. Thus, we obtain the expected linear bound.

4.6 Related Work

In this section we recall and comment on related work.

Term Abstractions

Panitz and Schmidt-Schauß [130] present a term abstraction for the termination analysis of a non-strict higher-order functional language.

Giesl et al. investigate term abstractions for the termination analysis of object-oriented bytecode programs [46–49, 129], higher-order functional programs [79], and logic programs [80]. For an overview we refer to [81, 82]. The approaches have been implemented in the AProVe[1] verifier.

Falke and Kapur [67], Falke et al. [68] present a complexity reflecting transformation from LLVM intermediate representation to `int`-based term rewrite systems for termination analysis. The transformation is conceptually similar to the abstraction presented here, though pointer access is abstracted by unconstrained assignments. The approach has been implemented in KITTeL[2].

A complexity reflecting transformation for higher-order functional programs to term rewrite systems is presented by Avanzini et al. [16]. The approach is implemented in the tool HoCA[3] and integrated in the T$_C$T framework.

[1] `http://aprove.informatik.rwth-aachen.de/`
[2] `https://github.com/s-falke/kittel-koat/`
[3] `http://cbr.uibk.ac.at/tools/hoca/`

Complexity Analysis of Java Programs with AProVe

Based on earlier work on termination analysis, Frohn and Giesl [76] present a *numeric* abstraction to constraint transition systems for the complexity analysis of Java bytecode programs in AProVe. Informally, the size of an object corresponds to the number of references together with the sum of all absolute values of integer fields that are reachable from the object.

The approach abstracts programs to *weighted CTSs*, which are CTSs where edges are additionally labelled with weights that indicate the cost of executing the transition. This provides a flexible analysis in terms of *cost models*, and a modular analysis in terms of *method summaries* that indicate the cost of executing a method. The approach uses complexity analysis tools such as CoFloCo[4] [73, 74] and KoAT[5] [51] as back-end.

In contrast to the approaches presented in the previous sections, constraints are formed over *symbolic heap locations* rather than the variables of the store. Incorporating heap invariants (cf. Brockschmidt et al. [46] and related work) allow more refined constraints, which inspect the fields of the objects separately, at the cost of additional variables.

The used notion of size is close to our definition of input size. One may would expect a linear bound for the tree traversal example. However, while the notion of size is precise the abstract semantics for *field access* is not. In particular, it is analogues to the path-length abstraction (cf. Frohn and Giesl [76]). Therefore, the runtime is exponential for the tree traversal example.

Cost and Termination Analyser (COSTA)

The COSTA[6] (COSt and Termination Analyser for Java bytecode) tool is developed by Albert et al. and provides automatic cost and termination analysis of Java bytecode programs [1, 4]. The tool provides a generic way to apply different cost models by associating instructions with cost expressions and often returns precise bounds.

For the cost analysis of Java bytecode programs, objects are abstracted to their *maximal path-length*, and programs are abstracted to *cost relation systems*, which provide a language independent abstract representation for cost analysis (Albert et al. [2]). A cost relation system consists of (multiple) cost relations of the following form

$$\langle C(\vec{x}) = e + \textstyle\sum_{i=1}^{m} D_i(\vec{y_i}) + \sum_{j=1}^{n} C(\vec{z_j}), \phi \rangle \ .$$

Informally, the cost of the call C with input $\vec{x}$ corresponds to the cost e plus the cost of non self-recursive calls $D_i(\vec{y_i})$ plus the cost of self-recursive calls $C(z_j)$. The constraint ϕ relates input and output of the arguments $\vec{x}$, $\vec{y_i}$, and $\vec{z_j}$. Closed-form representations of upper bounds on cost relations can be obtained by dedicated solvers, such as PUBS[7] [3, 5] or CoFloCo [73, 74].

We were not able to infer a bound from the motivating example using the available tools. However, an exponential bound for the recursive version of tree traversal is obtained.

[4] `https://github.com/aeflores/CoFloCo/`
[5] `https://github.com/s-falke/kittel-koat/`
[6] `http://costa.ls.fi.upm.es/web/`
[7] `https://costa.fdi.ucm.es/pubs/`

Symbolic Bound Analysis in Speed

Gulwani et al. [84, 87] present SPEED, which is a tool for the inference of symbolic complexity bounds for C/C++ programs that support conditional loops as well as iterations over user-defined data structures. The approach is based on *counter instrumentation*, that is, the program code is instrumented with (multiple) counter variables that represent the resource consumption. Upper bounds on the counter variables are obtained by numeric invariant analysis. See Section 3.3.6 on page 25 for further details.

To support custom data structures the approach makes use of an abstract domain that combines standard numeric domains and uninterpreted function symbols (Gulwani and Tiwari [85]). Data structures are associated with *quantitative functions* that represent numerical properties and *abstract operations* that are specified by constraints over the combined abstract domain. For instance, consider the quantitative functions for a singly-linked list

$$\mathsf{Len}(L) \triangleq \text{ length of list } L, \text{ and } \mathsf{Pos}(e, L) \triangleq \text{ position of element } e \text{ in list } L \ ,$$

together with the abstract semantics for assigning the next list segment

$$e = L.\,\mathsf{next}(f) \triangleq \mathsf{Pos}(e, L) = \mathsf{Pos}(f, L) + 1;\ \mathsf{Assume}(0 \leqslant \mathsf{Pos}(f, L) < \mathsf{Len}(L)) \ .$$

SPEED provides the bound $\mathsf{Len}(L) - \mathsf{Pos}(f, L)$ for $\texttt{for}(e = f; e \neq \texttt{null}; e = L.\,\mathsf{next}(e))$.

The proposed approach often provides intuitive and precise bounds. However, the method is not fully automatic and the constraints for the abstract semantics are complex for more sophisticated data structures or when side effects have to be considered.

Gulwani et al. [87] exemplifies the approach on a recursive version of tree traversal and provides a linear bound in terms of the number of nodes. The example, however, requires non-trivial loop invariants that have to be provided by the user.

Resource Static Analysis (RESA)

Atkey [10] presents an approach for the *amortised* resource analysis of imperative languages by embedding resource information within *separation logic* (Reynolds [135]). The approach relies on explicitly defined inductive predicates that represent the shape of the objects. For instance, the following *resource-aware* inductive predicate represents a list segment where resources R are associated to each element:

$$\mathsf{lseg}(R, x, y) \triangleq (x = y \wedge \mathsf{emp}) \vee \exists z, z'.[x \overset{\mathsf{data}}{\longmapsto} z] * [x \overset{\mathsf{next}}{\longmapsto} z'] * R * \mathsf{lseg}(R, z', y) \ .$$

Fenacci and MacKenzie [69] provide an implementation of this approach for the resource analysis of Java bytecode programs. The implementation is *semi-automatic* and requires user annotated postconditions and loop invariants. Furthermore, the proof search is customised for singly-linked lists and trees only. A linear bound in the number of calls is obtained for the recursive version of binary tree traversal using the predicate:

$$\mathsf{tree}(R, x) \triangleq (x = \mathit{null} \wedge \mathsf{emp}) \vee \exists y, z.[x \overset{\mathsf{left}}{\longmapsto} y] * [x \overset{\mathsf{right}}{\longmapsto} z] * R * \mathsf{tree}(y) * \mathsf{tree}(z) \ .$$

The motivating *Frying Pan* example in [10] is beyond the scope of our analysis, due to the abstraction of cyclic data.

Shape Norms

Fiedor et al. [70] investigate *numeric* abstractions for heap allocated data structures based on *shape norms*. A shape norm represents the maximal length of a path between two symbolic heap locations of interest. For instance, $x\langle next^* \rangle y$ represents the length of the path of a singly-linked list between the two heap locations referenced by x and y, and $x\langle (left + right)^* \rangle \texttt{null}$ represents the height of a binary tree. The approach makes use of *must-alias* and *may-alias* properties on heap locations. It has been realised in the tool Ranger[8], which makes use of the shape analyser Forester[9] [88] and the loop bound analyser Loopus[10] [145, 146, 158].

By inspecting the length of a path between two heap locations the approach also supports *cyclic* data structures. Consider for example, a program where variables x and y point to a cyclic single-linked list. Informally, for $l{:}\texttt{while}(x \mathrel{!}= y)\{\ x = x.next\ \}$ and norm $n = x\langle next^* \rangle y$ we obtain the numeric abstraction $\langle l, l, n' \geqslant 0 \wedge n' < n \rangle$.

Shape norms inspect the path-length between two symbolic heap locations. The approach conceptually generalises the path-length abstraction presented in this chapter. However, the number of nodes within a binary tree cannot be captured precisely with this abstraction.

The experiments in Fiedor et al. [70] demonstrate the viability of this approach for amortised complexity analysis providing precise upper bounds on some motivating examples of Atkey [10] fully automatically.

Type-Based Approach

Hainry and Péchoux [89, 90] provide a type-based approach to the complexity analysis of object-oriented programs. The type system is inspired by earlier works on *implicit computational complexity* and makes use of *safe recursion on notation* (Bellantoni and Cook [26]) and *non-interference* (Marion [114], Volpano et al. [152]) to control resource usage. The main idea is that variables that point to addresses in the initial heap are considered *safe* (have tier 1) and variables that point to fresh allocated addresses are considered *unsafe* (have tier 0). The typing system prohibits that information flows from tier 0 variables to tier 1 variables, and guarantees that arguments that control recursion and loop iteration are of tier 1.

Further, the system is decidable in polynomial time and provides a sound and complete characterisation of polynomial time computable functions on the object-oriented programming paradigm. For terminating and safe programs (a safe program is well-typed and conforms to a restricted notion of recursion) an upper bound of the form $\mathcal{O}(n^{k \times d})$ can be inferred. Here n captures the number of nodes in the heap together with the sum of all numbers of the initial configuration, k is the number of tier 1 variables of the initial configuration, and d depends on the maximal nesting depth on loops and recursive calls (which are transformed into loops).

[8] `http://www.fit.vutbr.cz/research/groups/verifit/tools/ranger/`
[9] `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`
[10] `https://forsyte.at/software/loopus/`

The authors provide additional prerequisites to restrict recursion. In particular, one criterion is akin to the notion of tree-shaped data and restricts the number of recursive calls invoked from distinct fields of an object to one. This allows to show a linear bound on a recursive implementation of tree *clone*, which is conceptually similar to traverse. The non-recursive version of tree traversal, as given above, cannot be typed. During the evaluation the stack variable is modified, and the system asserts that it has to be typed with tier 0. However, the loop iteration cannot be governed by a tier 0 variable.

4.7 Concluding Remarks

In this chapter we have discussed the automated resource analysis of imperative programs with heap allocated data. We have presented two known program abstractions from the literature in a uniform way, and paid special attention to the automated analysis of binary tree traversal. Data structures are often associated with intricate properties which are difficult to reconstruct within a general and automated setting. In the case of the running example we have been interested in measuring the progress, i.e. the number of visited nodes, in terms of the number of allocated nodes that are reachable from the store. Moreover, additional assumptions on the precise shape of the input are necessary to process the motivating example in the intended way. In the presence of sharing, capturing the precise quantitative relationship gets even more challenging.

Related work shows that the example, albeit standard, is non-trivial in a fully automated setting. The presented term abstraction handles the running example as intended, when supplemented with additional heap shape invariants to control sharing. However, the transformational approach, as depicted here is straight-lined, in the sense that the transformation captures very specific properties and informally, a class of problems. While the term abstraction captures construction and deconstruction of data, it suffers from the coarse approximation of side effects. This makes the approach inherently non-modular.

Chapter 5

Imperative Probabilistic Programs

In this chapter we are concerned with automated resource analysis of *probabilistic programs*. Before, we have discussed the analysis of a standard, i.e. non-probabilistic, model of computation. In non-probabilistic programs the runtime for a terminating run is given by the length of the program trace. In case of probabilistic programs, we are interested in the *average runtime*, i.e. the average length of the traces given by all probabilistic branches. This seems to make the analysis of probabilistic programs non-modular at first. Taking inspiration of known approaches to the modular resource analysis of non-probabilistic programs, we investigate under which conditions modularity is obtained again in the probabilistic setting. In what follows, we present a *fully automated and modular analysis* of imperative probabilistic programs.

The presentation is based on Avanzini, Schaper, and Moser [20]. This chapter outlines the central results. Additional (technical) details are given in the report [21]. Section 5.1 provides initial insights on the problem at hand. In Section 5.2 we present our probabilistic model of computation, while in Section 5.3 we introduce the programming language of interest. Then, in Section 5.4 we recall a compositional model for the computation of the expected resource consumption. This model forms the foundation of our analysis. We present the main result in Section 5.5, while Section 5.6 is concerned with its automation. Finally, we conclude this chapter in Section 5.7.

5.1 Introduction

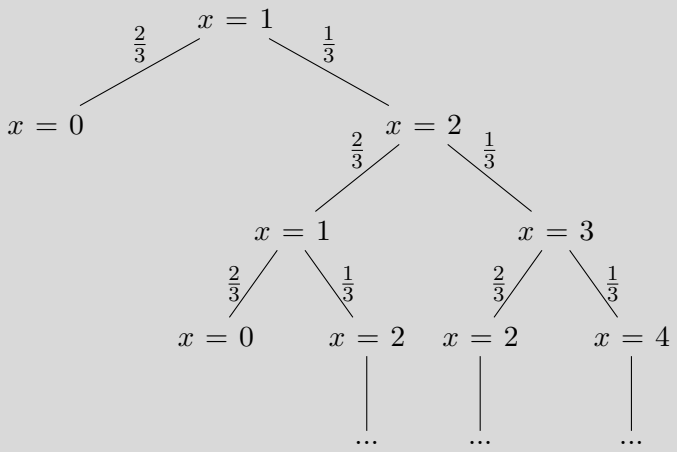
We are concerned with the *average case runtime complexity analysis* of a prototypical *imperative language* pWhile in the spirit of Dijkstra's *guarded command language*. This language is endowed with primitives for *sampling* and *probabilistic choice* so that *randomised algorithms* can be expressed. Complexity analysis in this setting is particularly appealing as *efficiency* is one striking reason why randomised algorithms have been introduced and studied. In many cases, the most efficient algorithm for the problem at hand is randomised (cf. Motwani and Raghavan [120]).

Next, we give some initial insights for reasoning about probabilistic programs.

Example 5.1 (Random Walk)**.** The following program illustrates a *random walk* over $\mathbb{N}$. In each iteration we flip a (biased) coin. With probability $p \in \mathbb{Q}$, in which $0 \leqslant p \leqslant 1$, x is decremented by one and with probability $1 - p$, x is incremented by one. The resource metric taken, via the command `tick`(1) in the loop body, gives the number of loop iterations.

$$\texttt{while}\ (x > 0)\ \{\ \texttt{tick}(1);\ \{x := x - 1\}\ [p]\ \{x := x + 1\}\ \}$$

We illustrate below, the *probability tree* of the first few iterations of the random walk with probability $p = \frac{2}{3}$ starting from the initial assignment $x = 1$.



The analysis of this program is intricate and the properties of interest depend on the probability p. At first, we inspect *termination* of the running example.

In the standard (or non-probabilistic) setting the question whether a program terminates on all inputs is a 'yes' or 'no' property. The existence of a non-terminating or infinite run implies non-termination. We can conceive the probability tree above as a way to represent all possible program traces that start from a given initial state. It is easy to see that for any fixed probability $0 \leqslant p < 1$ we can construct a tree of infinite height. Thus, for probabilistic programs a different notion is required which factors in the probability of the existence of non-terminating runs.

The well accepted analogon for termination of probabilistic programs is *almost sure termination* (AST for short), which states that the probability of eventually reaching a normal form is one, or equivalently, the existence of an infinite trace is zero (cf. Fioriti and Hermanns [71]). The random walk example is almost surely terminating for probability $p = \frac{1}{2}$ and $p > \frac{1}{2}$. AST guarantees that eventually a normal form is reached but does not take the number of steps to reach the normal form into account.

Bournez and Garnier [43] introduce a stronger property, termed *positive almost sure termination* (PAST for short). A program is PAST if it is AST and the *expected time* (or average time) to reach a normal form is finite. The random walk example is PAST for $p > \frac{1}{2}$. PAST states that the expected time to terminate is finite for each run. In the presence of non-determinism, however, PAST does not guarantee that there exists a bound on all possible runs.

Avanzini et al. [19] present an even stronger property, termed *strong almost sure termination* (SAST for short). A program is SAST if it is PAST and the expected time to reach a normal form is *strongly bounded*, i.e. there exists a bound on the expected runtime on all runs. A formal definition is given below in Definition 5.8 of the preliminaries. The notion of PAST and SAST coincide for programs without non-determinism. Thus, the random walk example is also SAST for probability $p > \frac{1}{2}$.

Next, we provide an intuition of *expected runtime*. In program analysis, we investigate properties of program traces. In the non-probabilistic setting we conceive program states as configurations, e.g. the valuation of the program variables. The runtime of an input configuration is then given by the length of its trace. In the deterministic case the normal form and the length of a terminating trace is unique for a given input, while in the non-deterministic case we take the maximal length of all possible traces.

In the probabilistic setting we can conceive program states as *distributions of configurations*, i.e. each configuration is associated with a probability that indicates how likely it is to reach the configuration. Informally, this means that even in the deterministic case multiple normal forms can be reached from a given input and the runtime of reaching the normal forms may vary. The average, or expected runtime of an input configuration is thus given by the expected value $\sum p_b \cdot m_b$ of the runtime m_b reaching a normal form σ_b with probability p_b. Alternatively, we can consider that a probabilistic choice is modelled with *probabilistic branches* as in the example above. The normal form and runtime of each branch may vary.

Our prototype implementation pWhile infers the bound $3 \cdot \mathsf{max}(0, x)$ on the expected number of loop iterations of the random walk with probability $p = \frac{2}{3}$.

Our staring point towards an *automated analysis* is the ert-*calculus* of Kaminski et al. [105], which constitutes a *sound and complete method* for deriving expected runtimes of probabilistic programs. The ert-calculus has been recently automated by Ngo et al. [124], showing encouraging results. Indeed, their prototype Absynth can derive accurate bounds on the expected runtime of a wealth of non-trivial, albeit academic, imperative programs with probabilistic choice.

Since the average case runtime of probabilistic programs seems to be non-modular (see e.g. Kaminski et al. [105]), different program fragments cannot be analysed in general independently within the ert-calculus. This work aims at overcoming this situation, by enriching the calculus with a form of *expected value analysis*. Conceptually, our result rests on the observation that if f and g measure the runtime of non-probabilistic programs `C` and `D` as a function in the variable assignment σ before executing the command, then $f(\sigma) + g(\sigma')$ for σ' the store after the execution of `C` gives the runtime of the composed command `C;D`. Estimating σ' in terms of `C` and σ, and ensuring monotonicity on g, gives rise to a modular analysis (see also the discussion in Section 3.3.4 on page 19). When `C` exhibits probabilistic behaviour though, the command `D` may be executed after `C` on several probabilistic branches b, each with probability p_b with a variable assignment σ_b. Assuming bounding functions f and g on the expected runtime of `C` and `D`, respectively, yields a bound $f(\sigma) + \sum_b p_b \cdot g(\sigma_b)$ on the expected runtime of the probabilistic program `C;D`. As the number of probabilistic branches b is unbounded for all but the most trivial programs `C`, estimating all assignments σ_b in terms of σ soon becomes infeasible. The

crux of our approach towards a modular analysis lies in the observation that if we can give the runtime of D in terms of a *concave function*, i.e. a real-valued function g that satisfies $g(p \cdot x + (1-p) \cdot y) \geqslant p \cdot g(x) + (1-p) \cdot g(y)$ for all $x, y, p \in \mathbb{R}$ and $0 \leqslant p \leqslant 1$, the expected runtime $\sum_b p_b \cdot g(\sigma_b)$ can be bounded in terms of g and the variable assignment $\sum_b p_b \cdot \sigma_b$ expected after executing C. This way, a modular analysis for *sequential programs* is recovered. This observation then also enables a modular analysis of *nested loops*.

To prove this machinery sound, we present a structural operational semantics in terms of *weighted probabilistic abstract reduction systems*. These constitute a refinement to probabilistic abstract reduction systems introduced by Bournez and Garnier [43] in which operations do not necessarily have uniform cost. Notably, probabilistic abstract reduction systems give rise to a reduction relation on (multi-)distributions that is equivalent to the standard operational semantic via stochastic processes (Avanzini et al. [19]). We then *generalise the* ert*-calculus* to one for reasoning about *expected costs* consumed by a command $\texttt{tick}(\cdot)$, and *expected values in final configurations*. This machinery is sound and complete with respect to the operational semantics. Finally, we conclude with additional insights on the *prototype implementation*.

5.2 Preliminaries

In this section we present *weighted probabilistic abstract reduction systems*, which induce a reduction relation on *multi-distributions* with *non-uniform cost*. These systems form our probabilistic model of computation.

Definition 5.2 (Multiset)**.** A *multiset* over a set A is a mapping $M : A \to \mathbb{N}$. The *union* $\biguplus_{i \in I} M_i$ of countably many multisets M_i is defined by $\left(\biguplus_{i \in I} M_i\right)(a) \triangleq \sum_{i \in I} M_i(a)$ which forms a multiset if and only if $\sum_{i \in I} M_i(a)$ is finite for every $a \in A$. The *sum* of a multiset M with respect to $f : A \to \mathbb{R}_{\geqslant 0}$ is defined by $\sum_{a \in M} f(a) \triangleq \sum_{a \in A} M(a) \cdot f(a)$.

We use set-like notations for multisets: $\varnothing$ denotes the *empty* multiset $\varnothing(a) \triangleq 0$, $\{\!\{a_i \mid i \in I\}\!\}$ is the multiset M with $M(a) = |\{i \in I \mid a_i = a\}|$, and $\{\!\{a_1, \ldots, a_n\}\!\}$ is its special case where $I = \{1, \ldots, n\}$ is finite.

Definition 5.3 (Multidistribution)**.** A *multidistribution* on a set A is a multiset μ of pairs of $a \in A$ and $0 < p \leqslant 1$, written $p : a$, satisfying $|\mu| \triangleq \sum_{p:a \in \mu} p \leqslant 1$. The set of multidistributions on A is denoted by $\mathsf{MDist}(A)$. Multidistributions are closed under *convex multiset unions* $\biguplus_{i \in I} p_i \cdot \mu_i \triangleq \sum_{i \in I} p_i \cdot |\mu_i| \leqslant 1$ for every finite or countable infinite index set I and probabilities $p_i > 0$ with $\sum_{i \in I} p_i \leqslant 1$. Here *scalar multiplication* is defined by $p \cdot \{\!\{q_i : a_i \mid i \in I\}\!\} \triangleq \{\!\{p \cdot q_i : a_i \mid i \in I\}\!\}$ for $0 < p \leqslant 1$. The *restriction* of a multidistribution $\mu \in \mathsf{MDist}(A)$ to a set $P \subseteq A$ is defined by $\mu{\restriction}P \triangleq \{\!\{p : a \mid p : a \in \mu, a \in P\}\!\}$.

Definition 5.4 (Expectation)**.** For $\mu \in \mathsf{MDist}(A)$, we define the *expectation* of a function $f \colon A \to \mathbb{R}^{\infty}_{\geqslant 0}$ as

$$\mathbb{E}_{\mu}(f) \triangleq \sum_{p:a \in \mu} p \cdot f(a) \,.$$

Notice that $\mathbb{E}_{\biguplus_{i \in I} p_i \cdot \mu_i}(f) = \sum_{i \in I} p_i \cdot \mathbb{E}_{\mu_i}(f)$.

Definition 5.5 (Weighted Probabilistic Abstract Reduction System)**.** A *weighted probabilistic abstract reduction system* (wPARS for short) is a pair $\mathcal{A} = (A, \to)$ consisting of a set A and a relation $\to \subseteq A \times \mathsf{MDist}(A) \times \mathbb{R}_{\geqslant 0}$. We write $a \xrightarrow{c} \mu$ instead of $(a, \mu, c) \in \to$. The *weighted one-step reduction relation* $\twoheadrightarrow \subseteq \mathsf{MDist}(A) \times \mathsf{MDist}(A) \times \mathbb{R}_{\geqslant 0}$ of $\to$ is defined by:

$$\frac{}{\mu \xrightarrow{0}\!\!\!\!\twoheadrightarrow \mu} \qquad \frac{a \xrightarrow{c} \mu}{\{\!\{1 : a\}\!\} \xrightarrow{c}\!\!\!\!\twoheadrightarrow \mu} \qquad \frac{\forall i \in I.\ \mu_i \xrightarrow{c_i}\!\!\!\!\twoheadrightarrow \nu_i \quad c = \sum_{i \in I} p_i \cdot c_i}{\biguplus_{i \in I} p_i \cdot \mu_i \xrightarrow{c}\!\!\!\!\twoheadrightarrow \biguplus_{i \in I} p_i \cdot \nu_i}$$

The *weighted multi-step reduction relation* $\twoheadrightarrow^* \subseteq \mathsf{MDist}(A) \times \mathsf{MDist}(A) \times \mathbb{R}_{\geqslant 0}$ of $\to$ is defined by:

$$\frac{\mu = \mu_0 \xrightarrow{w_1}\!\!\!\!\twoheadrightarrow \cdots \xrightarrow{w_n}\!\!\!\!\twoheadrightarrow \mu_n = \nu\ .\ w = \sum_{i=1}^{n} w_i}{\mu \xrightarrow{w}\!\!\!\!\twoheadrightarrow^* \nu}$$

Example 5.6 (Cont'd from Example 5.1)**.** Consider the random walk example with probability $p = \frac{2}{3}$ and initial input $x = 1$. Below, we depict the first few steps of the weighted one-step reduction relation which is induced by the program. Here $\{\!\{p_i : n_i\}\!\}$ denotes the probability of $x = n_i$. The first step has weight (or cost) one. When $x = 0$ then x is in normal form and cannot be reduced any more. Thus, in the second step we rewrite $\left\{\!\!\left\{\frac{1}{1} : 0\right\}\!\!\right\} \xrightarrow{0}\!\!\!\!\twoheadrightarrow \left\{\!\!\left\{\frac{1}{1} : 0\right\}\!\!\right\}$ with probability $\frac{2}{3}$ and $\left\{\!\!\left\{\frac{1}{1} : 2\right\}\!\!\right\} \xrightarrow{1}\!\!\!\!\twoheadrightarrow \left\{\!\!\left\{\frac{2}{3} : 1, \frac{1}{3} : 3\right\}\!\!\right\}$ with probability $\frac{1}{3}$. The weight associated with the second reduction step is $\frac{2}{3} \cdot 0 + \frac{1}{3} \cdot 1 = \frac{1}{3}$.

$$\left\{\!\!\left\{\tfrac{1}{1} : 1\right\}\!\!\right\} \xrightarrow{\frac{1}{1}}\!\!\!\!\twoheadrightarrow \left\{\!\!\left\{\tfrac{2}{3} : \underline{0}, \tfrac{1}{3} : 2\right\}\!\!\right\} \xrightarrow{\frac{1}{3}}\!\!\!\!\twoheadrightarrow \left\{\!\!\left\{\tfrac{2}{3} : \underline{0}, \tfrac{2}{9} : 1, \tfrac{1}{9} : 3\right\}\!\!\right\}$$
$$\xrightarrow{\frac{1}{3}}\!\!\!\!\twoheadrightarrow \left\{\!\!\left\{\tfrac{2}{3} : \underline{0}, \tfrac{4}{27} : \underline{0}, \tfrac{2}{27} : 2, \tfrac{2}{27} : 2, \tfrac{1}{27} : 4\right\}\!\!\right\} \xrightarrow{\frac{5}{27}}\!\!\!\!\twoheadrightarrow \cdots$$

It is easy to see that each multidistribution of the reduction corresponds to a level in the probability tree of Example 5.1 in which the probabilities have been accumulated along the path.

Definition 5.7 (Canonical Expected Cost, Canonical Expected Value)**.** Let $\mathcal{A} = (A, \to)$ be a wPARS. The *expected cost* $\mathsf{ec}_{\mathcal{A}} \colon \mathcal{P}(A) \to \mathbb{R}^{\infty}_{\geqslant 0}$ of $\mathcal{A}$ on $S \subseteq A$ is defined by

$$\mathsf{ec}_{\mathcal{A}}(S) \triangleq \mathsf{sup}\{w \mid a \xrightarrow{w}\!\!\!\!\twoheadrightarrow^* \mu \text{ and } a \in S\}\ .$$

Let $f \colon A \to \mathbb{R}^{\infty}_{\geqslant 0}$ be a function. The *expected value* $\mathsf{ev}^f_{\mathcal{A}} \colon \mathcal{P}(A) \to \mathbb{R}^{\infty}_{\geqslant 0}$ of $\mathcal{A}$ on $S \subseteq A$ with respect to f is defined by

$$\mathsf{ev}^f_{\mathcal{A}}(S) \triangleq \mathsf{sup}\{\mathbb{E}_{\mu}(f) \mid \exists w.\ a \xrightarrow{w}\!\!\!\!\twoheadrightarrow^* \mu\}\ .$$

Definition 5.8 (Strongly Bounded)**.** A wPARS $\mathcal{A} = (A, \to)$ is called *strongly bounded* on $S \subseteq A$ if there exists $p \in \mathbb{R}_{\geqslant 0}$ such that $a \xrightarrow{w}\!\!\!\!\twoheadrightarrow^* \mu$ implies $w \leqslant p$. This is equivalent to the statement $\mathsf{ec}_{\mathcal{A}}(S) < \infty$.

5.3 Probabilistic While Programs

We consider an imperative language pWhile in the spirit of Dijkstra's *guarded command language* [66], endowed with primitives for *sampling* from discrete distributions as well as *non-deterministic and probabilistic choice*. Let Var denote a finite set of integer-valued variables $x, y, \ldots$. We denote by $\Sigma \triangleq \mathsf{Var} \to \mathbb{Z}$ the set of *stores*, that associate variables with their integer contents. The syntax of *program commands* Cmd over Var is given by the following grammar:

$\mathtt{C}, \mathtt{D} \in \mathsf{Cmd} ::=$	$\mathtt{skip}$	*effectless operation*
	$\mid \mathtt{abort}$	*termination*
	$\mid \mathtt{tick}(r)$	*resource consumption*
	$\mid x \mathtt{ := } d$	*probabilistic assignment*
	$\mid \mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$	*conditional*
	$\mid \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$	*while loop*
	$\mid \{\mathtt{C}\}\ \mathtt{<>}\ \{\mathtt{D}\}$	*non-deterministic choice*
	$\mid \{\mathtt{C}\}[p]\{\mathtt{D}\}$	*probabilistic choice*
	$\mid \mathtt{C}; \mathtt{D}$	*sequential composition.*

In this grammar, $\phi, \psi \in \mathsf{BExp}$ denote *Boolean expressions* over Var and $d \in \mathsf{DExp}$ an *integer-valued distribution expression* over Var. With $[\![\cdot]\!] \colon \mathsf{DExp} \to \Sigma \to \mathcal{D}(\mathbb{Z})$ we denote the evaluation functions of distribution expressions, i.e. $[\![d]\!](\sigma)$ returns an integer distribution $\mathcal{D}(\mathbb{Z})$. We keep the precise notion of DExp and BExp abstract. Our prototype implementation supports, amongst others, *uniform* distribution expressions, e.g. $[\![\mathtt{uniform}[x-1, x, x+1]]\!](\sigma) = \{\frac{1}{3} : \sigma(x)-1, \frac{1}{3} : \sigma(x), \frac{1}{3} : \sigma(x)+1\}$ and standard Boolean and relational connectives. For Boolean expressions $\phi \in \mathsf{BExp}$ and store $\sigma \in \Sigma$, we indicate with $\sigma \vDash \phi$ that ϕ holds when the variables in ϕ take values according to σ.

Program commands are fairly standard. The command `skip` is a no-op, and `abort` terminates the execution. The command $\mathtt{tick}(r)$ consumes $r \in \mathbb{Q}_{\geqslant 0}$ resource units. The command $x \mathtt{ := } d$ assigns a value sampled from $[\![d]\!](\sigma)$ to x, for σ the current store. The usual non-probabilistic assignment $x \mathtt{ := } e$ for arithmetic expressions $e \in \mathsf{AExp}$ is recovered by the probabilistic assignment $x \mathtt{ := } d_e$, where $[\![d_e]\!](\sigma) \triangleq \{1 : [\![e]\!](\sigma)\}$.

The commands $\mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$ and $\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$ have the usual semantics, with $\psi \in \mathsf{BExp}$ an assertion that has to hold when entering the command. We abbreviate $\mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$ and $\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$ by $\mathtt{if}\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}$ and $\mathtt{while}\ (\phi)\ \{\mathtt{C}\}$ when ϕ is the trivial assertion $\top$ that is always true. Invariants can be inferred by dedicated tools or incorporated as *assumptions* by the user. Alternatively, we could have introduced an additional command to support assumptions, however, the chosen representation simplifies the implementation.

The command $\{\mathtt{C}\}\ \mathtt{<>}\ \{\mathtt{D}\}$ executes either C or D, in a non-deterministic fashion. Our analysis takes a *demonic* view on non-determinism, assuming that the branch with worst-case resource consumption is taken. In contrast, the probabilistic choice $\{\mathtt{C}\}[p]\{\mathtt{D}\}$

executes $\mathtt{C}$ with probability p and with probability $1-p$ the command $\mathtt{D}$, in which $p \in \mathbb{Q}_{\geqslant 0}$ and $0 \leqslant p \leqslant 1$.

We give the small step operational semantics for our language via a weighted probabilistic ARS $\to$ over *configurations*

$$\mathsf{Conf} \triangleq (\mathsf{Cmd} \times \Sigma) \cup \Sigma \cup \{\bot\} \,.$$

Elements $(\mathtt{C}, \sigma) \in \mathsf{Conf}$ are called *active* and denoted by $\langle \mathtt{C} \rangle(\sigma)$. Such an active configuration signals that the command $\mathtt{C}$ is to be executed under the current store σ, whereas $\sigma \in \mathsf{Conf}$ and $\bot \in \mathsf{Conf}$ indicate that the computation has halted. The former case gives the final store, whereas the later signals that the command terminated abnormally. The weighted probabilistic ARS $(\mathsf{Conf}, \to)$ is depicted in Figure 5.1.

$$\frac{}{\langle \mathtt{skip} \rangle(\sigma) \xrightarrow{0} \sigma}\,[\textsc{Skip}] \qquad \frac{}{\langle \mathtt{abort} \rangle(\sigma) \xrightarrow{0} \bot}\,[\textsc{Halt}] \qquad \frac{}{\langle \mathtt{tick}(r) \rangle(\sigma) \xrightarrow{r} \sigma}\,[\textsc{Tick}]$$

$$\frac{}{\langle x \mathtt{:=} d \rangle(\sigma) \xrightarrow{0} \{\!\{ p_i : \sigma[x := i] \mid i \in \mathbb{Z}, p_i = [\![d]\!](\sigma)(i) > 0 \}\!\}}\,[\textsc{Assign}]$$

$$\frac{\sigma \vDash \psi \wedge \phi}{\langle \mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \langle \mathtt{C} \rangle(\sigma)}\,[\textsc{IfT}] \qquad \frac{\sigma \vDash \psi \wedge \neg\phi}{\langle \mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \langle \mathtt{D} \rangle(\sigma)}\,[\textsc{IfF}]$$

$$\frac{\sigma \vDash \psi \wedge \phi}{\langle \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\} \rangle(\sigma) \xrightarrow{0} \langle \mathtt{C}; \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\} \rangle(\sigma)}\,[\textsc{WhT}]$$

$$\frac{\sigma \vDash \psi \wedge \neg\phi}{\langle \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\} \rangle(\sigma) \xrightarrow{0} \sigma}\,[\textsc{WhF}]$$

$$\frac{\sigma \vDash \neg\psi}{\langle \mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \bot}\,[\textsc{AbortIf}] \qquad \frac{\sigma \vDash \neg\psi}{\langle \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\} \rangle(\sigma) \xrightarrow{0} \bot}\,[\textsc{AbortWhile}]$$

$$\frac{}{\langle \{\mathtt{C}\} \mathtt{<>} \{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \langle \mathtt{C} \rangle(\sigma)}\,[\textsc{ChoiceL}] \qquad \frac{}{\langle \{\mathtt{C}\} \mathtt{<>} \{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \langle \mathtt{D} \rangle(\sigma)}\,[\textsc{ChoiceR}]$$

$$\frac{}{\langle \{\mathtt{C}\}\,[p]\,\{\mathtt{D}\} \rangle(\sigma) \xrightarrow{0} \{\!\{ p : \langle \mathtt{C} \rangle(\sigma), 1-p : \langle \mathtt{D} \rangle(\sigma) \}\!\}}\,[\textsc{ProbChoice}]$$

$$\frac{\langle \mathtt{C} \rangle(\sigma) \xrightarrow{r} \{\!\{ p_i : \gamma_i \}\!\}_{i \in I}}{\langle \mathtt{C}; \mathtt{D} \rangle(\sigma) \xrightarrow{r} \{\!\{ p_i : \mathsf{step}_{\mathtt{D}}(\gamma_i) \}\!\}_{i \in I}}\,[\textsc{Compose}] \quad \text{where } \mathsf{step}_{\mathtt{D}}(\gamma) \triangleq \begin{cases} \langle \mathtt{C}; \mathtt{D} \rangle(\sigma) & \text{if } \gamma = \langle \mathtt{C} \rangle(\sigma) \\ \langle \mathtt{D} \rangle(\sigma) & \text{if } \gamma = \sigma \in \Sigma \\ \bot & \text{if } \gamma = \bot. \end{cases}$$

Figure 5.1: Weighted Probabilistic ARS of pWhile.

In this reduction system, rules have the form $\gamma \xrightarrow{w} \mu$ for $\gamma \in \mathsf{Conf}$ and μ a *multidistribution* over Conf, i.e. countable multisets of the form $\{\!\{ p_i : \gamma_i \}\!\}_{i \in I}$ for *probabilities* $0 < p_i \leqslant 1$ with $\sum_{i \in I} p_i \leqslant 1$ and $\gamma_i \in \mathsf{Conf}$ $(i \in I)$. A rule $\gamma \xrightarrow{w} \{\!\{ p_i : \gamma_i \}\!\}_{i \in I}$ signals that γ reduces with probability p_i to γ_i, consuming cost w. By identifying dirac multidistributions $\{\!\{ 1 : \gamma' \}\!\}$ with γ', we may write $\gamma \xrightarrow{w} \gamma'$ for a reduction step without probabilistic effect.

Only the rules Assign, ProbChoice and Compose have probabilistic effect, all other rules are standard. The rule Skip indicates that any configuration $\langle \mathtt{skip} \rangle(\sigma)$

reduces to σ (or more precisely $\{\!\{1 : \sigma\}\!\}$) consuming 0 resources. Halt reduces any active configuration to the halting configuration $\bot$. The inference rule Tick indicates resource consumption and is the only rule with non-zero cost. With $[\![d]\!](\sigma)(i)$ we denote in Assign the probability that the distribution expression d evaluates to the integer i under the current store σ.

For the conditional commands, we use multiple rules to perform a case distinction on invariants and guards. If the invariant does not halt under the current store, any active configuration reduces to $\bot$. Iteration is induced in Wht by dynamically unrolling the loop. The non-deterministic choice is given by two rules ChoiceL and ChoiceR. On the other hand, the probabilistic choice ProbChoice returns a multidistribution weighting the individual branches with respect to probability p. The rule Compose incorporates the case distinction $\mathsf{step}_{\mathsf{D}}(\gamma)$ on active configurations and normal forms. The first case operates on active configurations which are returned for instance by WhT and IfT, while the other cases consider inactive configurations and $\bot$.

After defining the semantics of pWhile, we specialise the definition of expected cost and expected value earlier presented in Definition 5.7.

Definition 5.9 (Expected Cost, Expected Value)**.** Let $\mathcal{A} = (\mathsf{Conf}, \to)$ be the wPARS of a pWhile program, and let $\twoheadrightarrow$ be the *weighted one-step reduction relation* over $\mathsf{MDist}(\mathsf{Conf})$ induced by the wPARS $(\mathsf{Conf}, \to)$ of Figure 5.1. The *expected cost* $\mathsf{ec}[\cdot] : \mathsf{Cmd} \to \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$ of $\mathcal{A}$ is defined by

$$\mathsf{ec}[\mathtt{C}](\sigma) \triangleq \mathsf{sup}\{w \mid \langle \mathtt{C} \rangle(\sigma) \xrightarrow{w}\!\!\!\!\twoheadrightarrow^{*} \mu\} \,.$$

Let $f \colon \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$ be a function. The *expected value* $\mathsf{ev}[\cdot] : \mathsf{Cmd} \to (\Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}) \to \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$ of $\mathcal{A}$ with respect to f is defined by

$$\mathsf{ev}[\mathtt{C}](f)(\sigma) \triangleq \mathsf{sup}\{f(\mu\restriction\Sigma) \mid \langle \mathtt{C} \rangle(\sigma) \xrightarrow{w}\!\!\!\!\twoheadrightarrow^{*} \mu\} \,.$$

5.4 Expectation Transformers

This work is concerned with mechanising the resource analysis of probabilistic programs. In doing so, we make use of earlier work on expectation transformers, which provide an equivalent notion of expected cost and expected value.

Kaminski et al. [105] express the expected runtime of probabilistic programs in continuation passing style, via the *expected runtime transformer* $\mathsf{ert}[\cdot] \colon \mathsf{Cmd} \to \mathbb{T} \to \mathbb{T}$ over *expectations* $\mathbb{T} \triangleq \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$. Let $\mathtt{C} \in \mathsf{Cmd}$, then $\mathsf{ert}[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$. We suite this transformer to two transformers, the *expected cost transformer* $\mathsf{ect}[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$ and *expected value transformer* $\mathsf{evt}[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$, respectively. Their definition coincide up to the case where $\mathtt{C} = \mathtt{tick}(r)$, the former taking into account the cost r while the latter is ignoring it. We thus generalise $\mathsf{ect}[\mathtt{C}]$ and $\mathsf{evt}[\mathtt{C}]$ to a function $\mathsf{et}_c[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$ which is given in Figure 5.2.

Definition 5.10 (Expected Cost Transformer, Expected Value Transformer)**.** Let $\mathtt{C} \in \mathsf{Cmd}$. The *expected cost transformer* $\mathsf{ect}[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$, and *expected value transformer* $\mathsf{evt}[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$ are defined by

$$\mathsf{ect}[\mathtt{C}] \triangleq \mathsf{et}_{\top}[\mathtt{C}] \text{ and } \mathsf{evt}[\mathtt{C}] \triangleq \mathsf{et}_{\bot}[\mathtt{C}] \,.$$

$$
\begin{aligned}
\mathsf{et}_c[\texttt{skip}](f) &\triangleq f \\
\mathsf{et}_c[\texttt{abort}](f) &\triangleq \mathbf{0} \\
\mathsf{et}_c[\texttt{tick}(r)](f) &\triangleq [c] \cdot \mathbf{r} + f \\
\mathsf{et}_c[x \texttt{ := } d](f) &\triangleq \lambda\sigma.\mathbb{E}_{[\![d]\!](\sigma)}(\lambda i.f(\sigma[x := i])) \\
\mathsf{et}_c[\texttt{if } [\psi] \ (\phi) \ \{\mathtt{C}\} \ \{\mathtt{D}\}](f) &\triangleq [\psi \wedge \phi] \cdot \mathsf{et}_c[\mathtt{C}](f) + [\psi \wedge \neg\phi] \cdot \mathsf{et}_c[\mathtt{D}](f) \\
\mathsf{et}_c[\texttt{while } [\psi] \ (\phi) \ \{\mathtt{C}\}](f) &\triangleq \mu F.[\psi \wedge \phi] \cdot \mathsf{et}_c[\mathtt{C}](F) + [\psi \wedge \neg\phi] \cdot f \\
\mathsf{et}_c[\{\mathtt{C}\} \texttt{ <> } \{\mathtt{D}\}](f) &\triangleq \mathbf{max}(\mathsf{et}_c[\mathtt{C}](f), \mathsf{et}_c[\mathtt{D}](f)) \\
\mathsf{et}_c[\{\mathtt{C}\}\,[p]\,\{\mathtt{D}\}](f) &\triangleq \mathbf{p} \cdot \mathsf{et}_c[\mathtt{C}](f) + (\mathbf{1} - \mathbf{p}) \cdot \mathsf{et}_c[\mathtt{D}](f) \\
\mathsf{et}_c[\mathtt{C}; \mathtt{D}](f) &\triangleq \mathsf{et}_c[\mathtt{C}](\mathsf{et}_c[\mathtt{D}](f))
\end{aligned}
$$

Figure 5.2: Expectation Transformer $\mathsf{et}_c[\mathtt{C}] \colon \mathbb{T} \to \mathbb{T}$.

Here, functions $f \colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$ are extended pointwise on expectations and denoted in bold face, e.g. $\mathbf{0} \triangleq \lambda\sigma.0$ denotes the constant zero function, for each $r \in \mathbb{R}^\infty_{\geqslant 0}$ we have a constant function $\mathbf{r} \triangleq \lambda\sigma.r$, $f + g \triangleq \lambda\sigma.f(\sigma) + g(\sigma)$ for $f, g \in \mathbb{T}$ etc. Furthermore, we denote with $\preccurlyeq$ the *pointwise ordering on expectations*. The structure $(\mathbb{T}, \preccurlyeq)$ forms an ω-complete partial order (ω-CPO for short) with bottom element $\mathbf{0}$ and top element $\boldsymbol{\infty}$ (see Kaminski et al. [105]). For $\phi \in \mathsf{BExp}$ we use $[\phi]$ to denote the expectation function $[\phi](\sigma) \triangleq 1$ if $\sigma \vDash \phi$, and $[\phi](\sigma) \triangleq 0$ otherwise.

We comment on the definition of the expectation transformer given in Figure 5.2. Informally, $\mathsf{evt}[\mathtt{C}](f)(\sigma)$ represents the expectation of f applied to all normal forms that are obtained from the command $\mathtt{C}$ and store σ. The transformer $\mathsf{ect}[\mathtt{C}](f)(\sigma)$ additionally collects all costs that are given by the $\texttt{tick}(r)$ commands. In the process the collected resource is weighted with respect to the probabilistic branches that are induced by assignments and probabilistic choices.

The command `skip` consumes 0 resources and does not modify the store. Therefore, $\mathsf{et}_c[\texttt{skip}](f)$ returns f. The transformer applied on `abort` returns the constant zero function $\mathbf{0}$, which is the bottom element of $\mathbb{T}$. For the tick command the definition of expected cost and expected value is different. In the case $\mathsf{ect}[\texttt{tick}(r)](f)$ we have $[\top] = \mathbf{1}$ and return $\mathbf{r} + f$, and in the case $\mathsf{evt}[\texttt{tick}(r)](f)$ we have $[\bot] = \mathbf{0}$ and return f. The probabilistic assignment returns the expectation of f applied to the updated store.

The conditional flow is governed by the use of the bracket function $[\cdot]$. In the case of the commands $\texttt{if } [\psi] \ (\phi) \ \{\mathtt{C}\} \ \{\mathtt{D}\}$ and $\texttt{while } [\psi] \ (\phi) \ \{\mathtt{C}\}$ exactly one branch evaluates to $\mathbf{1}$ and one branch evaluates to $\mathbf{0}$ for any store. The expectation transformer of the while loop is given by a fixed point representation. With $\mu F.e$ we denote the *least fixed point* of the function $\lambda F.e \colon \mathbb{T} \to \mathbb{T}$ with respect to the pointwise ordering $\preccurlyeq$ on expectations. The transformer $\mathsf{et}_c[\mathtt{C}]$ is ω-continuous, and therefore, $\mathsf{et}_c[\mathtt{C}]$ is well-defined.

We are interested in the worst-case behaviour of programs, i.e. the maximal resource consumption and the maximal value with respect to some function f. Therefore, the non-deterministic choice maximises over both branches. The probabilistic choice returns

the expectation with respect to the given probability p. Finally, sequential application of commands is modelled by composition.

We note that $\mathsf{evt}[\mathtt{C}]$ coincides with the weakest precondition transformer $\mathsf{wp}[\mathtt{C}]$ of Olmedo et al. [128] on *fully probabilistic programs*, i.e. those without non-deterministic choice. While $\mathsf{evt}[\mathtt{C}]$ takes a *demonic* view on non-deterministic choice, $\mathsf{wp}[\mathtt{C}]$ takes an *angelic* view and minimises over non-deterministic choice.

In what follows we make the connection between the expected cost (expected value) of a program (see Definition 5.9) and the expected cost transformer (expected value transformer) (see Definition 5.10) precise. Here, we outline the central results, for details we refer to the technical report, Avanzini, Schaper, and Moser [21].

Recall that $\mathbb{T} \triangleq \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$. For expectations $f \colon \mathbb{T}$, we lift $\mathsf{et}_c[\mathtt{C}](f) \colon \Sigma \to \mathbb{R}^{\infty}_{\geqslant 0}$ from stores Σ to configurations Conf.

Definition 5.11. Let $\underline{\mathsf{et}}_c(f) \colon \mathsf{Conf} \to \mathbb{R}^{\infty}_{\geqslant 0}$ be defined by

$$\underline{\mathsf{et}}_c(f)(\langle \mathtt{C} \rangle(\sigma)) \triangleq \mathsf{et}_c[\mathtt{C}](f)(\sigma) \quad \underline{\mathsf{et}}_c(f)(\sigma) \triangleq f(\sigma) \quad \underline{\mathsf{et}}_c(f)(\bot) \triangleq 0 \,.$$

The following constitutes our first technical result which shows that $\mathsf{et}_c[\cdot](f)$ decreases in expectation along reductions, taking into account the cost in the case of $\mathsf{ect}[\cdot](f)$.

Theorem 5.12 (Decreasing wPARS)**.** $\mathbb{E}_{\mu}(\underline{\mathsf{et}}_c(f)) = \mathsf{sup}\{[c] \cdot w + \mathbb{E}_{\nu}(\underline{\mathsf{et}}_c(f)) \mid \mu \xrightarrow{w}\!\!\!\!\twoheadrightarrow^{*} \nu\}$.

To prove this theorem, we first show its variations based on the wPARS $\to$ and the single-step reduction relation $\twoheadrightarrow$ (see [21]). Both of these intermediate results follow by a straightforward induction on the corresponding reduction relation. The following is then immediate:

Corollary 5.13 (Soundness of the Expectation Transformers)**.** *Let* $\mathtt{C} \in \mathsf{Cmd}$ *be a command and* $\sigma \in \Sigma$ *be a store. Then,*

$$\textit{(i)}\ \mathsf{ec}[\mathtt{C}](\sigma) \leqslant \mathsf{ect}[\mathtt{C}](\mathbf{0})(\sigma)\ \textit{and (ii)}\ \mathsf{ev}[\mathtt{C}](f)(\sigma) \leqslant \mathsf{evt}[\mathtt{C}](f)(\sigma) \,.$$

By (i), the expected cost of running $\mathtt{C}$ is given by $\mathsf{ect}[\mathtt{C}](\mathbf{0})$. When $\mathtt{C}$ does not contain any loops, the latter is easily computable. To handle programs with loops, Kaminski et al. [105] propose to search for upper invariants. $I_f \in \mathbb{T}$ is an *upper invariant* for $\mathtt{D} = \mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$ with respect to $f \in \mathbb{T}$ if it is a pre-fixpoint of the cost through which $\mathsf{et}_c[\mathtt{D}](f)$ is defined, i.e. it satisfies $[\psi \wedge \phi] \cdot \mathsf{et}_c[\mathtt{D}](I_f) + [\psi \wedge \neg\phi] \cdot f \preccurlyeq I_f$.

Proposition 5.14 (Upper Invariant)**.** *Let* $\mathtt{C} \in \mathsf{Cmd}$ *be a command. Suppose* $I_f \in \mathbb{T}$ *is an upper invariant for* $\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$ *with respect to* $f \in \mathbb{T}$*. Then,*

$$[\psi \wedge \phi] \cdot \mathsf{et}_c[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}](I_f) + [\psi \wedge \neg\phi] \cdot f \preccurlyeq I_f \implies \mathsf{et}_c[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}](f) \preccurlyeq I_f.$$

This immediately suggests the following two stage approach towards an automated expected runtime analysis of a program $\mathtt{C}$ via Corollary 5.13. In the first stage, one evaluates $\mathsf{ect}[\mathtt{C}](\mathbf{0})$ *symbolically* on a notion of *cost expressions* CExp, generating constraints according to Proposition 5.14 whenever a $\mathtt{while}$ loop is encountered. Based on

the collection of generated constraints, in the second phase concrete upper invariants can be *synthesised*. From these, a symbolic upper bound on the expected cost $\mathsf{ec}[\mathtt{C}]$ can be constructed. Conceptually, this is the approach taken by Absynth [124], where $\mathsf{ert}[\mathtt{C}]$ is formulated in terms of a Hoare style calculus, and CExp is amenable to *linear programming*.

5.5 Towards A Modular Analysis

With Proposition 5.14 alone it is in general not possible to modularise this procedure so that individual components can be treated separately. In particular, nested loops generate mutual constraints that cannot be solved independently. In the course of this section we present Theorems 5.20 and 5.21, which provide conditions under which this global analysis can be broken down into a local one.

Example 5.15 (Compositional Analysis of Nested Loops)**.** The following example depicts a simple pWhile program with nested loops. We use A and B to indicate program labels for the loops.

$$
\begin{array}{ll}
A\colon & \texttt{while } (x > 0) \\
 & \quad \texttt{tick } 1;\ \{\ x = x - 1\ \}\ [\tfrac{2}{3}]\ \{\ x = x + 1\ \} \\
B\colon & \quad \texttt{while } (y > 0) \\
 & \qquad \texttt{tick } 1;\ \{\ y = y - 2\ \}\ [\tfrac{1}{2}]\ \{\ y = y + 1\ \}
\end{array}
$$

The following constraint system illustrates the application of the ect transformer together with Proposition 5.14. For the sake of readability we employ additional simplifications. The function symbols f_A and f_B denote unknown functions in terms of the integer-valued variables x and y. Due to the fixed point definition of the expectation transformer we obtain cyclic dependencies for f_A and f_B.

$$
\begin{aligned}
x > 0 &\implies \mathsf{f}_A(x,y) \geqslant 1 + \tfrac{2}{3} \cdot \mathsf{f}_B(x-1,y) + \tfrac{1}{3} \cdot \mathsf{f}_B(x+1,y) \\
x \leqslant 0 &\implies \mathsf{f}_A(x,y) \geqslant 0 \\
y > 0 &\implies \mathsf{f}_B(x,y) \geqslant 1 + \tfrac{1}{2} \cdot \mathsf{f}_B(x,y-2) + \tfrac{1}{2} \cdot \mathsf{f}_B(x,y+1) \\
y \leqslant 0 &\implies \mathsf{f}_B(x,y) \geqslant \mathsf{f}_A(x,y)
\end{aligned}
$$

In what follows, let $\vec{g} = g_1, \ldots, g_k$ denote expectations with $g_i \colon \Sigma \to \mathbb{R}^\infty_{\geqslant 0}$ and $f \colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$, The *composition* $\lambda\sigma.f(g_1(\sigma), \ldots, g_k(\sigma))$ is denoted by $f \circ \vec{g}$.

Definition 5.16 (Concave, Weakly Monotone)**.** Let p denote a probability with $0 \leqslant p \leqslant 1$. Call $f \colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$ *concave*, if

$$f(p \cdot \vec{r} + (1-p) \cdot \vec{s}) \geqslant p \cdot f(\vec{r}) + (1-p) \cdot f(\vec{s})\ .$$

Call $f \colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$ (weakly) *monotone*, if

$$\vec{r} \geqslant \vec{s} \implies f(\vec{r}) \geqslant f(\vec{s})\ .$$

Here, addition and multiplication, as well as the order $\geqslant$ are extended from $\mathbb{R}^\infty_{\geqslant 0}$ to $(\mathbb{R}^\infty_{\geqslant 0})^k$ componentwise.

Now, consider the expected cost transformer $\mathsf{ect}[\mathtt{C}](f)$ of command $\mathtt{C}$ with respect to expectation f. The first lemma states that it is enough to consider the expected cost $\mathsf{ec}[\mathtt{C}] = \mathsf{ect}[\mathtt{C}](\mathbf{0})$ plus the expected value transformer $\mathsf{evt}[\mathtt{C}](f)$ of $\mathtt{C}$ with respect to f. A similar lemma is given in Olmedo et al. [128] for a recursive probabilistic language. We suitably adapt it to pWhile.

Lemma 5.17. *Let* $\mathtt{C} \in \mathsf{Cmd}$ *and* $f \in \mathbb{T}$*. Then,* $\mathsf{ect}[\mathtt{C}](f) \preccurlyeq \mathsf{ec}[\mathtt{C}] + \mathsf{evt}[\mathtt{C}](f)$.

If $\mathtt{C}$ is fully-probabilistic, i.e. the command does not contain non-deterministic choice, then the stronger result $\mathsf{ect}[\mathtt{C}](f) = \mathsf{ec}[\mathtt{C}] + \mathsf{evt}[\mathtt{C}](f)$ holds. The following is immediate.

Corollary 5.18. *Let* $\mathtt{C};\mathtt{D} \in \mathsf{Cmd}$ *be commands. Then* $\mathsf{ec}[\mathtt{C};\mathtt{D}] \preccurlyeq \mathsf{ec}[\mathtt{C}] + \mathsf{evt}[\mathtt{C}](\mathsf{ec}[\mathtt{D}])$.

In Section 3.3.4 on page 19 we provide an informal discussion for the modular resource analysis of *sequential and nested* imperative programs. In the case of sequential programs $\mathtt{C};\mathtt{D}$ we investigate the runtime complexity of $\mathtt{C}$ and $\mathtt{D}$ separately. In doing so, we make use of size bounds, which assess the output of $\mathtt{C}$, on the runtime of $\mathtt{D}$.

Corollary 5.18 makes this discussion precise for the pWhile programming language. In non-probabilistic programs modularity is achieved by investigating size bounds on norms of the runtime function of $\mathtt{D}$ rather than on the runtime itself. For instance, consider that the runtime of $\mathtt{D}$ is $\mathsf{max}(0,x) \cdot \mathsf{max}(0,y)$. Then it is enough to obtain upper bounds on the output values of $\mathsf{max}(0,x)$ and $\mathsf{max}(0,y)$ with respect to $\mathtt{C}$. The only prerequisite is that the runtime function of $\mathtt{D}$ is *weakly monotone* in its arguments. We recover this observation for pWhile. However, for probabilistic programs we additionally require that the runtime (or expected cost) of $\mathtt{D}$ is *concave*. The next lemma presents our central observation.

Lemma 5.19. *Let* $\mathtt{C} \in \mathsf{Cmd}$ *be a command. Suppose* $g\colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$ *is a weakly monotone and concave function. Then,*

$$\mathsf{ect}[\mathtt{C}](g \circ (g_1, \ldots, g_k)) \preccurlyeq \mathsf{ec}[\mathtt{C}] + g \circ (\mathsf{evt}[\mathtt{C}](g_1), \ldots, \mathsf{evt}[\mathtt{C}](g_k)) \ .$$

The intuition behind this lemma is as follows. The functions $g_i\colon \Sigma \to \mathbb{R}^\infty_{\geqslant 0}$, also referred to as *norms*, represent an abstract view on program stores σ. In the most simple case, g_i could denote the absolute value of the i-th variable. Now consider a program $\mathtt{C};\mathtt{D}$. Let g measure the expected resource consumption of $\mathtt{D}$. The expected cost of $\mathtt{C};\mathtt{D}$ is thus the expected cost of $\mathtt{C}$, plus the expected cost of $\mathtt{D}$ measured in the values $\mathsf{evt}[\mathtt{C}](g_i)$ of the norms g_i expected after executing $\mathtt{C}$.

Next, we present two applications of the previous lemma for the modular analysis of sequential and nested programs.

Theorem 5.20 (Decomposition of Sequential Programs)**.** *Let* $\mathtt{C},\mathtt{D} \in \mathsf{Cmd}$ *be commands. Suppose* $g\colon (\mathbb{R}^\infty_{\geqslant 0})^k \to \mathbb{R}^\infty_{\geqslant 0}$ *is a weakly monotone and concave function. Then,*

$$\mathsf{ec}[\mathtt{D}] \preccurlyeq g \circ (g_1, \ldots, g_k) \implies \mathsf{ec}[\mathtt{C};\mathtt{D}] \preccurlyeq \mathsf{ec}[\mathtt{C}] + g \circ (\mathsf{evt}[\mathtt{C}](g_1), \ldots, \mathsf{evt}[\mathtt{C}](g_k)) \ .$$

We emphasize that concavity can be dropped when $\mathtt{C}$ admits no probabilistic behaviour. In combination with upper invariants (see Proposition 5.14) we obtain a modular method for loops.

Theorem 5.21 (Decomposition of Nested Programs)**.** *Let* $\mathtt{C}, \mathtt{D} \in \mathsf{Cmd}$ *be commands. Suppose* $g\colon (\mathbb{R}^{\infty}_{\geqslant 0})^k \to \mathbb{R}^{\infty}_{\geqslant 0}$ *is a weakly monotone and concave function. Then,*

$$[\psi \wedge \phi] \cdot (\mathsf{ec}[\mathtt{C}] + g \circ (\mathsf{evt}[\mathtt{C}](g_1), \ldots, \mathsf{evt}[\mathtt{C}](g_k))) \preccurlyeq g \circ (g_1, \ldots, g_k)$$
$$\wedge\, [\psi \wedge \neg\phi] \cdot f \preccurlyeq g \circ (g_1, \ldots, g_k) \implies \mathsf{ect}[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}](f) \preccurlyeq g \circ (g_1, \ldots, g_k)\ .$$

Consider a program $\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}$. With $\mathsf{ec}[\mathtt{C}]$ we assess the cost of one iteration of the loop body, while $\mathsf{evt}[\mathtt{C}](g_i)$ assesses the change in the norm g_i. Concavity and upper invariants are exploited to achieve modularity. We notice that if the program has multiple nested `while` loops, then each loop can be processed separately bottom-up.

5.6 Automation

In this section we provide additional insights about the implementation of the discussed techniques within the prototype pWhile, which is available online at

`http://cbr.uibk.ac.at/tools/pwhile/` .

At the time of writing the prototype implementation accepts pWhile programs with finite distributions over integer expressions in probabilistic assignments.

To provide an intuitive notion of cost functions and facilitate automation we introduce cost expressions. *Arithmetic expressions* $a, b \in \mathsf{AExp}$ and *cost expressions* $c, d \in \mathsf{CExp}$ over variables $x \in \mathsf{Var}$, integers $z \in \mathbb{Z}$ and constants $q \in \mathbb{Q}_{\geqslant 0}$ are given as follows:

$$a, b \in \mathsf{AExp} ::= x \mid z \mid a + b \mid a * b \mid \ldots$$
$$c, d \in \mathsf{CExp} ::= q \mid \mathsf{nat}(a) \mid [\phi] \cdot c \mid c + d \mid c \cdot d \mid \mathsf{max}(c, d)$$

Norms $\mathsf{nat}(a)$ lift expressions that depend on the store to cost expressions. We fix the interpretation of norms to $\mathsf{nat}(a) \triangleq \mathsf{max}(0, a)$. All other operations are interpreted in the expected way. The *evaluation function* of cost expressions is denoted by $[\![\cdot]\!]\colon \mathsf{CExp} \to \Sigma \to \mathbb{Q}_{\geqslant 0}$. Notice that $[\![c]\!] \in \mathbb{T}$ for all $c \in \mathsf{CExp}$.

To automate the cost inference of programs we provide a variation of the expectation transformer $\mathsf{et}_c[\mathtt{C}]\colon \mathbb{T} \to \mathbb{T}$ given in Figure 5.2. The *expectation transformer over cost expressions* $\mathsf{et}^{\sharp}_c[\mathtt{C}]\colon \mathsf{CExp} \to \mathsf{CExp}$, is defined in Figure 5.3. Furthermore, we define $\mathsf{ect}^{\sharp}[\mathtt{C}] \triangleq \mathsf{et}^{\sharp}_{\top}[\mathtt{C}]$ and $\mathsf{evt}^{\sharp}[\mathtt{C}] \triangleq \mathsf{et}^{\sharp}_{\bot}[\mathtt{C}]$. We intentionally omit the case for `while` loops, which we discuss below in more detail. The expectation transformer over cost expressions $\mathsf{et}^{\sharp}_c[\mathtt{C}]$ mimics $\mathsf{et}_c[\mathtt{C}]$ closely. In the case of probabilistic assignments we restrict to finite distributions over integer expressions, i.e. $\mathsf{et}^{\sharp}_c[x\ \mathtt{:=}\ \{p_1\colon a_1, \ldots, p_k\colon a_k\}](f) \triangleq \sum_{1 \leqslant i \leqslant k} p_i \cdot f[a_i/x]$, in which $f[a/x]$ denotes the substitution of variable x with expression a.

The expectation transformer over cost expressions is sound in the following sense.

Theorem 5.22 (Soundness of Expectation Transformer over Cost Expressions)**.** *Let* $\mathtt{C} \in \mathsf{Cmd}$ *be a command and* $f \in \mathsf{CExp}$ *be a cost expression. Then,*

$$\mathsf{et}_c[\mathtt{C}]([\![f]\!]) \preccurlyeq [\![\mathsf{et}^{\sharp}_c[\mathtt{C}](f)]\!]\ .$$

$$
\begin{aligned}
\mathsf{et}_c^\sharp[\mathtt{skip}](f) &\triangleq f \\
\mathsf{et}_c^\sharp[\mathtt{tick}(r)](f) &\triangleq [c] \cdot r + f \\
\mathsf{et}_c^\sharp[\mathtt{abort}](f) &\triangleq 0 \\
\mathsf{et}_c^\sharp[x := \{p_1 \colon a_1, \ldots, p_n \colon a_n\}](f) &\triangleq \textstyle\sum_i^n p_i \cdot f[a_i/x] \\
\mathsf{et}_c^\sharp[\mathtt{if}\ [\psi]\ (\phi)\ \{\mathtt{C}\}\ \{\mathtt{D}\}](f) &\triangleq [\psi \wedge \phi] \cdot \mathsf{et}_c^\sharp[\mathtt{C}](f) + [\psi \wedge \neg\phi] \cdot \mathsf{et}_c^\sharp[\mathtt{D}](f) \\
\mathsf{et}_c^\sharp[\{\mathtt{C}\} <> \{\mathtt{D}\}](f) &\triangleq \mathsf{max}(\mathsf{et}_c^\sharp[\mathtt{C}](f), \mathsf{et}_c^\sharp[\mathtt{D}](f)) \\
\mathsf{et}_c^\sharp[\{\mathtt{C}\}\,[p]\,\{\mathtt{D}\}](f) &\triangleq p \cdot \mathsf{et}_c^\sharp[\mathtt{C}](f) + (1-p) \cdot \mathsf{et}_c^\sharp[\mathtt{D}](f) \\
\mathsf{et}_c^\sharp[\mathtt{C};\mathtt{D}](f) &\triangleq \mathsf{et}_c^\sharp[\mathtt{C}](\mathsf{et}_c^\sharp[\mathtt{D}](f))
\end{aligned}
$$

Figure 5.3: Expectation Transformer over Cost Expressions $\mathsf{et}_c^\sharp[\mathtt{C}] \colon \mathsf{CExp} \to \mathsf{CExp}$.

We make use of Theorems 5.20 and 5.21 to decompose programs. Notably, using both theorems we can define a recursive strategy that infers bounds on `while` loops separately. We comment on the application of the theorems in the implementation.

Assume that we are interested to infer $\mathsf{ect}^\sharp[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}](f)$. First, we recursively compute $g_0 = \mathsf{ect}^\sharp[\mathtt{C}](0)$, which indicates the cost of one loop iteration. We heuristically select norms $g_1, \ldots, g_k$ based on the invariants and conditions of the program (e.g. $\mathsf{nat}(x-y)$ for some guard $x > y$). Second, we recursively compute $h_i = \mathsf{evt}^\sharp[\mathtt{C}](g_i)$ for all g_i. We have $\mathsf{ect}[\mathtt{C}](\mathbf{0}) \preccurlyeq [\![g_0]\!]$ and $\mathsf{evt}[\mathtt{C}]([\![g_i]\!]) \preccurlyeq [\![h_i]\!]$. Third, we express the necessary conditions as constraints over cost expressions:

$$
\begin{aligned}
\psi \wedge \phi &\vDash g_0 + \mathtt{h} \circ (h_1, \ldots, h_k) \leqslant \mathtt{h} \circ (g_1, \ldots, g_k) \\
\psi \wedge \neg\phi &\vDash f \leqslant \mathtt{h} \circ (g_1, \ldots, g_k)\ .
\end{aligned}
$$

Here $\mathtt{h}$ is a *template cost expression* with undetermined coefficients q_i. A constraint $\phi \vDash c \leqslant d$ holds if $[\![\phi]\!] \vDash [\![c]\!] \preccurlyeq [\![d]\!]$ holds for all stores $\sigma \in \Sigma$. When generating constraints only $\mathtt{h}$ is unknown.

To obtain a concrete cost expression for $\mathtt{h}$ we follow the method presented in Fuhs et al. [77]. Take for instance, the template cost expression $\lambda h_i. \sum q_i \cdot h_i$ for $\mathtt{h}$. Then, we are interested in finding an assignment of q_i such that all constraints hold and $q_i \geqslant 0$. In the implementation we restrict the template cost expression for $\mathtt{h}$ such that $[\![\mathtt{h}]\!]$ is weakly monotone and concave. We apply *case-elimination* and *case-distinction* to reduce the problem $\phi \vDash c \leqslant d$ to inequality constraints over polynomials. For example, given a norm $\mathsf{nat}(a) = \mathsf{max}(0, a)$, we eliminate max if we can show that $[\![a]\!] \geqslant 0$ for all assignments that satisfy ϕ. The obtained inequality constraints of polynomials have undetermined coefficient variables. We reduce the problem to certification of *non-negativity*, which can then be solved using SMT solvers.

Consequently, we have $\mathsf{ect}[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}]([\![f]\!]) \preccurlyeq [\![\mathsf{ect}^\sharp[\mathtt{while}\ [\psi]\ (\phi)\ \{\mathtt{C}\}](f)]\!]$ for all cost expression $f \in \mathsf{CExp}$. This statement follows immediately by Theorem 5.21 and the construction above.

Example 5.23 (Modular Analysis of Nested Loops)**.** We recall the motivating program of Example 5.15 and illustrate the modular approach to the expected runtime analysis of nested loops. Here, we abbreviate the body of the outer loop with command `C`.

```
        while (x > 0)
C:        tick 1; { x = x − 1 } [2/3] { x = x + 1 }
          while (y > 0)
            tick 1; { y = y − 2 } [1/2] { y = y + 1 }
```

We are interested in inferring an upper bound on the expected cost $\mathsf{ec}[\texttt{while}\ (x > 0)\ \{\texttt{C}\}]$. By Corollary 5.13 and Theorem 5.22 we have

$$\mathsf{ec}[\texttt{while}\ (x > 0)\ \{\texttt{C}\}] \preccurlyeq \mathsf{ect}[\texttt{while}\ (x > 0)\ \{\texttt{C}\}](\mathbf{0}) \preccurlyeq [\![\mathsf{ect}^\sharp[\texttt{while}\ (x > 0)\ \{\texttt{C}\}](0)]\!] \,.$$

Thus, we inspect $\mathsf{ect}^\sharp[\texttt{while}\ (x > 0)\ \{\texttt{C}\}](0)$. With respect to the previous discussion it is enough to consider constraints of the following form. For now, we keep h and the chosen norms $g_1, \ldots, g_k$ abstract.

$$x > 0 \vDash \mathsf{ect}^\sharp[\texttt{C}](0) + \mathsf{h} \circ (\mathsf{evt}^\sharp[\texttt{C}](g_1), \ldots, \mathsf{evt}^\sharp[\texttt{C}](g_k)) \leqslant \mathsf{h} \circ (g_1, \ldots, g_k)$$
$$x \leqslant 0 \vDash 0 \leqslant \mathsf{h} \circ (g_1, \ldots, g_k)$$

Since cost expressions evaluate to $\mathbb{Q}_{\geqslant 0}$ for all configurations, we restrict to the interesting first case.

$$\begin{aligned}
x > 0 &\vDash \mathsf{ect}^\sharp[\texttt{C}](0) + \mathsf{h} \circ (\mathsf{evt}^\sharp[\texttt{C}](g_1), \ldots, \mathsf{evt}^\sharp[\texttt{C}](g_k)) \leqslant \mathsf{h} \circ (g_1, \ldots, g_k) \\
x > 0 &\vDash 1 + 2 * \mathsf{nat}(y+2) + \mathsf{h} \circ (\mathsf{evt}^\sharp[\texttt{C}](g_1), \ldots, \mathsf{evt}^\sharp[\texttt{C}](g_k)) \leqslant \mathsf{h} \circ (g_1, \ldots, g_k) \\
x > 0 &\vDash 1 + 2 * \mathsf{nat}(y+2) + \sum q_i [\mathsf{evt}^\sharp[\texttt{C}](1), \mathsf{evt}[\texttt{C}](\mathsf{nat}(x)), \mathsf{evt}^\sharp[\texttt{C}](\mathsf{nat}(y+2))] \\
&\qquad\qquad \leqslant \sum q_i [1, \mathsf{nat}(x), \mathsf{nat}(y+2)] \\
x > 0 &\vDash 1 + 2 * \mathsf{nat}(y+2) + \sum q_i [1, \tfrac{2}{3}\,\mathsf{nat}(x-1) + \tfrac{1}{3}\,\mathsf{nat}(x+1), 1] \\
&\qquad\qquad \leqslant \sum q_i [1, \mathsf{nat}(x), \mathsf{nat}(y+2)]
\end{aligned}$$

In the first step, we recursively compute $\mathsf{ect}^\sharp[\texttt{C}](0)$. Then, the unknown cost expression h is replaced by the template cost expression $\sum q_i$, in which q_i denote unknown coefficients. Moreover, the prototype implementation infers the norms $g_1 = \mathsf{nat}(x)$, $g_2 = \mathsf{nat}(y+2)$ and the numeric constant $g_3 = 1$ from the program code. After the norms have been fixed, the expected value functions over cost expressions $\mathsf{evt}^\sharp[\texttt{C}](g_i)$ are computed recursively. In the last step we solve the resulting constraint system. In doing so, we perform case distinctions on the norms and require $q_i \geqslant 0$. We display only the relevant cases.

$$\begin{aligned}
&x > 0 \land y + 2 \geqslant 0 \\
&\quad \vDash 1 + 2(y+2) + q_1 + q_2 \tfrac{2}{3}(x-1) + q_2 \tfrac{1}{3}(x+1) + q3 \leqslant q_1 + q_2 x + q_3 (y+2) \\
&x > 0 \land 0 > y + 2 \\
&\quad \vDash 1 + q_1 + q_2 \tfrac{2}{3}(x-1) + q_2 \tfrac{1}{3}(x+1) \leqslant q_1 + q_2 x
\end{aligned}$$

A valid assignment is $q_1 = 0$, $q_2 = 9$ and $q_3 = 2$, which implies the upper bound $\mathsf{ec}[\texttt{while}\ (x > 0)\ \{\texttt{C}\}] \preccurlyeq [\![9 \cdot \mathsf{nat}(x) + 2 \cdot \mathsf{nat}(y+2)]\!]$.

5.7 Concluding Remarks

In this chapter we have discussed the automated resource analysis of an imperative probabilistic language pWhile. Our goal is the development of a modular analysis that can be automated. Taking inspiration of the resource analysis of non-probabilistic programs we have carefully investigated under which conditions modularity can be obtained again for probabilistic programs. The central observation states that we can improve upon a compositional analysis by restricting the shape of bound expressions to weakly monotone and concave functions. This allows to state proof rules for the modular analysis of sequential and nested programs. We have implemented a prototype that makes use of this observation. In the near future we are going to focus on the improvement of this prototype. In particular, we are interested in generalising assignments x `:=` d to infinite distributions. This allows to inspect the resource behaviour of challenging programs such as the *Coupon Collector*, in which `rand`(N) indicates a uniform distribution that depends on the input argument N.

```
assume (N> 0)
while(x > 0){
  tick(1)
  i := rand(N)
  if( i ⩾ x) { x := x − 1 } { skip }
}
```

Chapter 6

Framework for Automation

In this chapter we are concerned with *automating* resource analysis. Instead of inspecting the complexity problem of a concrete programming language or the implementation of a specific approach, we present a *framework for automation*. The framework brings together the principles that have been discussed in previous chapters, namely *abstract program representations*, *complexity reflecting transformations* and *modular bound analysis*, and is enriched with tactic-like combinators to facilitate *proof search*. It has been implemented and constitutes the core of the latest instalment of the *Tyrolean Complexity Tool* TcT. At the time of writing, the latest release is TcT-3.3, which supports the resource analysis of higher-order functional programs, term rewrite systems, object-oriented bytecode programs and constraint transition systems.

The presentation in this chapter is based on Avanzini, Moser, and Schaper [18]. Section 6.1 motivates the framework and provides a high-level overview. Then, we illustrate the *software architecture* of TcT in Section 6.2. In Section 6.3 we present the *combination framework* that forms the theoretical foundation of the tool. Section 6.4 provides details about the *implementation* of the complexity framework, and Section 6.5 demonstrates several *case studies*. Finally, we conclude this chapter in Section 6.6.

6.1 Introduction

In this section we motivate the complexity framework and provide a high-level overview. TcT is implemented in Haskell and available online at

`http://cl-informatik.uibk.ac.at/software/tct/` .

It features a command line, an interactive, and a web interface. In the setup of complexity analyser, TcT embodies a *transformational approach*, which is depicted in Figure 6.1.

First, the input program in relation to the resource of interest is *transformed* to an *abstract representation*. We refer to the result of applying such a transformation as *abstract program*. It has to be guaranteed that the employed transformations are *complexity reflecting*, that is, the resource bound on the obtained abstract program reflects upon the resource usage of the input program. More precisely, the complexity analysis deals with a general *complexity problem* that consists of a program together with the resource metric of interest as input. Second, we employ problem specific techniques to derive bounds on the given problem, and finally, the result of the analysis, i.e. a complexity bound or a notice of failure, is relayed back to the user.

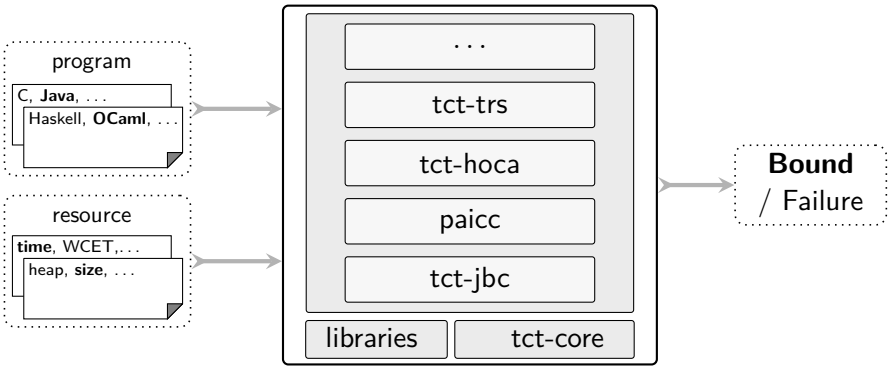


Figure 6.1: The Complexity Analyser TcT.

We emphasise that TcT does not make use of a *unique* abstract representation, but is designed to employ a *variety* of different representations. Moreover, different representations may interact with each other. This improves *modularity* of the approach and provides scalability and precision of the overall analysis. For now, we make use of *constraint transition systems* (CTSs for short) and various forms of *term rewrite systems* (TRSs for short). Concretising this abstract setup, TcT currently provides a fully automated runtime complexity analysis of pure OCaml programs as well as a runtime analysis of object-oriented bytecode programs. Furthermore, the tool provides runtime and size analysis of CTSs as well as runtime analysis of first-order rewrite systems. The latest instalment is a complete reimplementation of the tool that takes full advantage of the abstract *complexity framework* introduced by Avanzini and Moser [14]. TcT is open with respect to the complexity problem under investigation and problem specific techniques for the resource analysis. Moreover, it provides an expressive problem independent *strategy language* that facilitates *proof search*. In the rest of this chapter, we give insights about design choices, the implementation of the framework and report different case studies where we have applied TcT successfully.

6.2 Architectural Overview

In this section we give an overview of the *architecture* of the complexity analyser.

All components of TcT are written in the strongly typed, lazy functional programming language Haskell and released *open source* under BSD3. The core has approximately 2.200 lines of code, excluding external libraries. As depicted in Figure 6.1, the implementation of TcT is divided into separate components for the different program kinds and abstractions thereof supported. These separate components are no islands however. Rather, they *instantiate* the abstract *complexity framework* [14] for complexity analysis, from which TcT derives its power and modularity. In short, in this framework complexity techniques are modelled as *complexity processors* that give rise to a set of inferences over *complexity proofs*. From a completed complexity proof, a *complexity bound* can be inferred. The theoretical foundations of this framework are given in Section 6.3.

The abstract complexity framework is implemented in TcT's *core library*, termed tct-core, which is depicted in Figure 6.2 at the bottom layer. Central, it provides a common notion of a proof state, viz *proof trees*, and an interface for specifying processors. Furthermore, tct-core complements the framework with a simple but powerful *strategy language*. Strategies play the role of tactics in interactive theorem provers like Isabelle or Coq. They allow us to turn a set of processors into a sophisticated complexity analyser. The implementation details of the core library are provided in Section 6.4.

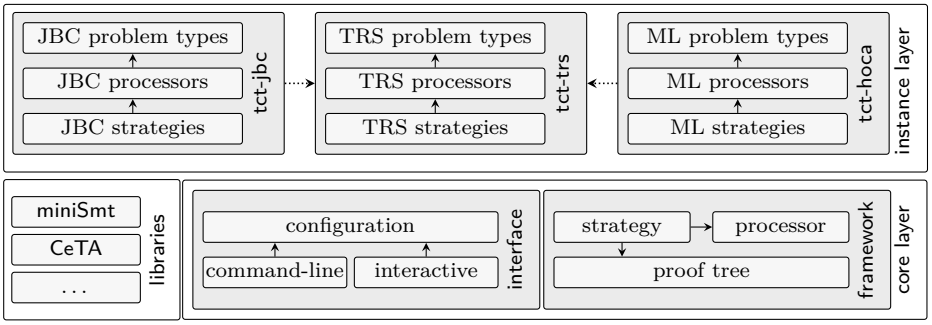


Figure 6.2: Software Architecture of TcT.

The complexity framework implemented in the core library leaves the type of *complexity problem*, consisting of the analysed program together with the resource metric of interest, abstract. Rather, concrete complexity problems are provided by concrete *instances*, such as the three *instances* tct-jbc, tct-trs and tct-hoca depicted in Figure 6.2. We will discuss some instances in detail in Section 6.5. Instances implement complexity techniques on defined problem types in the form of complexity processors, possibly relying on external libraries and tools such as SMT solvers. Optionally, instances may also specify strategies that compose the provided processors. Bridges between instances are easily specified as processors that implement conversions between problem types defined in different instances. For instance, in Chapter 4 we discuss the term abstraction of imperative programs with heap allocated data structures to (constraint) term rewrite systems. The module tct-jbc integrates the term abstraction as complexity processor and makes use of the module tct-trs as back-end to analyse the obtained complexity problem. We emphasise that the system is open to the seamless integration of alternative problem types through the specification of new instances.

Development Cycle. TcT was originally envisioned as a dedicated tool for the automated complexity analysis of first-order term rewrite systems. The first version was made available in 2008. Transformations of complexity problems have been implemented in version 1.7 and polished in version 2.0 (Avanzini and Moser [13]). Version 3.0 of the tool is a complete rewrite of the framework (Avanzini, Moser, and Schaper [18]). The main conceptual difference being, that the transformation framework, which is implemented in tct-core, is open to the type of the complexity problem. At the time of writing, the latest release is TcT-3.3, which provides minor incremental improvements.

6.3 A Formal Framework for Complexity Analysis

In this section we outline the theoretical framework upon which the complexity analyser T$_C$T is based. This framework was originally introduced by Avanzini and Moser [14] for the derivational and runtime complexity analysis of first-order term rewrite systems.

As mentioned before, both the input language (e.g. Java, OCaml, ...) and the resource under consideration (e.g. execution time, heap usage, ...) is kept abstract in the framework. That is, we assume that we are dealing with an abstract class of *complexity problems*. Each complexity problem $\mathcal{P}$ from this class is associated with a *complexity function* $\mathsf{cp}_{\mathcal{P}} : D \to D$, for a *complexity domain* D. Usually, the complexity domain D will be the set of natural numbers $\mathbb{N}$, however, more sophisticated choices of complexity functions such as those proposed by Albert et al. [6] and Danner et al. [63] fall into the realm of this framework.

Example 6.1 (Complexity Problem)**.** In Chapter 4 we discuss the worst-case runtime analysis of imperative programs with heap. Let $\mathcal{P}$ be a `Goto`R program and I denote a set of initial configurations. The worst-case runtime complexity of $\mathcal{P}$ on I is given by the maximal derivation height in terms of the input size $m \in \mathbb{N}$ (see Definition 4.4 on page 81). We suitably adapt this definition to complexity functions. The class of complexity problems is fixed to $\mathcal{P} = (\mathcal{P}, I)$ and the complexity domain is fixed to $D = \mathbb{N}$. Furthermore, let $\mathsf{cp}_{(\mathcal{P},I)}(m) \triangleq \mathsf{rc}^I_{\mathcal{P}}(m)$.

As indicated in the introduction, any transformational solver converts concrete programs into abstract ones, if not already interfaced with an abstract program. Based on the possible abstracted complexity problem $\mathcal{P}$ the analysis continues using a set of *complexity techniques*. In particular, a reasonable solver will also integrate some form of *decomposition techniques*, transforming an intermediate problem into various smaller *subproblems*, and analysing these subproblems separately, either again by some form of decomposition method, or eventually by some *base technique* which infers a suitable resource bound. At any stage in this *transformation chain*, a solver needs to keep track of computed complexity bounds, and relay these back to the initial problem.

To support this kind of reasoning, it is convenient to formalise the internals of a complexity analyser as an inference system over *complexity judgements*. In this framework, a complexity judgement has the shape $\vdash \mathcal{P} : B$, where $\mathcal{P}$ is a complexity problem and B is a set of *bounding functions* $f : D \to D$ for a complexity domain D. Such a judgement is *valid* if the complexity function of $\mathcal{P}$ lies in B, that is, $\mathsf{cp}_{\mathcal{P}} \in B$. Complexity techniques are modelled as *processors* within the framework. A processor defines a transformation of the *input problem* $\mathcal{P}$ into a list of subproblems $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ (if any), and it relates the complexity of the obtained subproblems to the complexity of the input problem. Processors are given as inferences

$$\frac{Pre(\mathcal{P}) \quad \vdash \mathcal{Q}_1 : B_1 \quad \cdots \quad \vdash \mathcal{Q}_n : B_n}{\vdash \mathcal{P} : B} \ ,$$

where $Pre(\mathcal{P})$ indicates a collection of *preconditions* on $\mathcal{P}$.

The processor is *sound* if under $Pre(\mathcal{P})$ the validity of judgements is preserved, that is,

$$Pre(\mathcal{P}) \wedge \mathsf{cp}_{\mathcal{Q}_1} \in B_1 \wedge \cdots \wedge \mathsf{cp}_{\mathcal{Q}_n} \in B_n \quad \Longrightarrow \quad \mathsf{cp}_{\mathcal{P}} \in B \ .$$

Dual, it is called *complete* if under the assumptions $Pre(\mathcal{P})$, validity of the judgement $\vdash \mathcal{P} : B$ implies validity of the judgements $\vdash \mathcal{Q}_i : B_i$.

A *proof* of a judgement $\vdash \mathcal{P} : B$ from the *assumptions* $\vdash \mathcal{Q}_1 : B_1, \ldots, \vdash \mathcal{Q}_n : B_n$ is a deduction using sound processors only. The proof is *closed* if its set of assumptions is empty. Soundness of processors guarantees that the formal system is correct. Application of complete processors on a valid judgement ensures that no invalid assumptions are derived. In this sense, the application of a complete processor is always safe.

Proposition 6.2. *If there exists a closed complexity proof $\vdash \mathcal{P} : B$, then the judgement $\vdash \mathcal{P} : B$ is valid.*

Example 6.3 (Complexity Processor)**.** We express the complexity reflecting transformations based on the path-length abstraction and the term abstraction of GotoR programs (see Theorem 4.17 on page 87 and Theorem 4.26 on page 92) as complexity processors:

$$\frac{\mathsf{acyclic}(I) \quad \vdash (\mathcal{P}^\circ, I^\circ) : f}{\vdash (\mathcal{P}, I) : \lambda m.f(k \cdot m)}\ \mathsf{path} \qquad \frac{\mathsf{treeshaped}(I) \quad \vdash (\mathcal{P}^\bullet, I^\bullet) : f}{\vdash (\mathcal{P}, I) : \lambda m.f(k \cdot m)}\ \mathsf{term}$$

Here, $(\mathcal{P}^\circ, I^\circ)$ and $(\mathcal{P}^\bullet, I^\bullet)$ denote the abstractions that are obtained by the transformations presented in Chapter 4. We express the necessary restrictions on the shape of the input as preconditions.

6.4 Implementing the Complexity Framework

The formal complexity framework described in the last section is implemented in the core library, termed tct-core. In the following we outline the two central components of this library:

(i) the generation of complexity proofs, and

(ii) common facilities for instantiating the framework to concrete tools.

6.4.1 Proof Trees, Processors, and Strategies

The library tct-core provides the verification of a valid complexity judgement $\vdash \mathcal{P} : B$ from a given input problem $\mathcal{P}$. More precisely, the library provides the environment to construct a complexity proof witnessing the validity of $\vdash \mathcal{P} : B$.

Since the class B of bounding functions is a result of the analysis, and not an input, the complexity proof can only be constructed once the analysis finished successfully. For this reason, proofs are not directly represented as trees over complexity judgements. Rather, the library features *proof trees*. Conceptually, a proof tree is a tree whose leaves are labelled by *open complexity problems*, that is, problems which remain to be analysed,

and whose internal nodes represent successful applications of processors. The complexity analysis of a problem $\mathcal{P}$ then amounts to the *expansion* of the proof tree whose single node is labelled by the open problem $\mathcal{P}$. *Processors* implement a single expansion step. To facilitate the expansion of proof trees, tct-core features a rich *strategy language*, similar to tactics in interactive theorem provers like Isabelle or Coq. Once a proof tree has been completely expanded, a complexity judgement for $\mathcal{P}$ together with the witnessing complexity proof can be computed from the proof tree.

In the following, we detail the central notions of proof tree, processor and strategy.

Proof Trees

Crucial to our representation of proof trees is that we abstract over the problem type. This allows concrete instantiations to precisely specify which problems are supported. Consequently, proof trees are parametrised in the type of complexity problems. The corresponding (generalised) algebraic data type `ProofTree` α (from module `Tct.Core.Data.ProofTree`) is depicted in Figure 6.3.

```
data ProofTree α where
  Open    :: α → ProofTree α                          -- open proof node
  Success :: Processor β ⇒                            -- successful application
               ProofNode β → CertFn → [ProofTree α] → ProofTree α
  Failure :: Reason → ProofTree α                     -- failed application
```

Figure 6.3: Data Type Declaration of Proof Trees in tct-core.

A constructor `Open` represents a leaf labelled by an open problem of type α. The ternary constructor `Success` represents the successful application of a processor of type β. Its first argument, a value of type `ProofNode` β, carries the applied processor, the current complexity problem under investigation as well as a proof object of type `ProofObject` β. This information is useful for proof analysis, and allows a detailed textual representation of proof trees. Note that `ProofObject` is a type-level function, the concrete representation of a proof object thus depends on the type of the applied processor. The second argument to `Success` is a *certificate-function*

```
type CertFn = [Certificate] → Certificate ,
```

which is used to relate the estimated complexity of generated subproblems to the analysed complexity problem. Thus currently, the set of bounding functions B occurring in the final complexity proof is fixed to those expressed by the data type `Certificate` (module `Tct.Core.Data.Certificate`). `Certificate` includes various representations of complexity classes, such as the class of polynomials, exponentials, primitive and multiple recursive functions, but also the more fine-grained classes of bounding functions $\mathcal{O}(n^k)$ for all $k \in \mathbb{N}$. The remaining argument to the constructor `Success` is a forest of proof trees, each individual proof tree represents the continuation of the analysis of a corresponding subproblem, which is generated by the applied processor. Finally, the constructor `Failure` indicates that the analysis failed. It results for example from the

application of a processor to an open problem which does not satisfy the preconditions of the processor. The argument of type `Reason` allows a textual representation of the failure-condition. The analysis will abort on proof trees containing such a *failure node*.

Processors

The interface for processors is specified by the type class `Processor`, which is defined in module `Tct.Core.Data.Processor` and depicted in Figure 6.4.

```
data Return α
 = NoProgress Reason
 | Progress (ProofObject α) CertFn [ProofTree (Out α)]

class (ProofData (ProofObject α)) ⇒ Processor α where
  type In α                                    -- type of input problem
  type Out α                                   -- type of output problems
  type ProofObject α                           -- meta information
  execute :: α → In α → TctM (Return α)        -- implementation

-- application of processor to a problem, resulting in a proof tree
apply :: Processor α ⇒ α → In α → TctM (ProofTree (Out α))
apply p i = (toProofTree <$> execute p i) `catchError` handler where
  toProofTree (NoProgress r)      = Failure r
  toProofTree (Progress ob cf ts) = Success (ProofNode p i ob) cf ts
  handler err                     = return (NoProgress (IOError err))
```

Figure 6.4: Data Type and Class Declaration of Processors in tct-core.

The type of input problem and generated subproblems are defined for processors on an individual basis, through the type-level functions `In` and `Out`, respectively. This eliminates the need for a global problem type, and facilitates the seamless combination of different instantiations of the core library. Each processor instance specifies additionally the type of proof objects `ProofObject` α, i.e. the meta information provided in case of a successful application. The proof object is constrained to instances of `ProofData`, which besides others, ensures that a textual representation can be obtained. Each instance of `Processor` has to implement a method `execute`, which given an input problem of type `In` α, evaluates to a `TctM` action that produces a value of type `Return` α. The monad `TctM` (defined in module `Tct.Core.Data.TctM`) extends the `IO` monad with access to runtime information, such as command line parameters and execution time. The data type `Return` α specifies the result of the application of a processor to its given input problem. In case of a successful application, the return value carries the proof object, a value of type `CertFn`, which relates complexity-bounds on subproblems to bounds on the input problem, and the list of generated subproblems. In fact, the type is slightly more liberal and allows for each generated subproblem a possibly open proof tree. This generalisation is useful in certain contexts, for instance, when the processor makes use of a second processor.

Strategies

To facilitate the expansion of a proof tree, tct-core features a simple but expressive *strategy language*. The strategy language is *deeply embedded*, via the generalised algebraic data type `Strategy α β`, which is depicted in Figure 6.5.

```
data Strategy α β where
  -- primitives for sequential processor application
  Id          :: Strategy α α
  Apply       :: (Processor γ) ⇒ γ → Strategy (In γ) (Out γ)
  Abort       :: Strategy α β
  Cond        :: (ProofTree β → Bool) → Strategy α β → Strategy β γ
                   → Strategy α γ → Strategy α γ
  -- primitives for parallel processor application
  Par         :: Strategy α β → Strategy α β
  Race        :: Strategy α β → Strategy α β → Strategy α β
  Better      :: (ProofTree β → ProofTree β → Ordering)
                    → Strategy α β → Strategy α β → Strategy α β
  -- control operators
  Timeout     :: Time → Strategy α β → Strategy α β
  Wait        :: Time → Strategy α β → Strategy α β
  WithStatus  :: (TcTStatus α → Strategy α β) → Strategy α β
  WithState   :: (TcTROState α → Strategy α β) → Strategy α β
```

Figure 6.5: Deep Embedding of the Strategy Language in tct-core.

Semantics over strategies are given by the function

```
evaluate :: Strategy α β → ProofTree α → TctM (ProofTree β) ,
```

defined in module `Tct.Core.Data.Strategy`. A strategy of type `Strategy α β` translates a proof tree with open problems of type α to one with open problems of type β.

The first four primitives defined in Figure 6.5 constitute our tool box for modelling sequential application of processors. The strategy `Id` is implemented by the identity function on proof trees. The remaining three primitives traverse the given proof tree in-order, acting on all open proof nodes. The strategy `Apply p` replaces the given open proof node with the proof tree resulting from an application of processor `p`. The strategy `Abort` signals that the computation should be aborted, replacing the given proof node by a failure node. Finally, the strategy `Cond predicate s1 s2 s3` implements a very specific conditional. It sequences the application of strategies `s1` and `s2`, provided the proof tree computed by `s1` satisfies the predicate `predicate`. For the case where the predicate is not satisfied, the conditional acts like the third strategy `s3`.

In Figure 6.6 we showcase the definition of derived strategy combinators. Sequencing `s1 ⋙ s2` of strategies `s1` and `s2` as well as a (left-biased) choice operator `s1 <|> s2` are derived from the conditional primitive `Cond`. When `s` fails then `try s` behaves as an identity. The combinator `force` complements the combinator `try`: the strategy `force s` enforces that strategy `s` produces a new proof node. The combinator `try` brings *backtracking* to our strategy language, i.e. the strategy `try s1 ⋙ s2` first applies strategy

```
-- auxiliary predicates on proof trees
nonFailing,progress :: ProofTree α → Bool
nonFailing t       = null [ Failure {} ← subTrees t ]
progress   Open{}  = False
progress   _       = True
-- choice
(<|>) :: Strategy α β → Strategy α β → Strategy α β
s1 <|> s2 = Cond nonFailing s1 Id s2
-- composition
(⋙) :: Strategy α β → Strategy β γ → Strategy α γ
s1 ⋙ s2 = Cond nonFailing s1 s2 Abort
-- backtracking
try :: Strategy α α → Strategy α α
try s = s <|> Id
force :: Strategy α β → Strategy α β
force s = Cond progress s Id Abort
-- iteration
exhaustive,exhaustive+ :: Strategy α α → Strategy α α
exhaustive  s = try (exhaustive+ s)
exhaustive+ s = force s ⋙ exhaustive s
```

Figure 6.6: Derived Strategy Combinators.

`s1`, backtracks in case of failure, and applies `s2` afterwards. In fact, in version 3.3 we use dedicated constructors for `s1 ⋙ s2`, `s1 <|> s2` and `force s1` , which provide better feedback when `s1` is failing. For the sake of brevity we follow the original presentation in [18]. Finally, the strategies `exhaustive s` applies `s` zero or more times, until strategy `s` fails. The combinator `exhaustive+` behaves similarly, but applies the given strategy at least once. The obtained combinators satisfy the expected laws, compare Figure 6.7 for an excerpt.

```
s1 ⋙ (s2 ⋙ s3) ≡ (s1 ⋙ s2) ⋙ s3          -- ⋙ is associative
s1<|>(s2<|>s3) ≡ (s1<|>s2)<|>s3          -- <|> is associative
s ⋙ Id ≡ s ≡ Id ⋙ s                      -- identity element of ⋙
s <|> Abort ≡ s ≡ Abort <|> s             -- identity element of <|>
s1 ⋙ (s2<|>s3) ≡ (s1 ⋙ s2)<|>(s1 ⋙ s3)   -- ⋙ distributes over <|>
(s1<|>s2) ⋙ s3 ≡ (s1 ⋙ s3)<|>(s2 ⋙ s3)
```

Figure 6.7: Laws of Derived Combinators (Excerpt).

Our language features also three dedicated constructors for parallel proof search. The strategy `Par s` implements a form of data level parallelism, applying strategy `s` to all open problems in the given proof tree in parallel. In contrast, the strategies `Race s1 s2` and `Better comp s1 s2` apply to each open problem the strategies `s1` and `s2` concurrently, and can be seen as parallel version of our choice operator. Whereas `Race s1 s2` simply returns the (non-failing) proof tree of whichever strategy returns first, `Better comp s1 s2` uses the provided comparison function `comp` to decide which proof tree to return.

We comment on the final four strategies depicted in Figure 6.5. The constructors `Wait` and `Timeout` are used to implement *wait* and *timeout*. The final two strategies provide means to construct strategies dynamically during execution. `TctStatus` includes global state, such as command line flags and the execution time, but also proof relevant state such as the current problem under investigation. `TctROState` is used to store runtime options for external tools such as SAT/SMT solvers.

6.4.2 From the Core to Executables

The framework is instantiated by providing a set of sound processors, together with their corresponding input and output types. At the end of the day the complexity framework has to give rise to an executable tool, which, given an initial problem, possibly provides a complexity certificate.

To ease the generation of such an executable, tct-core provides a default implementation of the `main` function, controlled by a `TctConfig` record (see module `Tct.Core.Main`). A minimal definition of `TctConfig` just requires the specification of a default strategy, and a parser for the initial complexity problem. Optionally, one can for example specify additional command line parameters, or a list of *declarations* for custom strategies, which allow the user to control the proof search. Strategy declarations wrap strategies with additional *meta information*, such as a *name*, a *description*, and a list of *parameters*. Firstly, this information is used for documentary purposes. If we call the default implementation with the command line flag `--list-strategies` it will present a documentation of the available processors and strategies to the user. Secondly, declarations facilitate the parser generation for custom strategies. Declarations, together with usage information, are defined in module `Tct.Core.Data.Declaration`. Given a path pointing to the file holding the initial complexity problem, the generated executable will perform the following actions in order:

1. Parse the command line options given to the executable, and reflect these in the aforementioned `TctStatus`.
2. Parse the given file according to the parser specified in the `TctConfig`.
3. Select a strategy based on the command line flags, and apply the selected strategy on the parsed input problem.
4. Should the analysis succeed, a textual representation of the obtained complexity judgement and corresponding proof tree is printed to the console. Otherwise, the uncompleted proof tree, including the `Reason` for failure is printed to the console.

Interactive. The library provides an *interactive mode* via the GHCi interpreter, similar to the one provided in TcT-2.0 (cf. Avanzini and Moser [13]). The implementation keeps track of a *proof state*, a list of proof trees that represents the history of the interactive session. We provide an interface to inspect and manipulate the proof state. Most noteworthy, the user can select individual sub-problems and apply strategies on them. The proof state is updated accordingly.

6.5 Case Studies

In this section we discuss several instantiations of this framework which have been established up to now. We keep the descriptions of the complexity problems informal and focus on the big picture. In the discussion we group abstract programs in contrast to real world programs.

6.5.1 Abstract Program Representations

Currently TcT employs *constraint transition systems* and *term rewrite systems* as abstract program representations. As mentioned before, the system is open to the seamless integration of alternative abstractions.

Constraint Transition Systems

In Chapter 3 we study the worst-case runtime analysis of integer-valued *constraint transition systems* (CTSs for short). CTSs naturally arise from imperative programs where loops, conditionals and assignments are formed over integer expressions, but can also be obtained from programs with user-defined data structures using suitable size abstractions, as illustrated for instance in Chapter 4. In Section 3.7 we discuss the tool paicc which provides an automated runtime analysis of integer programs based on the growth-rate analysis of loop programs (see also Schaper [139]). In what follows, we exemplify a typical configuration which can be compiled to an executable using the tct-core library. We depict the configuration file of paicc in Figure 6.8, and show the output of running paicc on the upcoming example in Figure 6.9. In the discussion of the example, we assume familiarity with the details of paicc given in Section 3.7.

The `main` method of the configuration file makes the program executable. The function `parseProblem` parses CTSs in a rule based format. The implementation in paicc uses several transformation steps. Each individual transformation step is implemented as a processor and lifted to a strategy using `apply` (see Figure 6.4). In doing so, we obtain a proof node for each transformation step in the output. The configuration file illustrates the combination of the individual transformation steps, thus forming a pipeline between different program representations. The final representation is then applicable to the implemented runtime analysis. We comment on the individual transformation steps below.

Example 6.4 (Motivating Example)**.** The following CTS depicts one of the running examples of Chapter 3 (page 62) in a rule based format that can be processed with paicc.

```
(VAR x y z)
(RULES
 l0(x,y,z) -> l1(x',y',z') :|:           x' = x   && y' = y   && z' = z
 l1(x,y,z) -> l1(x',y',z') :|: x > 0  && x' = x-1 && y' = x+y && z' = x+y
 l1(x,y,z) -> l1(x',y',z') :|: z > 0  && x' = x   && y' = y   && z' = z-1
 l1(x,y,z) -> l2(x',y',z') :|: x <= 0 && x' = x   && y' = y   && z' = z
)
```

```
main :: IO ()
main = runTct paiccConfig where
  paiccConfig = defaultTctConfig
    { parser = parseProblem, defaultStrategy = simple }

-- simplify (integer) constraint transition systems
simplify :: Strategy Its Its
simplify = try slicing ≫≫ try deadCodeElimination

data Greedy   = Greedy   | NoGreedy
data Minimize = Minimize | NoMinimize

-- abstract to BJK program
toLare :: Greedy → Minimize → Strategy Its Lare
toLare g m = decompose g ≫≫ abstractSize m ≫≫ abstractFlow

-- combine transformations and run the analysis
paicc :: Greedy → Minimize → Strategy Its lare
paicc g m = try simplify ≫≫ toLare g m ≫≫ analyseGrowthRate

simple,pervasive :: Strategy Its Lare
simple     = paicc NoGreedy NoMinimize
pervarsive = best
  [ paicc g m | g ← [Greedy, NoGreedy] , m ← [Minimize, NoMinimize] ]
```

Figure 6.8: Example Configuration of paicc (Simplified).

Simplification. We provide some standard program simplifications on CTSs which include *variable slicing* and *dead code elimination*. The growth-rate algorithm runs in polynomial time, however, the degree of the polynomial depends on the number of variables. Variable slicing removes individual variables based on a simple heuristic. Dead code elimination implements a syntax based reachability test and a feasibility check on constraints to eliminate transitions that cannot occur in any program run. Here, the modifier `try` indicates that we do not expect that the application is successful in the sense that the problem is actually simplified.

Decomposition. Control-flow in $\mathcal{BJK}$ programs is bounded via its loop structure. We recall that the loop structure is a nesting hierarchy of subprograms in which each subprogram is associated with a bound that limits the length of a trace when running it. The strategy `decompose greedy` signals the application of a processor which implements the algorithm presented in Section 3.7.2 on page 63 to infer a loop structure of the problem under consideration. The argument determines the policy of the algorithm. The flag `Greedy` indicates a policy that tries to infer a loop structure with minimal nesting depth, keeping in mind that the nesting depth implies a multiplicative interaction of the loop bounds. The result of this transformation step is a CTS that is decorated with a loop structure.

```
WORST_CASE(?,POLY)
* Step 1: Decompose WORST_CASE(?,POLY)
    + Considered Problem:
        Rules:
          0. l0(x,y,z) -> l1(x',y',z')  [x' = x && y' = y && z' = z]
          1. l1(x,y,z) -> l1(x',y',z')  [-1 + x >= 0 && x' = -1 + x && y' = x + y && z' = x + y]
          2. l1(x,y,z) -> l1(x',y',z')  [-1 + z >= 0 && x' = x && y' = y && z' = -1 + z]
          3. l1(x,y,z) -> l2(x',y',z')  [0 >= x && x' = x && y' = y && z' = z]
    + Applied Processor:
        Decompose Greedy
    + Details:
        We construct a loop structure:
          P: [0,1,2,3]
          \- p:[1,2] c: [1]
             \- p:[2] c: [2]
* Step 3: AbstractSize WORST_CASE(?,POLY)
    + Considered Problem:
        Rules:
          0. l0(x,y,z) -> l1(x',y',z')  [x' = x && y' = y && z' = z]
          1. l1(x,y,z) -> l1(x',y',z')  [-1 + x >= 0 && x' = -1 + x && y' = x + y && z' = x + y]
          2. l1(x,y,z) -> l1(x',y',z')  [-1 + z >= 0 && x' = x && y' = y && z' = -1 + z]
          3. l1(x,y,z) -> l2(x',y',z')  [0 >= x && x' = x && y' = y && z' = z]
        Loop Structure:
          P: [0,1,2,3]
          \- p:[1,2] c: [1]
             \- p:[2] c: [2]
    + Applied Processor:
        AbstractSize Minimize
* Step 4: AbstractFlow WORST_CASE(?,POLY)
    + Considered Problem:
        Program:
          Domain: [x,y,z] Bounds: [M,N]
           l0  ->  l1   [x <= x, y <= y, z <= z]
          + Loop [M <= K + x]
               l1  ->  l1   [x <= x, y <= x + y, z <= x + y]
               l1  ->  l1   [x <= x, y <= y, z <= z]
            + Loop [N <= K + z]
                 l1  ->  l1   [x <= x, y <= y, z <= z]
    + Applied Processor:
        AbstractFlow
* Step 5: Lare WORST_CASE(?,POLY)
    + Considered Problem:
        Program:
          Domain: [tick,huge,K,x,y,z] Bounds: [M,N]
           l0  ->  l1   []
          + Loop: [x -+> M,K -+> M]
               l1  ->  l1   [x -+> y,x -+> z,y -+> y,y -+> z]
               l1  ->  l1   []
            + Loop: [z -+> N,K -+> N]
                 l1  ->  l1   []
    + Applied Processor:
        Lare
    + Details:
         l0  ->  l2   [y -+> y, y -+> z, x -*> y, x -*> z, x -*> tick, y -*> tick,...]
        + <l1  ->  l1>  [y -+> y, y -+> z, x -*> y, x -*> z, x -*> tick, y -*> tick, ... ]
          + <l1  ->  l1>  [z -+> N, z -+> tick, tick -+> tick, ...]
```

Figure 6.9: Proof Output of paicc for the Running Example (Simplified).

Size Abstraction. The strategy `abstractSize m` infers transition invariants that conform to $\mathcal{BJK}$ constraints. The argument indicates whether optimisations, i.e. minimising coefficients, should be applied. We obtain a CTS with $\mathcal{BJK}$ constraints.

Flow Abstraction. The strategy `abstractFlow` abstracts $\mathcal{BJK}$ constraints into unary and binary flow information. In the proof output we use `x -=> y` to indicate an identity flow, `x -+> y` to indicate an additive flow, `x -*> y` to indicate a multiplicative flow, and `x -^> y` to indicate a super-polynomial flow from program variable `x` to variable `y`. Moreover, the program is augmented with a counter variable `tick` to represent the running time, a symbolic constant `K` to represent constant numbers and a symbolic constant `huge` which indicates unconstrained (or unbounded) growth.

Growth-Rate Analysis. Finally, `analyseGrowthRate` is a strategy that implements the growth-rate algorithm of Ben-Amram and Pineles [37]. Applying this processor results in a new proof node that has no subproblems. For the running program a polynomial growth-rate on the counter variable `tick` can be established, viz there exists a multiplicative flow `x -*> tick` and `y -*> tick` but there is no super-polynomial flow from any input variable to `tick`. The tool returns the complexity bound `Worst_CASE(?,POLY)` on the worst-case runtime.

The strategies `simple` and `pervasive` combine all transformation steps and the growth-rate analysis and can be directly applied on the complexity problem. The combinator `best` is the list version of `Better` and applies the strategies with all possible arguments in parallel, in doing so, it returns the proof tree witnessing the best bound. The proof tree that is depicted in Figure 6.9 is obtained by applying the `defaultStrategy` on the running example. It illustrates the result of each successfully applied transformation step.

Term Rewrite Systems

In Chapter 4 we use (constraint) term rewrite systems as target abstraction for programs with heap allocated data structures. The tct-trs instance provides automated resource analysis of *(first-order) term rewrite systems* (TRSs for short). Term rewriting forms an abstract Turing complete model of computation, which underlies much of declarative programming (cf. Baader and Nipkow [22]). Complexity analysis of TRSs has received significant attention in the last decade, see Moser [118] for details. For an overview of the techniques that are implemented in tct-trs, we refer to Avanzini [11].

We recall, a TRS consists of a set of rewrite rules, i.e. directed equations that can be applied from left to right. Computation is performed by *normalisation*, i.e. by successively applying rewrite rules until no more rules apply. As an example, consider the following TRS $\mathcal{R}_{\mathsf{sq}}$, which computes the squaring function on natural numbers given in unary notation.

Example 6.5 (Term Rewrite System $\mathcal{R}_{\mathsf{sq}}$)**.** The TRS $\mathcal{R}_{\mathsf{sq}}$ computes the *square* of a natural number in unary representation. We use infix notation for readability.

$$\mathsf{sq}(x) \to x * x \qquad x * 0 \to 0 \qquad x + 0 \to x$$
$$\mathsf{s}(x) * y \to y + (x * y) \qquad \mathsf{s}(x) + y \to \mathsf{s}(x + y)\ .$$

The *runtime complexity* of a TRS is naturally expressed as a function that measures the length of the longest reduction, in the sizes of (normalised) starting terms. Figure 6.10 depicts the proof output of tct-trs when applying a *polynomial interpretation* [109] processor with maximal degree 2 on $\mathcal{R}_{\mathsf{sq}}$. The resulting proof tree consists of a single progress node and returns the (optimal) quadratic asymptotic upper bound on the runtime complexity of $\mathcal{R}_{\mathsf{sq}}$.

```
WORST_CASE(?,O(n^2))
*** 1 Progress [(?,O(n^2))]  ***
 Considered Problem:
   Strict TRS Rules
     mult(0(),y)  -> 0()
     mult(s(x),y) -> plus(y,mult(x,y))
     plus(x,0())  -> x
     plus(s(x),y) -> s(plus(x,y))
     square(x)    -> mult(x,x)
   Signature:
     {mult/2,plus/2,square/1} / {0/0,s/1}
   Obligation: runtime innermost

 Applied Processor:
   NaturalPI {shape = Mixed 2}

 Proof:
   Polynomial Interpretation:
         p(0)  = 1
       p(mult) = 3*x1 + 2*x1*x2 + 2*x2
       p(plus) = 2 + 2*x1 + x2
         p(s)  = 1 + x1
     p(square) = 6 + 7*x1 + 2*x1^2
```

```
Orientation:
  mult(0(),y) = 3 + 4*y
              > 1
              = 0()

 mult(s(x),y) = 3 + 3*x + 2*x*y + 4*y
              > 2 + 3*x + 2*x*y + 4*y
              = plus(y,mult(x,y))

  plus(x,0()) = 3 + 2*x
              > x
              = x

 plus(s(x),y) = 4 + 2*x + y
              > 3 + 2*x + y
              = s(plus(x,y))

    square(x) = 6 + 7*x + 2*x^2
              > 5*x + 2*x^2
              = mult(x,x)
```

Figure 6.10: Polynomial Interpretation Proof of $\mathcal{R}_{\mathsf{sq}}$.

While the configuration for paicc, see above, models a simple transformation pipeline the tct-trs instance makes full use of the strategy combinator framework mixing more than 20 base techniques (not considering several variations of them) to a powerful strategy (see module `Tct.Trs.Strategy.Runtime`).

The tct-trs module also supports certified complexity proofs via the tool CeTA[1] (*Certified Tool Assertions*). The tool is written in the Isabelle/HOL theorem prover and provides

[1] `http://cl-informatik.uibk.ac.at/software/ceta/`

certification of several program properties such as (non-)termination, (non-)confluence, completion and complexity for term rewriting (see Avanzini et al. [17] for certification of complexity proofs). This is achieved, by providing an alternative proof output that conforms to the *certification problem format* which is processed by CeTA.

6.5.2 Real World Programs

One motivation for the complexity analysis of abstract programs is that these models are well-equipped to abstract over real-world programs whilst remaining conceptually simple.

Object-Oriented Bytecode Programs

In Chapter 4 we discuss the abstraction of programs with heap allocated data structures. The tct-jbc instance provides automated worst-case runtime analysis of object-oriented bytecode programs, more specifically, Jinja bytecode [107] (JBC for short) programs. We recall, given a JBC program, we measure the maximal number of bytecode instructions executed in any evaluation of the program. We employ techniques from data-flow analysis and abstract interpretation to obtain a *term* abstraction of JBC programs in terms of *constraint term rewrite systems* (cTRSs for short) (see Moser and Schaper [119] for details). CTRSs generalise TRSs and CTSs. Moreover, from the cTRSs, which are obtained from the term abstraction of JBC programs, we can extract a standard TRS or CTS fragment.

We have implemented the term abstraction in a dedicated tool termed jat[2] (*Jinja Analysation Tool*) and have integrated its functionality in the instance tct-jbc. The corresponding strategy, termed `jbc`, is depicted in Figure 6.11. We can use the instances tct-trs and paicc to analyse the problems which are obtained by the proposed transformation. We remark that in the current version tct-jbc uses the module tct-its, which provides a prototype implementation of the runtime and size analysis of integer programs by Brockschmidt et al. [51], instead of paicc. The framework is expressive enough to analyse both problems in parallel.

```
jbc :: Strategy ITS () → Strategy TRS () → Strategy JBC ()
jbc its trs = toCTRS ⋙ Race (toIts ⋙ its) (toTrs ⋙ trs)
```

Figure 6.11: JBC Transformation Pipeline modelled in tct-jbc.

Note that `Race s1 s2` requires that `s1` and `s2` have the same output problem type. We can model this with transformations to a dummy problem `()`. Nevertheless, as intended any witness that is obtained by a successful application of `its` or `trs` will be relayed back.

Example 6.6 (Binary Tree Traversal)**.** In Chapter 4 we discuss the runtime analysis of binary tree traversals. Example 4.29 on page 94 illustrates the TRS that is obtained by a successful application of `toTRS` together with a polynomial interpretation proof that induces a linear runtime bound.

[2] `http://cbr.uibk.ac.at/jat/`

Pure OCaml

In the case of higher-order functional programs, a successful application of the transformational approach has been demonstrated by Avanzini et al. [16], which study the runtime complexity of pure OCaml programs.

Example 6.7 (List Reversal)**.** The following OCaml program, which is taken from Bird's textbook on functional programming [40], illustrates reversal of a list.

```
let rec fold_left f acc = function
    [] → acc
  | x::xs → fold_left f (f acc x) xs ;;
let rev l = fold_left (fun xs x → x::xs) [] l ;;
```

A suitable adaption of Reynold's *defunctionalisation* [134] technique translates the given program into a slight generalisation of TRSs, so-called *applicative term rewrite systems* (ATRSs for short). In ATRSs closures are explicitly represented as first-order structures. Evaluation of these closures is defined via a global apply function (denoted by @). The structure of the defunctionalised program is necessarily intricate, even for simple programs. However, in conjunction with a sequence of sophisticated and in particular complexity reflecting transformations, one can bring the defunctionalised program in a form which can be effectively analysed by first-order complexity provers such as tct-trs (see Avanzini et al. [16] for the details).

The transformation from pure OCaml to term rewrite systems has been implemented in a prototype termed HoCA[3] (*Higher-Order Complexity Analysis*). Figure 6.12 depicts an example run of HoCA on the motivating example.

$$
\begin{aligned}
\mathsf{main}(x_0) &\to \mathsf{m}_1(x_0) \mathbin{@} \mathsf{f} & \mathsf{r}(x_0) \mathbin{@} x_1 &\to x_0 \mathbin{@} \mathsf{r}_1 \mathbin{@} [\,] \mathbin{@} x_1 \\
\mathsf{m}_1(x_0) \mathbin{@} x_1 &\to \mathsf{m}_2(x_0) \mathbin{@} \mathsf{r}(x_1) & \mathsf{r}_1 \mathbin{@} x_0 &\to \mathsf{r}_2(x_0) \\
\mathsf{m}_2(x_0) \mathbin{@} x_1 &\to x_1 \mathbin{@} x_0 & \mathsf{r}_2(x_0) \mathbin{@} x_1 &\to x_1 :: x_0 \\
\mathsf{f} \mathbin{@} x_0 &\to \mathsf{f}_1 \mathbin{@} x_0 & \mathsf{f}_3(x_0, x_1) \mathbin{@} x_2 &\to \mathsf{f}_4(x_2, x_0, x_1) \\
\mathsf{f}_1 \mathbin{@} x_1 &\to \mathsf{f}_2(x_1) & \mathsf{f}_4([\,], x_0, x_1) &\to x_1 \\
\mathsf{f}_2(x_1) \mathbin{@} x_2 &\to \mathsf{f}_3(x_1, x_2) & \mathsf{f}_4(x_0 :: x_1, x_2, x_3) &\to \mathsf{f} \mathbin{@} x_1 \mathbin{@} (x_2 \mathbin{@} x_3 \mathbin{@} x_0) \mathbin{@} x_2
\end{aligned}
$$

(a) Defunctionalised Applicative Term Rewrite System.

$$
\mathsf{main}(x_0) \to \mathsf{f}([\,], x_0) \qquad \mathsf{f}(x_0, [\,]) \to x_0 \qquad \mathsf{f}(x_0, x_1 :: x_2) \to \mathsf{f}(x_1 :: x_0, x_2)
$$

(b) Simplified First-Order Term Rewrite System.

Figure 6.12: Example Run of HoCA on List Reversal.

We have integrated the functionality of HoCA in the instance tct-hoca. The individual transformations underlying this tool are seamlessly modelled as processors, and its

[3] http://cbr.uibk.ac.at/tools/hoca/

transformation pipeline is naturally expressed in the strategy language. The corresponding strategy, termed `hoca`, is depicted in Figure 6.13. It takes an OCaml source fragment, of type `ML`, and turns it into a term rewrite system.

```
hoca :: Maybe String → Strategy ML TrsProblem
hoca name = mlToAtrs name ⋙ atrsToTrs ⋙ toTctProblem

mlToAtrs :: Maybe String → Strategy ML ATRS
mlToAtrs name = mlToPcf name ⋙ defunctionalise ⋙ try simplifyAtrs

atrsToTrs :: Strategy ATRS TRS
atrsToTrs = try cfa ⋙ uncurryAtrs ⋙ try simplifyTrs
```

Figure 6.13: HoCA Transformation Pipeline modelled in tct-hoca.

First, via `mlToAtrs` the source code is parsed and desugared, the resulting abstract syntax tree is turned into an expression of a typed λ-calculus with constants and fixpoints, akin to Plotkin's PCF [131]. All these steps are implemented via the strategy `mlToPcf :: Maybe String → Strategy ML TypedPCF`. The given parameter, an optional function name, can be used to select the analysed function. The function `defunctionalise :: Strategy TypedPCF ATRS` turns the program into an ATRS, which is simplified via the strategy `simplifyAtrs :: Strategy ATRS ATRS` modelling the heuristics implemented in HoCA.

Second, the strategy `atrsToTrs` uses the control flow analysis provided by HoCA to *instantiate* occurrences of higher-order variables [16]. The instantiated ATRS is then translated into a first-order rewrite system by *uncurrying* all function calls. Further simplifications, as foreseen by the HoCA prototype at this stage of the pipeline, are performed via the strategy `simplifyTrs :: Strategy TRS TRS`.

Currently, all involved processors are implemented via calls to the library shipped with the HoCA prototype, and operate on exported data types. The final strategy in the pipeline, `toTctProblem :: Strategy TRS TrsProblem`, converts HoCA's representation of a TRS to a complexity problem understood by tct-trs.

6.6 Concluding Remarks

In this chapter we have presented a framework for automating resource analysis. The framework is built upon a solid theoretical foundation in form of complexity processors, and has been realised in the complexity tool T$_C$T. We have illustrated several use cases which demonstrates the viability of the framework in practice, among them, the complexity analysis of higher-order functional programs, object-oriented bytecode programs, term rewrite systems and constraint transition systems.

Chapter 7

Conclusion

This thesis is devoted to resource analysis of imperative programs. In this work, we have addressed several aspects on recent developments.

Chapter 3 is concerned with resource analysis of imperative programs, in which the programs of interest conform to intermediate representations with a finite set of integer-valued variables and unstructured control flow. Central to our discussion are the resource analysis tools Loopus, KoAT and paicc, which have been developed in recent years. The latter one is maintained by the author. Due to the nature of the problem, the tools are complex and rely on heuristics. This makes a comparison between the tools, besides experimental evaluation, difficult. In this work, we have mitigated this problem in identifying the abstract program representations that are used internally as driving factor of the different approaches to the analysis. We have recalled and categorised the central concepts of the methods applied, and compared practical aspects between the tools using case studies. In addition, we have summarised known theoretical properties of related program representations from the literature. The properties of interest are termination, bounded termination and polynomial complexity.

Chapter 4 is concerned with resource analysis of imperative programs with heap allocated data structures. We have introduced a simple imperative language with statements for allocating and manipulating records. The presence of aliasing and sharing of data induces additional challenges in establishing quantitative properties of the program state. In this work, we have given a uniform presentation of two different program abstractions. We have recalled the path-length abstraction in which data is abstracted to the maximal number of pointers that is needed to traverse from a variable to the null value. In addition, we have presented a term abstraction in which data allocated on the heap is unfolded to terms. The latter abstraction is motivated by recent developments in automated runtime analysis of term rewrite systems. We have studied both abstractions with a challenging, albeit academic, example.

Chapter 5 is concerned with resource analysis of imperative probabilistic programs. We consider a guarded command language with support for probabilistic sampling and probabilistic choice. In the probabilistic setting, we are interested in averaging the resource usage over all probabilistic branches. This induces additional challenges and limitations in resource analysis. In this work, we have presented a fully automated analysis on the expected runtime and on the expected valuation of a function. The main contribution is a novel approach to modularity. We exploit concave bounding functions to transform a compositional analysis into a modular one. Here, modular indicates that in

our approach all loops of a program can be treated separately as obligations of constraints over expectations. At the time of writing we have developed the first prototype. In the near future, we are concerned with extending the probabilistic primitives that are supported by the prototype.

Finally, Chapter 6 is concerned with a framework for automating resource analysis. The Tyrolean Complexity Tool TCT is a library that helps to design and create new resource analysis tools. At the heart of this framework is a inference system for complexity problems. The implementation of this inference system embodies a transformational approach to resource analysis, which allows to reuse and combine individual instances of the library. Moreover, the implementation exposes an expressive strategy language that facilitates proof search. In this work, we have outlined the software architecture of the TCT library. We have given detailed insights on the implementation of the inference system and strategy language. In addition, we have demonstrated several case studies that exemplify the successful application of the framework. The case studies include resource analysis of higher-order functional programs, object-oriented bytecode programs, term rewrite systems and constraint transition systems.

Bibliography

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. V. Ramírez-Deantes, G. Román-Díez, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *ENTCS*, 258(1):109–121, 2009.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: A Language-Independent Target Language for Cost Analysis. *ENTCS*, 248:31–46, 2009.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *JAR*, 46(2):161–203, 2011.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *TCS*, 413(1):142–159, 2012.

[5] E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *TOCL*, 14(3):22:1–22:35, 2013.

[6] E. Albert, P. Arenas, S. Genaim, and G. Puebla. A Practical Comparator of Cost Functions and Its Applications. *SCP*, 111:483–504, 2015.

[7] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. A Formal Verification Framework for Static Analysis - As well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. *SOSYM*, 15(4): 987–1012, 2016.

[8] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Proc. 16th ATVA*, volume 11138 of *LNCS*, pages 513–520, 2018.

[9] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Proc. of 17th SAS*, volume 6337 of *LNCS*, pages 117–133, 2010.

[10] R. Atkey. Amortised Resource Analysis with Separation Logic. *LMCS*, 7(2), 2011.

[11] M. Avanzini. *Verifying Polytime Computability Automatically*. PhD thesis, University of Innsbruck, 2013.

[12] M. Avanzini and U. D. Lago. Automating Sized-Type Inference for Complexity Analysis. *PACMPL*, 1(ICFP):43:1–43:29, 2017.

[13] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.

[14] M. Avanzini and G. Moser. A Combination Framework for Complexity. *IC*, 248: 22–55, 2016.

[15] M. Avanzini and G. Moser. Complexity of Acyclic Term Graph Rewriting. In *Proc. of 1st FSCD*, volume 52 of *LIPIcs*, pages 10:1–10:18, 2016.

[16] M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Proc. of 20th ICFP*, pages 152–164, 2015.

[17] M. Avanzini, C. Sternagel, and R. Thiemann. Certification of Complexity Proofs using CeTA. In *Proc. of 26th RTA*, volume 36 of *LIPIcs*, pages 23–39, 2015.

[18] M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. of 22nd TACAS*, volume 9636 of *LNCS*, pages 407–423, 2016.

[19] M. Avanzini, U. Dal Lago, and A. Yamada. On Probabilistic Term Rewriting. In *Proc. of 14th FLOPS*, volume 10818 of *LNCS*, pages 132–148, 2018.

[20] M. Avanzini, M. Schaper, and G. Moser. Modular Runtime Complexity Analysis of Probabilistic While Programs. In *Proc. of 3rd DICE-FOPARA*, volume 298 of *EPTCS*, 2019.

[21] M. Avanzini, M. Schaper, and G. Moser. Modular Runtime Complexity Analysis of Probabilistic While Programs. *CoRR*, abs/1908.11343, 2019. Technical Report.

[22] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[23] R. Bagnara and F. Mesnard. Eventual Linear Ranking Functions. In *Proc. of 15th PPDP*, pages 229–238. ACM, 2013.

[24] R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR*, abs/cs/0512056, 2005.

[25] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A New Look at the Automatic Synthesis of Linear Ranking Functions. *IC*, 215:47–67, 2012.

[26] S. Bellantoni and S. A. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2:97–110, 1992.

[27] A. M. Ben-Amram. Size-Change Termination with Difference Constraints. *TOPLAS*, 30(3):16:1–16:31, 2008.

[28] A. M. Ben-Amram. On Decidable Growth-Rate Properties of Imperative Programs. In *Proc. of 1st DICE*, volume 23 of *EPTCS*, pages 1–14, 2010.

[29] A. M. Ben-Amram. Size-Change Termination, Monotonicity Constraints and Ranking Functions. *LMCS*, 6(3), 2010.

[30] A. M. Ben-Amram. Monotonicity Constraints for Termination in the Integer Domain. *LMCS*, 7(3), 2011.

[31] A. M. Ben-Amram and S. Genaim. Ranking Functions for Linear-Constraint Loops. *JACM*, 61(4):26:1–26:55, 2014.

[32] A. M. Ben-Amram and S. Genaim. Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions. In *Proc. of 27th CAV*, volume 9207 of *LNCS*, pages 304–321, 2015.

[33] A. M. Ben-Amram and G. W. Hamilton. Tight Worst-Case Bounds for Polynomial Loop Programs. In *Proc. of 22nd FoSSaCS*, volume 11425 of *LNCS*, pages 80–97, 2019.

[34] A. M. Ben-Amram and L. Kristiansen. On the Edge of Decidability in Complexity Analysis of Loop Programs. *IJFCS*, 23(7):1451–1464, 2012.

[35] A. M. Ben-Amram and C. S. Lee. Program Termination Analysis in Polynomial Time. *TOPLAS*, 29(1):5:1–5:37, 2007.

[36] A. M. Ben-Amram and A. Pineles. Growth-Rate Analysis of Flowchart Programs. Master's thesis, Academic College of Tel-Aviv Yaffo, 2014. URL `http://www2.mta.ac.il/~amirben/projects/aviad_final.pdf`. [Online; accessed 15-September-2019].

[37] A. M. Ben-Amram and A. Pineles. Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate. In *Proc. of 4th VPT*, volume 216 of *EPTCS*, pages 24–49, 2016.

[38] A. M. Ben-Amram and M. Vainer. Bounded Termination of Monotonicity-Constraint Transition Systems. *CoRR*, abs/1202.4281, 2012.

[39] A. M. Ben-Amram, N. D. Jones, and L. Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Proc. of 4th CiE*, volume 5028 of *LNCS*, pages 67–76, 2008.

[40] R. Bird. *Introduction to Functional Programming using Haskell, Second Edition.* Prentice Hall, 1998.

[41] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving Termination Through Conditional Termination. In *Proc. of 23rd TACAS*, volume 10205 of *LNCS*, pages 99–117, 2017.

[42] A. Bouajjani and R. Mayr. Model Checking Lossy Vector Addition Systems. In *Proc. of 16th STACS*, volume 1563 of *LNCS*, pages 323–333, 1999.

[43] O. Bournez and F. Garnier. Proving Positive Almost-Sure Termination. In *Proc. of 16th RTA*, volume 3467 of *LNCS*, pages 323–337, 2005.

[44] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear Ranking with Reachability. In *Proc. of 17th CAV*, volume 3576 of *LNCS*, pages 491–504, 2005.

[45] T. Brázdil, K. Chatterjee, A. Kucera, P. Novotný, D. Velan, and F. Zuleger. Efficient Algorithms for Asymptotic Bounds on Termination Time in VASS. In *Proc. of 33rd LICS*, pages 185–194. ACM, 2018.

[46] M. Brockschmidt, C. Otto, C. v. Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *LNCS*, pages 17–37, 2010.

[47] M. Brockschmidt, C. Otto, and J. Giesl. Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting. In *Proc. of 22nd RTA*, volume 10 of *LIPIcs*, pages 155–170, 2011.

[48] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *Proc. of 2nd FoVeOOS*, volume 7421 of *LNCS*, pages 123–141, 2011.

[49] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated Termination Proofs for Java Programs with Cyclic Data. In *Proc. of 24th CAV*, volume 7358 of *LNCS*, pages 105–122, 2012.

[50] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Proc. of 20th TACAS*, volume 8431 of *LNCS*, pages 140–155, 2014.

[51] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing Runtime and Size Complexity of Integer Programs. *TOPLAS*, 38(4):13:1–13:50, 2016.

[52] P. Cadek, C. Danninger, M. Sinn, and F. Zuleger. Using Loop Bound Analysis For Invariant Generation. In *Proc. of 18th FMCAD*, pages 1–9, 2018.

[53] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *Proc. of 36th PLDI*, pages 467–478, 2015.

[54] Q. Carbonneaux, J. Hoffmann, T. W. Reps, and Z. Shao. Automated Resource Analysis with Coq Proof Objects. In *Proc. of 29th CAV*, volume 10427 of *LNCS*, pages 64–85, 2017.

[55] T. Colcombet, L. Daviaud, and F. Zuleger. Size-Change Abstraction and Max-Plus Automata. In *Proc. of 39th MFCS*, volume 8634 of *LNCS*, pages 208–219, 2014.

[56] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear Invariant Generation Using Non-linear Constraint Solving. In *Proc. of 15th CAV*, volume 2725 of *LNCS*, pages 420–432, 2003.

[57] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *Proc. of 27th PLDI*, pages 415–426, 2006.

[58] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[59] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4th POPL*, pages 238–252, 1977.

[60] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. of 5th POPL*, pages 84–96, 1978.

[61] K. Crary and S. Weirich. Resource Bound Certification. In *Proc. of 27th POPL*, pages 184–198, 2000.

[62] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proc. of 12th USENIX*, 2003.

[63] N. Danner, J. Paykin, and J. S. Royer. A Static Cost Analysis for a Higher-order Language. In *Proc. of 7th PLPV*, pages 25–34, 2013.

[64] S. K. Debray and N. Lin. Cost Analysis of Logic Programs. *TOPLAS*, 15(5): 826–875, 1993.

[65] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of PLDI'90*, pages 174–188, 1990.

[66] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM*, 18(8):453–457, 1975.

[67] S. Falke and D. Kapur. A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In *Proc. of 22nd CADE*, volume 5663 of *LNCS*, pages 277–293, 2009.

[68] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *Proc. of 22nd RTA*, volume 10 of *LIPIcs*, pages 41–50, 2011.

[69] D. Fenacci and K. MacKenzie. Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic. *ENTCS*, 279(1):19–32, 2011.

[70] T. Fiedor, L. Holík, A. Rogalewicz, M. Sinn, T. Vojnar, and F. Zuleger. From Shapes to Amortized Complexity. In *Proc. of 19th VMCAI*, volume 10747 of *LNCS*, pages 205–225, 2018.

[71] L. M. F. Fioriti and H. Hermanns. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proc. of 42nd POPL*, pages 489–501, 2015.

[72] A. Flores-Montoya. Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations. In *Proc. of 21st FM*, volume 9995 of *LNCS*, pages 254–273, 2016.

[73] A. Flores-Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Darmstadt University of Technology, Germany, 2017.

[74] A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Proc. of 12th APLAS*, volume 8858 of *LNCS*, pages 275–295, 2014.

[75] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[76] F. Frohn and J. Giesl. Complexity Analysis for Java with AProVE. In *Proc. of 13th IFM*, volume 10510 of *LNCS*, pages 85–101, 2017.

[77] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proc. of 10th SAT*, volume 4501 of *LNCS*, pages 340–354, 2007.

[78] S. Genaim and D. Zanardini. Reachability-Based Acyclicity Analysis by Abstract Interpretation. *TCS*, 474:60–79, 2013.

[79] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *TOPLAS*, 33(2): 7:1–7:39, 2011.

[80] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic Evaluation Graphs and Term Rewriting - A General Methodology for Analyzing Logic Programs. In *Proc. of 22nd LOPSTR*, volume 7844 of *LNCS*, page 1, 2012.

[81] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proc. of 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.

[82] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing Program Termination and Complexity Automatically with AProVE. *JAR*, 58(1):3–31, 2017.

[83] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static Analysis of Energy Consumption for LLVM IR Programs. In *Proc. of 18th SCOPES*, pages 12–21, 2015.

[84] S. Gulwani. SPEED: Symbolic Complexity Bound Analysis. In *Proc. of 21st CAV*, volume 5643 of *LNCS*, pages 51–62, 2009.

[85] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. of 27th PLDI*, pages 376–386, 2006.

[86] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Proc. of 31st PLDI*, pages 292–304, 2010.

[87] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of 36th POPL*, pages 127–139, 2009.

[88] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. *FMSD*, 41(1):83–106, 2012.

[89] E. Hainry and R. Péchoux. Objects in Polynomial Time. In *Proc. of 14th APLAS*, pages 387–404, 2015.

[90] E. Hainry and R. Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. *IC*, 261:78–115, 2018.

[91] M. V. Hermenegildo, G. Puebla, F.Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization using Abstract Interpretation (and the Ciao System Preprocessor). *SCP*, 58(1-2):115–140, 2005.

[92] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5196 of *LNCS*, pages 364–379, 2008.

[93] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10): 576–580, 1969.

[94] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proc. of 19th ESOP*, volume 6012 of *LNCS*, pages 287–306, 2010.

[95] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In *Proc. of 24rd CAV*, volume 7358 of *LNCS*, pages 781–786, 2012.

[96] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. of 30th POPL*, pages 185–197, 2003.

[97] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *Proc. of 15th ESOP*, volume 3924 of *LNCS*, pages 22–37, 2006.

[98] M. Hofmann and G. Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Proc. of RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 272–286, 2014.

[99] M. Hofmann and D. Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Proc. of 18th CSL*, volume 5771 of *LNCS*, pages 317–331, 2009.

[100] M. Hofmann and D. Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *Proc. 22nd ESOP*, volume 7792 of *LNCS*, pages 593–613, 2013.

[101] J. Hopcroft and J. Pansiot. On the Reachability Problem for 5-dimensional Vector Addition Systems. *TCS*, 8(2):135 – 159, 1979.

[102] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proc. of 21st CAV*, volume 5643 of *LNCS*, pages 661–667, 2009.

[103] N. D. Jones and L. Kristiansen. A Flow Calculus of *mwp*-bounds for Complexity Analysis. *TOCS*, 10(4):28:1–28:41, 2009.

[104] S. Kahrs. The Primitive Recursive Functions are Recursively Enumerable, 2008.

[105] B. L. Kaminski, J. Katoen, C. Matheja, and F. Olmedo. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Proc. of 25th ESOP*, volume 9632 of *LNCS*, pages 364–389, 2016.

[106] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. W. Reps. Compositional Recurrence Analysis Revisited. In *Proc. of 38th PLDI*, pages 248–262, 2017.

[107] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *TOPLAS*, 28(4):619–695, 2006.

[108] L. Kristiansen and K. Niggl. On the Computational Complexity of Imperative Programming Languages. *TCS*, 318(1-2):139–161, 2004.

[109] D. Lankford. On Proving Term Rewriting Systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, 1979.

[110] C. S. Lee, N. D.Jones, and A. M.Ben-Amram. The Size-Change Principle for Program Termination. In *Proc. of 28th POPL*, pages 81–92, 2001.

[111] J. Leike and M. Heizmann. Ranking Templates for Linear Loops. *LMCS*, 11(1), 2015.

[112] J. Leroux and P. Schnoebelen. On Functions Weakly Computable by Petri Nets and Vector Addition Systems. In *Proc. of 8th RP*, volume 8762 of *LNCS*, pages 190–202, 2014.

[113] S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic Numeric Abstractions for Heap-Manipulating Programs. In *Proc. of 37th POPL*, pages 211–222, 2010.

[114] J.-Y. Marion. A Type System for Complexity Flow Analysis. In *Proc. of 26th LICS*, pages 123–132, 2011.

[115] J.-Y. Marion and R. Péchoux. Analyzing the Implicit Computational Complexity of Object-Oriented Programs. In *Proc. of 38th FSTTCS*, volume 2 of *LIPIcs*, pages 316–327, 2008.

[116] A. R. Meyer and D. M. Ritchie. The Complexity of Loop Programs. In *Proc. of 22nd NC*, pages 465–469, 1967.

[117] A. Miné. The Octagon Abstract Domain. *HOSC*, 19(1):31–100, 2006.

[118] G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

[119] G. Moser and M. Schaper. From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation. *IC*, 261:116–143, 2018.

[120] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[121] J.-Y. Moyen. Resource Control Graphs. *TOCL*, 10(4):29:1–29:44, 2009.

[122] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. Complexity Analysis for Term Rewriting by Integer Transition Systems. In *Proc. of 11th FroCoS*, volume 10483 of *LNCS*, pages 132–150, 2017.

[123] J. A. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. *ENTCS*, 253(5):65–82, 2009.

[124] N. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. of 39th PLDI*, pages 496–512, 2018.

[125] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[126] H. R. Nielson. A Hoare-Like Proof System for Analysing the Computation Time of Programs. *SCP*, 9(2):107–136, 1987.

[127] K. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM*, 35(5):1122–1147, 2006.

[128] F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. Reasoning about Recursive Probabilistic Programs. In *Proc. of 31st LICS*, pages 672–681, 2016.

[129] C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 259–276, 2010.

[130] S. Panitz and M. Schmidt-Schauß. TEA: Automatically Proving Termination of Programs in a Non-strict Higher-Order Functional Language. In *Proc. of 4th SAS*, volume 1302 of *LNCS*, pages 345–360, 1997.

[131] G. D. Plotkin. LCF Considered as a Programming Language. *TCS*, 5(3):223–255, 1977.

[132] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proc. 5th VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004.

[133] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. *TOPLAS*, 29(3):15, 2007.

[134] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *HOSC*, 11(4):363–397, 1998.

[135] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of 17th LICS*, pages 55–74, 2002.

[136] X. Rival and L. Mauborgne. The Trace Partitioning Abstract Domain. *TOPLAS*, 29(5), 2007.

[137] M. Rosendahl. Automatic Complexity Analysis. In *Proc. of 4th FPCA*, FPCA '89, pages 144–156, 1989.

[138] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Proc. of 7th VMCAI*, volume 3855 of *LNCS*, pages 95–110, 2006.

[139] M. Schaper. Automated Resource Analysis with paicc (Extended Abstract). `http://cl-informatik.uibk.ac.at/users/zini/events/dice18/abstracts/S.pdf`, 2018. [Online; accessed 15-September-2019].

[140] M. Schaper and G. Moser. On Abstract Program Representations for Automated Resource Analysis, 2018. Resubmitted.

[141] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1999.

[142] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of 12th SAS*, volume 3672 of *LNCS*, pages 320–335, 2005.

[143] A. Serrano, P. López-García, and M. V. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP*, 14(4-5): 739–754, 2014.

[144] M. Sinn. *Automated Complexity Analysis for Imperative Programs*. PhD thesis, TU Wien, Faculty of Informatics, Wien, Austria, 2016. URL `http://katalog.ub.tuwien.ac.at/AC13356888`.

[145] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.

[146] M. Sinn, F. Zuleger, and H. Veith. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *JAR*, 59(1):3–45, 2017.

[147] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[148] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *JACM*, 32(3): 652–686, 1985.

[149] F. Spoto. The Julia Static Analyzer for Java. In *Proc. of 23rd SAS*, volume 9837 of *LNCS*, pages 39–57, 2016.

[150] F. Spoto, F. Mesnard, and E. Payet. A Termination Analyzer for Java Bytecode based on Path-Length. *TOPLAS*, 32(3):8:1–8:70, 2010.

[151] R. E. Tarjan. Amortized Computational Complexity. *SIAM*, 6(2):306–318, 1985.

[152] D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *JCS*, 4(2/3):167–188, 1996.

[153] P. Wang, D. Wang, and A. Chlipala. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *PACMPL*, 1:79:1–79:26, 2017.

[154] B. Wegbreit. Mechanical Program Analysis. *CACM*, 18(9):528–539, 1975.

[155] B. Wegbreit. Verifying Program Performance. *JACM*, 23(4):691–699, 1976.

[156] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS*, 7(3):1–53, 2008.

[157] F. Zuleger. Asymptotically Precise Ranking Functions for Deterministic Size-Change Systems. In *Proc. of 10th CSR*, volume 9139 of *LNCS*, pages 426–442, 2015.

[158] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Proc. 18th SAS*, volume 6887 of *LNCS*, pages 280–297, 2011.