Decidability of Linear Tree Constraints for Resource Analysis of Object-oriented Programs

Sabine Bauer

München 2018

Decidability of Linear Tree Constraints for Resource Analysis of Object-oriented Programs

Sabine Bauer

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Sabine Bauer

München, den 8.11.2018

Erstgutachter: Prof. Dr. Dirk Beyer

Zweitgutachter: Prof. Dr. Jan Hoffmann

Tag der mündlichen Prüfung: 24.5.2019

Abstract

Program analysis aims to predict program behaviors already at compile time. For instance, one could ask how many memory resources are required or in which range the program variables take their values. Some of these questions are undecidable, which means no algorithm can exist that always returns for an arbitrary program as input the answer whether it has the property or not.

The property of interest in this thesis is the ability to execute a program within a certain amount of memory. Moreover, the obtained resource bounds are formally guaranteed. The examined programming language is a fragment of Java, called RAJA (Resource Aware Java), introduced and explored by Hofmann, Jost and Rodriguez. The problem of computing such resource bounds for RAJA can be reduced to satisfiability of linear tree constraints by type-based amortized analysis. Any solution to the system of constraints derived from a program leads to a guaranteed upper bound on the resource usage of that program.

These constraints are inequalities that concern infinite trees of arbitrary finite degree, having node labels being rational or real numbers or a symbol for "infinity". They generalize linear programming to an infinite number of variables, where inequality between trees is understood pointwise. A heuristic procedure for trees that have only finitely many different subtrees has been described in previous work. However, this limits the expressivity of resource bounds to linear functions. In order to obtain polynomial bounds, as often required in practice, we investigate the general case.

We first prove that the problem in its general formulation is at least as hard as the famous Skolem-Mahler-Lech problem, which is NP-hard and whose decidability status is still unknown. Therefore, we examine the procedure in the literature that generates constraints from RAJA code and are able to delineate an interesting subcase, having a concise formulation, that already covers all instances that appear when deriving the

constraints from realistic programs.

We prove that satisfiability of linear tree constraints in this subcase is decidable. Previously, it had been unknown whether such a system has a solution at all and if so, how it behaves mathematically, say, if the numbers in the nodes are constant, polynomially or exponentially growing with increasing levels, but also how to represent such an infinite tree in a finite way. We describe a procedure that decides satisfiability of the inequalities and we classify the growth rates of their minimal solutions.

Thus, we have the theoretical basis to automatically verify the behavior of realistic programs and to automatically calculate bounds on the amount of resources that arbitrary RAJA programs require. No interaction with the user is necessary.

Zusammenfassung

In der Analyse von Programmen geht es häufig darum, zur Kompilierzeit vorauszusagen, wie sich ein Programm verhalten wird, wenn man es ausführt. Man kann sich beispielsweise die Frage stellen, wieviele Resourcen es benötigt, ob es immer terminiert, in welchen Wertebereich sich die Programmvariablen bewegen und vieles mehr. Manche dieser Fragestellungen sind nicht entscheidbar, d.h. es kann keinen Algorithmus geben, der für jedes beliebige Programm als Eingabe die korrekte Antwort ausgibt, ob es die gewünschte Eigenschaft erfüllt oder nicht.

Diese Arbeit geht aus von der Frage, mit wieviel Speicherplatz ein objektorientiertes Programm garantiert auskommt. Betrachtet wird ein Fragment von Java mit dem Namen RAJA (Resource Aware JAva), das von Hofmann, Jost und Rodriguez eingeführt und untersucht wurde. Diese Problemstellung kann — mittels typenbasierter amortisierter Analyse — auf die Erfüllbarkeit spezieller Baumungleichungen zurückgeführt werden. Das heißt, wenn man eine Lösung des zu einem Programm gehörigen Ungleichungssystems findet, dann gibt es eine daraus berechenbare obere Schranke an den Platzverbrauch.

Die erwähnten Ungleichungen beziehen sich auf unendliche Bäume mit rationalen oder reellen Zahlen oder einem Symbol für "unendlich" in den Knoten und einem beliebigen endlichen Verzweigungsgrad. Sie sind punktweise zu verstehen und eine Möglichkeit, lineare Optimierung auf unendlich viele Variablen zu verallgemeinern. Für Bäume, die nur endlich viele verschiedene Teilbäume besitzen, wurde in Vorarbeiten eine heuristische Entscheidungsprozedur beschrieben. Dies erlaubt jedoch nur die Herleitung linearer Schranken an den Resourcenverbrauch.

Hier beginnen die in dieser Arbeit dargelegten Untersuchungen. Bisher war es in den meisten Fällen nicht klar, ob ein solches System eine Lösung hat, und wenn ja, wie sich diese Lösung mathematisch verhält, etwa ob die Zahlen in den Knoten mit steigendem

Level konstant, polynomiell oder exponentiell anwachsend sind, aber auch wie es möglich ist, einen solchen unendlichen Baum endlich darzustellen.

Wir beweisen, dass das Problem in seiner allgemeinen Version aus den Vorarbeiten bereits in einfachen Fällen mindestens so schwer ist wie das berühmte Skolem-Mahler-Lech Problem, von dem bislang nicht bekannt ist, ob es entscheidbar ist. Man weiß aber, dass es NP-schwer und damit aller Wahrscheinlichkeit nach nicht effizient lösbar ist.

Aus diesem Grund konzentrieren wir uns auf einen Spezialfall, der für unsere Zwecke gerechtfertigt ist, da es der einzige auftretende Fall ist, wenn die Ungleichungen von tatsächlichen Programmen abgeleitet werden, und der eine sehr einfache Formulierung hat. Wir zeigen, dass die Frage der Erfüllbarkeit linearer Baumconstraints von diesem Format entscheidbar ist. Wir geben einen Algorithmus an, der in allen Fällen die Erfüllbarkeit der Ungleichungen entscheidet und gleichzeitig das Wachstumsverhalten der optimalen (das sind die minimalen) Lösungen näher klassifiziert. Damit können wir entscheiden, ob diese Lösungen polynomiell wachsen oder exponentiell.

Als Ergebnis haben wir nun das theoretische Fundament, um mehr mögliche Programmverhalten automatisch zu verifizieren und Rückschlüsse auf den Resourcenverbrauch zu ziehen. Man benötigt hierzu keine Eingaben des Anwenders und es gibt keine Einschränkungen an die Gestalt der Lösungen — solange sie alle Constraints erfüllen.

Acknowledgement

First, I want to mention that my greatest thanks goes to Martin Hofmann who supervised and constantly supported me throughout my time at LMU. He died in an accident in early 2018. I am very grateful that I knew Martin as a person and also had the possibility to learn from him.

Great thanks to Dirk Beyer that he agreed to be my supervisor in Martin's place. The same holds for Steffen Jost, who also supervised me and supported me a lot.

Thanks to Jan Hoffmann for being the external assessor and for giving me the opportunity to present my work in his group at Carnegie-Mellon-University.

I also thank François Bry for being the chairman of the examination commitee and all members of the PUMA project (DGF Gradiuertenkolleg 1480, Program and Model Analysis), especially Helmut Seidl, for their interest and feedback.

I thank all my colleagues at LMU for everything. In particular, special thanks to Brigitte Pientka for her advice and explanations. The same holds for David Sabel, Andreas Abel and Ulrich Schöpp and for my friends Serdar Erbatur and Melanie Kreidenweis. Thank you for all our helpful and inspiring scientific and private discussions!

Thanks to my family and friends, in particular my amazing husband Johannes, who is always there for me and our children Daniel and Elias, and my parents Marianne and Franz, who never hesitated to pick up our children or help us when our time was short.

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Sabine Bauer

München, den 8.11.2018

Contents

List of Figures

Chapter 1

Introduction

Amortized Analysis is a powerful technique that provides guaranteed worst-case bounds for sequences of operations that are tighter than the sum of the worst-case costs of individual instructions. In this context, the costs are measured with several discrete resource metrics, for instance resource consumption such as memory usage, execution steps, etc. .

This goal is achieved by encoding resource information that abstracts all possible program states into a type system. Such a type system consists of *refined types* (i.e. types with "resource annotations") which were specially designed for resource analysis. Finding suitable state abstractions is complicated and is highly specific to the data-structures that are used. Most of the work previous to this thesis were concerned with purely functional programming languages. Adapting the automatic amortized analysis to widely used object-oriented imperative programming, such as Java, turned out to be difficult: even in the most simple case, type annotations being simple rational numbers in the functional world, turn into infinite trees of rational numbers. Moving from the initial verification of given bounds to a fully automated inference was achieved by Dulma Rodriguez [2, 3, 4]. The arising linear tree constraints, that appeared in the type inference algorithm and that could not yet be solved in general, are subject to this thesis. We present a decision procedure for them.

Organization of this chapter In Section 1.1, we formally define the list constraint problem, its syntax and what we mean by decidability. In Section 1.2, we explain the context in which this decidability problem was discovered and how we can apply its solution to analyze the resource requirements of Java-like programs. In section 1.3, we give an

overview of the contributions of this thesis.

1.1 Problem Explanation

The main subject of this thesis can be summarized in the following sentence:

It is about *satisfiability* of *linear pointwise inequalities* between *infinite lists and trees of real or rational numbers*.

We will now (in this chapter) explain what this means and why it is interesting to solve them, which is finally possible.

Let us for now just consider the following problem as given. In the next section we will explain how it is motivated from program analysis.

List constraints describe relations between variables that can be instantiated with infinite lists. Infinite lists are sequences of numbers in $\mathbb{Q}_0^+ \cup \{\infty\}$ or in a set $\mathbb{D}$, which consists of nonnegative reals and infinity. They are denoted by (possibly indexed) lower case letters $\mathbf{x}, \mathbf{y}, \ldots$ and have three operations: we can extract the number in the head of the list (using the **head**-operation), add two lists together pointwise and remove the head of the list to obtain another infinite list by the **tail**-operation.

The constraints are now pointwise inequalities between sums of such lists together with arithmetic conditions on selected entries at the beginning of the lists. The arithmetic constraints have the form of an ordinary linear program, but the list constraints encode infinitely many arithmetic inequalities.

Infinite trees are defined similarly, but with the difference that we have a finite number of operations like **tail**, that return the direct subtrees. The head node is called **root** in the tree case. Formally, let B be a finite set of branch labels. An infinite tree $T_{\mathbb{D}}^B$ with branches in B and labels in $\mathbb{D}$ is a map from $B^* \to \mathbb{D}$ (with B^* being all words over B). A tree s is a subtree of a tree t if there is $w \in B^*$ such that $s(p) = t(wp)$ for all $p \in B^*$. The definition of $T_{\mathbb{Q}_0^+ \cup \{\infty\}}^B$ is analogous.

We extend the constraints to *tree constraints*. For that, we take the set $B = \{\mathbf{l_1}, \ldots, \mathbf{l_n}\}$ of branch labels to replace the **tail** label from the list case. Variables then range over infinite

$$
\begin{aligned}
l &::= \mathbf{x} | \mathbf{tail}(l) && \text{(Atomic list)} \\
t &::= l | t + t && \text{(List term)} \\
c &::= t \geq t && \text{(List constraint)}
\end{aligned}
$$

Figure 1.1: Linear List Constraint Syntax

$$
\begin{aligned}
t &::= \mathbf{x} | \mathbf{l}(t), \text{ where } \mathbf{l} \in B \text{ with } |B| < \infty && \text{(Atomic tree)} \\
te &::= t | te + te && \text{(Tree term)} \\
c &::= te \geq te && \text{(Tree constraint)}
\end{aligned}
$$

Figure 1.2: Linear Tree Constraint Syntax

trees of degree n and labeled with numbers in $\mathbb{D}$ or $\mathbb{Q}_0^+ \cup \{\infty\}$. We write $l_i(\mathbf{x})$ for the i-th immediate subtree of the tree $\mathbf{x}$ and $\Diamond(\mathbf{x})$ for the root label of $\mathbf{x}$.

The constraints are build up using these operations and then comparing the results with the $\geq$ operator. We can nest the operations in any order that we want, if we define that the head of a sum of lists or trees is the sum of their heads and the same rule for the tails or subtrees (see below, (1.1), (1.2)).

Figures 1.1 and 1.2 present the formal version of the above description.

We do not make a difference in the notation for tree variables and concrete trees in $T_{\mathbb{D}}^B$, but use $\mathbf{x}, \mathbf{y}, \mathbf{z}, \ldots$ for both. When we talk of solutions, we always mean concrete trees, i.e. assignments of an infinite tree with nonnegative rational or real numbers that satisfy the inequalities stated by the constraints (for a tree variable).

An example of a list constraint is $\mathbf{tail}(\mathbf{tail}(\mathbf{x})) \geq \mathbf{tail}(\mathbf{x}) + \mathbf{x}$, which states that the unknown list $\mathbf{tail}(\mathbf{tail}(\mathbf{x}))$ is growing at least as fast as the Fibonacci sequence (depending on the initial variables $\mathbf{head}(\mathbf{x})$ and $\mathbf{head}(\mathbf{tail}(\mathbf{x}))$). The pointwise inequalities state not only that the i'th position of the list $\mathbf{x}$ has to be pointwise greater than or equal to i'th position of the Fibonacci list with starting variables $\mathbf{head}(\mathbf{x})$ and $\mathbf{head}(\mathbf{tail}(\mathbf{x}))$ for all i. They state something stronger, namely that in each position i the entry there has to be greater than or equal to the sum of the previous two entries. For example, the list $1, 1, 2, 99, 5, 8, 13, \ldots$

$v ::= n|\lambda|\mathbf{head}(l)$ (Arithmetic expression (number, variable or head of an atomic list))
$h ::= v|h + h$ (Head term)
$c ::= h \geq h$ (Arithmetic constraint)

Figure 1.3: Arithmetic Constraint Syntax

is greater than a Fibonacci sequence with initial values 1 and 1. But it doesn't fulfill the constraints since 5 is not greater than 2+99. The minimal continuation of this list beginning at index 5 is $101, 200, 301, 501, 802, \ldots$.

In addition to the list and tree constraints, we have arithmetic constraints given for the first element up to level n, that take the form of an arbitrary linear program with integer coefficients. Syntactically, they are the same as tree constraints with the difference that they can include numbers and hold only for the roots, which are arithmetic variables. An example for an arithmetic list constraint is $\mathbf{head}(\mathbf{x}) + 3 * \mathbf{head}(\mathbf{y}) >= 2 + \mathbf{head}(\mathbf{tail}(\mathbf{z}))$. Figure 1.3 gives the syntax for them.

We also include the rules for roots of sums and subtrees of sums:

$$\Diamond(\mathbf{x} + \mathbf{y}) = \Diamond(\mathbf{x}) + \Diamond(\mathbf{y}), \tag{1.1}$$

$$\mathbf{l}(\mathbf{x} + \mathbf{y}) = \mathbf{l}(\mathbf{x}) + \mathbf{l}(\mathbf{y}). \tag{1.2}$$

This will become useful later, when we manipulate tree constraints by *unfolding* (i.e. application of $\Diamond$ and labels to constraints together with an application of these rules, see Chapters 4 and 6).

With satisfiability we mean the possibility to assign values to the tree variables such that the constraints hold for these concrete trees. For all finite sets of arithmetic constraints, this is just the question whether a linear program has a solution. For the list constraints, we observe that each pointwise list inequality corresponds to infinitely many "ordinary" inequalities as they appear in a linear program. We also observe that there are three possibilities for the number of solutions: either, the entire solution list is defined uniquely or there are infinitely many different solutions. In the latter case, if the list is uniquely defined up to the initial values (i.e. only the initial values allow variations), then there are countably many solutions over $\mathbb{Q}_0^+$ and uncountably many over $\mathbb{D}$. Else, there are uncountably many

solutions over both $\mathbb{Q}_0^+$ and $\mathbb{D}$. Here are examples for constraint systems C_i in all cases: C_1 has countably many solutions over the rational numbers, C_2 has only one solution and C_3 has uncountably many solutions because the list is of infinite length and in each position there may be an arbitrary aberration from the bound.

$$C_1 = \{\mathbf{tail}(\mathbf{x}) \geq 2\mathbf{x} \wedge \mathbf{tail}(\mathbf{x}) \leq 2\mathbf{x}\}$$
$$C_2 = C_1 \cup \{\mathbf{head}(\mathbf{x}) = 1\}$$
$$C_3 = \{\mathbf{tail}(\mathbf{x}) \geq 2\mathbf{x}\}$$

We study satisfiability of such linear list constraint systems in general and in a special situation. This special situation is needed for the part of the thesis that gives a decision procedure for the satisfiability question and can be characterized as follows: We will restrict our attention to list constraint systems, where we have only one summand on the greater side of the inequalities, as they are formed according to the syntax. This is equivalent to not allowing negative summands on the side that is less or replacing the rule (List constraint) by $c ::= l \geq t$ (resp. (Tree constraint) by $c ::= t \geq te$). As we will show, this covers all cases that can be motivated by program analysis (see Sections 1.2 and 4).

Example 1. *The following constraint system is satisfiable:*

Arithmetic Constraints	*List Constraints*
$\mathbf{head}(\mathbf{x}) = \mathbf{head}(\mathbf{z}) = 2$	$\mathbf{tail}(\mathbf{y}) \geq \mathbf{y}$
$\mathbf{head}(\mathbf{tail}(\mathbf{x})) = 5$	$\mathbf{z} \geq \mathbf{tail}(\mathbf{z}) + \mathbf{tail}(\mathbf{z})$
$\mathbf{head}(\mathbf{y}) \geq 1$	$\mathbf{tail}(\mathbf{tail}(\mathbf{x}))) \geq \mathbf{tail}(\mathbf{x}) + \mathbf{x} + \mathbf{tail}(\mathbf{y}) + \mathbf{tail}(\mathbf{y})$

and has the equivalent formulation in array notation

Arithmetic Constraints	*List Constraints*
$\mathbf{x}[0] = \mathbf{z}[0] = 2$	$\mathbf{y}[i+1] \geq \mathbf{y}[i] \forall i \geq 0$
$\mathbf{x}[1] = 5$	$\mathbf{z}[i] \geq \mathbf{z}[i+1] + \mathbf{z}[i+1] \forall i \geq 0$
$\mathbf{y}[0] \geq 1$	$\mathbf{x}[i+2] \geq \mathbf{x}[i+1] + \mathbf{x}[i] + \mathbf{y}[i+1] + \mathbf{y}[i+1] \forall i \geq 0$

It has the (minimal) solutions $\mathbf{y}$ *a constant list,* $\mathbf{z}$ *a list with first element 2 and the rest zero and* $\mathbf{x}$ *a "Fibonacci list" as above, with an additional summand.*

$$\mathbf{y} = 1, 1, \ldots, \ \mathbf{z} = 2, 0, 0, \ldots,$$
$$\mathbf{x} = 2, 5, 9, 16, 27, 45, .., \mathbf{x}_{n-1}, \mathbf{x}_n, \mathbf{x}_n + \mathbf{x}_{n-1} + 2, \ldots \ .$$

If we add the constraint $\mathbf{x} \leq \mathbf{z}$*, (i.e.* $\mathbf{x}[i] \leq \mathbf{z}[i], \forall i \geq 0$*) the constraints become unsatisfiable, since a maximal solution to the list constraint on* $\mathbf{z}$ *is an exponentially decreasing list*

$$\mathbf{z} = 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots, \frac{1}{2^n}, \ldots \ .$$

These two constraint systems are in the above mentioned special form and thus can be treated by our algorithm.

We remark that tree constraints without arithmetic constraints are always satisfiable with infinite lists of zeros.

Example 2. *Let* $\mathbf{l}, \mathbf{r}$ *denote the labels of a binary tree.*

1. *The constraint system*

$$\Diamond(\mathbf{t_1}) = \Diamond(\mathbf{t_2}) = 1 \ \wedge \ \mathbf{r}(\mathbf{t_1}) = \mathbf{r}(\mathbf{t_2}) + \mathbf{r}(\mathbf{t_2}) \ \wedge \ \mathbf{l}(\mathbf{t_2}) + \mathbf{l}(\mathbf{t_2}) = \mathbf{l}(\mathbf{t_1})$$

   where the $\Diamond(\cdot)$ *symbol denotes the variable in the root of a tree, admits a solution with only finitely many different subtrees (namely a tree consisting only of 1s, and a tree with root number 1 and two subtrees with only 2s),*

2. *and the system consisting of the arithmetic constraint* $\Diamond(\mathbf{t_1}) = 1$ *and the tree constraints*

$$\mathbf{r}(\mathbf{t_1}) \geq 2\mathbf{t_1}, \mathbf{l}(\mathbf{t_1}) \geq 3\mathbf{t_1}, \mathbf{t_1} \geq \mathbf{l}(\mathbf{r}(\mathbf{t_1})),$$

   is unsatisfiable, because it implies

$$1 = \Diamond(\mathbf{t_1}) \geq \Diamond(\mathbf{l}(\mathbf{r}(\mathbf{t_1}))) \geq 2\Diamond(\mathbf{l}(\mathbf{t_1})) \geq 6\Diamond(\mathbf{t_1}) = 6.$$

3. *The system*

$$\Diamond(\mathbf{t_2}) = 1, \mathbf{r}(\mathbf{t_2}) = 2\mathbf{t_2}, \mathbf{l}(\mathbf{t_2}) = 3\mathbf{t_2}$$

   has the solution in Figure 1.4. This tree has infinitely many different subtrees; going right means duplicating the root of the subtree and going left means multiplying by

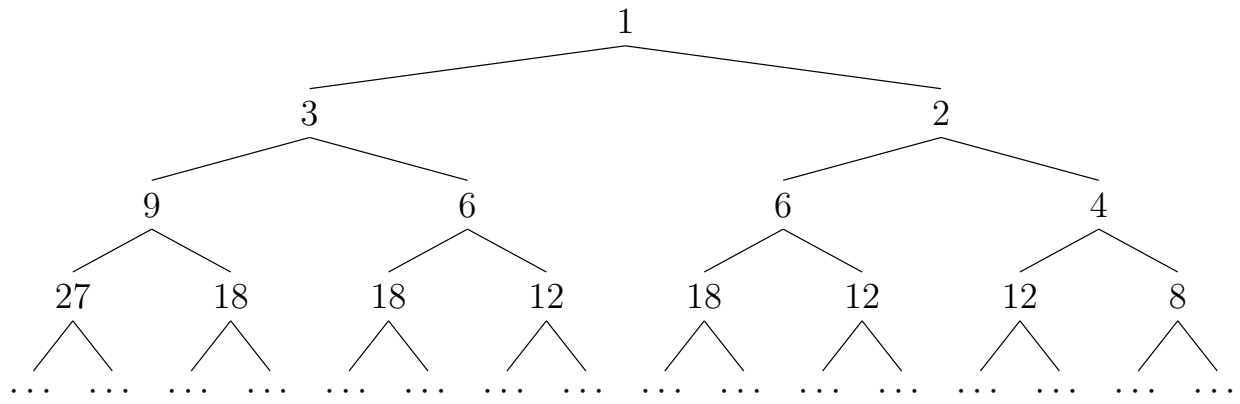


Figure 1.4: Tree with an Infinite Number of Different Subtrees

three:

$$\forall w \in (\mathbf{l}|\mathbf{r})^* : \Diamond(w(\mathbf{t_2})) = 2^i 3^j,\ i = \textit{number of } \mathbf{r}\textit{'s in } w,\ j = \textit{number of } \mathbf{l}\textit{'s in } w.$$

Remark 1. *We can derive new valid constraints by combining existing constraints and by unfolding. Unfolding results in equivalent constraints, whereas combinations of constraints by transitivity do not replace the existing constraints equivalently.*

Example 3.

- *The constraints*

$$\mathbf{r}(\mathbf{r}(\mathbf{x})) \geq \mathbf{r}(\mathbf{x}) + \mathbf{r}(\mathbf{x}),\ \mathbf{r}(\mathbf{x}) \geq \mathbf{x} + \mathbf{x} \tag{1.3}$$

imply together

$$\mathbf{r}(\mathbf{r}(\mathbf{x})) \geq \mathbf{x} + \mathbf{x} + \mathbf{x} + \mathbf{x}, \tag{1.4}$$

but not vice versa. The list $1, 1, 4, 4, 16, 16, \ldots$ *is a solution to (1.4), but not to (1.3).*

- *The constraint*

$$\mathbf{r}(\mathbf{r}(\mathbf{l}(\mathbf{x}))) \geq \mathbf{x} + \mathbf{y}$$

is equivalent to its unfolded version, which is the system

$$\begin{aligned}\mathbf{r}(\mathbf{r}(\mathbf{r}(\mathbf{l}(\mathbf{x})))) &\geq \mathbf{r}(\mathbf{x}) + \mathbf{r}(\mathbf{y}),\\ \mathbf{l}(\mathbf{r}(\mathbf{r}(\mathbf{l}(\mathbf{x})))) &\geq \mathbf{l}(\mathbf{x}) + \mathbf{l}(\mathbf{y}),\\ \Diamond(\mathbf{r}(\mathbf{r}(\mathbf{l}(\mathbf{x})))) &\geq \Diamond(\mathbf{x}) + \Diamond(\mathbf{y}).\end{aligned}$$

Both operations will be used in Chapter 6.

According to the problem description, the list constraint satisfiability question is a generalization of the feasibility question for linear programming to infinitely many unknowns. This is already a reason for investigations about decidability etc., but there is also a direct implication for program analysis. The decidable subcase of the problem corresponds to inference of resource types in an object-oriented program, which will be the topic of the next section.

1.2 Motivation from Programming

This thesis presents a procedure to solve linear tree constraints as introduced in the previous section. In this section, we explain informally where the constraints come from —namely amortized analysis — and how they are generated from programs.

An Example for the use of amortized analysis We give intuition for the analysis in the example of a doubly linked list that is constructed from a singly linked list and then transformed back again into a singly linked list. The purpose of the example is just showing how the analysis adapts to changes of data structures during a computation. As usual in amortized analysis, we assign a nonnegative value, called the *potential* to each access path that leads to a certain object. In the example in Figure 1.5, there are infinitely many different (circular) access paths in the doubly linked list, most of them having potential zero. Figuratively speaking, the potential can be considered as the amount of dollars at hand for this object and all operations that are invoked for it must be payed from its potential (e.g. allocating a new object, which is defined to have cost 1).

The design of this analysis ensures that inspite of aliasing and update, there is no unnecessary

doubling or multiplying of the potential. The reason that makes this work is that it carries out a balancing using typing rules. This will become clearer in the next chapter.

The potential of the input list depends on its length and can be defined only via the access paths. This is illustrated in Figure 1.5: in step 1, we see this singly linked list consisting of `Cons` objects with a pointer to the next list and to the element (el), e.g an integer number. This list is still present in step 2, and in addition to it, we now also have constructed a doubly liked list with `DCons` objects and two empty lists on each end of it. These `DCons` objects are again deallocated in step 3, when we build a new singly linked list and set the pointers such that we obtain the original list again. All yellow objects are allocated freshly and consume one memory unit. We can see that the resource consumption of this program is the length of the input list plus three.

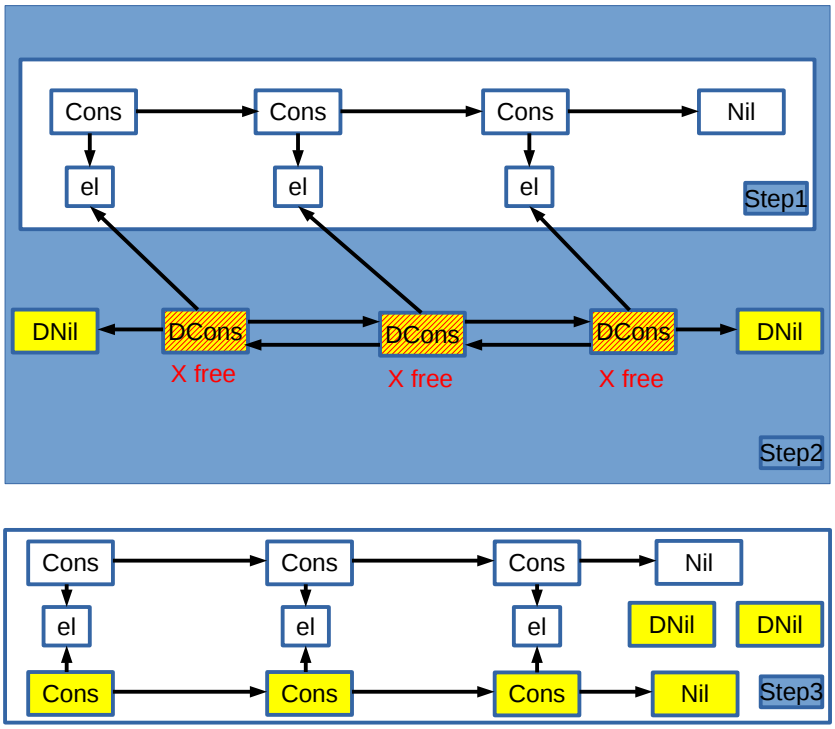


Figure 1.5: Doubly Linked List (DList)

We want to point out another interesting aspect. In our approach we have two different "views" of objects, one (covariant) for reading and one (contravariant) for writing. This concept allows us to treat aliasing correctly. Conceptually, each single access path, no

matter how complicated it is, contributes to the potential. This means that one and the same object can have different potentials depending on the program location. The potential assignments for both kinds of views then are expressed using unknowns. This amounts to an infinite number of unknowns, which are found by solving type constraints. To obtain these constraints, the local balancing rules are encoded in typing rules. In this example, the constraints ensure that the write-view for each pointer to the previous entry in class `DList` has potential zero and thus the (circular) access paths using these pointers do not contribute to the overall potential.

Constraints and Programs There are two important aspects about the role of the constraints in amortized resource analysis:

- Intuitively, the constraints are conditions for an object-oriented program with resource annotated types to be well-typed.
- Solutions of the constraints deliver a valid typing with resource types and allow to compute bounds on the resource consumption, more precisely the heap space usage.

The constraint satisfaction problem arising from the inference of resource types in automatic amortized analysis for object-oriented programs was discovered and introduced by Rodriguez and Hofmann. They also solve some instances of the problem; the above example can be treated already in Rodriguez' work presented in [1].

Their work was based on similar work for functional programs in the past. Linear arithmetic constraints were used by Hofmann and Jost in the context of automatic type-based amortized resource analysis by the potential method [5] where it was applied to first-order functional programs. The constraint systems appearing in this analysis have finitely many variables and can be reduced to linear programming. The same was true for subsequent extensions to higher-order functions and more complex potential functions. The extension of this method to object-oriented programs [6, 1, 3] led to the use of constraints involving infinite lists or trees whose entries are numerical variables. In this case, it is no longer a direct option to solve the constraints with linear programming. The reason for that lies in the following difference. In functional programs it is not straightforward to build circular data structures, whereas in object-oriented language there may be arbitrary and unpredictable chains of data structures where one is an attribute of the other (such as a list denoted by tail(x) is an attribute of a list x).

Thus one needed to take into account also infinite data structures. They are represented as infinite trees, where the objects are nodes and the degree of a node is the number of attributes the object has and its children are exactly these attributes, which are again objects. A heuristic procedure was developed ([1]) which allowed to find solutions (corresponding to linear bounds, i.e. regular solution trees) in many cases but the general question of decidability of these infinitary constraint systems remained open. It is precisely this question that we treat in this thesis.

To put the problem in perspective we consider a piece of code in the Java-like language RAJA (as defined in [1], also known as Featherweight Java Extended with Update (FJEU)), which implements the sieve of Eratosthenes and which can not be analyzed with the methods presented in [1]. Here the classes `List`, `Cons` and `Nil` are defined as usual, such that `List` is a superclass of `Nil` (the empty list) and `Cons` (a nonempty list) and the methods `eratos` and `filter` of the class `List` are defined in the two subclasses.

```
class Cons extends List{
  int elem;
  List next;
  //RAJA syntax: let _ = a <- b is the same as a = b
  List filter(int a){
    let l = new Cons in
    let b = this.elem - (a * this.elem / a) in
    if  (b == 0) then
      let _ = l.next<-this.next.filter(a) in
      return l.next
    else
      let _ = l.elem <- this.elem in
      let _ = l.next<-this.next.filter(a) in
      return l;
  }
  List eratos(){
    let l = new Cons in
    let _ = l.next<-this.next.filter(this.elem).eratos() in
    return l;
  }
}
```

In [7], this program was implemented in the functional language RAML, and analyzed with the result that its resource consumption is quadratic. In our case, the existing analysis tool — also named RAJA — cannot handle the appearing constraints because they do not admit regular solutions over $\mathbb{D}$ or $\mathbb{R}_0^+$, i.e. solutions representing linear potential and hence space usage. If we try to run the programm, we get an error that we run out of memory because the reserved freelist size is too small.

If we implement the function differently without creating a new list to store the results and with the side effect of changing `this`, we obtain linear potential. The reason is that, if we modify `this` instead, we do not need additional potential for creating new lists in each recursive step. Then the tool can solve the constraints.

```
List filter(int a){
 let _ = this.elem <- this.elem - (a * this.elem / a) in
   if  (this.elem == 0)
     return this.next.filter(a)
   else
     let _ = this.next<-this.next.filter(a) in
     return this;
}

List eratos(){
 let _ = this.next<-this.next.filter(this.elem).eratos()
   in return this;
}
```

In contrast to the RAJA tool, which guesses the constraint solutions and reserves a freelist of linear length and then runs out of memory for the first program, we can solve the generated constraints in both cases and thus obtain the right freelist size.

We give another example that can be analyzed using the RAJA tool, namely the (quadratic time) sorting algorithm Bubblesort. The numbers of recursive calls to Bubblesort for *each list element* depends on the lists length and is not a fixed number. Despite these difficulties, the RAJA system is capable of analyzing this program:

```
class List{
  List sort(int n){
    return null;
```

```
    }
  List swap(){
    return null;
    }
  List sortaux(){
    return null;
    }
  void printL(){
    return null;
    }
  int length(){
    return null;
    }
  List sort(int n){
    return null;
    }
}

class Nil extends List{
  List sort(int n){
    return this;
  }
  void printL(){
    return print("");
  }
  List swap(){
    return this;
  }
  List sortaux(){
    return this;
  }
  int length(){
    return 0;
  }
  List sort(int n){
    return this;
  }
```

```
}

class Cons extends List  {
  List next;
  int elem;

  int length(){
    let next = this.next in
    return next.length() + 1;
  }

  void printL(){
    let _ = print(this.elem) in
    let _ = print(", ") in
    return this.next.printL();
  }

  List swap(){ // swaps the first two list elements
    if this.next instanceof Nil then
      return this
    else
      let a = this.elem in
      let _=  this.elem <- ((Cons) this.next).elem
      in let _ =((Cons) this.next).elem <- a in
      return this;
  }

  List sortaux(){ // does one bubble step
    if this.next instanceof Nil then
      return this
    else
      let thiselem = this.elem in
      let thisnextelem = ((Cons) this.next).elem in
      if (thiselem < thisnextelem) then
      let _ = ((Cons) this.next).sortaux() in return this
      else let _= this.swap() in
      let _= ((Cons) this.next).sortaux()
```

```
      in return this;
  }

  List sort(int n){
    if (n==0) then return this
    else
      return this.sortaux().sort(n-1);
    }
  }

class Main {

  void printList(List l){
    let _ = print("[") in
    let _ = l.printL() in
    return print("]");
  }

  List main(List list){
    let main = new Main in
    let list2 = list.sort(list.length()) in
    let _ = main.printList(list2) in
    return list2;
  }
}
```

Here the auxiliary function `sortaux` does the bubble steps and the function `sort` repeats `sortaux` until the list is sorted. This is surely the case after n steps, where n is the length of the input list. Note that this program does not do new object allocations and consumes no heap cells. Thus the potential can be set to zero despite the recursive calls, since we have no arithmetic constraints that ensure that at the beginning we have positive potential. Indeed, in absence of arithmetic constraints, all tree constraints have trivial solutions.

Now we illustrate how the constraint generation works. This will be made more precise in Chapter 2. Since the constraints generated by the RAJA type inference algorithm for this program and the corresponding variables are hundreds in number and the analysis is

very involved, we decided to pick simpler (schematic) program fragments to explain the constraint generation. Although they may seem artificial, there are imaginable cases, where one wants to write such code. However, the aim of these programs is to be as simple as possible and to generate constraints of a certain format. We start with an example that has a cascade of recursive calls, which results in constraints with solution an exponentially increasing list and exponential heap space consumption (in the size of the input list):

```
class Cons extends List{
  List next;
  int elem;
  List m(int i) {
    let res = new Cons in
    let res'= new Cons in
    // ... some calculation ...
    let _ = res.next <- this.next.m(i) in
    let _ = res'.next <- this.next.m(i+1) in
    return res;
  }
}
```

The resulting constraints include

$$\mathbf{tail}(\mathbf{x}) \geq \mathbf{x} + \mathbf{x}.$$

where $\mathbf{x}$ ranges over infinite lists of nonnegative rational numbers (or infinity) and $\mathbf{tail}(\mathbf{x})$ denotes the `next` attribute of $\mathbf{x}$. Addition (+) is understood pointwise. Constraints of that form do not admit regular solutions (except the list consisting only of infinity), but can be treated by the method we describe here.

We now explain the meaning of the above constraint. The list $\mathbf{x} = x_0, x_1, x_2, \ldots$ models the potential assigned to the argument, i.e. `this`: x_0 "dollars" for `this` itself, x_1 dollars for `this.next` (if present), etc.. The constraint then models the sharing of potential of `this.next` between its two occurrences $\texttt{this.next}_i, i = 1, 2$ in the let-expressions, which means that their potential sums to the potential of `this.next` itself. Since we invoke the method `m` of `this` recursively with $\texttt{this.next}_i$, they must be subtypes of the type of `this`. This means that they have at least the same potential as `this`.

Since this inequality holds pointwise, we have—if we read it as a recurrence relation—an exponentially increasing list.

One can modify this program by replacing one of the recursive calls by a call to another method of the `this` object. This would result in a constraint of the form

$$\mathbf{tail}(\mathbf{x}) \geq \mathbf{x} + \mathbf{y},$$

where the potential requirements of the other method are captured in $\mathbf{y}$. If the method requires nontrivial (e.g. polynomial) potential, then we have $\mathbf{x}$ a list with polynomially growing entries, where the degree has increased by one.

Our new algorithm allows us to treat a wider class of programs than it was the case previously. We can decide satisfiability (i.e. the possibility of correct typing refined with potential) and in many cases we are able to detect minimal solutions with respect to the heap space consumption. In these cases we can guarantee that the program can execute successfully with a certain amount of memory as a function of its input size. Unlike in [1] this function does not have to be linear.

We remark that it is essential to be able to treat programs with superlinear resource consumption, which can not be analyzed with the existing RAJA framework. In addition to those examples that can directly be seen to require nonlinear potential, also other programs can have this property. The reason for that is that the algorithm generates a vast amount of constraints already for simple programs and these are then substituted and combined using an elimination procedure that brings them into a normal form. This normal form differs significantly from the original set of constraints, and contains also nonlinear constraints that are very difficult to detect by just considering the underlying program. Thus it is very likely that nonlinearity is more frequent than it may seem at the first glimpse.

1.3 Contributions and Outline

This thesis presents a decision procedure that solves all linear tree constraints that are relevant for RAJA type inference. Parts of this work have been published in conference proceedings ("TYPES-2015" (collection of abstracts), "LPAR-21" and "LPAR-22") [8, 9, 10]. These results were discovered and written by myself and discussed with Martin Hofmann

and Steffen Jost, who also revised the presentation. Theorem 3 and the completeness proof for the derivation system in Figure 6.3 was the result of considerations and discussions with Martin Hofmann. The proof of Lemma 10 was restructured by Martin Hofmann.

We have the contributions listed here:

- We show that the list constraint problem in its original and most general formulation admits a reduction from the famous Skolem-Mahler-Lech problem, which is computationally hard and whose decidability status is unknown until now. This makes the list constraint problem unlikely to be decidable, in particular because it offers additional difficulties such as inequalities, multiple constraints on the same variable and mutual dependencies between variables. Of course then the tree problem is much more involved, because there are complicated combinatorial aspects that arise in the combination of paths.

- We show that this hard instance cannot be generated from programs and identify the constraints that can really be generated by the type inference and give a formal syntax for them.

- This latter subproblem concerns trees of degree at least two, because the procedure presented in [1] always results in trees, even for list programs. We show that this is an avoidable effect by simplifying the type inference in this point. More precisely, we replace the tree constraints by list constraints from which the binary trees can be recomputed and which are equisatisfiable.

- We give a decision procedure for satisfiability in this practical case by a reduction to linear programming.

- The growth rates of solutions of satisfiable systems are determined and classified.

- We illustrate our algorithm by solving constraints that are generated by RAJA programs which cannot be analyzed by the RAJA tool.

This is also the ordering in which the results are presented, except that we first present the decision procedure for lists separately in order to ease the understanding, and then generalize it to trees. More precisely, we give an overview of related work and amortized analysis in Chapter 2. Chapters 3 to 6 contain new theoretical results. The results regarding lists [8, 9] are presented in much more detail and further explained Chapter 4. Similarly,

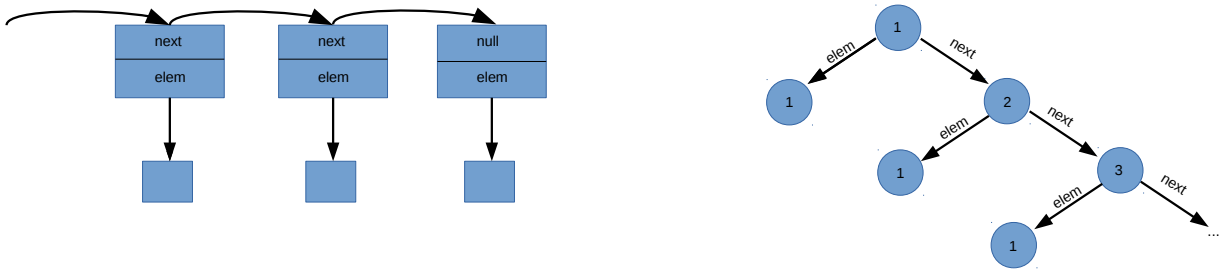


Figure 1.6: Tree for a Singly Linked List

the decision procedure for trees as in [10] is included in Chapter 6.2 and further illustrated and motivated and slightly extended by the growth rate estimation in the rest of Chapter 6. Chapter 7 gives a description of programs that are beyond the scope of the RAJA tool but can be analyzed with our method. Chapter 8 gives a summary and shows ideas for future work.

Linear Tree Constraints - use of the new results in this thesis Our decision procedure allows us to extend the analysis to a wider class of potential functions. As potential functions for the amortized analysis, infinite trees labeled with nonnegative real numbers are used. Such a potential function applied to an object in some heap gives the labels of all nodes that actually exist as access path of the object at hand.

For example, if the potential function is given by a tree corresponding to a singly linked list as in Figure 1.6, then the potential of the this object adds up to $9 = 1+2+3+1+1+1$. Potentials of acyclic objects will in general be finite unless a tree node contains the label ∞, whereas potentials of cyclic data structures are usually infinite. Also notice that physical nodes in an alias data structure will contribute several times to the potential, once per individual access path.

Until now, only trees with a regular structure in the common sense were considered.

Recall that the translation from programs to linear constraints on infinite data structures allows us to read off the resource consumption of these programs - once we can solve the constraints. Dulma Rodriguez already gave an (incomplete, but still in many cases well working) procedure to solve them. This procedure has the restriction that the trees containing the potential annotations have to be "regular" in the sense that they have only finitely many different subtrees. Such trees directly correspond to linear resource bounds (in the size of the input), whereas for instance linearly growing trees are not regular and deliver quadratic resource bounds. Clearly, it is desirable to be also able to model nonlinear resource consumption, like it can be done in the functional setting [11]. In our language, we deal with graph-like data structures and complicated pointer arithmetic; these are concepts that are not easy to model in a functional language. In addition, fundamental imperative features like aliasing often lead to nonlinear resource behaviors.

We also observe that there is a close connection between list constraints and recurrences. Notice that even for nonlinear resource bounds the constraints are still linear. This is akin to the situation for linear recurrences, where the vast majority of interesting cases of solutions is exponentially growing and can then be expressed using linear algebra with matrix theory and Eigenvalues. Since the potential is never negative, we are interested only in nonnegative constraint solutions.

We consider all kinds of trees without any restriction. In the prototype implementation of the RAJA language, one of the examples is sorting a list using mergesort. In the example linear bounds are possible by using static garbage collection, namely free expressions that make additional potential available for the further computation. There is research in this direction [12, 13], but by now there are still open questions about the realization of a static garbage collector in Java. If we omit the free expressions in the code, the program is no longer analyzable, which means that it then requires nonlinear potential annotations. Nonlinear bounds now make our analysis independent of this construction and thus closer to real Java. Further, other nonlinear examples as dynamic programming (with list-like data), longest common subsequence etc. become analyzable using our decision procedure for lists.

Once we have obtained a valid resource typing, it can then also be verified using the typechecker in [1], given that it has the regular structure required there.

Limits Our analysis is capable of analyzing arbitrary RAJA programs. But this does not mean that we obtain nontrivial upper bounds in all cases. There are two theoretical limitations, which have their reason in the undecidability of the halting problem. We are considering heap space, not running time, but one could easily introduce a new heap cell allocation for each execution of a loop and then one could know if they are finite in number.

For instance, for some programs we may obtain trees labeled with infinity as the only constraint solution. This means we overapproximate their resource usage by an infinite amount of heap cells needed.

In other cases, the program may generate unsatisfiable constraints, which means that it is not correctly typable with our resource types. Still, the program may be welltyped without the resource annotations.

Chapter 2

Related Work

In this chapter, we present the state of the art in the field of automatic amortized analysis, which is the main application of the tree constraints. We remark that although they come from this context, they are of independent interest, since they are a generalization of linear programming to infinitely many variables, a problem for which one does not yet have many other results.

In Section 2.1, we informally explain the idea of the amortized analysis method, and in Section 2.2 we introduce the language RAJA, which is the subject of our considerations. We explain how the constraints are generated, which role they play in program analysis, and show the attempts that have already been made to solve them, including a heuristic procedure by Hofmann and Rodriguez [3].

Then, in Section 2.3, we describe other related work that follows different approaches or is not as closely related as the previous.

2.1 Amortized Analysis by the Potential Method

In this section we give a short overview of related work in amortized analysis, because that is where the constraint problem comes from. The idea of amortized analysis goes back to the 1980s [14]. It is an approach that takes into account not only the worst case resource consumption of programs (which may be much more than one has in practice) but the worst

case *average* resource usage of sequences of operations (for a more detailed explanation see for instance [11]). The benefit from that is that in the change of data structures during a computation additional resources may become available, i.e. there may be operations that bring the data into a state such that the following operations can be carried out more efficiently. Then one knows that in the next step not the worst case bound will hold.

One well known example is a FIFO queue that is modeled with two stacks [6]. There one starts with pushing the elements on the first stack and when the first pop is done, one has to move all elements to the next stack to reverse their order. Then the next pops are simple because most of the work was done by this moving to the other stack. Another example are the so called self-adjusting data structures [14].

Hofmann and Jost first applied amortized analysis by the potential method to first-order functional programs in [5]. They annotated the types in the programs with the available resources of the data structure and then introduced typing rules to reason about the resource consumption of functions. There they had the restriction that the potential functions where required to be linear. The approach was later generalized to multivariate polynomial potential by Hoffmann [11].

This was the starting point for many other investigations in this direction [15, 16, 17, 18]. Hofmann and Moser applied amortized analysis to term rewriting [19, 20], Hoffmann refined his work, made it fully automated and carried over the analysis to concurrent programs in C and OCaml [21, 22, 23, 24, 25, 26, 27]. Hofmann and Jost introduced an amortized analysis for a fragment of Java [6], which features object oriented programming, polymorphic functions and monomorphic recursion [3, 4, 1] and is called RAJA (Resource Aware JAva). Rodriguez gave a reduction from type inference for RAJA to solving linear inequalities between trees.

This latter analysis system is the origin of our tree constraints. In the next section, we will explain how the system works, how the constraints are generated and which cases the heuristic constraint solving procedure is able to treat.

2.2 Type Inference in the Object-oriented Language RAJA

2.2.1 Constraint Generation from RAJA Programs

The language RAJA, where the name stands for resource aware Java, consists of the following statements [1]:

$$
\begin{array}{rr}
c ::= \text{ class } C \text{ [extends } D] \ \{A; M\} & (Class) \\
A ::= C\ a & (Attribute) \\
M ::= H\ m(E_1 x_1, \ldots, E_j x_j)\{ \text{ return } e; \} & (Method) \\
e ::= x & (Variable) \\
\ |\text{null} & (Constant) \\
\ |\text{new } C & (Construction) \\
\ |\text{free}(x) & (Destruction) \\
\ |(C)\ x & (Cast) \\
\ |x.a_i & (Access) \\
\ |x.a_i <- x & (Update) \\
\ |x.m(x) & (Invocation) \\
\ |\ \text{if } x \text{ instanceof } C \text{ then } e_1 \text{ else } e_2 & (Conditional) \\
\ |\ \text{let } [D]\ x = e_1 \text{ in } e_2 & (Let)
\end{array}
$$

Figure 2.1: RAJA Syntax

The update syntax differs a bit from Java, since the expression $x.a_i <- y$ evaluates to x. The reason for that is that the number of sharings in the program is reduced by avoiding additional copies of y. However, the Java update is also possible by writing let $u = (x.a <- y)$ in y.

Rodriguez describes a type inference for the subset RAJA^m of this language, where only monomorphic recursion is allowed. This is, that a recursive method can not be called such that it has different types in the calls. As an example, the following code would be beyond the scope of the RAJA system[1].

[1]This concrete example still works in RAJA, because there is no mechanism that detects the polymorphic

```
class List{
  int length(){return 0;}
}
class Nil extends List{
  int length(){return 0;}
}
class Cons extends List{
  int elem;
  List next;
  int length(){
  let next = this.next in
return next.length() + 1;}
}

class NestedNil extends NestedList{
  int length(){return 0;}
}

class NestedList extends Cons{
  List elem2;
  NestedList next;
  int length(){
    let elem2 = this.elem2 in
    let next = this.next in
    return elem2.length() + next.length() + 1;
    }
}

class Main{
  List main(Cons list){
    let main = new Main in
    let l = new NestedList in
    let l2 =  new NestedList in
    let l3 = new NestedNil in
```

recursion — which is formally not allowed — and here it succeeds in analyzing it. But in general, the soundness proofs do not cover all cases of polymorphic recursion.

```
    let _ = l2.elem <-1 in
    let _ = l2.elem2 <- list in
    let _ = l.elem <- 1 in
    let _ = l.elem2 <-  list in
    let _ = l.next <- l2 in
    let _ = l2.next <- l3 in
    let x = l.length() in
    let _ = print(x) in return list;
  }
}
```

Note that "really nested" structures as e.g. in [28] can not be implemented in RAJA because it has no type variables or generic classes.

Moreover, resource-polymorphic recursion is allowed. This means if we regard identical types with only different potential annotations as different types, then this sort of polymorphism is explicitly allowed and often needed for the analysis.

From now on, when writing RAJA, we actually mean RAJA^m. Rodriguez offers a polynomial type checking algorithm [2] as well as a reduction from type inference to linear constraint solving [3, 4]. The constraint generation there is sound and complete. This means, that satisfiable constraints for a certain expression e allow to us compute the potential annotations and the types for e and, for the other direction, that constraints which are generated from a typeable expression are satisfiable.

The key idea is to assign potential to all objects and all their attributes (again possibly objects) and to regard them through the so called *views*, that capture this potential information. The set of views if defined coinductively. There are two types of views, the get-views that are needed when reading an attribute and the set-views for writing. Subtyping is covariant in the get-views and contravariant in the set-views. This will be defined formally in Chapter 3, where we simplify this view construction.

The constraints then make statements about the potential and how it changes in each step of the program.

There is no room in this thesis to explain all the rules for the constraint generation because this requires many prerequisites from type theory that would take too much space and

$$\frac{\mathcal{C} = (E^v <: C^u \wedge p' \leq p)}{M; \Xi; \Gamma, x : E^v \left|\frac{p}{p'}\right. x : C^u \& \mathcal{C}} \text{ (Var)}$$

$$\frac{\mathcal{C} = (\bigwedge_{E<:C} (C.a)^{A^{get}(E^v,a)} <: D^u \wedge p \leq p')}{M; \Xi; \Gamma, x : C^v \left|\frac{p}{p'}\right. x.a : D^u \& \mathcal{C}} \text{ (Access)}$$

$$\mathcal{C} = (\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \textstyle\bigwedge_i u_i \sqsubseteq v_i \oplus w_i)$$

$$\frac{M; \Xi; \Gamma, \vec{y} : F^{\vec{v}} \left|\frac{p'}{p}\right. e_1 : D^a \& \mathcal{C}_1 \qquad M; \Xi; \Gamma, \vec{y} : F^{\vec{w}}, x : D^a \left|\frac{p'}{p''}\right. e_2 : C^b \& \mathcal{C}_2}{M; \Xi; \Gamma, \vec{y} : F^{\vec{u}} \left|\frac{p}{p''}\right. \text{ let D } x = e_1 \text{ in } e_2 : C^b \& \mathcal{C}} \text{ (Let)}$$

Figure 2.2: Example Rules for Constraint Generation ([1])

is already explained in [1]. But for the purpose of illustration and in order to give an evidence for the *unilateral* constraints that play an important role in this thesis, we give three example rules and explain their meaning. They can be found in in Figure 2.2.

Besides that they are among the most interesting rules — they describe sharing of potential and subtyping, which is most important for our constraints since it corresponds to sums and inequalities — they are also comparatively easy to read.

Basically, they show for each syntax construct, how new constraints arise when evaluating it. The statement $\Gamma \left|\frac{p}{p'}\right. B : T \& \mathcal{C}$ means that in a context Γ, expression B evaluates to resource type T and thereby consumes p memory units and returns p' memory units to the freelist. Afterwards the constraints $\mathcal{C}$ must hold. M and Ξ are maps that cover the dependency structure of the program and ensure that recursions are typeable. We will not explain them in more detail.

The first rule (Var) is for introducing a new variable. It states that the constraints say that the *refined type* E^v is a subtype of the refined type C^u (i.e. E is a subclass of C and v is a subview of u, which is defined in Chapter 3) and that the nonnegative number p' is less or equal to p. These constraints $\mathcal{C}$ are generated when evaluating an expression where a

new variable is defined in the context Γ, which maps object variables to refined types, and which is extended by the statement that x is of type E^v. We then (if $\mathcal{C}$ is fulfilled) can use any positive amount of potential $p - p'$ to evaluate x to be of type C^u. Basically this just means that an object of a certain type is also an instance of all its supertypes.

The second rule (Access) models accessing a field. There the constraints ensure that for all subclasses E of C the get-type of the attribute a of class C under the get-view for the subclass is a subtype of a type D^u. This D^u is the type that $x.a$ can be typed with if x is of refined type C^v in the context Γ.

The third rule (Let) is the syntax-directed rule for the sharing relation. The constraint system in the first line also serves as a premise and contains pointwise sums of views. It states that if a variable y has two aliases, then the potential is divided between them, i.e. their potential sums to the overall potential of y. We first evaluate e_1 and then e_2 which is assigned the final type C^b and collect all constraints that arise during these evaluations (possibly by other rules) and then add the sum constraints. Of course, these (Let) rules can be chained for more than two occurrences.

There are to other very important rules, namely the rules for creating a new object and for deallocating objects and returning heap cells to the freelist, where the free heap units lie. They are as one might expect, with the simplification that creating a new object takes exactly one heap cell. Code that matches these two rules is the only place in a program where the freelist is manipulated. This means that `new` expressions are the only source of increasing potential. The amount of potential needed then depends on the program structure, for instance how often these `new` statements are executed in a recursive call.

In [6], it is remarked that one could easily adapt this analysis to other heap quantities or resource metrics as multiple freelists or stack size.

The rule (Let) is the only possible situation where sums of lists (resp. trees) are generated. We have inspected all constraint generation rules and come to the result that there is no case where a sum on the greater side of the inequalities can appear. This greatly simplifies the constraints that are to be solved, and, as we will see in Chapter 4, gives rise to a decision procedure for the subcase that covers all instances needed for RAJA type inference.

2.2.2 Elimination Procedure

So far, we have seen how constraints are generated from RAJA programs. It turns out that many of them are redundant and that one can eliminate most of the variables while keeping satisfiability (resp. unsatisfiability).

$$\frac{\mathcal{C}(\mathbf{y}^-) \text{ or } \mathcal{C}(\mathbf{y}^+), \mathcal{C}' = \text{ erase } \mathbf{y} \text{ from } \mathcal{C}}{\mathcal{C} \rightarrow_{elim_{\mathbf{y}}} \mathcal{C}'} \text{ (Prune)}$$

$$\frac{\bigcup_{i=1,\ldots,n}\{\mathbf{y} \sqsubseteq \mathbf{te_i}\} \cup D(\mathbf{y}^+), AC(\mathbf{y}^+)}{\mathcal{C} \rightarrow_{elim_{\mathbf{y}}} \bigcup_{i=1,\ldots,n}(D, AC)[\mathbf{te_i}/\mathbf{y}]} \text{ (Elim}^+\text{)}$$

$$\frac{\bigcup_{i=1,\ldots,n}\{\mathbf{te_i} \sqsubseteq \mathbf{y}\} \cup D(\mathbf{y}^-), AC(\mathbf{y}^-)}{\mathcal{C} \rightarrow_{elim_{\mathbf{y}}} \bigcup_{i=1,\ldots,n}(D, AC)[\mathbf{te_i}/\mathbf{y}]} \text{ (Elim}^-\text{)}$$

$$\mathcal{C}(\mathbf{y}^+, \mathbf{y}^-), \mathcal{C}(\mathbf{y}^{proj}) \cap \mathcal{C}(\mathbf{y}^{whole}) = \emptyset, nd(\mathbf{y}) > 0$$
$$\frac{l_i \in L, \vec{\mathbf{z}}, \lambda \text{ new }, \mathcal{C}' = \mathcal{C}(\mathbf{y}^{proj}) \cap \text{unfold}(\mathcal{C}(\mathbf{y}^{whole})), \mathcal{C}'' = \mathcal{C}'[\mathbf{z_i}/l_i(\mathbf{y})][\lambda/\Diamond(\mathbf{y})]}{\mathcal{C} \rightarrow_{elim_{\mathbf{y}}} \mathcal{C}''} \text{ (Elim}^{\pm}\text{)}$$

Figure 2.3: Elimination Procedure by Rodriguez [1]

Let a tree constraint system $\mathcal{C} = (AC, TC)$ consisting of arithmetic inequalities AC and tree inequalities TC be given. We say that a list variable $\mathbf{y}$ appears positively in TC or AC and write $TC(\mathbf{y}^+), AC(\mathbf{y}^+)$ (resp. negatively, $TC(\mathbf{y}^-), AC(\mathbf{y}^-)$), if it is on the left (resp. right) side of the $\geq$ operator, or simply $\mathcal{C}(\mathbf{y}^+)$ or $\mathcal{C}(\mathbf{y}^-)$, no matter if it is in an arithmetic or a tree inequality. The elimination procedure consists of four cases that match the syntax of a constraint. The first case is that a variable occurs either only positively or only negatively. Then we can delete this variable from the constraint system without effecting its satisfiability properties, because these properties remain unchanged when setting the variables to lists with all entries infinity (resp. zero). The second and third case

are symmetric to each other: in the second, a variable $\mathbf{y}$ (without prefixed labels) is greater or equal to several right hand sides $\mathbf{te_i}$ that can consist of sums not containing $\mathbf{y}$, and in the remaining constraints $\mathbf{y}$ occurs only negatively. Then we can replace $\mathbf{y}$ by the $\mathbf{te}_i$ in all these constraints. In the third case we have that $\mathbf{y}$ (again without prefixed labels) is less than some $\mathbf{te}_i$ (in which $\mathbf{y}$ is not allowed to appear) and otherwise occurs only positively, such that we can replace $\mathbf{y}$ by its upper bounds.

The last case is where $\mathbf{y}$ appears positively and negatively and there exists no constraint in which $\mathbf{y}$ appears without prefixed labels (*as a whole*, denoted by $\mathbf{y}^{whole}$) as well as with prefixed labels or with the root symbol $\Diamond(\cdot)$ applied to it (*projected*, $\mathbf{y}^{proj}$). The expressions $\mathcal{C}(\mathbf{y}^{whole})$ and $\mathcal{C}(\mathbf{y}^{proj})$ denote the constraints in which $\mathbf{y}$ occurs as a whole or projected. Further, $\mathbf{y}$ must not occur only without prefixed root or label symbols in all constraints (i.e. the nesting depth *nd* is not zero). For instance, in the list case a constraint $\mathbf{tl}(\mathbf{y}) \geq \mathbf{y}$ would not fit for this rule. Similarly, a constraint system like $\mathbf{y} + \mathbf{tl}(\mathbf{y}) \geq \mathbf{z} \wedge \mathbf{z} + \mathbf{z} \geq \mathbf{y}$ is not amenable to it since we can neither eliminate $\mathbf{y}$ (because it occurs with and without labels in the same constraint) nor $\mathbf{z}$ (because it only occurs without root or label symbols in all constraints).

An example where the rule is applicable is, in case of a binary tree, the constraint $\mathbf{l}(\mathbf{x}) \geq \mathbf{r}(\mathbf{x}) + \mathbf{r}(\mathbf{x})$. In this case we unfold the inequality by adding the constraints for the head to AC and the corresponding constraints for the two immediate subtrees to TC. The subtrees get fresh names in all constraints. Here the left and the right subtree of $\mathbf{x}$ become independent of each other, which means one is not a subtree of the other. This may lead to a new situation where the other three rules are applicable. In this case we can eliminate all constraints by $Prune$.

A more complicated example is the following. If we have two binary tree inequalities $\mathbf{ll}(\mathbf{y}) \geq \mathbf{r}(\mathbf{y})$ and $\mathbf{y} \geq \mathbf{z}$, we derive the new constraints $\Diamond(\mathbf{y}) \geq \Diamond(\mathbf{z}) \in AC$ and $\mathbf{r}(\mathbf{y}) \geq \mathbf{r}(\mathbf{z}), \mathbf{l}(\mathbf{y}) \geq \mathbf{l}(\mathbf{z}) \in TC$ by unfolding the second constraint. After the renaming $\Diamond(\mathbf{y}) = \lambda, \mathbf{r}(\mathbf{y}) = \mathbf{x}, \mathbf{l}(\mathbf{y}) = \mathbf{t}$, the constraints are $\lambda \geq \Diamond(\mathbf{z})$ and $\mathbf{x} \geq \mathbf{r}(\mathbf{z}), \mathbf{t} \geq \mathbf{l}(\mathbf{z}), \mathbf{l}(\mathbf{t}) \geq \mathbf{x} \in TC$. Now we can first eliminate $\mathbf{t}$ by the first rule $Prune$ and then do the same with $\mathbf{x}$ and $\mathbf{z}$.

In compact notation, the elimination procedure given by Rodriguez looks as presented in Figure 2.3. In her doctoral thesis, Rodriguez also shows that this procedure terminates by using the nesting depth of expressions (i.e. the maximum length of label or root prefixes) as wellfounded descending function that decreases by one when applying the fourth rule.

For lists, we are particularly interested in constraint systems where the elimination procedure can no longer be applied. We call these systems in normal form. A first observation is the following.

Lemma 1.

- *The elimination procedure stops if for all variables* $\mathbf{y}$
  1. $\mathbf{y}$ *appears both positively and negatively and*
  2. *either* $\mathcal{C}(\mathbf{y}^{whole}) \cap \mathcal{C}(\mathbf{y}^{proj}) \neq \emptyset$ *or the nesting depth is zero and*
  3. $Elim^+$ *and* $Elim^-$ *are not applicable.*
- $Elim^+$ *(resp.* $Elim^-$*) is not applicable if for all variables* $\mathbf{y}$ *one of the following options is taken:*
  1. $\mathbf{y}$ *appears on the left (resp. right) at least twice in the same constraint or*
  2. *it appears on the left (resp. right) only with prefixed labels or with prefixed* $\Diamond$ *applications or*
  3. *it appears on both sides of one constraint.*

Proof. If this is the case the premises of all rules are wrong. □

In Chapter 4, we will modify this elimination procedure for lists in order to make it fit into our unilateral setting and we will split away the linear program belonging to the initial values and solve it independently from the list constraint. This way we can eliminate more variables. For instance in a constraint system like

$$\mathbf{r}(\mathbf{x}) \geq \mathbf{r}(\mathbf{y}) + \mathbf{y},$$
$$\Diamond(\mathbf{y}) \geq \Diamond\mathbf{x} + \Diamond(\mathbf{l}(\mathbf{x})),$$

which is in normal form according to the original elimination procedure, we can further eliminate $\mathbf{y}$ because we have only a lower bound condition on the first element of $\mathbf{y}$ and the rest of $\mathbf{y}$ is allowed to be zero. For more details, see Chapter 4.

However, in inequalities that have the form $R_1 + \mathbf{l}(\mathbf{x}) \geq \mathbf{x} + R_2$ or $R_1 + \mathbf{l}(\mathbf{x}) \leq \mathbf{x} + R_2$, where R_1, R_2 are sums of tree variables, we can never try to eliminate the variable $\mathbf{x}$, because

then the procedure would not terminate (cf. [1]).

In the case of trees of higher degree, it turned out that there is no benefit in modifying the elimination procedure, at least the form where it can no longer be applied is not particularly suitable for the algorithm in Chapter 6. For this reason, the decision procedure for trees is given such that it does not make a difference whether the elimination procedure has been applied before or not (for deciding satisfiability of the tree constraints). Nevertheless, eliminating variables before running the decision procedure will make it work faster. Additionally, it will be easier to determine the growth rates of a satisfiable tree constraint system if we assume that the elimination procedure cannot be applied any more (see Chapter 6, Section 6.4).

2.2.3 Heuristic Procedure for Solving Linear Constraints over Regular Trees

Rodriguez gives an heuristic method which works in case of existing regular solution trees, i.e. trees that have only finitely many subtrees. The list case makes obvious that this is a very strong restriction: regular lists are periodic.

The algorithm consists of two ingredients: firstly, one assumes that the constraints admit a regular solution. In addition to that, one needs a (finite) representation of all trees as regular trees, called "tree schema". This representation can then be used to calculate all inequalities that hold for these (sub-)trees by an iterative procedure. Since all subtrees are repeated at some point, Rodriguez is able to prove that the translation of the constraints to arithmetic inequalities for the nodes of the trees results in a finite set of arithmetic constraints. These are satisfiable, if and only if there is a solution with trees that fit into the tree schema. Recall that the arithmetic constraints have the form of a linear program, and we can check with an LP-solver whether it is feasible or not.

Secondly, she gives a method to construct such a tree schema in particular cases. In order to formulate this algorithm, there is another restriction, the constraint system has to be in a special linear form. Basically, this means that all trees that appear in chains like $\mathbf{x} \geq \mathbf{y} \geq \mathbf{z} \geq \cdots \geq \mathbf{ll}(\mathbf{x})$ can be set all equal to $\mathbf{x}$, which makes it possible to construct a tree schema for these constraints. Further, only chains with nonincreasing or nondecreasing label word lengths are allowed and no variables may appear in both sorts of chains.

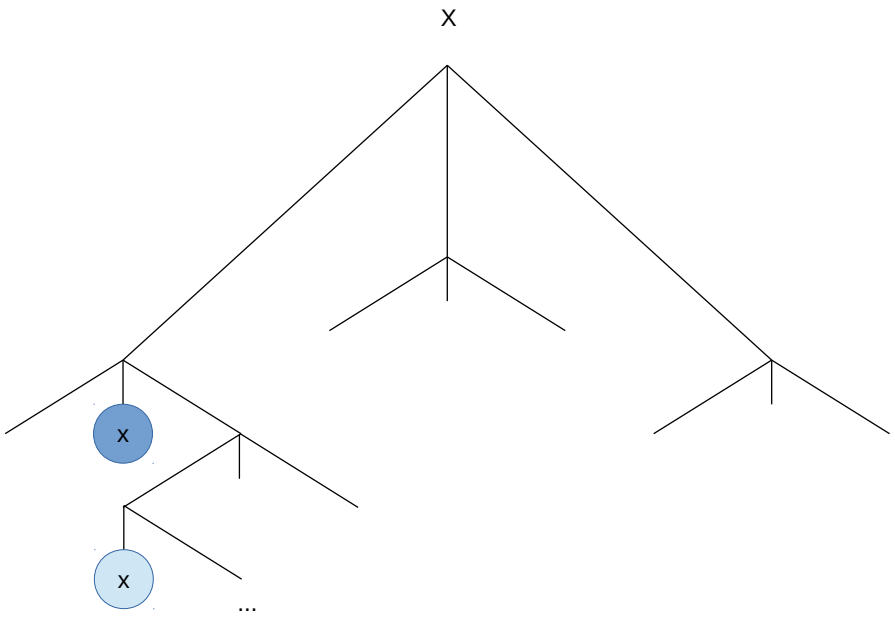


Figure 2.4: Tree with only Finitely many Different Elements but without Tree Schema

Clearly, the following cases and constraints are not amenable to this heuristic:

- As soon as we have sums in the chains and one variable that appears several times, all variables that are part of the chains must be either infinity or all but one summand is zero in order to have a regular solution. For instance, the constraints

$$\begin{aligned} \mathbf{x} &\geq \mathbf{y} + \mathbf{y}, \\ \mathbf{y} &\geq \mathbf{z}, \\ \mathbf{z} &\geq \mathbf{l}(\mathbf{x}). \end{aligned}$$

  could be handled by the algorithm by setting $\mathbf{x} = \mathbf{y} + \mathbf{y} = \mathbf{z} + \mathbf{z} = \mathbf{lx} + \mathbf{lx}$ and thus a regular solution is only possible if $\mathbf{x} = \hat{0}$, i.e. a tree that has zero in each node. Here $\mathbf{x}, \mathbf{y}, \mathbf{z}$ can be arbitrary tree expressions (i.e. trees with possibly prefixed label words). In [1], the problem is avoided by restricting the chains to contain no sums at all.

- The list constraints

$$\begin{aligned} \mathbf{l}(\mathbf{x}) &\geq \mathbf{x}, \\ \mathbf{x} &\geq \mathbf{l}(\mathbf{l}(\mathbf{l}(\mathbf{x}))) \end{aligned}$$

  have only periodic solution lists (see Lemma 8), but they are not suitable for the existing algorithm.

- Also in the case, where we may have regular solutions, the algorithm can not always find a suitable tree schema. Consider the tree constraints for a tree $\mathbf{x}$ with three labels

$$\begin{aligned} \mathbf{x} &\geq \mathbf{m}(\mathbf{l}(\mathbf{x})), \\ \mathbf{x} &\geq \mathbf{r}(\mathbf{l}(\mathbf{x})), \\ \mathbf{l}(\mathbf{x}) &\geq \mathbf{x}, \\ \mathbf{l}(\mathbf{r}(\mathbf{x})) &\geq \mathbf{x}, \\ \mathbf{m}(\mathbf{x}) &\geq \mathbf{x}. \end{aligned}$$

  Then we have cycles of the form (with the notation $l\mathbf{x}$ for $\mathbf{l}(\mathbf{x})$)

$$\begin{aligned} &\mathbf{x} \geq ml\mathbf{x} \geq mlrl\mathbf{x} \geq mlr\mathbf{x} \geq m\mathbf{x} \geq \mathbf{x}, \\ &\mathbf{x} \geq ml\mathbf{x} \geq mlrl\mathbf{x} \geq mlrlrl\mathbf{x} \geq mlrlr\mathbf{x} \geq mlr\mathbf{x} \geq m\mathbf{x} \geq \mathbf{x}, \\ &\quad \ldots, \\ &\mathbf{x} \geq ml\mathbf{x} \geq mlrl\mathbf{x} \geq \cdots \geq ml(rl)^i\mathbf{x} = m(lr)^i l\mathbf{x} \geq m(lr)^i\mathbf{x} \geq \cdots \geq mlr\mathbf{x} \geq m\mathbf{x} \geq \mathbf{x}, \end{aligned}$$

  which do not deliver a valid tree schema, because all repetitions of the tree $\mathbf{x}$ lie on such a path that the tree schema would contain an infinite amount of statements, which is not allowed (see the picture in Figure 2.4).

All these cases are unilateral, which means they can be treated by our new algorithm. It is presented in Chapter 4 for lists and in Chapter 6 for trees of arbitrary finite degree. Finally, we give an example that the algorithm in [1] can handle and explain how it works. Consider

the constraints

$$\mathbf{l}(\mathbf{x}_1) \geq \mathbf{x}_2,$$
$$\mathbf{l}(\mathbf{x}_2) \geq \mathbf{x}_1.$$

They imply

$$\mathbf{l}(\mathbf{l}(\mathbf{x}_1)) \geq \mathbf{l}(\mathbf{x}_2) \geq \mathbf{x}_1,$$
$$\mathbf{l}(\mathbf{l}(\mathbf{x}_2)) \geq \mathbf{l}(\mathbf{x}_1) \geq \mathbf{x}_2.$$

Here we only have one label, and thus we can assume without loss of generality that we are in the list case. (Of course, we could assume a tree of arbitrary degree but with all other subtrees not subject to any inequality and thus equal to zero.) We can interpret this statement as that both lists have to be nondecreasing in each second position. The algorithm would in this case replace the $\geq$ relations with equality and deliver the solutions

$$\mathbf{x}_1 = \mathbf{head}(\mathbf{x}_1), \mathbf{head}(\mathbf{x}_2), \mathbf{head}(\mathbf{x}_1), \mathbf{head}(\mathbf{x}_2), \cdots = \overline{\mathbf{head}(\mathbf{x}_1), \mathbf{head}(\mathbf{x}_2)},$$
$$\mathbf{x}_2 = \mathbf{head}(\mathbf{x}_2), \mathbf{head}(\mathbf{x}_1), \mathbf{head}(\mathbf{x}_2), \mathbf{head}(\mathbf{x}_1), \cdots = \overline{\mathbf{head}(\mathbf{x}_2), \mathbf{head}(\mathbf{x}_1)}$$

The latter inequality can also be written as

$$\mathbf{x}_2 = \mathbf{tail}(\mathbf{x}_1),$$

which again illustrates our notation for pointwise inequalities.

2.3 Other Related Work

Resource analysis encounters a broad interest and different approaches have been introduced. Florian Zuleger et al. have developed an amortized analysis for imperative programs using vector addition systems [29]. Elvira Albert and her group work on a resource inference by recurrence solving [30, 31], whereas Sumit Gulwani et al. compute bounds on the number of statements a procedure executes by generating invariants between counter variables [32].

The most closely related work uses type systems and amortization and has been initiated by Jan Hoffmann and his colleagues (started from [11], to [21], which is a state-of-the-art

system description in that context).

There is also industrial research as AbsInt [33], considering loop oriented programming without recursive procedures and data structures. Another related work on resource analysis is by Jürgen Giesl et al. . They analyze the complexity of term rewrite systems by counting the innermost rewriting steps, and use the result to determine the total runtime complexity [34]. This is part of their development of the tool AProVE for complexity analysis of Java programs [35]. For that, they are using symbolic execution to show that aliasing and cyclic structures, that complicate the analysis, appear only rarely, and then translate the respective program into an integer program, which they can then analyze with standard tools. AProVe is able to show termination and can compute heap space bounds.

Alexandra Silva et al. present a coinductive version of infinite binary trees [36]. We tried to prove decidability of the tree constraints with coinduction, but this method does not seem to be suitable for our setting. In particular, we do not know how to handle the arithmetic constraints in the top of the tree coinductively. Dulma Rodriguez gives many of her definitions on infinite trees by coinduction and remarks that they carry a final coalgebra structure [1]. We decided against presenting syntax and semantics this way, because the meaning of pointwise inequalities is very intuitive, and for our methods, we do not need these formalizations.

There is also related work on constraint systems [37], but these systems differ from ours, for instance because of their finite domain. Erich Grädel et al. define the notion of automatic structures that can be described by certain regular languages [38]. The solution trees for our tree constraints do not fall into this framework; still, we are going to use regular languages in a different context to describe selected parts of the trees in Chapter 6.

Peter Habermehl et al. examine decision problems over infinite arrays, but with a different index arithmetic than ours, which often leads to undecidability [39]. In [40], another kind of periodic relations, but similar to linear list constraints, is considered. These results have other premises and do not subsume our list case arguments.

There is a huge amount of work on recurrence solving, linear programming, Perron-Frobenius theory and linear algebra, which we will apply to our setting. For an introduction see [41, 42, 43, 44]. For the results we use we refer the interested reader to [45, 46] for growth rate detection using matrix theory. The situation for recurrences and decision problems there are described in [47, 48, 49, 50]. In [51], a proof of the Skolem-Mahler-Lech theorem which states

that the indices of the zeros in a recurrence sequence are an arithmetic progression, is given. This proof is complicated and uses p-adic numbers. Also the open Skolem-Mahler-Lech problem is formulated there, which asks if the entries of a recurrence are all nonnegative, and which will become important in the next chapter, where we reduce this problem to the general list constraint satisfiability question. In [52], it is shown that this problem is NP-hard, which also has implications for our constraints. We do not develop further theory in these contexts. Our main contribution there is to bring the constraints into an equivalent form that is suitable to apply these results.

Chapter 3

General Linear List Constraints

In this chapter we simplify the constraint generation such that list programs can be handled more easily. We then formally define the general list constraint problem as formulated by Hofmann and Rodriguez in [3] and prove computational hardness of this variant. We also explain the connection to rational generating functions and give a procedure to split certain lists into sublists while keeping the nonnegative coefficients of the corresponding recurrence relation unchanged. This procedure works for general lists which have not yet the property to be *coprime* as defined below and is used later for the estimation of growth rates in Chapter 5.

3.1 Simplifying the Problem

Since the translation from programs to constraints given in [1] results in at least binary trees, we give a special set of views, which are induced by two infinite lists and comprise all relevant sorts of list constraints.

For this, we introduce the coinductive definition of views given by Rodriguez. Recall that these views are annotations for classes; a class C together with a view r build the *refined type* C^r.

Definition 1 (Views in [1])**.** *The set of views $\mathcal{V}$ is defined via*

- *a map $\Diamond(\cdot)$, which assigns to each view r and class C a nonnegative number $\Diamond(C^r)$,*

- *another map* $A^{get}(\cdot,\cdot)$ *that assigns to each view* $r \in \mathcal{V}$ *and class* C *and field* a *of* C *a view* $s = A^{get}(C^r, a)$,
- *and the similar map* $A^{set}(\cdot,\cdot)$ *that assigns to each view* $r \in \mathcal{V}$ *and class* C *and field* a *of* C *a view* $s = A^{set}(C^r, a)$.

We also define the inequality relation $r \sqsubseteq s$ *between views* r *and* s *coinductively as*

$$\forall C.\Diamond(C^r) \geq \Diamond(C^s),$$
$$\forall C.\forall a.A^{get}(C^r, a) \sqsubseteq A^{get}(C^s, a),$$
$$\forall C.\forall a.A^{set}(C^r, a) \sqsupseteq A^{set}(C^s, a).$$

Last, we define sharing of potential between a view r *and a multiset of views* D *(denoted by* $\curlyvee(r \mid D)$*) as follows:*

- *If* $\curlyvee(r \mid D)$ *then for all classes* C*:* $\Diamond(C^r) \geq \sum_{s \in D} \Diamond(C^s)$,
- $\forall s \in D.r \sqsubseteq s$,
- $\forall a.\ \curlyvee(A^{get}(C^r, a) \mid A^{get}(C^D, a))$
- $\forall a.\ \curlywedge(A^{set}(C^r, a) \mid A^{set}(C^D, a))$

and

- *If* $\curlywedge(r \mid D)$ *then for all classes* C*:* $\min_{s \in D} \Diamond(C^s) \geq \Diamond(C^r)$,
- $\forall s \in D.r \sqsupseteq s$,
- $\forall a.\ \curlywedge(A^{get}(C^r, a) \mid A^{get}(C^D, a))$
- $\forall a.\ \curlyvee(A^{set}(C^r, a) \mid A^{set}(C^D, a))$

where $A^{get}(C^D, a) = \{A^{get}(C^s, a) | s \in D\}$ *and* $A^{set}(C^D, a) = \{A^{set}(C^s, a) | s \in D\}$.

This definition allows many different kinds of views. We restrict ourselves to the following view type.

Definition 2 (Views induced by lists)**.** *Let* $\mathbf{x}_{get}$ *and* $\mathbf{x}_{set}$ *be two infinite lists. The view*

$\mathbf{x} = \langle \mathbf{x}_{get}, \mathbf{x}_{set} \rangle$ *is the view with*

$$\Diamond(\mathbf{List}^{\mathbf{x}}) = \Diamond(\mathbf{x}_{get}), \tag{3.1}$$

$$A^{get}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}) = \langle \mathbf{tail}(\mathbf{x}_{get}), \mathbf{tail}(\mathbf{x}_{set}) \rangle, \tag{3.2}$$

$$A^{set}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}) = \langle \mathbf{tail}(\mathbf{x}_{set}), \mathbf{tail}(\mathbf{x}_{get}) \rangle. \tag{3.3}$$

These views are a proper subset of all views, with the additional property, that

$$A^{set}(\mathbf{List}^{A^{set}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail})}, \mathbf{tail}) = A^{get}(\mathbf{List}^{A^{get}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail})}, \mathbf{tail}). \tag{3.4}$$

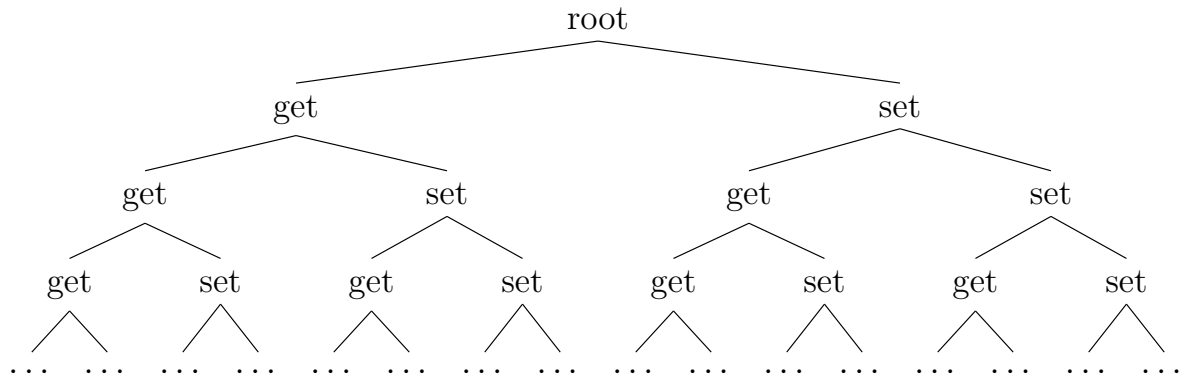


Figure 3.1: Binary Tree View for List-like Data

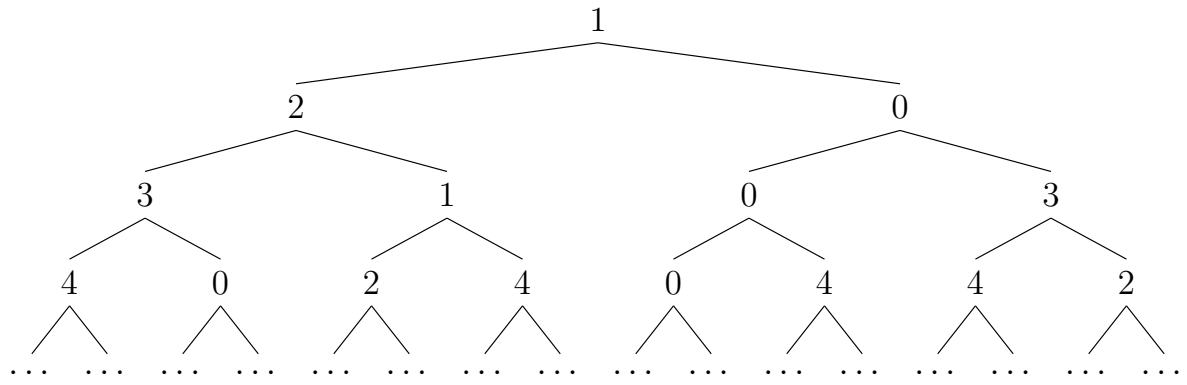


Figure 3.2: Example for a Binary Tree View for List-like Data

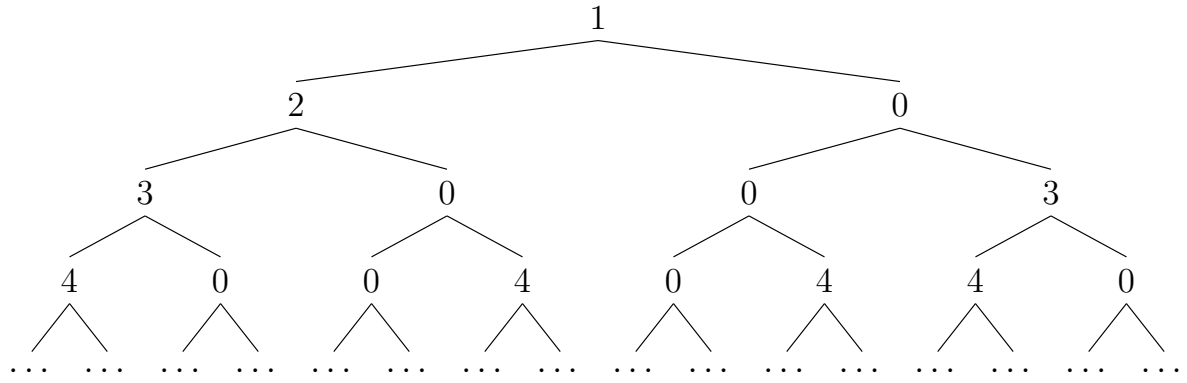


Figure 3.3: Example for a Binary Tree View for List-like Data Constructed from two List-Induced Views for Lists 1,2,3,4,... and 0,0,0,...

The original views can be depicted as binary trees like in Figure 3.1. The list-induced views then have the same shape as in this Figure, but with the difference that all information is stored in the two lists that belong to the leftmost path in the tree and the right children of the respective nodes, and the rest is filled in such that property (3.4) is not violated. Consequently, the list-induced views are less general than the binary tree views. For instance, the tree in Figure 3.2 is not induced by lists, because property (3.4) does not hold, whereas the tree in Figure 3.3 is. Note that the subtrees of the tree **x** in Figure 3.3 satisfy $\mathbf{llx} = \mathbf{rrx}$ and $\mathbf{lrx} = \mathbf{rlx}$.

The definitions of subtyping and sharing and the property to be "main" that is needed to guarantee that writing and reading of attributes are sound w.r.t. the resource typing (see the next definition,[1]) simplify for these views.

Definition 3 (The "main" property from [1])**.** *Let C^r be a refined type. We say that C^r is a* main *RAJA type, if for each attribute a, which lies in the set of all attributes of class C, denoted by $A(C)$, holds $A^{set}(C^r, a) \sqsubseteq A^{get}(C^r, a)$.*

This property states that we can write equal or more specific types into an attribute than we can read, and excludes cases where we write something of a (too general) type T into the field and then read it as a subtype of T, which might be incorrect. In [1], this property only has to hold for the views for the main function. This is in order to avoid additional,

unnecessary constraints, for instance for objects that are never updated.

It is sufficient for solving constraints for list programs to check whether a relation between these special views is fulfilled, as we will show.

Lemma 2 (Subtyping for simplified list views)**.** *For list views* $\mathbf{x}, \mathbf{y}$*, we have*

$$\mathbf{x} \sqsubseteq \mathbf{y} \Leftrightarrow$$

$$\mathbf{x}_{get} \geq \mathbf{y}_{get} \tag{3.5}$$

$$\wedge \mathbf{tail}(\mathbf{x}_{set}) \leq \mathbf{tail}(\mathbf{y}_{set}). \tag{3.6}$$

Proof. By definition, the subtyping relation means that

$$\Diamond(\mathbf{x}_{get}) \geq \Diamond(\mathbf{y}_{get}), \tag{3.7}$$

$$A^{get}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}) = \langle \mathbf{tail}(\mathbf{x}_{get}), \mathbf{tail}(\mathbf{x}_{set}) \rangle$$

$$\sqsubseteq \langle \mathbf{tail}(\mathbf{y}_{get}), \mathbf{tail}(\mathbf{y}_{set}) \rangle = A^{get}(\mathbf{List}^{\mathbf{y}}, \mathbf{tail}), \tag{3.8}$$

$$A^{set}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}) = \langle \mathbf{tail}(\mathbf{x}_{set}), \mathbf{tail}(\mathbf{x}_{get}) \rangle$$

$$\sqsupseteq \langle \mathbf{tail}(\mathbf{y}_{set}), \mathbf{tail}(\mathbf{y}_{get}) \rangle = A^{set}(\mathbf{List}^{\mathbf{y}}, \mathbf{tail}). \tag{3.9}$$

Case " $\Leftarrow$ ": The formula (3.5) follows from $\Diamond(\mathbf{x}_{get}) \geq \Diamond(\mathbf{y}_{get})$ and $\mathbf{tail}(\mathbf{x}_{get}) \geq \mathbf{tail}(\mathbf{y}_{get})$, which follows by the coinduction hypothesis. (3.6) follows by (3.9) and by coinduction.

Case " $\Rightarrow$ ": This direction follows by coinduction, since (3.5) implies

$$\mathbf{tail}(\mathbf{x}_{get}) \geq \mathbf{tail}(\mathbf{y}_{get}),$$

which shows, together with

$$\mathbf{tail}(\mathbf{tail}(\mathbf{y}_{set})) \geq \mathbf{tail}(\mathbf{tail}(\mathbf{x}_{set})),$$

the formula (3.8). Similarly, $\mathbf{tail}(\mathbf{tail}(\mathbf{x}_{get})) \geq \mathbf{tail}(\mathbf{tail}(\mathbf{y}_{get}))$, together with the property (3.6), shows (3.9). □

Lemma 3. *The list view* $\mathbf{x}$ *is main in the sense of [1], if* $\mathbf{x}_{set} \geq \mathbf{x}_{get}$*.*

Remark 2. *The view in Figure 3.3 is not main.*

Proof. This follows by the previous lemma, since being main is according to (3.2) and (3.3))

exactly the property

$$\langle \mathbf{tail}(\mathbf{x}_{set}), \mathbf{tail}(\mathbf{x}_{get})\rangle \sqsubseteq \langle \mathbf{tail}(\mathbf{x}_{get}), \mathbf{tail}(\mathbf{x}_{set})\rangle.$$

□

Now we come to the sharing relation $\curlyvee$ as defined in [1].

Lemma 4. *Let* $\mathbf{x}, \mathbf{y}, \mathbf{z}$ *be list views. Then*

$$\curlyvee\,(\mathbf{x}, \mathbf{y}|\mathbf{z}) \Leftrightarrow \mathbf{x}_{get} + \mathbf{y}_{get} \leq \mathbf{z}_{get} \wedge \min(\mathbf{tail}(\mathbf{x}_{set}), \mathbf{tail}(\mathbf{y}_{set})) \geq \mathbf{tail}(\mathbf{z}_{set}). \tag{3.10}$$

Proof. The sharing relation says that

$$\Diamond(\mathbf{x}_{get}) + \Diamond(\mathbf{y}_{get}) \leq \Diamond(\mathbf{z}_{get}), \tag{3.11}$$

$$\mathbf{y}, \mathbf{x} \sqsupseteq \mathbf{z}, \text{ i.e. } \mathbf{x}_{get}, \mathbf{y}_{get} \leq \mathbf{z}_{get} \text{ and } \mathbf{x}_{set}, \mathbf{y}_{set} \geq \mathbf{z}_{set}, \tag{3.12}$$

$$\curlyvee\,(A^{get}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}), A^{get}(\mathbf{List}^{\mathbf{y}}, \mathbf{tail})|A^{get}(\mathbf{List}^{\mathbf{z}}, \mathbf{tail})), \tag{3.13}$$

$$\curlywedge\,(A^{set}(\mathbf{List}^{\mathbf{x}}, \mathbf{tail}), A^{set}(\mathbf{List}^{\mathbf{y}}, \mathbf{tail})|A^{set}(\mathbf{List}^{\mathbf{z}}, \mathbf{tail})), \tag{3.14}$$

where the statement

$$\curlywedge\,(A^{get}(\mathbf{List}^{\mathbf{a}}, \mathbf{tail}), A^{get}(\mathbf{List}^{\mathbf{b}}, \mathbf{tail})|A^{get}(\mathbf{List}^{\mathbf{c}}, \mathbf{tail})) \tag{3.15}$$

means

$$\min(\Diamond(\mathbf{a}_{get}), \Diamond(\mathbf{b}_{get})) \geq \Diamond(\mathbf{c}_{get}), \tag{3.16}$$

$$\curlywedge\,(A^{get}(\mathbf{List}^{\mathbf{a}}, \mathbf{tail}), A^{get}(\mathbf{List}^{\mathbf{b}}, \mathbf{tail})|A^{get}(\mathbf{List}^{\mathbf{c}}, \mathbf{tail})), \tag{3.17}$$

$$\curlyvee\,((A^{set}(\mathbf{List}^{\mathbf{a}}, \mathbf{tail}), A^{set}(\mathbf{List}^{\mathbf{b}}, \mathbf{tail})|A^{set}(\mathbf{List}^{\mathbf{c}}, \mathbf{tail})). \tag{3.18}$$

Case “ $\Rightarrow$ ”:

Statement (3.11) and the coinduction hypothesis, which implies that

$$\mathbf{tail}(\mathbf{x}_{get}) + \mathbf{tail}(\mathbf{y}_{get}) \leq \mathbf{tail}(\mathbf{z}_{get}),$$

show that $\mathbf{x}_{get} + \mathbf{y}_{get} \leq \mathbf{z}_{get}$. We also have by coinduction hypothesis that

$$\min(\mathbf{tail}(\mathbf{tail}(\mathbf{x}_{set})), \mathbf{tail}(\mathbf{tail}(\mathbf{y}_{set}))) \geq \mathbf{tail}(\mathbf{tail}(\mathbf{z}_{set}))$$

and further that $\min(\Diamond(\mathbf{tail}(\mathbf{x}_{set})), \Diamond(\mathbf{tail}(\mathbf{y}_{set}))) \geq \Diamond(\mathbf{tail}(\mathbf{z}_{set}))$ because of (3.14) and (3.16).

Case "$\Leftarrow$": First, (3.11) holds by definition, and (3.12) holds, since all lists have nonnegative entries. The coinduction hypothesis implies (3.13) and (3.14). □

Solving these inequalities is more difficult than our original problem as defined in the introduction. In the previous work, a one-to-one correspondence between solving the subtyping constraints and solving constraints over infinite trees has been established. The translation of the (co- and contravariant) views by Rodriguez to infinite trees with simple pointwise inequalities returns binary trees, whereas our (also co- and contravariant) views belong to trees of degree one, i.e. infinite lists, which are also compared pointwise. We now argue that both are equivalent regarding satisfiability.

There is a set of views as defined in [1], which satisfies a set of conditions, if and only if there is a set of views induced by two infinite lists which also satisfies the conditions. We give an example to illustrate the idea. Let a binary tree view u for a list be given and let C be the set that consists of the two constraints $A^{get}(u, \mathbf{tail}) \sqsubseteq u, \Diamond(u) \geq 1$, which is satisfiable and originally belongs to constraints over a binary tree with children the get and the set views. Then we have the positive (covariant) part v^+ and the negative (contravariant) part v^- of the solution view v given by

$$\begin{aligned}\Diamond(v^+) &= \Diamond(List^v),\\ \Diamond(v^-) &= 0,\end{aligned}$$

and

$$\begin{aligned}\mathbf{l}(v^+) &= A^{get}(List^v, \mathbf{tail})^+,\\ \mathbf{l}(v^-) &= A^{set}(List^v, \mathbf{tail})^-,\\ \mathbf{r}(v^+) &= A^{get}(List^v, \mathbf{tail})^-,\\ \mathbf{r}(v^-) &= A^{set}(List^v, \mathbf{tail})^+.\end{aligned}$$

We can define the new view $\langle \mathbf{x}_{get}, \mathbf{x}_{set} \rangle$ via

$$\begin{aligned} \mathbf{head}(\mathbf{x}_{get}) &= \Diamond(v^+), \\ \mathbf{head}(\mathbf{x}_{set}) &= 0, \\ \mathbf{tail}(\mathbf{x}_{get}) &= [\Diamond(\mathbf{l}^i(v^+))]_i, \\ \mathbf{tail}(\mathbf{x}_{set}) &= [\Diamond(\mathbf{l}^i(v^-))]_i. \end{aligned}$$

By the translation of the subtyping rules for the binary tree views to list views, the list view $\langle \mathbf{x}_{get}, \mathbf{x}_{set} \rangle$ satisfies the constraints:

$$\begin{aligned} \mathbf{head}(\mathbf{x}_{get}) = \Diamond(List^v) &= 1, \\ \mathbf{tail}(\mathbf{x}_{get}) = [\Diamond(\mathbf{l}^{i+1}(v^+))]_i &\geq [\Diamond(\mathbf{l}^i(v^+))]_i = \mathbf{x}_{get}, \\ \mathbf{tail}^2(\mathbf{x}_{set}) = [\Diamond(\mathbf{l}^{i+1}(v^-))]_i &\leq [\Diamond(\mathbf{l}^i(v^-))]_i = \mathbf{tail}(\mathbf{x}_{set}). \end{aligned}$$

The first inequality follows from $v^+ \sqsubseteq A^{get}(List^v, \mathbf{tail})^+$ and the second inequality holds since $v^- \sqsupseteq A^{get}(List^v, \mathbf{tail})^-$.

Similarly, we can define such a set of special views for trees in the straightforward way. We have two trees that induce a view by the following rules.

Definition 4. *Let $\mathbf{x}_{get}, \mathbf{x}_{set}$ be two infinite trees of degree n. The view $\mathbf{x} = \langle \mathbf{x}_{get}, \mathbf{x}_{set} \rangle$ is the view with*

$$\Diamond(\mathbf{Tree}^{\mathbf{x}}) = \Diamond(\mathbf{x}_{get}),$$

and for all subtrees,

$$\begin{aligned} A^{get}(\mathbf{Tree}^{\mathbf{x}}, \mathbf{l_1}) &= \langle \mathbf{l_1}(\mathbf{x}_{get}), \mathbf{l_1}(\mathbf{x}_{set}) \rangle, \\ A^{set}(\mathbf{Tree}^{\mathbf{x}}, \mathbf{l_1}) &= \langle \mathbf{l_1}(\mathbf{x}_{set}), \mathbf{l_1}(\mathbf{x}_{get}) \rangle, \\ &\dots \\ A^{get}(\mathbf{Tree}^{\mathbf{x}}, \mathbf{l_n}) &= \langle \mathbf{l_n}(\mathbf{x}_{get}), \mathbf{l_n}(\mathbf{x}_{set}) \rangle, \\ A^{set}(\mathbf{Tree}^{\mathbf{x}}, \mathbf{l_n}) &= \langle \mathbf{l_n}(\mathbf{x}_{set}), \mathbf{l_n}(\mathbf{x}_{get}) \rangle. \end{aligned}$$

The proofs are similar to the list case. The result are tree constraints of the same degree as the corresponding views, which means half the size of the original views in [1]. As we

proved, these simplified views are consistent with the underlying type theory.

3.2 Syntax and Semantics

Recall that the tree constraint problem has the following shape. Tree constraints are build by the grammar:

$$\begin{aligned} tae &\coloneqq x|\mathbf{l}(tae) && \text{(Atomic Tree Expression)} \\ te &\coloneqq tae|te + te && \text{(Tree Expression)} \\ tc &\coloneqq te \geq te. && \text{(Tree Constraint)} \end{aligned}$$

Here x is a variable, $\mathbf{l}$ a label in $B = \{\mathbf{l_1}, \ldots \mathbf{l_n}\}$ and tc is the final constraint. Arithmetic constraints are defined as follows:

$$\begin{aligned} ae &\coloneqq n|\lambda|ae + ae|\Diamond(tae) \\ ac &\coloneqq ae \geq ae, \end{aligned}$$

where n is a number, λ an arithmetic variable and ac the resulting constraint. In the next section, we consider list constraints of that shape.

3.3 The General Case

In this section, we show that in this general form, the list case of the constraint satisfiability problem is hard for the famous Skolem-Mahler-Lech problem whose decidability status is open but which is at least NP-hard. We also draw a connection to the theory of rational generating functions and describe how we can modify list constraints in order to get a certain format that we will need later for our decision procedure.

3.3.1 Hardness of the General List Case

In order to establish the announced hardness and also for the subsequent development it is useful to draw a connection between the case of infinite lists and rational generating functions. In particular, this shows that certain behaviors cannot arise as unique solutions to type inference problems. A concrete example is the harmonic sequence because it has a logarithmic generating function [42], whereas all uniquely expressible sequences have a rational generating function.

In this section, we use the connection to recurrences to show the hardness of the list constraint problem. We consider the following three decision problems and establish a relation between them.

- **Skolem-Mahler-Lech Problem (SML)**

  Given: A homogeneous linear recurrent sequence of degree k with initial values $b_1, \ldots, b_k$ and constant rational coefficients $a_1, \ldots, a_k$ of the form

  $$\begin{aligned} x_n &= a_1 x_{n-1} + \cdots + a_k x_{n-k}, n > k \\ x_1 &= b_1, \ldots, x_k = b_k \\ a_i, b_i &\in \mathbb{Q}, b_i \geq 0, \text{ for all } i = 1, \ldots, k, a_k \neq 0 \end{aligned}$$

  Asked: Is there an index n such that $x_n = 0$?

- **List Constraint Satisfiability Problem (LC)**

  Given: A finite system of list constraints and arithmetic constraints

  Asked: Is there a set of lists, which simultaneously satisfies all constraints of the system in $\mathbb{D}$?

- **Tree Constraint Satisfiability Problem (TC)**

  Given: A finite system of tree constraints and arithmetic constraints

  Asked: Is there a set of trees, which simultaneously satisfies all constraints of the system in $\mathbb{D}$?

The SML problem is not known to be decidable and is believed to be computationally very hard. (see [48, 52]).

Theorem 1. *The list constraint satisfiability problem is hard for the Skolem-Mahler-Lech-Problem.*

Proof. The famous Skolem-Mahler-Lech problem can be reduced to the positivity problem for recurrences (SML′) [48], which as we show now can be reduced to satisfiability of list constraints: Let

$$x_n = a_1 x_{n-1} + \cdots + a_k x_{n-k},$$
$$x_1 = b_1, \ldots, x_k = b_k$$

be a given linear recurrent sequence. Then we can express the question

$$\text{Is } x_i \geq 0 \text{ for all } i? \text{ (SML}')$$

in terms of the constraints.

In order to do this, we multiply rational coefficients with the least common multiple of their denominators. We then rewrite the recurrence relation such that the x_i with negative coefficient c_i go to the left hand side of the equation. Then the recurrence relation takes the form

$$c_n x_n + \sum_i c_i x_{n-i} = \sum_i d_i x_{n-i}. \tag{3.19}$$

We translate (3.19) into a constraint by replacing the subscript n in the recurrence by $\mathbf{tail}^k$, where k is the order of the recurrence. This means dropping the first k elements of the list, and the subscript $n - j$, $j = 1, \ldots, k$ we replace by $\mathbf{tail}^{k-j}$. Next, we encode constant factors $c_i x_{n-i}$ by repeated addition $x_{n-i} + \cdots + x_{n-i}$. Last, we build the same constraint with $\leq$ instead of $\geq$. This is in the desired syntax for our constraints.

We can construct such a constraint system C for every recurrence with rational coefficients. Now C is satisfiable if and only if (SML′) for $\mathbf{x}$ has a positive answer.

□

Example 4. *Let*

$$x_n = -\frac{1}{2}x_{n-1} + 2x_{n-3} + x_{n-10}, n > 10$$
$$x_1 = \cdots = x_{10} = 1.$$

be the given recurrence. The constraint system C consisting of the constraints for $R_1 =$ " $\geq$ ", $R_2 =$ " $\leq$ "

$$\begin{aligned}&\mathbf{tail}^{10}(\mathbf{x}) + \mathbf{tail}^{10}(\mathbf{x}) + \mathbf{tail}^{9}(\mathbf{x})\\ &R_i\ \mathbf{tail}^{7}(\mathbf{x}) + \mathbf{tail}^{7}(\mathbf{x}) + \mathbf{tail}^{7}(\mathbf{x}) + \mathbf{tail}^{7}(\mathbf{x}) + \mathbf{x} + \mathbf{x}, i = 1, 2\\ &\wedge \mathbf{head}(\mathbf{x}) = \mathbf{head}(\mathbf{tail}(\mathbf{x})) = \mathbf{head}(\mathbf{tail}^2(\mathbf{x})) = \cdots = \mathbf{head}(\mathbf{tail}^9(\mathbf{x})) = 1\end{aligned}$$

describes the list with entries exactly the elements of the recurrence, since the inequality in the constraint holds pointwise for all entries of the list after position 10.

Corollary 1. *The problems TC and LC are NP-hard.*

This follows from the NP-hardness of the SML problem ([52]) and from the fact that the reduction was polynomial many-to-one. The size of the constraint system C is linear in the length of the encoding for the recurrence. Note that LC is a special case of TC and thus TC is also at least as hard as SML.

3.3.2 Rational Generating Function

In this section we show that we cannot express more sequences *uniquely* with our constraints than with linear recurrences (which correspond directly to rational generating functions). This is worth mentioning, because we have additional to recurrence systems possibly multiple conditions on the same variable as well as inequalities.

Lemma 5. *If there is a unique solution to a set of list constraints, then it has a rational generating function.*

Proof. We can translate lists into recurrences; we do this first for the case where we have equality constraints. Then the constraint system with mutually dependent, not necessarily distinct lists $\mathbf{x}_i$ for $i = 1, ..., n$ (which means that there can be more than one constraint on

the same variable) has the form (all a's are rationals and $\mathbf{x}^{(i)}$ meaning $\mathbf{tail}^i(\mathbf{x})$):

$$\begin{aligned}
\mathbf{x}_1^{(i_1)} &= a_{1,1,1}\mathbf{x}_1^{(1)} + \cdots + a_{1,1,i_1-1}\mathbf{x}_1^{(i_1-1)} + a_{1,2,1}\mathbf{x}_2^{(1)} + \cdots + a_{1,2,m_{1,2}}\mathbf{x}_2^{(m_{1,2})} \\
&\quad + a_{1,n,1}\mathbf{x}_n^{(1)} + \cdots + a_{1,n,m_{1,n}}\mathbf{x}_n^{(m_{1,n})} \\
&\quad \cdots \\
\mathbf{x}_n^{(i_n)} &= a_{n,n,1}\mathbf{x}_{n,1} + \cdots + a_n^{(n,i_n-1)}\mathbf{x}_{n,i_n-1} + a_{n,2,1}\mathbf{x}_2^{(1)} + \cdots + a_{n,2,m_{n,2}}\mathbf{x}_2^{(m_{n,2})} \\
&\quad + a_{n,1,1}\mathbf{x}_1^{(1)} + \cdots + a_{n,1,m_{n,1}}\mathbf{x}_1^{(m_{n,1})}
\end{aligned}$$

We can write the generating function $G(z)$ with additive terms for the initial values as done in [42]. The formal power series $G(z)$ in z has as coefficients exactly the entries of the infinite lists defined by the constraints.

$$\begin{aligned}
&G_{\mathbf{x}_1}(z) = a_{1,1,1}z^{i_1-1}G_{\mathbf{x}_1}(z) + \cdots + a_{1,1,i_1-1}zG_{\mathbf{x}_1}(z) + \\
&a_{1,2,1}z^{i_1-1}G_{\mathbf{x}_2}(z) + \cdots + a_{1,2,m_{1,2}}z^{i_1-m_{1,2}}G_{\mathbf{x}_n}(z) + a_{1,n,1}z^{i_1-1}G_{\mathbf{x}_n}(z) \\
&+ \cdots + a_{1,n,m_{1,n}}z^{i_1-m_{1,n}}G_{\mathbf{x}_n}(z) + \underbrace{G_{init_1}(z)}_{\text{polynomial in z}}, \\
&G_{\mathbf{x}_n}(z) = a_{n,n,1}z^{i_n-1}G_{\mathbf{x}_n}(z) + \cdots + a_{n,n,i_n-1}zG_{\mathbf{x}_n}(z) + \\
&a_{n,2,1}z^{i_n-1}G_{\mathbf{x}_2}(z) + \cdots + a_{n,2,m_{n,2}}z^{i_n-m_{n,2}}G_{\mathbf{x}_2}(z) + a_{n,1,1}z^{i_n-1}G_{\mathbf{x}_1}(z) \\
&+ \cdots + a_{n,1,m_{n,1}}z^{i_n-m_{n,1}}G_{\mathbf{x}_1}(z) + \underbrace{G_{init_n}(z)}_{\text{polynomial in z}}.
\end{aligned}$$

We solve the linear equality system over the field $\mathbb{R}(X)$. Note that if there is no contradiction, there has to be a solution in the field, which means that the constraint solutions have rational generating functions.

If we have other relations than equality, which define a solution list uniquely, we introduce slack variables s_i for each inequality. Then we have a bijection between values of $\mathbf{x}$, which satisfy the inequality, and values of s. If $\mathbf{x}$ is a solution that is defined uniquely, then also s is defined uniquely. In particular, s is nonnegative. We can now again solve the linear equality system with additional variables s_i over the field $\mathbb{R}(X)$. □

Example 5. *The harmonic sequence* $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, ...$ *is not the unique solution of a constraint system. Thus there is no resource behavior that must be exactly this. The reason is that it has the generating function* $-ln(1-z)$.

Remark 3. *The harmonic list is a non-unique solution to a constraint system, e.g. to* $\mathbf{x}^{(2)} \leq \mathbf{x}^{(1)}$.

3.3.3 List Splitting

The point of the following statements is that one can split a list into sublists while keeping the original recurrence relation nearly unchanged. This will be needed later, when we split lists into sublists preserving the nonnegativity (and other properties, see Chapter 5) of the coefficients.

Lemma 6. *An infinite list* $\mathbf{x}$*, which is a solution to a list constraint*

$$\begin{aligned} &\mathbf{x}^{(i)} \ R \ a_1\mathbf{x}^{(j_1)} + ... + a_k\mathbf{x}^{(j_m)}, \\ &\quad a_k \neq 0 \text{ for all } k \ , j_1 > ... > j_m, R \in \{=, \geq, \leq\} \end{aligned}$$

can be formulated in the form $\mathbf{x} = x@\mathbf{x}$*, where "@" is the append-operation,* x *a finite list consisting of the arithmetic variables of length* k *and*

$$\mathbf{x}^{(l)} = \begin{cases} \mathbf{y}_0^{(\frac{l}{n})}, \ \text{if } l \equiv 0 \mod n \\ \mathbf{y}_1^{(\frac{l-1}{n})}, \ \text{if } l \equiv 1 \mod n \\ ... \\ \mathbf{y}_{n-1}^{(\frac{l-(n-1)}{n})}, \ \text{if } l \equiv n-1 \mod n \end{cases}$$

$$k = \max\{i, \{l | \exists C \text{ arithmetic constraint which contains } \Diamond(\mathbf{x}^{(l)})\}\}$$

with $n = \gcd(i - j_1, ..., i - j_m)$ *and lists* $\mathbf{y}_0, ...\mathbf{y}_{n-1}$.

Remark 4. *The greatest common divisor of the differences between the shifts of the* $\mathbf{y}_1, ..., \mathbf{y}_{n-1}$ *is 1. More precisely, the new recurrences relations have the form*

$$x_{\lceil \frac{i}{n} \rceil} = a_1 x_{\lceil \frac{j_1}{n} \rceil} + ... + a_k x_{\lceil \frac{j_m}{n} \rceil}.$$

with $n = \gcd(i - j_1, ..., i - j_m)$ *as above.*

Proof. The claim that the greatest common divisor equals 1 follows from the fact that $\mathbb{Z} \ni \frac{i-j_l}{n} = \lceil \frac{i}{n} \rceil - \lceil \frac{j_l}{n} \rceil$ and that the $\frac{i-j_l}{n}$ are relative prime. □

From now on we call such lists *relatively prime* or *coprime lists*.

Example 6. *Let the constraints be*

$$\begin{aligned}\mathbf{x}^{(5)} &= \mathbf{x}^{(3)} + \mathbf{x}^{(1)},\\ \Diamond(\mathbf{x}^{(0)}) = \Diamond(\mathbf{x}^{(2)}) = \Diamond(\mathbf{x}^{(4)}) &= 1 \wedge \Diamond(\mathbf{x}^{(1)}) = \Diamond(x^{(3)}) = 2\end{aligned}$$

Then we have $n = 2$ *and the solution list*

$$x@\mathbf{x} = (1,2,1,2,1,4,2,6,3,10,5,...) = [1,2,1,2,1]@(4,2,6,3,10,5,...)$$

$$\mathbf{x}^{(i)} = \begin{cases}\mathbf{y}_0^{(\frac{i}{2})}, & \text{if } i \equiv 0 \mod 2\\ \mathbf{y}_1^{(\frac{i-1}{2})}, & \text{if } i \equiv 1 \mod 2\end{cases}$$

where $\mathbf{y}_0 = (2,3,5,...)$ *and* $\mathbf{y}_1 = (4,6,10,...)$*. The lists* $\mathbf{y}_0$ *and* $\mathbf{y}_1$ *both can be defined by the Fibonacci sequence, with different initial values.*

Remark 5. *It is not possible to split coprime lists further while keeping this property. For instance, if we split a Fibonacci list further into two sublists*

$$1,2,5,13,\ldots,$$
$$1,3,8,21,\ldots,$$

the coefficients of the recurrence change and get negative. Here, both lists satisfy

$$\mathbf{x}_n = 3\mathbf{x_{n-1}} - \mathbf{x_{n-2}}.$$

For theory on splitting recurrences into subsequences, see for instance [43]. Finding the coefficients of the new recurrences in general is complicated (cf. [49]).

Chapter 4

Unilateral List Constraints: The Decidable Subcase for Lists

In this chapter, we identify a special format of list and tree constraints which contains the image of the translation from the amortized analysis (Section 4.1) and show that for this fragment in the list case satisfiability is decidable by a reduction to linear programming (Section 4.2). In Section 4.3, we discuss this result.

4.1 Unilateral Constraints

We are able to better delineate the image of the translation from type inference to the constraint satisfaction problem, i.e. we have found that the constraints that really appear in program analysis are a proper subcase of the general constraint form.

We have identified this subproblem which can still be used for type-based resource inference but has better algorithmic properties.

Let us call a constraint *unilateral*, if it is of the form

$$\mathbf{x} \geq \mathbf{y}_1 + \cdots + \mathbf{y}_m,$$

where $\mathbf{x}, \mathbf{y}_i$ are tree variables (possibly with prefixed labels).

Definition 5. *ULC is the special case of LC where all constraints are unilateral. The analogous tree problem is called UTC.*

For instance, the constraint for a binary tree with labels $\mathbf{r}, \mathbf{l}$

$$\mathbf{l}(\mathbf{r}(\mathbf{x})) \geq \mathbf{r}(\mathbf{r}(\mathbf{x})) + \mathbf{r}(\mathbf{r}(\mathbf{x})) + \mathbf{x}$$

is unilateral.

It is easy to see by direct inspection of the constraint generation rules of RAJA that all generated constraints are unilateral. Since these generation rules do not play any further role in the results of this thesis other than providing a strong motivation for unilateral constraints we decided against showing them completely in this thesis and refer the interested reader to Chapter 2 or [1, Chapter 5.3.1.].

Basically, the proof that all generated constraints are unilateral would be a structural induction over the number n of type derivation steps for a program, showing that the number of summands on the left equals one for all n.

Note that in order to treat all possible RAJA programs, we must be able to decide UTC. As a first step in this direction, which enables us to analyze the class of list programs, we give a procedure that decides the list case ULC.

Modifying the elimination procedure for unilateral constraints We recall the elimination procedure for general list constraints as defined in [1] and referred to in Chapter 2. If we assume that there is only one summand on the greater side of the inequalities, all rules slightly change, except the rule (Prune) for deleting instances of lists that appear *only positively* or *only negatively* (as defined in Chapter 2, Section 2.2.2). These lists are set to zero or infinity. In this chapter, we will prove that we can set even more lists to zero or infinity, namely the wider class of lists that are *bounded in only one direction.*

Example 7. *The list constraint*

$$\mathbf{x}^{(6)} \geq \mathbf{x}^{(1)}$$

defines only a lower bound on the list $\mathbf{x}$*, which states it is nondecreasing in each fifth position. (Recall that with* $\mathbf{x}^{(i)}$ *we denote* $\mathbf{tail}^i(\mathbf{x})$*.) In contrast to that, it appears positively (i.e. on the smaller side of the inequality) and negatively (i.e. on the greater side).*

We also make the simplification that we assume that all list constraints are shifted to a level where no arithmetic constraint appear any more.

Example 8. *The constraint system*

$$\begin{aligned}\mathbf{x}^{(2)} &\geq \mathbf{y} + \mathbf{y},\\ \Diamond(\mathbf{y}) &\geq \Diamond(\mathbf{y}^{(2)}) = 1,\end{aligned}$$

contains $\mathbf{y}$ *positively and negatively, but it is equivalent to a linear program*

$$\begin{aligned}P &= \{\Diamond(\mathbf{y}) \geq \Diamond(\mathbf{y}^{(2)}) = 1\}\\ &= \{a \geq b = 1\}\end{aligned}$$

in conjunction with the constraint

$$\mathbf{x}^{(5)} \geq \mathbf{y}^{(3)} + \mathbf{y}^{(3)},$$

that we obtain by iterated **tail** *applications. This has the advantage that the linear program and the constraints can be considered independently and thus we obtain more list variables that occur only positively or only negatively and that thus can be eliminated more directly.*

The other rules now take the shape in Figure 4.1 for unilateral list constraints, with the notation from Chapter 2.

The rules that can be applied are not always unique. If $\mathbf{y}$ occurs only negatively elsewhere and is greater than a sum of expressions at the same time, then we can sometimes apply (Elim$^-$) and (Elim$^+$) to $\mathbf{y}$. More precisely, if (Elim$^-$) is applicable for $\mathbf{y}$, then we have constraints of the form

$$\mathbf{y} \geq \mathbf{te_i} \quad (\mathbf{te_i} \text{ possibly a sum}), i = 1, \ldots, m \tag{4.1}$$

$$\mathbf{z_1} \geq \mathbf{y} + \mathbf{R_1} (\text{according to the unilateral syntax, there is only one variable on the left}), \tag{4.2}$$

$$\ldots,$$

$$\mathbf{z_n} \geq \mathbf{y} + \mathbf{R_n}. \tag{4.3}$$

If all $\mathbf{R_i}$ are empty sums, this is also an instance of the (Elim$^+$) rule: $\mathbf{y}$ occurs positively in the inequalities (4.1) and inequalities (4.2) to (4.3) also fit to the rule, when the $\mathbf{z_i}$ are the

$$\frac{\bigcup_{i=1,\ldots,n}\{\mathbf{te_i} \sqsubseteq \mathbf{y}\} \cup D(\mathbf{y}^-)}{C \rightarrow_{elim_{\mathbf{y}}} \bigcup_{i=1,\ldots,n}(D, AC)[\mathbf{te_i}/\mathbf{y}]} \text{ (Elim}^-\text{)}$$

$$\frac{\bigcup_{i=1,\ldots,n}\{\mathbf{y} \sqsubseteq \mathbf{te_i}\} \cup D(\mathbf{y}^+)}{C \rightarrow_{elim_{\mathbf{y}}} \bigcup_{i=1,\ldots,n}(D, AC)[\mathbf{te_i}/\mathbf{y}]} \text{ (Elim}^+\text{)}$$

$$\frac{\begin{array}{c} C(\mathbf{y}^+, \mathbf{y}^-), C(\mathbf{y}^{proj}) \cap C(\mathbf{y}^{whole}) = \emptyset \\ \vec{z}, \lambda \text{ new }, C' = C(\mathbf{y}^{proj}) \cap \text{unfold}(C(\mathbf{y}^{whole})), C'' = C'[\mathbf{z}/(\mathbf{tail}(\mathbf{y}))][\lambda/\Diamond(\mathbf{y})] \end{array}}{C \rightarrow_{elim_{\mathbf{y}}} C''} \text{ (Elim}^{\pm}\text{)}$$

Figure 4.1: Elimination Procedure for Unilateral List Constraints

$\mathbf{te_i}$.

For example, the constraint

$$\begin{array}{c} \mathbf{z} \geq \mathbf{y}, \\ \mathbf{tail}(\mathbf{y}) \geq \mathbf{tail}(\mathbf{tail}(\mathbf{z})), \end{array}$$

where we assume that $\mathbf{z}$ is not eliminable by any rule, leads to

$$\mathbf{tail}(\mathbf{tail}(\mathbf{z})) \leq \mathbf{tail}(\mathbf{z}).$$

with the (Elim$^-$) rule, but we could also use (Elim$^+$) for $\mathbf{tail}(\mathbf{y})$ (then with one additional **tail** application in the first constraint) and obtain the same result.

The rule (Elim$^{\pm}$) received the modification that the case where the nesting depth (nd) is zero is no longer interesting and thus we need not explicitly exclude it. In the general case it was, for instance one could have constraints like $\mathbf{x}+\mathbf{x} \geq \mathbf{y}+\mathbf{y}+\mathbf{y} \wedge \mathbf{y}+\mathbf{y}+\mathbf{y} \geq \mathbf{x}+\mathbf{x}$, which are in normal form according to the elimination procedure in [1]. In the unilateral case, either $nd = 0$ or we have the same list on different sides of inequalities. If there are no shifts, (i.e. the nesting depth is zero), then either (Elim$^+$) would be immediately

applicable due to the unilaterality or we would have $\mathbf{y} \geq a \cdot \mathbf{y} + \mathbf{R}$, with $a \in \mathbb{N}$ and no summand $\mathbf{y}$ in $\mathbf{R}$. Then, we would have to keep track of $\mathbf{y}$: if $\mathbf{R}$ is nonempty, $\mathbf{y}$ must be infinity, and else try both variants, one with $\mathbf{y}$ equal to a list of zeros, and $\mathbf{y}$ equal to a list of infinity-symbols. We omit this case for simplicity.

4.2 Decidability of ULC

We now show our decidability result for the unilateral list case.

We write $n\mathbf{x}$ for the repeated addition $\mathbf{x} + \cdots + \mathbf{x}$. Let an unilateral constraint system be given. We apply the elimination procedure in [3] to these constraints in order to bring it in normal form for our algorithm. This procedure essentially eliminates variables, which appear only on one side of the inequalities in the style of Fourier-Motzkin elimination.

Lemma 7. *We can bring the constraints of the system into following form using the procedure in Figure 4.1:*

$$\mathbf{x}^{(i)} \geq \mathbf{lin}(\mathbf{x}) + \mathbf{R}, \tag{4.4}$$

where

$\mathbf{lin}(\mathbf{x})$ *is a* nonempty *linear combination of* shifts*(i.e. iterated applications of* **tail**$)$ *of* $\mathbf{x}$ *with positive integer coefficients and not all summands equal to* $\mathbf{x}^{(i)}$,
$\mathbf{R}$ *is a (possibly empty) sum of other terms of the form* $\mathbf{lin}(\mathbf{y})$.

We call $\mathbf{R}$ *the inhomogeneity.*

Additionally, we may have constraints on the same variables (i.e. those variables for which a constraint of the form (4.4) exists): $\mathbf{x}^{(i)} \geq \mathbf{R}$.

In other words, the constraints consist of inequalities between *non-eliminable* (w.r.t. the procedure in [3]) variables, namely those which appear on two sides of an inequality.

Proof. Assume that no rule is applicable for $\mathbf{y}$. This is precisely the case when we have constraints such that $\mathbf{y}$ occurs positively and negatively and the following conditions hold:

- The set $C(\mathbf{y}^{proj}) \cap C(\mathbf{y}^{whole})$ is nonempty.

- The variable $\mathbf{y}$ occurs on the left only with prefixed labels.
- The variable $\mathbf{y}$ occurs on the right only with prefixed labels or in sums.

Thus either $\mathbf{y}$ already has a constraint as in Formula (4.4), or the constraints take the form

$$\begin{aligned}\mathbf{tail}^j(\mathbf{y}) &\geq \mathbf{z_1} + \mathbf{R},\\ \mathbf{z_2} &\geq \mathbf{tail}^i(\mathbf{y}) + \mathbf{y} + \mathbf{R}',\end{aligned}$$

such we do not have $i = 0$ and such that $\mathbf{z_1}$ and $\mathbf{z_2}$ cannot be eliminated. But in this case, we can apply **tail** j times to the second inequality and then eliminate y by (Elim$^+$) and obtain

$$\mathbf{tail}^j\mathbf{z_2} \geq \mathbf{tail}^i(\mathbf{z_1}) + \mathbf{z_1} + \mathbf{R}'',$$

which agrees with our normal form, since the $\mathbf{z_i}$ are un-eliminable. □

We also remark that we could further restrict our normal form to have only two summands on the right by introducing new variables. While this was very useful to ease the proofs in [1], in our case the amount of new variables will complicate the understanding, especially for the growth rate estimation in Chapter 5.

An example for eliminating variables is the constraint system

$$\mathbf{x}^{(3)} \geq \mathbf{y}^{(3)}, \mathbf{y}^{(3)} \geq \mathbf{x}^{(2)} + \mathbf{z}^{(1)}, \mathbf{z}^{(1)} \geq \mathbf{x}^{(1)},$$

which results after two applications of the elimination procedure (eliminating $\mathbf{y}$ and $\mathbf{z}$) in

$$\mathbf{x}^{(3)} \geq \mathbf{x}^{(2)} + \mathbf{x}^{(1)}$$

and is now in normal form and the elimination procedure can no longer be applied. Since the constraints hold pointwise, this implies

$$\mathbf{x}^{(3+i)} \geq \mathbf{x}^{(2+i)} + \mathbf{x}^{(1+i)},$$

for all i. We will refer to this operation as *shift* of i.

Example 9. *The constraints*

$$\mathbf{x}^{(2)} \geq \mathbf{x}^{(1)} + \mathbf{y}^{(3)}, \tag{4.5}$$
$$\mathbf{x}^{(2)} \geq 2\mathbf{x}^{(1)} + 3\mathbf{y}, \tag{4.6}$$
$$\mathbf{y}^{(2)} \geq 4\mathbf{y}^{(1)} + \mathbf{x}^{(1)}, \tag{4.7}$$
$$\mathbf{head}(\mathbf{x}) = 2, \tag{4.8}$$
$$\mathbf{head}(\mathbf{y}) = 1. \tag{4.9}$$

form a constraint system from LC with two constraints on the variable $\mathbf{x}$*. It is unsatisfiable in* $\mathbb{D}$*, since the constraints (4.5) and (4.7) imply*

$$\mathbf{x}^{(2)} \geq \mathbf{x}^{(1)} + 4\mathbf{y}^{(2)} + \mathbf{x}^{(2)},$$

which means $\mathbf{x}^{(1)}$ *and* $\mathbf{y}^{(2)}$ *have to vanish. But then (4.6) cannot be satisfied.*

Let us call an infinite list $x_0, x_1, x_2, \ldots$ *periodic* if there exist $0 \leq k < l$ such that $x_i = x_{i+l}$ holds for all $i \geq k$.

The main ingredient for the proof for the list case is the following observation.

Lemma 8. *If there are two constraints of the form*

$$\mathbf{x}^{(i)} \geq \mathbf{x}^{(j)} + \mathbf{R}, \qquad (LB)$$
$$\mathbf{x}^{(i)} \geq \mathbf{x}^{(k)} + \mathbf{R}', \qquad (UB)$$

with $j < i, i < k$*, every satisfying solution maps* $\mathbf{x}$ *to a periodic list.*

Moreover, the period length is bounded by $\mathrm{lcm}(i-j, k-i)$ *and the periodicity starts after at most* i *steps.*

Remark 6. *We will call constraints of the forms LB and UB lower and upper bounds, respectively. Upper bounds have as solutions at most constant (resp. periodic) lists and lower bounds have solution lists that are at least constant (resp. periodic).*

Proof of Lemma 8. Since $\mathbf{R}$ is nonnegative, we have $\mathbf{x}^{(i)} \geq \mathbf{x}^{(j)}$, and thus we know that $\mathbf{x}$ is increasing in each $(i-j)$'th position beginning from entry number j. We also have $\mathbf{x}^{(i)} \geq \mathbf{x}^{(k)}$, which means it is decreasing in each $(k-i)$'th position from position i on; thus it is periodic with the period length at most $\mathrm{lcm}((i-j)(k-i))$. □

Remark 7. *This result can be sharpened by replacing the* lcm *by* gcd.

Proof. This follows from the proof of the theorem of Fine and Wilf [53]: Let

$$\mathbf{x}^{(i)} \geq \mathbf{x}^{(i-j)},$$
$$\mathbf{x}^{(i)} \geq \mathbf{x}^{(i+k)}, k, j > 0.$$

By the extended Euclidean algorithm, there are integers $r, s \geq 0$ such that $rj - sk = \gcd(j, k)$. Thus we have for great enough n

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(n-rj)} \geq \mathbf{x}^{(n-rj+sk)} = \mathbf{x}^{(n+\gcd(j,k))}.$$

Together with the periodicity, which we already have, this implies that the period length is at most $\gcd(j, k)$. □

Lemma 8 allows us to replace a list variable with finitely many arithmetic variables.

Corollary 2. *If*

$$\mathbf{x}^{(i)} \geq a_1\mathbf{x}^{(j_1)} + \cdots + a_m\mathbf{x}^{(j_m)} + \mathbf{R}$$

is a unilateral list constraint with the inhomogeneity $\mathbf{R}$*, then either* $j_k > i$ *for all* k *or* $j_k < i$ *for all* k *or every solution list is periodic.*

Proof. Let us assume that we have a list constraint

$$\mathbf{x}^{(i)} \geq \mathbf{x}^{(k)} + \mathbf{x}^{(j)} + \mathbf{lin}(\mathbf{x}) + \mathbf{R}, i < k, j < i.$$

Then, since all list entries are nonnegative, we know that $\mathbf{x}^{(i)} \geq \mathbf{x}^{(j)}$, and $\mathbf{x}^{(i)} \geq \mathbf{x}^{(k)}$, which means we are in the situation of Lemma 8. □

Example 10. *Let the constraint be*

$$\mathbf{x}^{(4)} \geq \mathbf{x}^{(7)} + \mathbf{x}^{(2)}.$$

Then we know by nonnegativity $\mathbf{x}^{(4)} \geq \mathbf{x}^{(7)}$ *and* $\mathbf{x}^{(2)} \leq \mathbf{x}^{(4)}$*. We obtain by shifting that* $\mathbf{x}^{(4+i)} \geq \mathbf{x}^{(7+i)}$ *and thus* $\mathbf{x}^{(4)} \geq \mathbf{x}^{(7)} \geq \mathbf{x}^{(10)}$ *and further* $\mathbf{x}^{(4)} \leq \mathbf{x}^{(6)} \leq \cdots \leq \mathbf{x}^{(10)}$*. Then* $\mathbf{x}^{(4)} = \mathbf{x}^{(10)}$ *and every solution list is periodic with period length dividing* $6 = \mathrm{lcm}(7-4, 4-2)$.

We now give a procedure to deduce periodic constraints (i.e. list constraints as in Lemma 8 or in Example 10) in list constraint systems with inhomogeneities.

Example 11. *Let the following constraints be given:*

$$\mathbf{x}^{(10)} \geq \mathbf{x}^{(9)} + \mathbf{y}^{(10)},$$
$$\mathbf{y}^{(10)} \geq \mathbf{y}^{(9)} + \mathbf{x}^{(12)}.$$

We can deduce a new periodic constraint

$$\mathbf{x}^{(10)} \geq \mathbf{x}^{(9)} + \mathbf{y}^{(9)} + \mathbf{x}^{(12)}$$

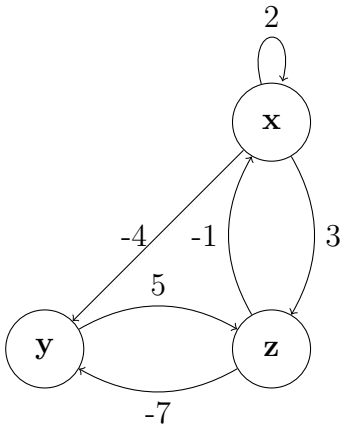
Lemma 9. *We can detect periodic constraints in polynomial time in the number of variables.*

Proof. We introduce a weighted graph for the list constraints by creating a node for each variable. Note that we have only terms of the form $\mathbf{v}^{(n)}$ on the left hand side of the inequalities, with $\mathbf{v}$ a variable and n (possibly after shifting) always the same number. We create an edge weighted with $i \in \mathbb{Z}$ from node $\mathbf{x}$ to node $\mathbf{y}$ if in a constraint for $\mathbf{x}$ a shift of $n + i$ of $\mathbf{y}$ appears.

We use the Bellman-Ford algorithm to detect negative cycles in the graph beginning at node $\mathbf{x}$. We can also find positive cycles by multiplying the labels with -1. Negative cycles correspond to lower bounds and positive cycles to upper bounds on $\mathbf{x}$. Thus we have a periodic constraint if there are both positive and negative cycles. We set up a table with rows labeled with the variables and columns labeled with + and -. We will have 1 in row $\mathbf{x}$ and column + (resp. -) if there is a positive (resp. negative) cycle beginning from $\mathbf{x}$, else we fill in zero. We start with filling in the cycles that contain each variable only once and update the table in the following way: If a positive (resp. negative) cycle from variable $\mathbf{x}$ contains a variable $\mathbf{y}$ (i.e. they are in the same strongly connected component) and if $\mathbf{y}$ has a negative (resp. positive) cycle, we can conclude that $\mathbf{x}$ has a negative (resp. positive) cycle. This follows from the fact that we can run around this additional cycle from $\mathbf{y}$ as many times as are necessary to change the sign of the weight of the resulting cycle.

□

$$\mathbf{x}^{(10)} \geq \mathbf{x}^{(12)} + \mathbf{y}^{(6)} + \mathbf{z}^{(13)}$$
$$\mathbf{y}^{(10)} \geq \mathbf{z}^{(15)}$$
$$\mathbf{z}^{(10)} \geq \mathbf{x}^{(9)} + \mathbf{y}^{(3)}$$



Step 1	+ cycles	- cycles
x	1	0
y	0	1
z	1	1

Step 2	+ cycles	- cycles
x	1	1
y	1	1
z	1	1

Figure 4.2: Example: Constructing a Graph from a Set of Constraints

For instance, if we start with $\mathbf{x}^{(20)}$ for the constraints in Figure 4.2, we get

$$\begin{aligned}
\mathbf{x}^{(20)} &\geq \underline{\mathbf{x}^{(22)}} + \mathbf{y}^{(16)} + \mathbf{z}^{(23)},\\
\mathbf{z}^{(23)} &\geq \mathbf{x}^{(22)} + \mathbf{y}^{(16)},\\
\mathbf{y}^{(16)} &\geq \mathbf{z}^{(21)},\\
\mathbf{z}^{(21)} &\geq \mathbf{x}^{(20)} + \mathbf{y}^{(14)},\\
\mathbf{y}^{(14)} &\geq \mathbf{z}^{(19)},\\
\mathbf{z}^{(19)} &\geq \mathbf{x}^{(18)} + \mathbf{y}^{(12)},\\
\mathbf{y}^{(12)} &\geq \mathbf{z}^{(17)},\\
\mathbf{z}^{(17)} &\geq \underline{\mathbf{x}^{(16)}} + \mathbf{y}^{(10)}
\end{aligned}$$

Thus $\mathbf{x}$ has bounds in both directions:

$$\mathbf{x}^{(20)} \geq \mathbf{x}^{(22)} + \mathbf{x}^{(16)} + \mathbf{R}$$

Note that if we have cycles with zero weight, e.g. $\mathbf{x}^{(10)} \geq \mathbf{x}^{(10)} + \mathbf{R}$, then all lists in $\mathbf{R}$ have to vanish.

Remark 8. *This implies also that constraints of the form*

$$\begin{aligned}\mathbf{x}^{(i)} &\geq \mathbf{x}^{(i-j)} + \mathbf{x}^{(i+k)},\\ i, j, k \in \mathbb{N}, j &< i,\end{aligned}$$

are always only satisfiable with zero lists.

Example 12. *For the constraints*

$$\mathbf{x}^{(4)} \geq \mathbf{x}^{(6)} + \mathbf{x}^{(2)},$$

we have a zero weight cycle corresponding to the derivable constraint

$$\begin{aligned}\mathbf{x}^{(6)} \geq \mathbf{x}^{(8)} + \mathbf{x}^{(4)} &\geq \mathbf{x}^{(8)} + \mathbf{x}^{(6)} + \mathbf{x}^{(2)},\\ R = \mathbf{x}^{(8)} &+ \mathbf{x}^{(2)},\\ \Rightarrow \mathbf{x}^{(2)} = \hat{0} &\vee \mathbf{x}^{(2)} = \hat{\infty},\end{aligned}$$

which means $\mathbf{x}$ *is a list that finally consists only of zeros or infinity.*

Remark 9. *Similarly, upper bounds of the form*

$$\mathbf{x}^{i} \geq \mathbf{x}^{(i+j)} + \mathbf{x}^{(i+k)}$$

can only have exponentially decreasing lists as nontrivial solutions. For instance,

$$\mathbf{x}^{i} \geq k * \mathbf{x}^{(i+j)}$$

has as solution an exponentially decreasing list (or with a zero list). More precisely, if $k = 5$ *and* $j = 1$*, we have a list*

$$\mathbf{head}(\mathbf{x}), \frac{\mathbf{head}(\mathbf{x})}{5}, \frac{\mathbf{head}(\mathbf{x})}{25}, \frac{\mathbf{head}(\mathbf{x})}{125}, \ldots, \frac{\mathbf{head}(\mathbf{x})}{5^n}, \ldots$$

Proof of Remark 9. We define the characteristic polynomial of a recurrence relation of degree d:

$$x_n = a_1 x_{n-1} + ... + a_d x_{n-d}, a_i \in \mathbb{Q} \text{ for all } i = 1, ..., d, n \geq k + 1$$

as

$$p(z) = z^n - a_1 z^{n-1} - ... - a_d z^{n-d}.$$

In the case of upper bounds, we have constraints of the form

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(n+k_1)} + \cdots + \mathbf{x}^{(n+k_l)},$$

which means

$$\mathbf{x}^{(n+\max_j\{k_j\})} \leq \cdots - \mathbf{x}^{(n+k_i)} - \cdots + \mathbf{x}^{(n)},$$

and has characteristic polynomial

$$p(z) = z^{n+\max_j\{k_j\}} + \cdots + z^{n+k_i} + \cdots - z^n.$$

If the roots of this polynomial with maximal modulus, the so-called characteristic roots are negative or complex, then the constraints are unsatisfiable, because then we have an oscillation of the values between positive and negative ones, according to [48]. In the other cases, the characteristic roots must be less than 1, and so the growth rate is exponentially decreasing (see Chapter 5 for more details). □

We also note that something similar holds for inhomogeneities in form of additional summands y: they must be summable zero sequences in order to satisfy the upper bound constraints.

We use these observations to show that lists with such upper bounds can never have also lower bounds in a satisfiable constraint system, unless their initial values are infinity. Thus the satisfiability of a constraint system is not touched by assuming these lists are periodic: if they are bounded only from above in the sense below (in the proof of Theorem 2), we can even set them to be finally zero. Of course, we have to determine a position from which we can set the respective list to zero and there consider all arithmetic constraints and interactions with other lists.

We now state our decidability result.

Theorem 2. *The satisfiability problem for our list constraint systems in $\mathbb{D}$ is decidable.*

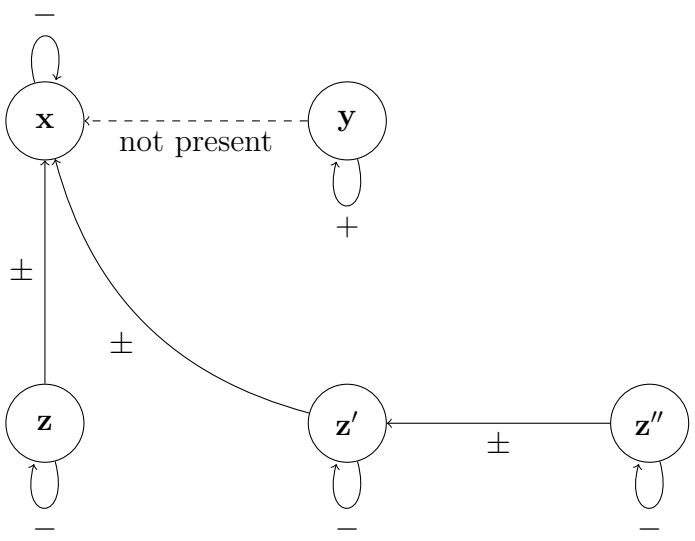


Figure 4.3: Theorem 2: Case 1

Proof. We construct an equisatisfiable linear program from the constraint system. Let G be the graph constructed as in the proof of Lemma 9. First, we carry out the procedure to detect bounds in two directions as in Lemma 8. For those, we replace the periodic lists with arithmetic variables.

Now only those variables remain for which we have either only positive or only negative cycles.

We say that a variable $\mathbf{x}$ has a lower (resp. upper) bound if it has a negative (resp. positive) cycle or if it is greater (resp. less) than another variable $\mathbf{y}$.

If there are only lower bounds for some list $\mathbf{x}$, i.e. the node $\mathbf{x}$ in the graph has only negative cycles and no incoming edges from other variables, then we set $\mathbf{x}^{(n)}$ to the list $\hat{\infty}$, which consists only of infinity. If there are only upper bounds for a list $\mathbf{x}^{(n)}$, which means that the node $\mathbf{x}$ has only positive cycles and no outgoing edges to other variables, we set $\mathbf{x}^{(n)}$ to the zero list $\hat{0}$.

If there are upper and lower bounds on a list $\mathbf{x}^{(n)}$, which do not fit into the situation from Lemma 8, then there are three possible cases:

- Case 1 (see Figure 4.3): there is a negative cycle in $\mathbf{x}$ and $\mathbf{x}$ is not reachable from other variables $\mathbf{y}$ with a positive cycle. This means $\mathbf{x}$ is bounded from above only by other variables $\mathbf{z}$ that are all *unbounded* from above. Then we set also $\mathbf{x}^{(n)}$ to ∞.

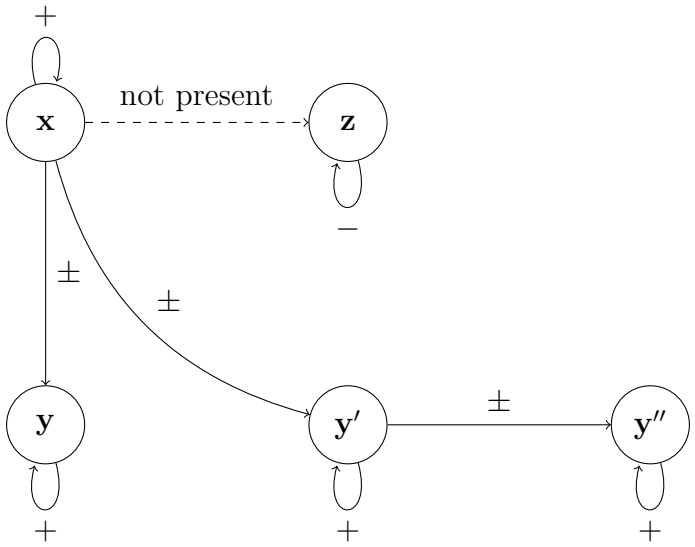


Figure 4.4: Theorem 2: Case 2

- Case 2 (see Figure 4.4): In this case $\mathbf{x}$ has a positive cycle and no other variables $\mathbf{y}$ with negative cycle are reachable from $\mathbf{x}$. This means that $\mathbf{x}$ is only bounded from below by variables that have no lower bound. In this case we intend to set $\mathbf{x}$ to a list that finally vanishes. But we cannot simply set $\mathbf{x}^{(n)}$ to $\hat{0}$, because there may be a constraint on $\mathbf{x}$ like

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(n+j)} + \mathbf{y}^{(i)}, i < n \tag{4.10}$$

  and, for instance, an arithmetic constraint $\mathbf{head}(\mathbf{y}^{(i)}) \geq 1$.(Recall that until position $n-1$ we can have arithmetic constraints.) We observe that, if we have a negative path of weight i from $\mathbf{x}$ to $\mathbf{y}$, as in (4.10), we cannot have a path with weight greater than $-i$ from $\mathbf{y}$ to $\mathbf{x}$ because then we would have a negative cycle in $\mathbf{x}$.

  In the example (4.10) we can set $\mathbf{y}^{(n)} = \hat{0}$ and $\mathbf{x}^{(2n)} = \hat{0}$. According to the observation, all constraints on $\mathbf{y}$ only contain shifts of $\mathbf{x}$ of at least $2n$ positions and from this position on $\mathbf{x}$ vanishes, so the problem does not occur any more.

  In order to generalize this to more variables, we now build a new directed graph G', which has no negative cycles. As nodes, we take all variables with only positive cycles. The edges of G' are the negative weighted edges in the original graph G. We begin with the variables that have only incoming edges and set all of them to lists vanishing

from the n'th position. All other nodes $\mathbf{x}$ have edges to variables $\mathbf{y}_1, \ldots \mathbf{y}_k$. If all $\mathbf{y}_i^{(n')}$ have been set to $\hat{0}$ in the previous steps, we set also $\mathbf{x}^{(n+n')}$ to $\hat{0}$.

- Case 3 (see Figure 4.5): Now only the case remains where we have a variable $\mathbf{x}$ with a negative cycle which is reachable from $\mathbf{y}$, which has a positive cycle. This means that a list with lower bound constraint is less or equal to a list with upper bound constraint. We argue that then there is a solution with $\mathbf{x}$ and $\mathbf{y}$ periodic lists, i.e.

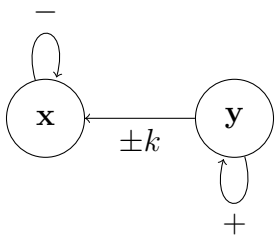


Figure 4.5: Theorem 2: Case 3

if there is a constraint that enforces one of the variables to be strictly increasing or decreasing, then the system immediately becomes unsatisfiable.

We first describe the situation where this happens: Constraints that admit as solution only strictly falling lists can have no solutions for which a lower bound constraint exists (except that this bound is zero and thus periodic). The reason for this is that lists with lower bounds are at least constant or periodic. Thus these lists must fall in case 2 and can be set to $\hat{0}$ preserving satisfiability.

On the other side, constraints that admit as solutions only strictly increasing lists are either of the form

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{n-j} + \mathbf{R}, \tag{4.11}$$

with $\mathbf{R}$ having a lower bound (i.e. being at least constant or periodic) or

$$\mathbf{x}^{n} \geq \mathbf{R}, \tag{4.12}$$

with $\mathbf{R}$ strictly increasing. One observes that at least for one variable in the set $\{v | v \text{ variable in } \mathbf{R}\} \cup \{\mathbf{x}\}$ a constraint of the form (4.11) must hold, and thus $\mathbf{x}$ is at least linear (except if all lists vanish, see also Chapter 5).

Now we go back to the general situation of case 3 without the strictness assumption. Due to the normal form of the constraint syntax, which allows sums only on the smaller side of the inequality, $\mathbf{y}$ has the constraint $\mathbf{y}^{n+j} \leq \mathbf{y}^{(n)} - \mathbf{R}$ with $\mathbf{R}$ nonnegative. Thus $\mathbf{y}$ is at most constant or periodic. Similarly, $\mathbf{x}$ is at least constant or periodic.

Thus $\mathbf{y}$ can only be greater than $\mathbf{x}$ if $\mathbf{x}$ is bounded from above by a constant a and $\mathbf{y}$ is bounded from below by a constant $b \geq a$. According to the above facts, then there must be a solution with both lists periodic.

So the idea is to identify $\mathbf{x}$ and $\mathbf{y}$ with periodic lists and replace them also with finitely many arithmetic variables while keeping the satisfiability of the remaining system.

Let n' be the maximum of the lengths of the aperiodic initial parts and k the least common multiple of all period lengths of the lists that have been set to periodic lists so far. Further, let m be the number of variables that remain. For all these remaining variables $\mathbf{x}$ we set $\mathbf{x}^{(mn')} = \mathbf{x}^{(mn'+k)}$ and replace $\mathbf{x}^{(mn')}$ by k additional arithmetic variables.

Now that we have replaced all lists by finitely many variables (namely polynomially many variables in the number of lists), we translate the constraints in inequalities between these arithmetic variables (see Example 13). Recall that we may have more than one constraint per variable that we must take all into account. More precisely, for each constraint C of the form

$$\mathbf{x} \geq \mathbf{y}_1 + \cdots + \mathbf{y}_k,$$

where some of the $\mathbf{y}_i$ may be shifts of $\mathbf{x}$, we have the equivalent formulation

$$\begin{aligned} x_0 x_1 \dots x_m (x'_1 \dots x'_l)^\omega \geq & y_{1,0} y_{1,1} \dots y_{1,m_1} (y'_{1,1} \dots y'_{1,l_1})^\omega + \dots \\ & + y_{k,0} y_{k,1} \dots y_{k,m_k} (y'_{k,1} \dots y'_{k,l_k})^\omega. \end{aligned}$$

For this inequality, we iteratively carry out the following procedure to translate it into arithmetic constraints. We add an arithmetic constraint

$$\begin{aligned} \mathbf{head}(x_0 x_1 \dots x_m (x'_1 \dots x'_l)^\omega) \geq & \mathbf{head}(y_{1,0} y_{1,1} \dots y_{1,m_1} (y'_{1,1} \dots y'_{1,l_1})^\omega) + \dots \\ & + \mathbf{head}(y_{k,0} y_{k,1} \dots y_{k,m_k} (y'_{k,1} \dots y'_{k,l_k})^\omega), \end{aligned}$$

and then drop the first element of all lists and continue with the shifted constraint

$$\begin{aligned}\mathbf{tail}(x_0x_1\dots x_m(x'_1\dots x'_l)^\omega) \geq &\mathbf{tail}(y_{1,0}y_{1,1}\dots y_{1,m_1}(y'_{1,1}\dots y'_{1,l_1})^\omega)+\dots\\ &+\mathbf{tail}(y_{k,0}y_{k,1}\dots y_{k,m_k}(y'_{k,1}\dots y'_{k,l_k})^\omega) \Leftrightarrow\\ x_1\dots x_m(x'_1\dots x'_l)^\omega) \geq &y_{1,1}\dots y_{1,m_1}(y'_{1,1}\dots y'_{1,l_1})^\omega)+\dots\\ &+y_{k,1}\dots y_{k,m_k}(y'_{k,1}\dots y'_{k,l_k})^\omega),\end{aligned}$$

This procedure stops after at most $\max_i(m, m_i + \mathrm{lcm}_j(l, l_j)) + 1$ steps, when no new constraints are derived any more.

Since all these operations do not effect satisfiability, we have transformed the list constraint system into an equisatisfiable linear program over $\mathbb{D}$. This program can again be transformed into an ordinary linear program. By examining the proof of Theorem 2, we see that we have set selected arithmetic variables to be infinity (case 4.4). We can delete all inequalities with infinity on the left or on both sides, and all inequalities with infinity only on the right cause unsatisfiability.

The other variables that have not been assigned a concrete value *are allowed* to be infinity (cf. case 4.5 and Lemma 8). Assume the linear program is not satisfiable with all arithmetic variables in $\mathbb{R}_0^+ = \mathbb{D} \setminus \{\infty\}$. Then we can try all subsets of the set of arithmetic variables assigned with infinity, and then solve all corresponding linear programs as described above. The structure of this proof is depicted in Figure 4.6. □

Example 13. *Consider the constraints*

$$\begin{aligned}\mathbf{x}^{(4)} &\geq \mathbf{x}^{(2)} + 2\mathbf{x}^{(3)},\\ \mathbf{x}^{(4)} &\geq \mathbf{x}^{(7)}.\end{aligned}$$

As in Example 10, we can conclude that we have a periodic list of period starting at position 4 and of length at most 6, i.e. $\mathbf{x}^{(4)} = \mathbf{x}^{(10)}$*. Thus we have*

$$\mathbf{x} = x_1x_2x_3x_4(abcdef)^\omega, \mathbf{x}^{(2)} = x_3x_4(abcdef)^\omega, \mathbf{x}^{(3)} = x_4(abcdef)^\omega, \mathbf{x}^{(4)} = (abcdef)^\omega,$$

with $x_1, x_2, x_3, x_4, a, b, c, d, e, f$ *arithmetic variables. If we translate the list constraints into*

arithmetic constraints for these variables, we obtain

$$a \geq x_2 + 2x_3, b \geq x_3 + 2a,$$
$$c \geq a + 2b, d \geq b + 2c, e \geq c + 2d, f \geq d + 2e, a \geq e + 2f, b \geq f + 2a.$$

In this example the only solution is a list with x_1, x_2 arbitrary and the rest zero or infinity. For all nontrivial lists we have a contradiction since the first constraint ensures exponential growth (in contrast to example 10) and the second is a constant upper bound.

From the proof of Theorem 2 and from Lemma 9 we can derive the following.
Corollary 3. *The decidability problem for ULC with initial values in $\mathbb{R}_0^+$ can be solved in polynomial time.*

This is the case for our application, because there we do not intend to set initial values to infinity, because we can not derive bounds from the corresponding lists any more. From the theoretical point of view, we can still decide satisfiability — but in general, we need exponential time.

Overall, we have the following result.
Corollary 4. *Let a RAJA program that involves only list-like date (i.e. objects with only one attribute that is not a basic data type) be given.*

- *The question whether this program can be correctly typed with resource types is decidable.*
- *If the program is correctly typable with resource types that encode finite resource information, can be decided in PTIME.*

4.3 Discussion

As a fundamental restriction of the resource types, we have to mention that programs may allow for a computation of a certain finite resource bound and yet be untypable with resource types. This means, the constraints for the program are unsatisfiable, despite the fact that it can execute with a certain (finite) amount of memory. This is also a theoretical limitation due to the halting problem.

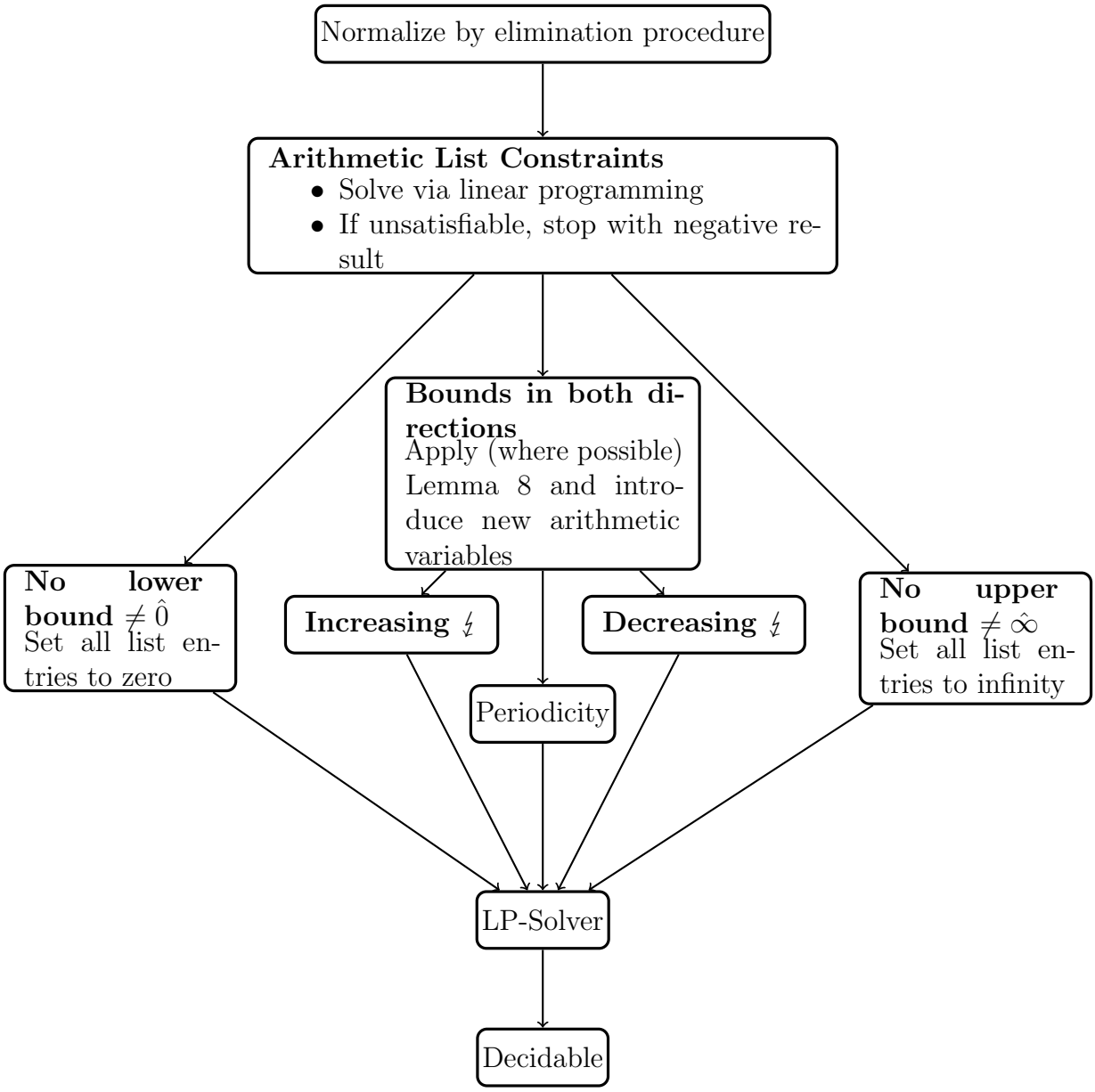


Figure 4.6: Concept of the Decision Procedure for Lists

However, we are now able to analyze more programs than it was the case previously.

Relation to previous work The reader may have noticed that we proceed similarly to Rodriguez, at least regarding the point that we replace all lists by periodic lists (possibly consisting only of infinity). There are three main differences.

According to the unilateral syntax, it is possible to set all increasing lists to $\hat{\infty}$ a priori and just for the decision procedure, because this operation does not effect satisfiability. In the related work, these instances can not be solved and are thus left out by the algorithm. This has either the reason that one cannot treat all programs without this infinity symbol because then one would solve the halting problem. Another possible reason is that the algorithm can not handle the program despite the fact that it is theoretically possible to find upper bounds. The latter is a point where we extend the algorithm. However, if the (existing and the new) algorithm returns upper bounds for a program, then they are correct.

We also prove that certain sorts of constraints force lists to be periodic in Lemma 8. Among them are many constraints that also could not be treated by the heuristic algorithm.

Secondly, setting certain lists to infinity is not a proper restriction, because we are able to show in the next section that all lists set to infinity this way can never have upper bounds in the satisfiable case. We carry out a growth rate analysis which allows us to find approximations and concrete formulas for these lists after the decidability has been checked.

Thirdly, the list case was included for didactic reasons and to point out the difficulties that appear and to illustrate the solution strategies. Chapter 6, where we prove decidability of unilateral tree constraints, covers also the list case.

Related issues We now discuss what happens if we consider $\mathbb{Q}_0^+$ instead of $\mathbb{D}$. Regarding satisfiability, it remains true that once we have found periodic constraints we can replace them by finitely many arithmetic variables as we did and solve the resulting constraints with linear programming. Variables with lower bounds can not be instantiated with constant lists consisting of infinity, but a solution can be found by reading them as recurrence relations, cf. Chapter 5. This all works without changes.

One reviewer of [9] asked what happens if variables range over (nonnegative) integers or reals. The use of (irrational) real numbers is not motivated by the application, so we have

not considered it, but due to the linear nature of the constraints we expect no changes at all. Regarding integers, the polynomial time result no longer holds, as integer linear programming is NP-complete. Beyond that, we expect no further complications. For the amortized approach to resource analysis this kind of integrality constraints do not arise.

We remark that as a consequence of the decision procedure in case of unsatisfiability the contradiction can be proved with a finite number of **head** and **tail** applications and translation of the resulting constraints into arithmetic variables. This defines a complete proof procedure. The number of steps is not known a priori, but it is known after carrying out the decision procedure and thus can be used as a certificate for unsatisfiability in the style of Farkas' lemma.

Chapter 5

Optimal Solutions of Satisfiable List Constraints

By now, we have seen pure decidability of the constraint satisfiability question, but for a real resource analysis, it is not possible to set potentials of entire subtrees to infinity. In practical applications it will also be of importance to get information about the growth rate of solutions or, even better, to compute solutions of minimal growth rate. For the general list constraints (those that are hard for SML) the two problems are closely related: In order to know whether a list $\mathbf{x}$ is less than, say, a fixed polynomial p, just add constraints to that effect and check whether the resulting system is still satisfiable. However, such constraints do not fall into the fragment ULC.

On the other hand, ULC has the property that in presence of nontrivial upper bounds (i.e. constraints of the form (UB) in Chapter 4), all growth behaviors are bounded by a constant. This also means that if there are such upper bounds and strictly increasing lower bounds on the same variables, the system immediately becomes unsatisfiable (see the proof of Theorem 2). From now on we assume that we have no initial values equal to infinity.

Proposition 1. *Let C be a ULC problem and write C' for C with all upper bounds removed, such that only constraints of the following form remain:*

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(i_1)} + \cdots + \mathbf{x}^{(i_m)} + \mathbf{R}, \forall l : i_l < n \tag{5.1}$$

(with $\mathbf{R}$ restricted not to contain $\mathbf{y}^{(j)}$ for which constraints $\mathbf{y}^{(j)} \geq \mathbf{x}^{(k)}, k \geq n$ can be obtained

as in Figure 4.2). If C is satisfiable then the minimal solutions of C and C' coincide.

Thus it suffices to consider only lower bound constraints for the estimation of growth rates. E.g., we want to assure that the n'th position in the minimal solution list is less or equal to a polynomial in n, or to an exponential expression in n.

Now we draw a connection between matrix recurrences and constraint systems. For a system of lower bound constraints we consider the associated recurrence relations, which consist of all constraints with inequality replaced by equality. Homogeneous recurrence relations $(x_n)_n$ of order d, that correspond to constraints with only one list variable $\mathbf{x}$, can be rewritten using their *companion matrix* [54]. This matrix contains the coefficients of the recurrence in its first row, 1 on the lower minor diagonal and the rest zero. It yields the vector $(x_n, \ldots, x_{n-d})$ when applied to the vector $(x_{n-1}, \ldots, x_{n-d-1})$. The rational expansion theorem for linear recurrences [42] ensures that, if the greatest root (w.r.t. the modulus) of the matrix is simple and positive, the sequence becomes monotone with exponential growth rate and is in particular not oscillating between negative and positive values [52, 48]. Now we generalize the companion matrix to constraint systems. The following example shows the idea.

Example 14. *For the constraints*

$$\begin{aligned}\mathbf{x}^{(n)} &\geq \mathbf{x}^{(n-2)} + 2\mathbf{y}^{(n-1)},\\ \mathbf{y}^{(n)} &\geq 3\mathbf{y}^{(n-1)} + 2\mathbf{x}^{(n-1)}\end{aligned}$$

the matrix version is

$$\vec{\mathbf{x}}_n = \begin{pmatrix} \mathbf{head}(\mathbf{x}^{(n)}) \\ \mathbf{head}(\mathbf{x}^{(n-1)}) \\ \mathbf{head}(\mathbf{y}^{(n)}) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 3 \end{pmatrix} \begin{pmatrix} \mathbf{head}(\mathbf{x}^{(n-1)}) \\ \mathbf{head}(\mathbf{x}^{(n-2)}) \\ \mathbf{head}(\mathbf{y}^{(n-1)}) \end{pmatrix} = A\vec{\mathbf{x}}_{n-1} \ .$$

We want to point out the difficulties that occur. There are three facts that make our constraint system differ from an ordinary system of recurrences:

- We have inequalities instead of equalities.
- We may have multiple constraints on the same variable. This becomes problematic, when we want to *compare* lower bounds and one constraint is not 'pointwise' less than the other. We will come back to this below.

- All entries must be nonnegative.

In general, where we have more than one constraint per variable, the associated recurrence for a constraint system contains in addition to + also the max operator:

$$\vec{\mathbf{x}}_n = \max(A_1\vec{\mathbf{x}}_{n-1}, \ldots, A_k\vec{\mathbf{x}}_{n-1}),$$

where $\vec{\mathbf{x}}_n$ is a d-dimensional vector and the A_i are $d \times d$ matrices with nonnegative integers as entries. We first treat the case, where $k = 1$ and where we have no maximum.

We will see in Section 5.1, that we can precisely estimate the growth rate for constraints that only consist of mutually dependent variables in constant time. In Section 5.2, we use linear algebra to estimate the growth rates for more general constraints, with the restriction that there is only one constraint per variable. Then, in Section 5.3, we describe which problems occur when we have multiple constraints on the same variable. We connect this situation to the results of Uriel Rothblum and Kousha Etessami [45, 55] in section 5.4, and apply the latter to obtain the asymptotic growth rate in polynomial time.

In the next sections we make use of Perron-Frobenius theory. Results in this context can be found for instance in [56, 57, 58].

5.1 Estimation of Growth Rates with Perron-Frobenius Theory

In this section we report on partial progress in this topic based on the observation that recurrences stemming from ULC constraints have a unique dominating eigenvalue which is real.

We now prove the existence of a such a zero for homogeneous recurrences that correspond to unilateral lower bound constraints in the following sense. Let

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(i_1)} + \cdots + \mathbf{x}^{(i_m)}$$

be a lower bound constraint, i.e. $i_l < n$ holds for all l.

If we replace the inequality with equality, we obtain a recurrence relation whose solutions

form minimal solutions to the constraint.

Remark 10. *We can uniquely express each constraint solution* $\mathbf{x}$ *as a list of aberrations* $\varphi(\mathbf{x})$ *from this minimal list* x *such that*

$$\mathbf{x}^{(n)} = x_n + \varphi(\mathbf{x})_n.$$

Note that this differs from the pointwise *aberration from the inequality in each step. The list* $\varphi(\mathbf{x})$ *collects all these pointwise aberrations and adds them together.*

For the characteristic polynomial of these recurrences we have the following.

Lemma 10. *Let* $p(x) = x^n - \sum_{i=0}^{(n-1)} a_i x^i$ *be a polynomial with* $a_i \geq 0$ *and* $a_0 > 0$ *and let* $0 \leq i_1 < \cdots < i_m \leq n-1$ *be such that* $a_i > 0 \iff i = i_j$ *for some* j*. If* $gcd(n - i_1, \ldots, n - i_m) = 1$ *then* p *has a single real root* $\alpha \in \mathbb{R}^+$ *which is dominating in the sense that* $p(\gamma) = 0$ *implies* $|\gamma| < \alpha$ *for all* $\gamma \in \mathbb{C}$*.*

Proof. We first show that α has maximal modulus among all roots. The Descartes sign rule applied to p shows that there is one positive real root. Call it α. Now, if $p(\beta) = 0$ for $0 \neq \beta \in \mathbb{R} \setminus \mathbb{C}, |\beta| > \alpha$ then

$$0 = |\beta^n| - |\sum_i a_i \beta^i| \geq |\beta|^n - \sum_i a_i |\beta|^i$$

but this is impossible for such a β. Since $\lim_{x\to\infty} p(x) = \infty$, and also $p(|\beta|) < 0$ (else, $|\beta|$ would be a second real root), $|\beta| > \alpha$ would imply the existence of another real root greater than α (by the intermediate value theorem). So there is no complex root of greater modulus than α.

To prove that α is the only root of maximal modulus, now suppose that $p(\gamma) = 0$ and, for a contradiction that $|\gamma| = \alpha$. Write $\gamma = \alpha\zeta$ with $|\zeta| = 1$. Dividing $p(\gamma) = 0$ by ζ^n gives $\alpha^n = \sum_i a_i \alpha^i \zeta^{(i-n)} = \sum_i a_i \alpha^i$. Since the a_i and α are positive reals it follows that $\zeta^{(i_j - n)} = 1$ since $a + b = |a| + |b|$ implies $a = |a|$ and $b = |b|$ for $a, b \in \mathbb{C}$. By the assumption on the gcd it then follows that $\zeta = 1$ and hence $\gamma = \alpha$. □

If $\gcd(n - i_1, \ldots, n - i_m) = k > 1$, we can split any satisfying recurrence sequence into k subsequences (see [43]) with indices the k'th arithmetic progression with common difference k and such that the coefficients of the recurrence remain positive (see Chapter 3, Section 3.3.3). Then we are again in the situation of Lemma 10. In particular, we then know that the

sequence becomes monotone with exponential growth rate ([42]) and is not oscillating ([52, 48]).

This generalizes to constraint systems of mutually dependent variables. If there is only one constraint per variable, we can build the matrix for the corresponding recurrence system—the above defined companion matrix. We call a $n \times n$ matrix A *irreducible* if the graph that we obtain by considering it as an adjacency matrix (which has n nodes and an edge from node i to node j, if $A_{ij} > 0$), is strongly connected. We call an irreducible matrix *aperiodic*, if the gcd of the lengths of all directed closed paths is equal to 1. One can show Lemma 12 below by induction over the number of list variables and by splitting the lists in gcd-1-sublists similar as for Lemma 10. It then allows for an estimation of growth rates by the Perron-Frobenius Theorem (a matrix with strictly positive real entries has a dominant real positive eigenvalue (as in Lemma 10)).

First, we bring the constraints into a form such that we can construct a (always quadratic) matrix for them and such that they fulfill the requirement that no constraint on a variable $\mathbf{y}$ involves higher shifts on a variable $\mathbf{x}$ than the constraint on $\mathbf{x}$ itself. This means that the constraint on $\mathbf{x}$, seen as a homogeneous recurrence (with the inhomogeneities left out) has as order the maximal number of tail applications that appear in the constraints. We focus on the lower bound constraints — for the upper bounds we have the zero list as an optimal solution. We can set up a system of recurrence relations C'' for each combination C of constraints (i.e. each subset of constraints where there is exactly one constraint per variable) by replacing inequalities by equalities. Then we write the system C'' as a matrix problem.

For constraint systems with only one constraint per variable, we have the following.

Lemma 11. *We can express minimal solutions of lower bound constraint systems as a matrix exponentiation problem such that each constraint on a variable* $\mathbf{x}$ *contains the maximal shift of* $\mathbf{x}$ *that appears at all.*

Proof. We build the companion matrix as e.g. in [54]. Then we replace the equality by inequality to describe minimal solutions. This is only possible since all coefficients on the right side of the equality are nonnegative. The only additional difficulty that can occur is when a constraint on a variable $\mathbf{y}$ involves higher shifts on a variable $\mathbf{x}$ than the constraint on $\mathbf{x}$ itself. Then we do the modification which is illustrated in the next example.

Example 15. *The constraints*

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(n-1)} + \mathbf{y}^{(n-3)},$$
$$\mathbf{y}^{(n)} \geq \mathbf{y}^{(n-1)} + \mathbf{x}^{(n-2)}.$$

lead to a companion matrix that does not fit into our framework. But we can rewrite the system by two substitutions as

$$\mathbf{x}^{(n+3)} \geq \mathbf{x}^{(n+2)} + \mathbf{y}^{(n-1)} + \mathbf{x}^{(n-2)},$$
$$\mathbf{y}^{(n+3)} \geq \mathbf{y}^{(n+2)} + \mathbf{x}^{(n)} + \mathbf{y}^{(n-2)},$$

such that the corresponding recurrence coefficient a_k (that belongs to the lowest power of x that appears in the recurrence, i.e. k is its order) is nonzero.

□

If there is one constraint per variable, the problem can be solved by the eigenvalue method, which gives us an optimal solution where equality holds instead of the $\geq$ relation.

Lemma 12. *We can transform a lower bound list constraint system with mutually dependent lists in such a way such that its companion matrix is irreducible, aperiodic and nonnegative.*

Proof. Two constraint variables $\mathbf{x}, \mathbf{y}$ are said to be dependent, if there is a constraint on $\mathbf{x}$ with $\mathbf{y}$ on the right hand side or vice versa. With this definition we can set up a dependency graph and split it into the tree of strongly connected components.

We now show that a strongly connected component of the constraint dependency graph has an irreducible corresponding matrix. We start the induction with the case where we have only one variable and a recurrence of degree k. Then the $(k+1) \times (k+1)$-matrix looks as follows:

$$\begin{pmatrix} \mathbf{x}_n \\ \mathbf{x}_{n-1} \\ \vdots \\ \mathbf{x}_{n-k} \end{pmatrix} \geq \begin{pmatrix} \alpha_{n-1} & \dots & \dots & \dots & \alpha_{n-k-1} \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & \dots & 1 & 0 & 0 \\ 0 & \dots & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x}_{n-1} \\ \mathbf{x}_{n-2} \\ \vdots \\ \mathbf{x}_{n-k-1} \end{pmatrix},$$

where all α's are nonnegative. If we consider it as an adjacency matrix for $k+1$ nodes v_i, we see that from v_1 we can reach v_{k+1}, from v_2 we can reach v_1, from v_3 we come to v_2 and so on until v_{k+1} to v_k. Let us call this irreducible matrix A_x and the vector on the left hand side of the inequality $\vec{\mathbf{x}}_n$ and the vector on the right $\vec{\mathbf{x}}_{n-1}$, respectively.

For the induction step we assume that we have $n+1$ variables $\mathbf{x}_1, \ldots \mathbf{x}_n, \mathbf{y}$, all of them of possibly higher order with a strongly connected graph for the recurrence that belongs to each variable. Then we consider the matrix belonging to $\mathbf{x}_1, \ldots, \mathbf{x_n}$. Since the variables are strongly connected w.r.t. the dependency defined above, we have one of the $\mathbf{x}$'s on the right hand side of the constraint for $\mathbf{y}$, say $\mathbf{x}_j$ and $\mathbf{y}$ on the right hand side of the constraint for, say $\mathbf{x}_i$. Since ordering of variables does not affect irreducibility of a matrix, we write the system as

$$\begin{pmatrix} \vec{\mathbf{x}}_{1,n} \\ \vec{\mathbf{x}}_{2,n} \\ \vdots \\ \vec{\mathbf{x}}_{i,n} \\ \vec{\mathbf{x}}_{j,n} \\ \vec{\mathbf{y}}_n \end{pmatrix} \geq \begin{pmatrix} A_{x_1} & \ldots & & & & & \\ & A_{x_2} & \ldots & & & & \\ & & \ldots & \ldots & & & \\ \vdots & & & & & & \\ & & & & A_{x_i} & \ldots & B \\ & & & & & A_{x_j} & \ldots \\ & & & & & B' & A_y \end{pmatrix} \begin{pmatrix} \vec{\mathbf{x}}_{1,n-1} \\ \vec{\mathbf{x}}_{2,n-1} \\ \vdots \\ \vec{\mathbf{x}}_{i,n-1} \\ \vec{\mathbf{x}}_{j,n-1} \\ \vec{\mathbf{y}}_{n-1} \end{pmatrix},$$

where B and B' are $k_{\mathbf{x}_i} \times k_{\mathbf{y}}$ (resp. $k_{\mathbf{y}} \times k_{\mathbf{x}_j}$) matrices ($k_{\mathbf{x}}$ denotes the order of the variable $\mathbf{x}$) that contain an entry greater or equal to 1 in their first row. We have that a vertex w, which contains a variable from $\mathbf{x}_j$ is connected to another vertex w' corresponding to $\mathbf{y}$, as well as a vertex t belonging to $\mathbf{y}$ is connected to another vertex t' of $\mathbf{x}_i$, which is connected to w by induction. Since for all k, A_{x_k} and A_y are irreducible, the proof of irreducibility is completed.

The greatest common divisor of a list constraint system C with variables $\mathbf{x}_1, \ldots \mathbf{x}_n$, of the form

$$\begin{aligned} \mathbf{x}_1^{(m)} &\geq \sum_i a_{(1,i)} \mathbf{x}_1^{(i)} + \mathbf{lin}(\mathbf{x}_2, \ldots, \mathbf{x}_n), \\ &\ldots \\ \mathbf{x}_n^{(m)} &\geq \sum_i a_{(n,i)} \mathbf{x}_n^{(i)} + \mathbf{lin}(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}), \end{aligned}$$

is defined as $n = \gcd\{m' - i' | \exists k.\mathbf{x}_k^{(m')} \geq \mathbf{x}_k^{(i')} \text{ follows from } C\}$.

If the $n \geq 2$, then we can split all lists into n sublists and obtain a similar recurrence for these sublists. Lemma 6 in Chapter 3 describes this for single lists and generalizes to systems of list constraints.

Further, the gcd of the system equals 1 if and only if the companion matrix is aperiodic. This is based on the observation that if the right hand side of an inequality on a variable $\mathbf{x}$ contains a shift of $n - i$, then there is a cycle of length $n - i$ in the companion matrix, considered as an adjacency matrix. □

In the other case, there are k constraints on $\mathbf{x}$

$$\begin{aligned}\mathbf{x}^{(n)} &\geq a_{1,1}\mathbf{x}^{(n-1)} + \cdots + a_{1,m}\mathbf{x}^{(n-m)},\\ &\ldots,\\ \mathbf{x}^{(n)} &\geq a_{k,1}\mathbf{x}^{(n-1)} + \cdots + a_{k,m}\mathbf{x}^{(n-m)}.\end{aligned}$$

where m is the maximum order of the constraints and without loss of generality all indices on the left hand side equal n (by shifting the inequalities). Then we could build several companion matrices and calculate in each step of the recurrence the maximal value for $\mathbf{x}$. But it is possible that the matrices where the maximum is taken are always alternating, such that we do not obtain a closed formula (cf. Section 5.3). Alternatively, we can replace the multiple constraints on $\mathbf{x}$ by the new constraint

$$\mathbf{x}^{(n)} \geq \max(a_{1,1}, \ldots a_{k,1})\mathbf{x}^{(n-1)} + \cdots + \max(a_{1,m}, \ldots, a_{k,m})\mathbf{x}^{(n-m)} \tag{5.2}$$

which is an upper bound on the optimal solution. Note that this is only necessary, if for each two constraints there are indices i, i', j, k such that $a_{i,j} < a_{i',j}$ and $a_{i,k} > a_{i',k}$. If there are inhomogeneities, we take analogously the maximal coefficients of the additional terms.

Lemma 13. *The matrix that belongs to the constraints defined in (5.2) is nonnegative, aperiodic and irreducible.*

Proof. The new matrix contains a positive entry at row i and column j if and only if one of the companion matrices A_l for the system with the l'th constraint on $\mathbf{x}$ has a positive entry $(a_l)_{ij}$. A greater number of positive entries preserves the properties irreducible and aperiodic. □

The growth rate of the system defined by the matrix can be read off from its dominant eigenvalue ρ, which is real, positive and simple by the Perron-Frobenius Theorem. It is $\mathbf{x}^{(n)} = \Omega(\rho^n)$ for all initial values with positive coordinates. For those initial values of $\mathbf{x}$ that have a nonzero component in the direction of the eigenvector, that belongs to ρ, the power method is applicable and implies that after iterated applications of the matrix, the (normed) result vectors converge to this eigenvector. This means we have a closed formula in this special case.

Now we come to the case, where the list variables are not all necessarily mutually dependent. We say that $\mathbf{x}$ depends on $\mathbf{y}$, if $\mathbf{y}$ appears on the right hand side of the constraint on $\mathbf{x}$. We calculate the strongly connected components in the dependency graph of the constraint system. Let X be a set of list variables, which appears in a leaf of the tree of strongly connected components. If $|X| = \{\mathbf{x}\}$, the constraints are of the form

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(j)} + \mathbf{R}(\mathbf{x}),$$

where $\mathbf{R}(\mathbf{x})$ are shifts of the variable $\mathbf{x}$. If $\mathbf{R}(\mathbf{x})$ is empty, we have as a minimal solution a list which is constant or periodic. If $\mathbf{R}(\mathbf{x})$ contains other shifts of $\mathbf{x}$ less than n, we know that we are in the matrix case and can build a companion matrix for the recurrence. Then all solutions grow exponentially [42].

We now assume that we are in the nth strongly connected component that contains only one variable and that the components that $\mathbf{x}$ depends on are already processed. If in

$$\mathbf{x}^{(n)} \geq \mathbf{x}^{(j)} + \mathbf{R}, \mathbf{R} \text{ not empty}$$

one of the summands in $\mathbf{R}$ is exponential or a shift of $\mathbf{x}$, the growth of all solutions is exponential; if all of the summands can be set to periodic or polynomial lists of maximal degree d, the minimal solution list is polynomial of degree $d + 1$. For the additional constraints of the form $\mathbf{x}^{(i)} \geq \mathbf{R}$ we set the growth of $\mathbf{x}$ to the maximal growth in $\mathbf{R}$.

For the inner nodes of the tree of strongly connected components with variable set X' with $|X'| > 1$, the growth is exponential.

Overall, we have the following. Let a system of lower bound list constraints be given.

- We can decide if the growth rate of the system is polynomial (including degree) or exponential.

- If there is only one constraint per variable, we can determine the exact growth rate also in the exponential case.
- If there are at least two constraints for one variable, we can determine a nontrivial upper bound (i.e. not $\hat{\infty}$) on the optimal solution.

The exponential growth for matrices that are not irreducible, will be estimated more accurately in Sections 5.2 and 5.4.

5.2 Optimization for one Constraint per Variable

In case that we have only one constraint per variable, the companion matrix for the system is unique and the following holds.

Let us fix a constraint set with one constraint per variable and also fix a particular variable in that set. We are interested in determining the asymptotically least solution for that variable. Let us write $\mathbb{L}$ for the set of solutions to that variable. Also let A be the companion matrix to the constraint system.

Lemma 14. *There is $d \in \mathbb{N}$ and polynomials $p_i(n)$ and complex numbers λ_i for $i = 1 \dots d$ and a d-dimensional convex set $\mathcal{C}$ specified by finitely many linear inequalities such that*

- *For each $(v_1, \dots, v_d) \in \mathcal{C}$ the list $(x(n))_n$ with $x(n) = \sum_{i=1}^{d} v_i p_i(n) \lambda_i^n$ is in $\mathbb{L}$*
- *For each $(x(n))_n \in \mathbb{L}$ there exists $(v_1, \dots, v_d) \in \mathcal{C}$ such that the list $(x(n))_n$ satisfies $x(n) \geq \sum_{i=1}^{d} v_i p_i(n) \lambda_i^n$.*

Proof. Clearly, every solution is bounded below by a solution to the associated system of linear recurrences obtained by replacing inequality by equality. These solutions can be written in the form $x(n) = e^T A^n v$ where A is the companion matrix, e is a unit vector singling out the variable we're focusing on and v is a start vector which must satisfy the arithmetic constraints. Since the latter form a convex set we can conclude using the standard formula [46] for powers of a Jordan normal form. We also notice that the λ_i are among the eigenvalues of A. □

Lemma 15. *Let $\lambda_1, \dots \lambda_d \in \mathbb{C}$ be pairwise distinct and for all i let $|\lambda_i| = 1$. Let the*

numbers $v_1, \ldots, v_d$ not all be 0. Then there is a constant c, such that

$$\left| \sum_i v_i \lambda_i^n \right| \geq c \text{ for infinitely many } n$$

Proof. Let $z_n = \sum_i v_i \lambda_i^n$. Not all of the $z_0, z_1, ...z_{d-1}$ can be zero. This can be seen using the Vandermonde matrix (which is the companion matrix here) and the formula

$$\begin{pmatrix} z_0 \\ \vdots \\ z_{d-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ \lambda_1 & \lambda_2 & \ldots & \lambda_d \\ \lambda_1^2 & \lambda_2^2 & \ldots & \lambda_d^2 \\ & \ldots & & \\ \lambda_1^{d-1} & \lambda_2^{d-1} & \ldots & \lambda_d^{d-1} \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$$

where the matrix determinant is nonzero given that the λ_i are distinct. Thus let $z_j \neq 0$. If the arguments of all λ_i are rational multiples of 2π, then let N be the least common multiple of the denominators and we have $z_{j+kN} = z_j, k = 0, 1, 2, 3, 4, \ldots$. In the other case, $c = |z_j|$ may not be reached again, but since we can approximate irrational arguments with arbitrary precision by rationals, we know that at least $\frac{c}{2}$ will be exceeded infinitely often. □

Lemma 15 then yields the following result.

Theorem 3. *Let $\mathbb{L}$ be the set of solutions for a particular variable $\mathbf{x}$ in a list constraint system with one constraint per variable. We can effectively find (in polynomial time) $k \in \mathbb{N}$ and $\mathbb{R} \ni r \geq 0$ and $\mathbb{R} \ni c > 0$ such that*

- *for all $\mathbf{x} \in \mathbb{L}$*

$$\mathbf{head}(\mathbf{x}^{(n)}) \geq cn^k r^n \text{ for infinitely many } n.$$

- *there exist a constant $c' \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{L}$ with*

$$\mathbf{head}(\mathbf{x}^{(n)}) \leq c' n^k r^n.$$

Thus the asymptotic growth of the minimal solution is $n^k r^n$.

Proof. Write $x_1(n), \ldots, x_d(n)$ for the functions $x_i(n) = p_i(n)\lambda_i^n$ from Lemma 14 and let $\mathcal{C}$ be the convex set of start vectors from that lemma. We assume w.l.o.g. that the x_i are sorted in a way that those x_i which belong to an eigenvalue of greater modulus have a greater index than those x_i with a smaller eigenvalue and also that those x_i with equal eigenvalue are grouped together and those x_i with same eigenvalue are sorted by the degree of their polynomial p_i.

Now we want to find out, whether there are initial values $v \in \mathcal{C}$ such that in the representation of v as a linear combination of the x_i the "greatest" x_i (i.e. those that belong to the greatest eigenvalue λ_{x_i}), do not appear. For instance, if x_d and x_{d-1} belong to the eigenvalue λ with $|\lambda| > \max(\lambda_{x_{d-2}}, \ldots\ \lambda_{x_1})$, then we check whether there is $v \in \mathcal{C}$ with $(0 \ldots 001)v = 0$ and $(0 \ldots 010)v = 0$. We continue this way until we have found $r \in \mathbb{R}$, such that there is necessarily an x_i with corresponding eigenvalue λ in the linear combinations for all $v \in \mathcal{C}$ with $|\lambda| = r$, but none of modulus greater than r.

Let us now consider all x_i with corresponding eigenvalue of modulus r. Those x_i with smaller corresponding eigenvalue can be neglected. Now we try, by an iteration that is similar to the one above, to find $v \in \mathcal{C}$ such that the coefficient of the x_i whose polynomial has maximal degree, is zero. For example, let λ and μ be two eigenvalues of modulus r and x_i functions $ax^m + \mathcal{O}(x^{m-1}\lambda^n)$ (and respectively $bx^m + \mathcal{O}(x^{m-1}\lambda^n)$ and $ex^m + \mathcal{O}(x^{m-1}\mu^n)$) and all other x_i with polynomials of degree less than m. Let v_a, v_b, v_e be the components of v for the functions x_i. We then check if there is a $v \in \mathcal{C}$ with $av_a + bv_b = 0$ and $v_e = 0$. If yes, the $\deg(k) < \deg(m)$; if no, it is exactly k, according to Lemma 15.

□

5.3 Difficulties in the Estimation of Growth Rates

If there are several (say $k_{\mathbf{x}}$)constraints per variable $\mathbf{x}$, where m' is the maximum order of the constraints and without loss of generality all indices on the left hand side equal n, then we could build several companion matrices, one for each of the $m = \prod_{\mathbf{x}} k_{\mathbf{x}}$ combinations of constraints, $A_1, \ldots, A_m$, and calculate in each step of the recurrence the maximal value for $\vec{\mathbf{x}}_n$, such that $\vec{\mathbf{x}}_n \geq \max_i(A_i\vec{\mathbf{x}}_{n-1})$.

If there are two or more constraints on the same variable, and all variables mutually

dependent, we can again construct nonnegative, aperiodic and irreducible matrices for each of them and such that they have the same dimension. We first bring them on the same order. Then the matrix may become reducible, or be no longer aperiodic.

Example 16. *Let the constraints be*

$$\mathbf{x}^{(n)} \geq 3\mathbf{x}^{(n-1)},$$
$$\mathbf{x}^{(n)} \geq 2\mathbf{x}^{(n-1)} + 2\mathbf{x}^{(n-2)}.$$

The second constraint is of order two, the other is of order one. Then we can set up a matrix for the first constraint, namely

$$A = \begin{pmatrix} 3 & 0 \\ 1 & 0 \end{pmatrix},$$

which is not irreducible, or we can bring it to order two by substituting and obtain

$$\mathbf{x}^{(n)} \geq 9\mathbf{x}^{(n-2)}, B = \begin{pmatrix} 0 & 9 \\ 1 & 0 \end{pmatrix},$$

which is not aperiodic and not sound since we only have the implication

$$\mathbf{x}^{(n)} \geq 3\mathbf{x}^{(n-1)} \Rightarrow \mathbf{x}^{(n)} \geq 9\mathbf{x}^{(n-2)},$$

but not necessarily the converse.

We choose the first option and show that the Perron-eigenvector remains if we lift the order in the following way: Let A' be the quadratic $(k \times k)$ submatrix corresponding to the recurrence part of the constraint on $\mathbf{x}$. Then

$$A = \begin{pmatrix} A' & B \\ D & C \end{pmatrix},$$

for suitable matrices B, D and a square matrix C. The new matrix A^{new} is defined as

$$A^{new} = \begin{pmatrix} A' & (0,\ldots,0)^t & B \\ (0,\ldots,0,1) & 0 & (0,\ldots,0) \\ D & (0,\ldots,0)^t & C \end{pmatrix}.$$

This matrix need not be irreducible, but it still has the same (unique) maximal eigenvalue, which is positive and real and simple. By calculating the characteristic polynomial of the matrix with application of the Laplace formula for the $(dim(A')+1)$'st row, we get

$$\det\begin{pmatrix} A'-\lambda Id & (0,\ldots,0)^t & B \\ (0,\ldots,0,1) & -\lambda & (0,\ldots,0) \\ D & (0,\ldots,0)^t & C-\lambda Id \end{pmatrix} = (-1)^{k+1}(-\lambda)\det\begin{pmatrix} A'-\lambda Id & B \\ D & C-\lambda Id \end{pmatrix},$$

because the other summand $(-1)^{k-1}\cdot 1\cdot \det(A^{new}_{k+1,k})$ is zero since the matrix $(A^{new}_{k,k+1})$ is obtained from A^{new} by deleting row $k+1$ and column k has a column that consists only of zeros. Thus the characteristic polynomial is multiplied by λ and there is only an additional zero eigenvalue and the growth rate remains unchanged.

But unfortunately, another problem appears: it is possible that the matrices where the maximum is taken are always changing, such that we do not obtain a formula that can be efficiently computed. For the multiple constraints, we form the maximum of several matrix multiplications in each step: Let $\mathbf{x}_1,...,\mathbf{x}_n$ be the variables in the constraint system and k_i the number of constraints on the variable $\mathbf{x}_i$. Then the recursively defined sequence

$$\mathbf{x}_n = \begin{pmatrix} \max((A_1\mathbf{x}_{n-1})_1,...,(A_l\mathbf{x}_{n-1})_1) \\ ... \\ \max((A_1\mathbf{x}_{n-1})_n,...,(A_l\mathbf{x}_{n-1})_n) \end{pmatrix}$$

with matrices $A_1,...,A_l$, where $l=\prod_i k_i$, is an optimal solution. If the maximum is taken always for the same matrix from the ith step on, say for A_1, which means that

$$\forall n\geq 1.\forall w\in(A_1|,\ldots|A_k)^*A^n, |w|=i.\forall k.A_1^n w\mathbf{x}\geq A_kA_1^{n-1}w\mathbf{x},$$

then $\forall\mathbf{x}.\mathbf{x}_{n+i}\geq\mathbf{x}_{minn+i}\coloneqq A_1^n w\mathbf{x}$.

But there is not necessarily one matrix, for which the maximum value is taken in all steps beginning from a finite position. An example are the matrices

$$A=\begin{pmatrix}2&3\\4&1\end{pmatrix}, B=\begin{pmatrix}3&2\\4&1\end{pmatrix}. \tag{5.3}$$

These matrices belong to a system of two constraints $\mathbf{x}^{(n)}\geq 2\mathbf{x}^{(n-1)}+3\mathbf{y}^{(n-1)}$ and $\mathbf{x}^{(n)}\geq$

$3\mathbf{x}^{(n-1)} + 2\mathbf{y}^{(n-1)}$ on the first variable $\mathbf{x}$ and only one constraint $\mathbf{y}^{(n)} \geq 4\mathbf{x}^{(n-1)} + \mathbf{y}^{(n-1)}$ for the second variable $\mathbf{y}$. We have for all initial vectors v

$$(Av > Bv \Rightarrow BAv > AAv) \wedge$$
$$(Av < Bv \Rightarrow ABv > BBv).$$

This can be seen by a case distinction. If and only if $v = (v_1, v_2)^t$ and $v_1 > v_2$, Av is less than Bv. Thus in this step B will be the chosen matrix. After application of B, the resulting vector $v' = (v'_1, v'_2)^t$ will have the second component v'_2 greater than the first v'_1 and thus in the next step A will be chosen and so on. If we list the matrices where the maximum is taken in each step, there are sequences which are not finally constant (as in the example), and other examples suggest that we do not know yet if they are even finally periodic.

For instance, this list for the matrices

$$\begin{pmatrix} 2 & 7 & 7 & 0 \\ 6 & 4 & 5 & 1 \\ 0 & 8 & 5 & 3 \\ 7 & 3 & 2 & 4 \end{pmatrix} \text{ and } \begin{pmatrix} 5 & 7 & 3 & 1 \\ 6 & 4 & 5 & 1 \\ 0 & 8 & 5 & 3 \\ 7 & 3 & 2 & 4 \end{pmatrix}$$

seems to be aperiodic or at least has an aperiodic initial part of length more than 100 when starting with initial vector $(0, 2, 5, 4)^t$.

Although there is an infinite alternation, in the case of formula (5.3) the problem is easily solved by taking the new matrix

$$C \coloneqq \begin{pmatrix} 0 & B \\ A & 0 \end{pmatrix}$$

and

$$\begin{pmatrix} \vec{\mathbf{x}}_n \\ \vec{\mathbf{z}}_n \end{pmatrix} = C \begin{pmatrix} \vec{\mathbf{x}}_{n-1} \\ \vec{\mathbf{z}}_{n-1} \end{pmatrix}$$

with $\vec{\mathbf{z}}$ an auxiliary vector. More precisely, we set $\vec{\mathbf{z}}_n = A\vec{\mathbf{x}}_n$ and get

$$\vec{\mathbf{x}}_n = \begin{cases} \begin{pmatrix} (C^n(\vec{\mathbf{x}}_1, \vec{\mathbf{z}}_1)^t)_1 \\ (C^n(\vec{\mathbf{x}}_1, \vec{\mathbf{z}}_1)^t)_2 \end{pmatrix}, & \text{if } n \text{ even,} \\ \\ \begin{pmatrix} (C^n(\vec{\mathbf{x}}_1, \vec{\mathbf{z}}_1)^t)_3 \\ (C^n(\vec{\mathbf{x}}_1, \vec{\mathbf{z}}_1)^t)_4 \end{pmatrix}, & \text{if } n \text{ odd.} \end{cases}$$

Note that according to Remark 1, we cannot take the matrix product AB and introduce a new constraint $ABx \geq x$, because the information on the intermediate step ($Bx \geq x$) would be lost and that violates soundness, because there other minimal solutions than those of the original system become possible.

In case there is in total only one variable (i.e. we have two or more homogeneous recurrences $r_1, \ldots, r_n$), we can already prove that if the maximum is obtained for the same matrix A as many times as the degree d_1 of the recurrence r_1 (w.l.o.g. with the index 1) for A is, then it will always be taken for A. Assume for any B

$$A^i\vec{\mathbf{x}}_n \geq BA^{i-1}\vec{\mathbf{x}}_n, i = 1, ..., d_1.$$

Consider the characteristic polynomial p of the recurrence r_1. We have r_1 equal to $x_n = \sum_{i=1}^{d_1} a_i x_{n-i}$ and p equal to $x^{d_1} - \sum_{i=0}^{d_1-1} a_{d_1-i}x^i$, of which A is a zero by the Cayley-Hamilton theorem, and calculate

$$A^{d_1+1}\vec{\mathbf{x}}_n = \sum_{i=1}^{d_1} a_{d_1-i-1}A^i\vec{\mathbf{x}}_n \geq \sum_{i=1}^{d_1} a_{d_1-i-1}BA^{i-1}\vec{\mathbf{x}}_n = B\sum_{i=0}^{d_1-1} a_{d_1-i}A^i\vec{\mathbf{x}}_n = BA^{d_1}\vec{\mathbf{x}}_n \ .$$

The above considerations lead us from linear list constraints to the field of (possibly inhomogeneous) positive recurrences with maximums. We give a heuristic to approximate the behavior of optimal solutions in Section 5.1, which applies here and allows us to find a nontrivial upper bound on the systems growth.

A more accurate estimation is always possible, if the chain of "stronger" matrices (i.e. matrices that deliver stronger bounds) becomes periodic. We conjecture that this is the case.

We make another observation. If there is more than one constraint on a variable $\mathbf{x}$, we have at least two matrices A_1, A_2 of (w.l.o.g. common) dimension k such that the vector $\vec{\mathbf{x}}_n = (\Diamond(\mathbf{x}^{(n)}), \ldots, \Diamond(\mathbf{x}^{(n-k)}))$ is defined as $\vec{\mathbf{x}}_n \coloneqq \max(A_1, A_2)\vec{\mathbf{x}}_{n-1}$. Then we observe that

$$\vec{\mathbf{x}}_n \leq \max(||A_1||, ||A_2||)^n \vec{\mathbf{x}}_0,$$

where $||A||$ denotes the spectral norm of A.

Remark 11. *There are constraints which lead to matrices where this bound is taken.*

Example 17. *The matrices*

$$A = \begin{pmatrix} 0 & 101 \\ 100 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & 100 \\ 101 & 0 \end{pmatrix}$$

both have spectral norm 101 and starting with the vector $(0,1)^t$ *the maximum is taken alternating between* A *and* B*. We have*

$$x_{2n} = ||(BA)^n(0,1)^t|| = ||(0, 101^{2n})|| = 101^{2n} = \max(||A||, ||B||)^{2n}||(0,1)^t||.$$

In this case, the heuristic would take the optimal combination of constraints and get the matrix

$$C = \begin{pmatrix} 0 & 101 \\ 101 & 0 \end{pmatrix}.$$

We can show that in dimension two or in the radial case (i.e. when we have matrices with spectral radius equals to their spectral norm) if there is infinite alternation for x and y then the growth rates (i.e. the dominant eigenpairs) coincide. Let λ be the Perron-eigenvalue for A and x the corresponding eigenvector, i.e. $Ax = \lambda x$. Let similarly $By = \mu y$.

Let first A and B be radial, i.e. the spectral radius equals the spectral norm.

Then we have for the $w = w_1 ... w_n \in (A|B)^n$ that makes the optimal choices of matrices, that

$$\lambda^n x = A^n x \leq wx \leq ||w_1||||w_2||...||w_n||x = \lambda^k \mu^l x, k + l = n,$$

where k is the number of As in w and l the number of Bs. Thus $\mu \geq \lambda$. On the other hand,

we also have

$$B^n y = \mu^n y \leq wy \leq ||w_1||||w_2||...||w_n||y = \lambda^k \mu^l y, k + l = n,$$

and thus $\lambda \geq \mu$. So $\lambda = \mu$ is the Perron-eigenvalue and $x = y$.

Now let the dimension be two without the radial assumption. Let

$$A = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}, B = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}, a_3 = b_3, a_4 = b_4.$$

We have the following because x and y are eigenvectors:

$$\begin{aligned} a_1 y_1 + a_2 y_2 &= c y_1, \\ a_1 x_1 + a_2 x_2 &= \lambda x_1, \\ b_1 x_1 + b_2 x_2 &= d x_1, \\ b_1 y_1 + b_2 y_2 &= \mu y_1, \\ a_3 y_1 + a_4 y_2 &= \mu y_2, \\ a_3 x_1 + a_4 x_2 &= \lambda x_2. \end{aligned}$$

In matrix form with the roles of the x, y's and the a, b's changed, this is

$$\begin{aligned} \begin{pmatrix} y_1 & y_2 \\ x_1 & x_2 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} &= \begin{pmatrix} c y_1 \\ \lambda x_1 \end{pmatrix}, \\ \begin{pmatrix} y_1 & y_2 \\ x_1 & x_2 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} &= \begin{pmatrix} \mu y_1 \\ d x_1 \end{pmatrix}, \\ \begin{pmatrix} y_1 & y_2 \\ x_1 & x_2 \end{pmatrix} \begin{pmatrix} a_3 \\ a_4 \end{pmatrix} &= \begin{pmatrix} \mu y_2 \\ \lambda x_2 \end{pmatrix}. \end{aligned}$$

Let

$$e = \det \begin{pmatrix} y_1 & y_2 \\ x_1 & x_2 \end{pmatrix} = y_1 x_2 - x_1 y_2.$$

We assume that x and y are linearly independent, thus $e \neq 0$. We now invert the matrix C

such that

$$C^{-1} = \frac{1}{e}\begin{pmatrix} x_2 & -y_2 \\ -x_1 & y_1 \end{pmatrix}$$

and

$$C^{-1}\begin{pmatrix} cy_1 \\ \lambda x_1 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix},$$
$$C^{-1}\begin{pmatrix} \mu y_2 \\ \lambda x_2 \end{pmatrix} = \begin{pmatrix} a_3 \\ a_4 \end{pmatrix},$$
$$C^{-1}\begin{pmatrix} \mu y_1 \\ dx_1 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Then $0 < a_3 = x_2\mu y_2 - y_2\lambda x_2$ and thus $\mu > \lambda$ if $e \geq 0$ and if $e \leq 0$ then $\lambda > \mu$. If $e \geq 0$, further $-x_1\mu y_1 + dy_1x_1 > 0 \Leftrightarrow d > \mu$ and $-x_1cy_1 + y_1\lambda x_1 > 0 \Leftrightarrow \lambda > c$. It follows that $d > \lambda$ and $\mu > c$, this means that $Ax < Bx$ (meaning inequality w.r.t. the only unequal component, in other words that B is stronger than A in x) and $Ay < By$ or, in the symmetric case that $e < 0$, that $Ax > Bx$ and $Ay > By$.

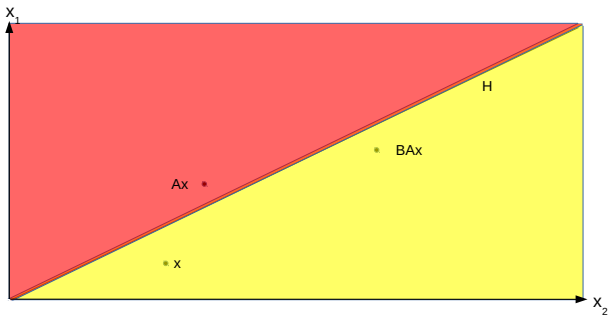


Figure 5.1: Partition by the Hyperplane H

This means that x and y lie on the same side of the hyperplane $H = \{x|Ax = Bx\}$, which is here a line. Let us assume they are on the A side. The situation can be imagined like in

Figure 5.1. On the yellow side, A is stronger, and on the red side, it is B. If x lies "below" y (i.e. considered as vectors or numbers in the complex plane, x has the smaller angle), and $y = ax + bv$, where v is the other eigenvector of A (which exists, since the Perron-eigenvalue is simple, and which belongs to an eigenvalue less than λ), then $Ay = \lambda ax + cbv$, where $|c| < \lambda$. Thus Ay lies below y and hence again on the A side of the line H, see Figure 5.2.

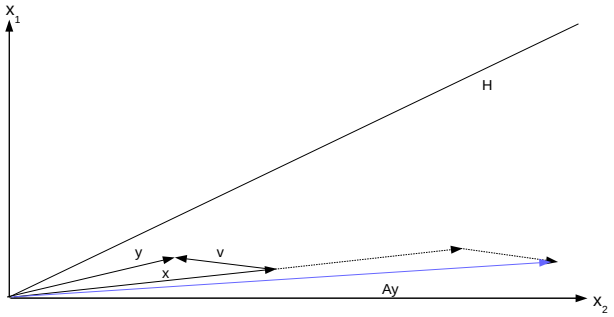


Figure 5.2: Eigenvalues and Eigenvectors: x below y

Now, if y lies "below" x, then again $y = ax + bv$ and $Ay = \lambda ax + cbv$, where $|c| < \lambda$. This case is a bit more complicated, see Figure 5.3.

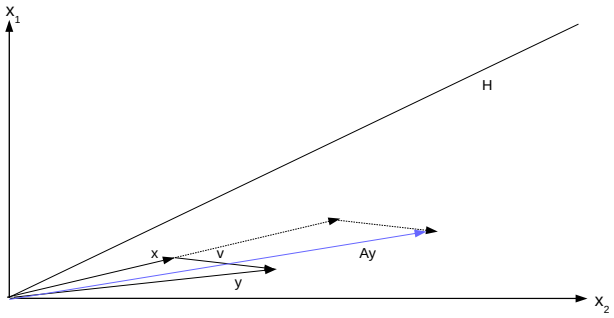


Figure 5.3: Eigenvalues and Eigenvectors: y below x

If Ay was on the B side, then, since $(BAy)_1 > (AAy)_1$, also BAy is on the B side. Moreover, by the same argument, we have the same for $BBAy$ and so on. Thus there is no oscillation between both sides.

If both eigenvectors are on the B side, then if x is below y a symmetric argument applies and if y is below x, then either Bx always stays on the B side or at some point it jumps to the A side where it stops oscillating. Overall, there is no oscillation for eigenvectors on the A or B side, and we know that there can only be oscillation if both eigenvectors are equal and lie in H.

5.4 Growth Rate Estimation in Polynomial Time for more than one Constraint per Variable

Luckily, we found out later, that there is related work that subsumes this conjecture and generalizes its proof to higher dimensions [45, 59]. Rothblum et al. characterize *totally expanding* matrix systems, which means that iterated application of a matrix to a starting vector return increasing, unbounded vectors as results. These matrix systems consist of a set of rows that can be combined arbitrarily by a so-called *policy* that picks rows and combines them to a quadratic matrix. Our constraints can be directly seen as rows, and we can combine them independently to obtain a companion matrix. A special case is the iteration

$$x(n+1) = \max(Ax(n), Bx(n), Cx(n)).$$

with component-wise maximum. Then for each n, there exists a matrix D, such that $x(n+1) = Dx(n)$ where the i'th line of D is either the i'th line of A, B or C. D is chosen according to the policy that delivers the maximal vectors, which is unknown.

We can now consider a policy or choice function π, which delivers such a matrix for each n. This is, a function π with $x(n+1) = \pi(n) \cdot x(n)$ for the right π that chooses according to the maximum. Vice versa, let the initial vector be fixed, then we can consider for each arbitrary π the sequence $x_\pi(n)$. That is, $x_\pi(n+1) = \pi(n)x_\pi(n)$.

Define now the (coordinate-dependent) geometric growth rate as

$$\gamma_\pi^i = \inf\{a > 0 \mid \lim_{n->\infty} \frac{|x_\pi(n)_i|}{a^n} = 0\}$$

Thus for given π is γ_π is the smallest exponential basis, that *flattens* the sequence x_π.

Further, define the actual geometric growth rate of the sequence as

$$\gamma^i = \inf\{a > 0 \mid \lim_{n->\infty} \frac{|x(n)_i|}{a^n} = 0\}$$

It is clear that $\gamma_\pi \leq \gamma$ for all π and $\gamma = \gamma_\pi$, for the particular π that makes the right choices. Rothblums result states that

$$\gamma = \gamma_\pi \text{ for a constant } \pi,$$

which chooses always the same matrix D independently of n. So, for such a constant π, the sequence is not identical to the maximal sequence, but it has the same asymptotic growth rate. So one could try all matrices generated by each policy, and take the maximal spectral radius or even better, the growth rate as a polynomial times an exponential expression as in Chapter 5.2. This is a priori exponential in the number of available rows or matrices, but Etessami recently presented a polynomial time algorithm for calculating π [55].

Our constraints fit into this framework. For instance, if we have the constraints

$$\begin{aligned}
\mathbf{x}^{(n+1)} &\geq 2\mathbf{x}^{(n)} + 5\mathbf{y}^{(n)}, \\
\mathbf{y}^{(n+1)} &\geq \mathbf{x}^{(n)} + 3\mathbf{y}^{(n)}, \\
\mathbf{x}^{(n+1)} &\geq 20\mathbf{x}^{(n)} + \mathbf{y}^{(n)}, \\
\mathbf{y}^{(n+1)} &\geq \mathbf{x}^{(n)} + 30\mathbf{y}^{(n)},
\end{aligned}$$

the possible policies give the matrices

$$\begin{pmatrix} 2 & 5 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 2 & 5 \\ 1 & 30 \end{pmatrix}, \begin{pmatrix} 20 & 1 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 20 & 1 \\ 1 & 30 \end{pmatrix}.$$

Thus we do not obtain a closed formula (i.e. we cannot calculate the exact $x(n)$ by this method), but the asymptotic growth rate by constructing a sequence which has the same

growth but is easier to calculate. In this example the maximal policy belongs to the last matrix.

There are a few things to note. Firstly, if we have an irreducible, aperiodic matrix, the geometric growth rate is equal for all coordinates.

Secondly, the results in loc. cit. hold in the general case and are independent of the initial value (some degenerate initial values may have smaller growth rate than 1). In our context, we must determine the initial vectors that also lie in the convex set given by the arithmetic constraints and in our case, we first search for such vectors with degenerate coordinates and pick those which belong to the smallest eigenvalues (cf. Section 5.2).

Lastly, our systems are always totally expanding in the sense of [45], because otherwise we would have no strictly increasing lower bound constraints.

Overall, we can always detect the growth rate of minimal solutions to a constraint system as a polynomial times an exponential expression. In case of an Perron-Frobenius matrix, we even have closed formulas in most cases, in all other cases, we have recursive matrix formulas for the optimal solutions.

Chapter 6

Decidability of Unilateral Tree Constraints

This chapter presents the theory of linear tree constraints. The main result is the decidability proof for the tree setting. Section 6.1 explains the difficulties that occur in the tree case and why it cannot be directly reduced to the list case. Section 6.2 presents the decidability proof. There, we use regular languages to describe parts of the trees that are zero, infinity or which have nodes in certain directly computable intervals. In contrast to the decision procedure, the growth rates results for lists can directly be adapted to trees by an observation that we show in Section 6.4. The characterizations in the decidability proof and the growth rates can also be used to construct concrete minimal solution trees for satisfiable systems. We explain that at the end of Section 6.4 and come back to it in Chapter 8.

6.1 Discussion of Solution Strategies

Our main theorem is a generalization of the list argument and states decidability also for the tree constraint satisfiability problem.

Before we state our decidability result, we observe that after the elimination procedure for trees the constraints are in a normal form for which all (sub)tree variables on the left hand side that are not eliminable, have at least one dependent (sub)tree on the right hand side. Additionally, one of the occurrences of each such variable in such a constraint must

be whole. Further, we have additional inequalities on these uneliminable subtrees. This normal form is a generalization of the normal form for unilateral linear list constraints, but it only increases the efficiency of the procedure of the next section, and it does not allow for a simpler structure, as in the list case. Also, a reduction to the list case is not directly possible.

When adapting the procedure to find cycles in the case of lists to the tree case, one has to note that there are two possible forms of substituting, since the order of labels matters. In the first case, we have constraints (from now on we denote labels and variables by normal italic letters, and v, w are finite successions of labels)

$$x \geq vy \wedge y \geq z, \tag{6.1}$$

which imply

$$x \geq vz.$$

In the second case, which never occurs simultaneously to the first, we have constraints

$$x \geq y \wedge wy \geq z, \tag{6.2}$$

from which follows

$$wx \geq z. \tag{6.3}$$

In the list case, we exploited the fact that all lists have "self-references" either back or forward in the list, the negative or positive cycles. They delivered bounds on the growth of these lists. In the tree case, we do not necessarily have these bounds. Although we can bring a single constraint into the form that it has only subtrees on the right that depend on the left (sub)tree, this property is not preserved when we bring all constraints on the same level, which was also a premise for the list case. This is best explained by an example.

Example 18. *We have the constraints*

$$x \geq lr(x),$$
$$r(x) \geq 3x.$$

Then these two constraints start on different levels of the tree. Unfolding yields

$$\begin{aligned} r(x) &\geq rlr(x), \\ l(x) &\geq llr(x), \qquad (6.4) \\ r(x) &\geq 3x. \qquad (6.5) \end{aligned}$$

Fist, we observe that inequality (6.4) is no longer in the normal form and that it also implies, together with (6.5), that $lx \geq 3ll(x)$, and that the left subtree of x was a priori undefined.

This example shows that we have to consider combinations of paths that make new nontrivial bounds derivable.

The next example shows that we also need to take into account inhomogeneities in form of independent summands.

Example 19. *The constraints*

$$\begin{aligned} \Diamond(x) &= 0, \\ \Diamond(llx) &\geq 1, \\ rx &\geq 2x, \\ lx &\geq 2x, \\ x &\geq lrx. \end{aligned}$$

There is a cycle $x \geq lrx \geq lx \geq x$ and implies that the whole tree x is repeated when going right and then left. Here it is possible to find a regular solution, namely

$$\begin{aligned} \Diamond(rx) = \Diamond(lx) &= 0, \\ llx == \hat{\infty}, rrx &= \hat{\infty}, \\ rlx = \hat{\infty}, lrx &= x. \end{aligned}$$

But this naive approach does not always lead to a solution.

Example 20. *Let the following constraints be given.*

$$\begin{aligned} lx &\geq x + y, \\ rx &\geq x + y, \\ \Diamond(y) &= 1, \\ y &\geq ry, \\ y &\geq ly. \end{aligned}$$

We assume that it is possible to set y to a tree with root 1 and the rest zeros. From level two on, we have the same bounds holding for all trees on level two and thus we can consider setting

$$\begin{aligned} llx = lx, rrx = rx, \\ lrx = rx, rlx = lx, \end{aligned}$$

obtaining a regular solution. But what if we have set

$$lrx = lx, rlx = rx?$$

This would also work in this concrete example. In general, we have no algorithm that carries out this choice and is able to pick the right trees from an infinite amount of possibilities.

As a first approach, we also tried to find a satisfying regular tree assignment. But this seems to be not always possible (cf. Example 23 below, or equivalently Chapter 2, Figure 2.4). If we would "abbreviate" the path, $ml(rl)^i$ at some point and set $(lr)^j lx = (lr)^k lx$ for $k \leq j$, this would fit in the situation that we have no other constraints, but as soon as we have interactions with other constraints, it becomes difficult to pick j and k. For instance, if we can (somehow) derive that $lrlx \geq 2lx$, then $k = 0$ and $j = 1$ are no longer possible, but we have not found a way to ensure that a certain choice of j and k will be suitable. This led to the construction in the next section, which is independent of a regularity assumption on the trees themselves.

Our next approach was to try to find all cycles and assign a weight to them. For instance,

if we modify Example 23 such that the constraints are

$$\begin{aligned} mlx &\leq x, \\ lx &\geq 2x, \\ lrx &\geq 5x, \\ rlx &\leq x, \\ mx &\geq 3x, \end{aligned}$$

then we obtain a cycle $x \geq mlx \geq mlrlx \geq 2mlrx \geq 10mx \geq 30x$, which implies that the root of x is zero. But not all cycles are finite. Here the set of roots equal to zero are of infinite number: $x, mlrx, mlrlx, mlrlrlx, \ldots$, and we have an infinite amount of cycles depending on which constraints we apply.

Thus it becomes more complicated to detect cycles. If we would draw a weighted graph as for the list case (or construct a weighted automaton, cf. [60]), we would have to include the "direction" using label information in each path. This already reminds of a finite automaton and indeed, it turns out that a nondeterministic finite automaton instead of a weighted graph is more suitable. We show later that such a construction solves the problem of finding cycles and represents them as a regular language (when the starting point is fixed).

Otherwise, if we do not include label information, we have to introduce variables for some of the subtrees and include paths, e.g. from node x to nodes lx and rx. Then, in contrast to the list case, these paths do not deliver a stable set under composition. For lists, if we add a new constraint for each closed path, it is not possible to derive new paths by repeating them. For instance, if the closed path is $lllllx \geq x$, then, the repetition yields $lllllllllx \geq x$. For trees, we have multiple labels and thus not all cycles are chained existing cycles. In the running example 23, we have (among others) cycles $ml(rl)^i x = x$, and none of them can be obtained just by repetition of existing cycles.

That complicates the calculation and the automaton needed for it is introduced in the next section. The idea for constructing this automaton, which is explained in detail in the next section, is to define a stack automaton as an intermediate step (cf. section 6.2 for the construction). There we write x on the stack, modify the stack content as the constraints describe, and then compare the result to y. If they are equal, we know that $x \geq y$ is a consequence of the constraints.

Then, we show that although the stack size can increase beyond an arbitrary amount, it will finally decrease again and we can "abbreviate" these calculation steps.

Let L be the set of pairs of trees with possibly prefixed labels x, y such that the constraints imply $x \geq y$. In Section 6.2, we will show that L can be interpreted as a contextfree language by the encoding $xy^R \in L$ for the pair (x, y) that satisfies $x \geq y$. We will show the following for fixed trees x and y:

Claim 1. *Let x be a whole tree and y an arbitrary (sub)tree (i.e. a whole tree possibly with prefixed labels). The language $L_{x,y} \coloneqq \{u \in \Sigma^* | C \text{ implies } ux \geq y\}$, where Σ is the label set, is regular.*

For the list case, we argue directly by Parikh's theorem as follows. A special case of this theorem is that context free languages over an unary alphabet are regular. $L_{x,y}$ is context free, by defining $L'_{x,y} = L \cap \Sigma^* xy^R$ and because context free languages are closed under intersection and quotient with regular languages, here

$$L_{x,y} = L'_{x,y}/\{xy^R\} = \{w | w.xy^R \in L'_{x,y}\}.$$

The remaining alphabet is unary, because it consists only of tl applications, it follows that $L_{x,y}$ is regular.

The proof idea of Claim 1 is as follows. We claim that the following formula holds for label words $u, u', \ldots$:

$$\begin{aligned}
&\vdash ux \geq y \Leftrightarrow \\
&\exists u', u''.u = u'u'' \wedge \exists \text{ Constraint } u''x \geq vz \\
&\wedge (|v| < |u''| \vee (\exists z'. \vdash u'vz \geq wz' \wedge |w| < |u|)) \\
&\wedge \vdash vz \geq y
\end{aligned}$$

There are three things to notice here. First, we do not necessarily have an explicit constraint that states $u'vz \geq wz'$. This may be the consequence of several constraints applied after each other. Second, the length of the label prefix decreases — and this happens after a finite number of steps. Third, the variables z, z', x are not allowed to be subtrees, whereas y is allowed to contain prefixed labels.

This is best explained in a picture. Every step consists of one constraint application. We just jump over the intermediate peaks successively and go directly to a state where the

stack content has been removed again. In the next section, we will formally prove that this is possible by introducing a proof system to derive inequalities and by giving rules for the computation of the language $L_{x,y}$ using dynamic programming and nondeterministic finite automata.

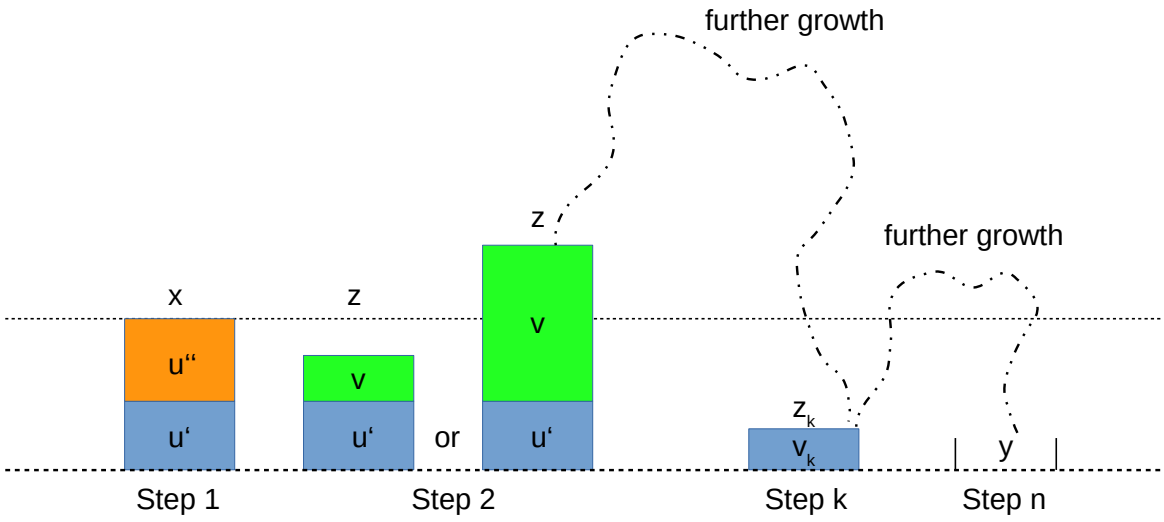


Figure 6.1: Stack Size Determined by Constraints

6.2 Decidability

We now establish our main result (Theorem 7), namely that satisfiability of unilateral linear tree constraints is decidable. The proof is structured as follows: We first observe that unsatisfiability is semi-decidable. We then show that we can reduce a set of constraint systems that contains all satisfiable ones to linear programming using the following arguments:

- We describe how to derive inequalities that follow from a set of constraints using a sound and complete proof system,
- characterise the set of trees greater than a fixed tree as a regular language,
- use these languages to find all trees bounded from above and from below,
- show that all other trees can be set to zero or infinity without changing the satisfiability properties of the system, and finally
- reduce the constraints to an equisatisfiable linear program.

This means satisfiability is semi-decidable. Together with the semi-decidability of unsatisfiability, this implies that satisfiability is decidable.

6.2.1 Unsatisfiability is Semi-Decidable

From now on we are in the realm of UTC and omit the word "unilateral". An *unfolding step* for a constraint $x \geq S_1 + \cdots + S_n$, where S_i are (sums of) tree variables, consists of adding the arithmetic constraint $\Diamond(x) \geq \Diamond(S_1) + \cdots + \Diamond(S_n)$ and application of the (LabelSum) rule in Figure 6.2 to obtain the constraints for the next step.

$$\frac{S = \sum_i S_i \qquad TC \vdash x \geq S}{TC \vdash l(x) \geq \sum_i l(S_i)} \text{ (LabelSum)}$$

$$\frac{S = \sum_i S_i \qquad TC \vdash x \geq S}{TC \vdash \Diamond(x) \geq \sum_i \Diamond(S_i)} \text{ (Root)}$$

Figure 6.2: Label Application Rules

Each such step delivers a new, bigger set of arithmetic constraints that can be seen as a linear program. We have a succession of programs $(P_i)_{i\geq 0}$.

Lemma 16. *The constraint system (AC, TC), where AC is a set of arithmetic and TC a set of tree constraints, is unsatisfiable if and only if one of the linear programs P_i is unsatisfiable.*

Proof. The proof is basically the same as the compactness proof for infinite dimensional 0-1-programming in [61]. It uses the facts that $\mathbb{D}$ is a compact set and that the matrix that belongs to the constraints has a certain quasi-finite structure. When interpreting the constraints as infinitely many arithmetic inequalities, we can see the matrix that describes them as an infinite dimensional object. The key observation is now that we can represent the constraints using a matrix such that in each row, there are only finitely many nonzero entries. In [61], they call this property "finite support" of an infinite dimensional matrix.

With these two properties we can show that an infinite set of conditions is violated, if and only if already one of the finite subsets consisting on the first n conditions, contains a contradiction for some n. □

Thus, if there is a contradiction, we find it, but if the system is satisfiable, this will not terminate. In the remainder of this section, we give a procedure that terminates in the satisfiable case.

6.2.2 The Set of Trees Greater than a Fixed Tree is a Regular Language

We now describe the implications of a constraint system as given in Figure 6.3. Intuitively, if a constraint $x \geq y$ holds for trees x and y, then also each subtree of x is greater than or equal to the subtree of y with the same label, and similarly the root of x must be greater or equal to the root of y. Further, the greater-or-equal relation must be transitive.

$$\frac{}{TC \vdash ux \geq ux} \text{ (Reflexivity)}$$

$$\frac{TC \vdash x \geq y}{TC \vdash l(x) \geq l(y)} \text{ (Label)}$$

$$\frac{x \geq y_1 + ... + y_n \in TC \qquad TC \vdash uy_i \geq z}{TC \vdash ux \geq z} \text{ (Transitivity)}$$

Figure 6.3: Proof System for Unilateral Tree Constraints

Tree expressions are of the form ux where $u : \Sigma^*$ and Σ is the set of tree labels like *left*, *right*, etc., and x is a variable. We use letters x, y, z for variables and for tree expressions.

The judgement $TC \models x \geq y$, where x and y are expressions, has the meaning that $x \geq y$ follows semantically from the tree constraints in TC. That is, every valuation that satisfies TC also satisfies $x \geq y$. The judgement $TC \vdash x \geq y$ means that the inequality $x \geq y$ is derivable by the rules in Figure 6.3.

Theorem 4. *The proof system in Figure 6.3 is sound and complete (i.e. $TC \models x \geq y \Leftrightarrow TC \vdash x \geq y$).*

Proof. Soundness is trivial. For completeness we argue as follows. Let T be the set of all tree expressions over the variables in TC and define a graph $G = (V, E)$ where $V = T$ and

$$\begin{aligned} E = &\{(x, y)| \text{ there is a constraint } x' \geq y_1 + \cdots + y_n \in TC \text{ and} \\ &u : \Sigma^*.x = ux' \text{ and } y = uy_i \text{ for some } i\}. \end{aligned}$$

Now fix a tree expression x_0 and define a valuation η in such a way that $\eta(\Diamond(z)) = 0$ if z is reachable from x_0 in G and $\eta(\Diamond(z)) = \infty$ otherwise. We claim that η satisfies TC. Indeed, suppose that $x \geq y_1 + \cdots + y_n$ is a constraint in TC. We must show that $\eta(\Diamond(ux)) \geq \eta(\Diamond(uy_1)) + \cdots + \eta(\Diamond(uy_n))$ holds for all $u : \Sigma^*$. If ux is unreachable from x_0 then $\eta(\Diamond(ux)) = \infty$ and the inequality holds. Otherwise, if ux is reachable then $uy_1, \ldots, uy_n$ are also reachable and the inequality holds as well.

Now suppose that $x_0 \geq y$ is an inequality that is not derivable from TC. In this case, y is not reachable from x_0 in G. The valuation η constructed above then satisfies TC yet $\eta(x_0) = 0$ and $\eta(y) = \infty$ so $x_0 \geq y$ is not a semantic consequence of TC. □

For the next step, we are interested in the set $L_z^{\geq}$ of tree expressions greater or equal to a fixed tree expression z; in short all x such that $TC \vdash x \geq z$. Let us define the language $L_{x,y} \coloneqq \{u \mid TC \vdash ux \geq y\}$ with x and y fixed tree variables as an auxiliary step to compute $L_z^{\geq}$.

Theorem 5. *The language $L_{x,y}$ is regular.*

Proof. We construct a finite automaton that accepts a word w, if and only if $TC \vdash wx \geq y$.

With the proof system in Figure 6.3, we can first build a pushdown automaton $\mathcal{A}$ from TC, that reads no input and such that $u : L_{x,y}$ if and only if $\mathcal{A}$ accepts beginning from stack ux.

We give the idea for the construction of a slightly more general pushdown automaton, namely an automaton that accepts a word $vxyw^r$ (w^r denotes the word w reversed) if and only if the constraints imply $vx \geq wy$. Acceptance is by empty stack, and we start by writing v on the stack while we are in a so-called "write-state", then go into a state named "x", there modify vx nondeterministically and without reading from the input, as the constraints describe (possibly going to state "y" for another variable y). After that, we leave state "y" (or "x") and go into a "compare-state" where we compare the obtained stack with w and empty it if they both are equal.

Example 21. *Consider the constraints*

$$lr(x) \geq rr(x) \qquad\qquad lr(x) \geq l(y)$$

The pushdown automaton $\mathcal{A}_0 = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ such that

$$Z = \{z_0, z_\infty, z_x, z_y, z', z''\}, \Sigma = \{l, r, x, y\}, \Gamma = \{L, R, \#\}$$

and the transition relation δ is defined as depicted in Figure 6.4. Note that the lower case input symbols in Σ correspond to the according upper case letters in the stack alphabet Γ. Here z_0 is the write-state, z_∞ the compare-state and z', z'' are auxiliary states. For instance, in Figure 6.4, the auxiliary states are used as intermediate steps to rewrite $lr(x)$ to $rr(x)$ or to $l(y)$. We use the usual notation with triples for the current state, the read input symbol

$$
\begin{aligned}
\delta(z_0, a, B) &= (z_0, AB),\\
\delta(z_0, x, B) &= (z_x, B),\\
\delta(z_0, y, B) &= (z_y, B),\\
\delta(z_x, \epsilon, RB) &= (z', B),\\
\delta(z', \epsilon, LB) &= (z'', B),\\
\delta(z'', \epsilon, B) &= \{(z_x, RRB), (z_y, LB)\},\\
\delta(z_x, x, B) &= (z_\infty, B),\\
\delta(z_y, y, B) &= (z_\infty, B),\\
\delta(z_\infty, r, RB) &= (z_\infty, B),\\
\delta(z_\infty, l, LB) &= (z_\infty, B),\\
\delta(z_\infty, \epsilon, \#) &= (z_\infty, \epsilon),\\
B \in \Gamma^*, a &\in \{l, r\}.
\end{aligned}
$$

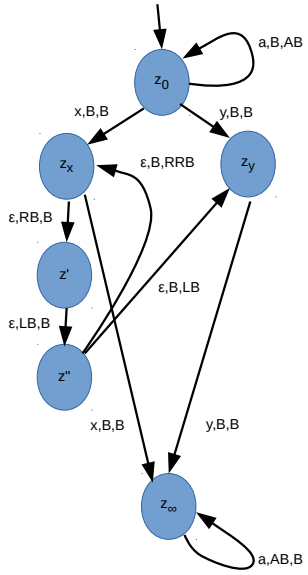


Figure 6.4: Example Pushdown Automaton

and the stack content, that are then mapped to the next state and the new stack content by δ. In the picture, the triples on the arrows mean the input symbol, the stack before and the stack after the transition.

The language L corresponding to this generalisation is not always a regular language: this can be seen with the Pumping Lemma. Assume L is regular and let p be the Pumping Lemma number and let the constraints be $lrx \geq lrlrx$ and consider words $(lr)^i xx(rl)^{i+1}$, which are implied by the constraints and thus in L. Then the word $\alpha = (lr)^p$ is obviously longer than p. For each division of $\alpha xx(rl)^{p+1} = uvw$ in three words u, v, w such that $|uv|$ is not longer than p, for instance with $v = lr$, holds $\forall k.(lr)^{p+k} xx(rl)^{p+1} \in L$, which is not implied by the constraints. In any case we have that the label word before the x's is longer than the label word after them, which is not a consequence of the constraint. This is a contradiction.

However, we can use L to show the regularity of the language $L_{x,y} = \{u \mid TC \vdash ux \geq y\}$ where the second variable y including its label word is fixed and x is fixed, but not its label word.[1] For that, we build the above mentioned pushdown automaton $\mathcal{A}$ similar to the

[1] In the list case, where we have only one label, this is a direct consequence of Parikh's theorem.

construction for $\mathcal{A}_0$ above. We can assume w.l.o.g. (possibly by introducing new states), that $\mathcal{A}$ has only transitions of the form $x \to y, x \xrightarrow{pop(a)} y$, or $x \xrightarrow{push(a)} y$, with x, y states that belong to variables. We then define the set

$$Q = \{(x, y) | x \to^* y\}$$

and enumerate it using dynamic programming and the rules in Figure 6.5. Now we obtain for $L_{x,y}$ the representation in Figure 6.6. From this we can read off a finite automaton $\mathcal{B}$

$$\frac{x \to y}{(x, y) : Q}$$

$$\frac{x \longrightarrow y \qquad (y, z) : Q \qquad z \longrightarrow w}{(x, w) : Q}$$

$$\frac{x \xrightarrow{push(a)} y \qquad (y, z) : Q \qquad z \xrightarrow{pop(a)} w}{(x, w) : Q}$$

Figure 6.5: Rules for Q

for $L_{x,y}$ directly: the states are the states of $\mathcal{A}$, for each pair $(x, y) \in Q$ we introduce an ϵ-move from state x to state y, and the $\mathcal{A}$-transitions $x' \xrightarrow{pop(a)} x''$ are the nontrivial moves that consume the letter a. □

We then also have $L_{ux,y} = \{w \mid wu : L_{x,y}\}, L_{x,vy}$ and $L_{ux,vy}$ regular. The disjoint union over

$$\frac{(x, y) : Q}{\epsilon : L_{x,y}}$$

$$\frac{u : L_{x''',y} \qquad (x'', x''') : Q \qquad x' \xrightarrow{pop(a)} x'' \qquad (x, x') : Q}{ua : L_{x,y}}$$

Figure 6.6: Rules for $L_{x,y}$

the sets $L_{x,vy}$ for all x equals $L^{\geq}_{z}$, for the expression $z = vy$.

From now on we omit the brackets for trees with prefixed labels and write lx instead of $l(x)$.

Example 22. *Let the constraints be*

$$lx \geq x \qquad x \geq rz \qquad lrz \geq lly \qquad ly \geq y$$

then $L_{x,y} = L_{x,ly} = l^+$ *and* $L_{lx,y} = l^*$. *The language* $L^{\leq}_{lrz} = \{lr_z, ll_y, l_y, \epsilon_y\}$, *where the subscript* x *at label word* w *means that* $w \in L_{x,lrz}$.

Tree constraints systems without arithmetic constraints are always trivially satisfiable by setting all tree entries to zero. Analogously, all nodes that have bounds only in one direction (i.e. are only implied to be greater than a set of arithmetic variables a_i or only less) can be set to zero or infinity. The only interesting case appears when we have subtrees x whose root $\Diamond(x)$ lies between two arithmetic variables a and b. The set $L^{\geq}_{a} \cap L^{\leq}_{b} \coloneqq \{x \mid a \leq \Diamond(x) \leq b\}$ can be computed using the languages $L^{\geq}_{x}$ and $L^{\leq}_{y}$ for certain subtrees x, y. These trees are defined as the subtree starting at the point where the arithmetic variables a, b are located. For instance, if $a = \Diamond(lrrz)$, then the subtree x is $lrrz$. Thus we can write $L^{\geq}_{a} \cap L^{\leq}_{b} = \{x \mid TC \vdash y_a \leq x \leq y_b\}$, where y_a (resp. y_b) is the tree with root $\Diamond(y_a) = a$ (resp. b).

Example 23. *Consider the constraints*

$$\Diamond(x) = 1 \qquad lrx \geq x \qquad lx \geq x \qquad mx \geq x \qquad x \geq rlx \qquad x \geq mlx$$

The language of trees with root being either greater-than-or-equal, equal, or less-than-or-equal to the root of x *are obtained by iteratively applying the constraints and transitivity.*

$$L^{\geq}_{\Diamond(x)} = (lr \mid l \mid m)^*_x, \quad L^{=}_{\Diamond(x)} = L^{\leq}_{\Diamond(x)} \cap L^{\geq}_{\Diamond(x)} = m(lr)^* l_x = ml(rl)^*_x, \quad L^{\leq}_{\Diamond(x)} = (ml \mid rl)^*_x$$

6.2.3 Normal Form for Tree Constraints

We now bring the constraints into a normal form to start our procedure. Constraints in this normal form all have a variable with label word of length n on the left hand side, and all label words on the right are at most of length n. The variables with label word of length exactly n can be represented as a directed acyclic graph with an edge between x and y if

and only if $TC \vdash x \leq y$. Further, there are no arithmetic constraints below level $n-1$ (i.e. for trees with label word of length more than $n-1$).

To obtain constraints in this normal form, we examine the form of each constraint. If there is only one label word of maximal length, we take this word and isolate it on the left side of the inequality. If there is more than one such word, we write k-times repeated addition of the same summand s as $k \cdot s$. If there is then only one summand s of maximal length, we bring it on the left hand side and divide both sides by k. Otherwise, we build l constraints by bringing the l'th of the longest summands on the left (possibly again by dividing by a positive integer). Then, we apply the rules in Figure 6.2 to all the obtained constraints until all label words on the left have the same length. The result is an equivalent constraint system (i.e. a system with exactly the same solutions) consisting of unfolded tree constraints and a new, bigger set of arithmetic constraints.

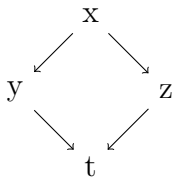
Depending on the position of this longest label word and according to the unilateral syntax, we have — after bringing the longest label word on the left side — three kinds of constraints:

- *lower bounds* are of the form $wx \geq w_1 y_1 + \cdots + w_i y_i$, with all w_i shorter than w.
- *upper bounds* have the shape $wx \leq w'y - w_1 y_1 - \cdots - w_i y_i$, with all w_i, w' shorter than w.
- *undirected constraints* have two label words of the same length, as for instance $x \geq y + z, x \leq y - z$.

The last set can be transformed into a directed acyclic graph by removing cycles as follows: If we can derive $x \geq y_1 \geq y_2 \geq \cdots \geq x + R$ by just using transitivity (not through label application, which would require again a regular language), then we conclude that $x = y_i \forall i$ and that R is identical to the tree consisting only of zeros. In this graph, we have now encoded upper and lower bounds simultaneously.

Example 24. *Let the constraints be $x \geq y + z$, $z \geq t$, $y \geq t$. They correspond to the graph with an edge from x to y and to z and from y and z to t, shown in Figure 6.7. We then add the four constraints $t \leq z$, $t \leq y$, $y \leq x - z$, $z \leq x - y$ to our system. We must traverse this graph in two directions (i.e. we need both kinds of bounds that are implied by it) to obtain an order in which we treat the nodes. The need for this will be explained in Example 25.*

Figure 6.7: Graph for Example 24



6.2.4 Idea and Examples

Let us briefly explain the intuition behind our procedure before covering the technical details. The idea is that we label all nodes in the trees which are in a set $L_a^{\geq} \cap L_b^{\leq}$ for arithmetic variables a and b with sets of intervals to which the number in the node has to belong to. These intervals are derived from the constraints. Nodes with the same set of intervals are defined to be in the same *class*. It can then be shown that there are only finitely many different classes for all constraint systems that contain the satisfiable ones.

Last, we translate the statement that all these intervals are nonempty into a finite set of linear inequalities between the arithmetic variables. This linear program is equisatisfiable to the constraints (i.e. if the intervals are nonempty, then there exists a solution with the valuation of each $\Diamond(x)$ in the interval assigned to $\Diamond(x)$). If they are satisfiable, we thus get the answer in terms of a satisfiable linear program, and, in addition to that, an assignment of a class to each of the nodes that can be seen as a certificate for satisfiability.

Combining this with Lemma 16, we have a decision procedure that either returns an unsatisfiable linear program implied by the constraints or a schematic notation for the intervals that contain their solutions from which it is directly possible to compute a solution.

Example 25. *Consider the (list-) constraints*

$$\Diamond(x) = \Diamond(y) = 1 \qquad x \geq y \qquad lx \leq x \qquad ly \geq y$$

They are equivalent to the system $\Diamond(x) = \Diamond(y) = 1$ *in conjunction with* $C = \{c_1, c_2, c_3, c_4\}$*, where*

$$c_1 \coloneqq ly \leq lx \qquad c_2 \coloneqq lx \geq ly \qquad c_3 \coloneqq lx \leq x \qquad c_4 \coloneqq ly \geq y$$

which is in normal form (with constraint c_1 *being redundant in this case). The constraint* c_3 *delivers the interval* $[0, 1]$ *for* $\Diamond(lx)$ *(resp.* c_4 *delivers* $[1, \infty]$ *for* $\Diamond(ly)$*). Thus* c_1 *gives*

us the interval $[1,1]$ *for* $\Diamond(ly)$ *and* c_2 *gives the same interval for* $\Diamond(lx)$*. By duplicating the constraint, we ensure that we treat the nodes in subsequent levels of the DAG (as constructed above) correctly. Imagine we had only* c_1 *without* c_2*, then we would miss the bound on* $\Diamond(ly)$*. In the next steps, we derive no new bounds any more.*

The system in Example 25 is satisfiable by the trees consisting of only 1s. If we modify it slightly, it becomes unsatisfiable:

Example 26. *The constraints* $x \geq y, \Diamond(y) \geq 1, lx + lx \leq x, ly \geq y + y$ *are equivalent to* $\Diamond(x) \geq \Diamond(y) \geq 1$ *and* $D = \{d_1, d_2, d_3, d_4\}$*, where*

$$d_1 \coloneqq ly \leq lx \qquad d_2 \coloneqq lx \geq ly \qquad d_3 \coloneqq lx \leq 0.5x \qquad d_4 \coloneqq ly \geq 2y$$

We have the intervals $[0, 0.5\Diamond(x)]$ *and* $[2\Diamond(y), \infty]$ *for* $\Diamond(lx)$ *by* d_3 *and* d_2*, and* $[2\Diamond(y), \infty]$ *and* $[0, 0.5\Diamond(x)]$ *for* $\Diamond(ly)$ *by* d_4 *and* d_1*. In the next steps, the factors will be 0.25 and 4, etc. The list* x *is exponentially decreasing, whereas* y *grows exponentially. So if the roots of* x *and* y *are neither zero nor infinity, then no matter which number they are,* x *will at some point be less than* y*. Here we see that there is a contradiction, but we can not say after how many iterations we will find it. There the other part of our algorithm, namely Lemma 16, applies.*

6.2.5 Satisfiability is Decidable

Before we prove the central facts (Theorem 6 and Theorem 7) of this chapter, we need some word-combinatorial preliminaries. We say that two label words are *dependent* if one is a suffix of the other. The next two lemmas are well known and can be found for instance in [62].

Lemma 17. *Let* $u, s, t \in \Sigma^+$ *such that* $tu = st$*. There then exist* $q, r \in \Sigma^*$ *and* $i \in \mathbb{N}$ *such that* $s = qr$*,* $u = rq$*,* $t = q(rq)^i$*.*

Lemma 18. *If for word* x, y, z *holds* $x^n y^m = z^k$*, with* $n, m, k \geq 2$*, then exists* t *such that* $x, y, z \in t^*$*.*

Lemma 19. *Let* c *be a unilateral tree constraint. If it is of the form*

$$a_1 a_2 \dots a_n x \geq c_1 \cdot a_2 \dots a_n x + \dots + c_n \cdot a_n x + c_{n+1} \cdot x \tag{6.6}$$

with all $a_i \neq \epsilon, c_j \in \mathbb{N}_0$ *and label words and* x *a tree, then it can be transformed (by*

	$a_1a_2\dots a_n$	$a_3\dots a_n$
a_1a_2	$a_3\dots a_na_1a_2$	$a_3\dots a_n$

Figure 6.8: Commuting Words a_1a_2 and $a_3\dots a_n$

application of labels from the left) into a constraint with all $a_k \in p^+$ for a suitable word p and all other summands independent.

Proof. We assume that there is a label word $t \in \Sigma^*$ which we can apply from the left such that all summands stay dependent of $a_1a_2\dots a_nx$. (If such a t does not exist, then all summands are already independent.) This is,

$$ta_1a_2\dots a_n \text{ has the suffixes } ta_2\dots a_n, ta_3\dots a_n, \dots ta_n, t. \tag{6.7}$$

We apply Lemma 17 to $ta_1a_2\dots a_n = st$ with $u = a_1\dots a_n$ and obtain $r, q \in \Sigma^*$ such that $a_1\dots a_n = rq, t = q(rq)^i = q(a_1\dots a_n)^i$. Thus $ta_2\dots a_n = q(a_1\dots a_n)^ia_2\dots a_n$. According to (6.7), $ta_2\dots a_n$ is a suffix of $ta_1a_2\dots a_n = q(a_1\dots a_n)^{i+1}$. This means that a_1 and $a_2\dots a_n$ commute. Hence there is p_1 such that both are in p_1^+.

Similarly, $ta_1a_2\dots a_n$ has the suffix $ta_3\dots a_n = q(a_1\dots a_n)^ia_3\dots a_n$, and thus a_1a_2 and $a_3\dots a_n$ commute (see Figure 6.8, where words of the same length are written in boxes). We can thus conclude that there is p_2 with a_1a_2 and $a_3\dots a_n$ are in p_2^+. We proceed the same way until we obtain in the last step that $a_1\dots a_{n-1}$ and a_n commute. We now write $a_1 = p_1^{i_1}, a_2\dots a_n = p_1^{j_1}$ and $a_1a_2 = p_2^{i_1}, a_3\dots a_n = p_2^{j_2}$, etc. Application of Lemma 18 allows us to conclude from $(a_1\dots a_n)^2 = p_1^{i_1+j_1}p_2^{i_2+j_2} = p_3^{2(i_3+j_3)}$ that $p_1, p_2, p_3 \in p^*$ for a certain p. Thus all p_i are in p^+ and for all i, we have $a_i \in p^+$. □

Theorem 6. *Satisfiability of linear tree constraints is semi-decidable.*

Proof. Assume that all constraints are in the normal form described above. Introduce an arithmetic variable for each node above level n. Recall that there are no arithmetic constraints below level $n-1$ and all left hand sides of the constraints have label word of length exactly n. Now calculate the sets $L_a^{\geq} \cap L_b^{\leq}$ for all pairs of arithmetic variables a, b. One may assume w.l.o.g. that all a, b are neither zero nor infinity: Otherwise, let A, B be disjoint subsets of the set of arithmetic variables and test all combinations of additional constraints $A \ni a_i = 0$ and $B \ni a_i = \infty$. If one of them is satisfiable, we return this as the result.

Our procedure starts with step 1 at level n and assigns a set of intervals to each node. For the lower bounds, which have the form $wx \geq w_1y_1 + \cdots + w_my_m$, with all w_i shorter than w, we add the interval $[\sum_i \Diamond(w_iy_i), \infty]$. For the upper bounds, that have the shape $wx \leq w'y - w_1y_1 - \cdots - w_my_m$, with all w_i and w' shorter than w, we add the interval $[0, \Diamond(w'y) - \sum_i \Diamond(w_iy_i)]$.

For the undirected constraints, we observe that the membership of all nodes in $L_a^{\geq} \cap L_b^{\leq}$ ensures that we have already an interval for the starting nodes of the DAG constructed above. We traverse it in both directions and add for constraints $x \geq y_1 + \cdots + y_m$ the new set of intervals $\big\{[\sum_i a_i, \infty] \;\big|\; [a_i, b_i] \text{ is an interval for } \Diamond(y_i)\big\}$; respectively for $x \leq y - z_1 - \cdots - z_m$ the new intervals $\big\{[0, b - \sum_i c_i] \;\big|\; [a, b] \text{ is an interval for } \Diamond(y) \text{ and } [c_i, d_i] \text{ is an interval for } \Diamond(z_i)\big\}$.

Further, we set all nodes that have bounds in only one direction to $[0, 0]$ or $[\infty, \infty]$. We denote the set of intervals for node $\Diamond(x)$ with $I(x)$, and I_n is the set of all $I(x)$ obtained until step n.

The unilateral constraint syntax allows us to define a meaningful addition and subtraction on interval sets that formalises how we compute new interval sets.

$$I(x) + I(y) = \big\{[a + c, \infty] \mid [a, b] \in I(x), [c, d] \in I(y)\big\}$$
$$I(x) - I(y) = \big\{\,[0,\ b - c] \mid [a, b] \in I(x), [c, d] \in I(y)\big\}$$

Observation 1. *The order of evaluation does not play any role for sums of interval sets, i.e.* $I(x) - I(y) - I(z) = I(x) - \big(I(y) + I(z)\big)$.

To prove this, let w.l.o.g. be $I(x) = [a, b]$, $I(y) = [c, d]$ and $I(z) = [e, f]$. Then

$$\begin{aligned}[a, b] - [c, d] - [e, f] &= [0, b - c - e] = [0, b - (c + e)] \\ &= [a, b] - [c + e, \infty] = [a, b] - ([c, d] + [e, f]).\end{aligned}$$

In step $n + 1$, we apply the rule (LabelSum) from Figure 6.2 to the constraints to increase the length of the label word of the left sides by 1. Then, for the lower bounds we no longer necessarily have arithmetic variables as roots of the trees on the right, but also nodes equipped with intervals. Thus, we proceed in a similar way as for the undirected constraints on level n, namely add the intervals that can be derived from the variables on the right. We do the same for the upper bounds and the undirected constraints on level $n + 1$. More

precisely, for the lower bounds, we set $I(x) = I(y_1) + \cdots + I(y_m)$, and for the upper bounds $I(x) = I(y) - I(z_1) \cdots - I(z_m)$.

We claim that after a finite amount of steps, no new intervals are derived any more: If we see the set of intervals that belong to a node as its *class*, then there are only finitely many different classes. The reason is that if the intersection of one of the interval sets would be constantly shrinking, we would infinitely often add a nonzero number to the lower bound or subtract a nonzero number from the upper bound or divide the upper bound by a positive integer (by the assumption that all arithmetic variables are neither zero nor infinity). Since all considered nodes are bounded from above and below, we would at some point obtain a contradiction (see Example 26).

It is thus necessary to give a criterion that ensures that we have found all classes already and must not search for new classes anymore. Having found all intervals, the condition that the intersection of all intervals that belong to the same node is nonempty delivers an equisatisfiable linear program.

We now define S as the least common multiple of all differences of label word lengths that appear in the constraint system. For instance, S for the single constraint $lrrlx \geq x + lx$ is $12 = \mathrm{lcm}(4, 3)$. Note that in the list case, S is a bound on the period length of the solution lists (cf. [9]). The criterion looks as follows: Let L be the set defined by

$$L = \cup_{a,b}(L_a^{\geq} \cap L_b^{\leq}),$$

which describes the nodes with bounds in two directions. If in S iterations no new interval sets for the nodes in L are derived any more (i.e. for each node x on a certain level and word p with $|p| = S$, the intersection of all intervals that belong to px is equal to the intersection of the intervals for x), then we have found all of them.

There are two things to show, namely that the premise of this criterion implies $I = I_n$ for a $n \in \mathbb{N}$ and that this premise will finally hold.

Claim 2 (Part 1)**.** *If there is a $n \in \mathbb{N}$ such that for all $x \in L$ on level $n, \ldots, n + S$ and for all label words p with $|p| = S$, the set $I(px)$ is equal to $I(x)$, then $I = \cup_{j \in \mathbb{N}} I_j = I_{n+S}$.*

Claim 3 (Part 2)**.** *There is a $n \in \mathbb{N}$ such that for all $x \in L$ on level $n, \ldots, n + S$ and for all label words p with $|p| = S$, the set $I(px)$ is equal to $I(x)$.*

To prove the first, we show that for all $k = 0, \ldots, S$ and for all $l \in \mathbb{N}$, we have $I_{n+k+l\cdot S} = I_{n+k}$. We consider three cases:

1. If we have a lower bound constraint $px \geq \sum_i y_i$, we know that for all label words p, q with $px, qpx \in L$ and $|p|, |q| = S$, this implies $qpx \geq \sum_i qy_i$. The lower bounds of the intervals for qy_i are not stronger than those for y_i. This follows from the assumption if $qy_i \in L$. It is also true if $qy_i \notin L$, because then the interval for qy_i must be $[0, 0]$ (since it is less or equal to qpx, which is at most b and so it can not be $[\infty, \infty]$) and thus it delivers no new bounds at all. We mark this property by $(\star)$.

2. Similarly, in case of upper bounds $px \leq y - \sum_i z_i$, the upper bounds for qy and the lower bounds of the intervals for qz_i are not stronger than those for y and z_i. Again, if $qy, qz_i \in L$, this is a consequence of the assumption, and if $qy \in L$ and $qz_i \notin L$, then qz_i has interval $[0, 0]$. Last, if $qy \notin L$ then qy has interval $[\infty, \infty]$ and delivers no new bounds. This property is called $(\star\star)$.

In these two cases, the right hand side in $(\star)$ and $(\star\star)$ is on level less or equal to the level on the left. We may assume that all $py_i, qy, qz_i \in L$. For all label words q, p with $qpx \in L$ and $|q|, |p| = S$, we have $I(qy_i) = I(y_i)$, $I(qy) = I(y)$, $I(qz_i) = I(z_i)$ and the set of intervals for qpx, which is the sum of the intervals for the qy (resp. the difference between the intervals of qy and the qz_i) is (after intersection) not smaller than the set of intervals for qpx (according to $(\star)$ and $(\star\star)$), and also not smaller than the interval for x. More precisely, we have

$$I(qpx) = I(qy_i) + \cdots + I(qy_m) = I(y_i) + \cdots + I(y_m) = I(px) = I(x), \text{ or resp.}$$
$$I(qpx) = I(qy) - I(qz_i) - \cdots - I(qz_m) = I(y) - I(z_1) - \cdots - I(z_m) = I(px) = I(x).$$

This means that if we jump S levels down in the tree and nothing changes regarding the intervals, then the next iteration will never again obtain a new interval.

3. The last case occurs if we have an undirected constraint $x \geq y + z$. This implies $px \geq py + pz$, with $I(px) = I(x)$, $I(py) = I(y)$, $I(pz) = I(z)$, and for the new interval obtained from the undirected constraint $I_{new}(px) = I(py) + I(pz) = I(y) + I(z) = I(x)$, etc. If there are no changes in the intervals for p, q of length S, then there are no changes at all and $I = I_{n+S}$.

To prove the second part of the claim, we assume that for all levels there is a p of length S and x on that level (optionally plus a number between 1 and S) such that $I(px) \neq I(x)$. In

other words, we derive at least one new interval on each Sth level.

All constraints on the variables in L (except a subset of the undirected constraints where all label words have the same length) imply constraints of the form $qy \geq z + R$ (resp. $qy \leq z - R$) with $|q| = S$. For the lower bounds, we have to consider all possibilities for the choice of z in the following, whereas for the upper bounds there is only one positive summand and thus z is unique.

We can assume that for all x holds $I(px) \subset I(x)$ because of the choice of S: if this was not the case and pqy had strictly weaker lower bounds (resp. strictly weaker upper bounds) than pz, the interval $I(pqy)$ would contain more than $I(pz)$ and thus qy could not have z as a bound.

Thus we have $I(pqy) = I(pz) + I(R)$ (resp. $I(pqy) = I(pz) - I(R)$). We now assume w.l.o.g. that y plays the role of the x above and that $z = y$ holds[2]. So we have a constraint $px \geq x + R$ or $px \leq x - R$ which is derivable just by unfolding using the (LabelSum) rule in Figure 6.2. We just treat the first since both are similar.

According to Lemma 19, either some label words of the summands in $R = r_1 + \cdots + r_m$ and p are powers of the same path t, or in the next step all summands in $qR \coloneqq qr_1 + \cdots + qr_m$ are independent of qpx for all q of length S. If the second happens, if no tree $t^i x$ is reachable[3] from R, then either this constraint delivers no new bounds below level $n + S$, or R must contain a tree z that is at least constant when seen as a list $(p'^i z)_i$ along a path $p' \in t^*$ of length S. This implies $p = p'$, which again implies $ppx = t^j px = t^{2j} x \geq t^j x + t^{kj} z + R'$, and that means $p^l x \geq p^{l-1} x + p^{l+k-2} z$ holds — just like in the first case. Overall, we have that either the constraint does not deliver an infinite amount of new bounds or has the form of a strictly increasing list along the path $(\Diamond(p^i))_i$. In both cases, this is a contradiction, since only finitely many $p^i x$ can be in L, thus $I(p^i x)$ is at some point equal to $[\infty, \infty]$ and then stops changing.

Claim 1 and Claim 2 ensure that in a set of cases including the satisfiable ones, we will only derive finitely many different intervals. Thus the problem to decide whether the values of the arithmetic variables can be chosen such that these intervals are all nonempty can be solved by linear programming. If and only if they can be chosen this way, the constraints

[2] If this is not the case for the initial y, the next candidate for x is z.

[3] No lower bound constraint on any summand $t'x$ in R with $t' \in t^*$ on the right exists — which is decidable according to Theorem 5.

are satisfiable. □

Example 27. *Consider the constraint system*

$$llx \geq rx + mx \qquad\qquad \Diamond(rx) = 1$$

There are no arithmetic constraints below level 1*, and the system is already in normal form. All summands on the right hand side of the tree constraint are independent of the variable on the left hand side. We derive the interval* $[1, \infty]$ *for* llx *and continue with unfolding the constraint to obtain*

$$rllx \geq rrx + rmx \qquad lllx \geq lrx + lmx \qquad mllx \geq mrx + mmx$$

Because of the independency, we obtain no new intervals at all, since none of the trees on the right sides are in L*.*

Example 28. *Let the constraints be*

$$\Diamond(y) = 1 \qquad y \geq ly \qquad y \geq ry \qquad lx \geq x + y \qquad rx \geq x + y \qquad x \geq lrx$$

Then all nodes in $(l|r)^+y$ *are assigned the intervals* $[0, 0]$*. Similarly,* l^+x *and* r^+x *and all other nodes are set to* $[\infty, \infty]$*, except those nodes that have bounds in two directions (i.e. are in* L*). The only nodes in* L *are the roots of* lr^*x *and* $\Diamond(y)$*. Thus we only need to compute intervals for* lr^*x*. The root* $\Diamond(x)$ *gets the interval* $[1, 1]$*. On level two, there is no node in* L*. Then, on level three,* $\Diamond(lrx)$ *is labeled with the intervals* $[1, \infty]$ *and* $[0, 1]$*. Thus their intersection is equal to* $[1, 1]$*. The same happens on level* $4, 6, 8,$ *etc. Indeed, we can easily check that another solution than one with* $\forall w \in (lr)^*.\Diamond(wx) = 1$ *is not possible.*

Combined with the semi-decidable unsatisfiability, we can decide UTC.

Theorem 7. *The unilateral tree constraint problem is decidable.*

Proof. For a given constraint system, we run the two semi-decision procedures for satisfiability and unsatisfiability in parallel. As we proved, one of them will terminate and this yields our result. □

In the next example, we sketch how we can solve list constraints with this procedure.

Example 29. *Consider the constraints*

$$llllx \geq llx + 2lllx, llllx \geq lllllllx,$$

as they appeared (with different notation) in Example 13 in Chapter 4. As in Example 10, we can conclude that we have a periodic list of period starting at position 4 and of length at most 6 (and effectively 1 by Remark 7), but this time by computing the languages

$$L_{llx} = L_{lllx} = l^*ll.$$

Thus we have the following constraints in normal form

$$l^7x \leq l^4x, \text{ (upper bound)}$$
$$l^7x \geq l^5x + l^6x \text{ (lower bound)}$$

and the interval condition to be checked for all list elements except the first two:

$$[c + 2c, \infty] \cap [0, c] \neq \emptyset$$

with $a = \Diamond(x), b = \Diamond(lx), c = \Diamond(llx)$ arithmetic variables. The result is the same as in Example 13.

6.3 Complexity

If the constraints are unsatisfiable, we cannot determine the complexity at all, because this contradiction may be located arbitrarily deep in the tree, depending on the initial values. For instance, if we have tree constraints

$$\begin{aligned} lrx &\geq x, \\ y &\geq 2lry, \\ y &\geq x, \\ \Diamond(y) &\geq 1024 = 2^{10}, \\ \Diamond(x) &= 1, \end{aligned}$$

then we will need 10 steps to find a contradiction, if $\Diamond(y) = 1024$, but if it forced by an arithmetic constraint to be equal to 2^{100}, we will need 100 steps. Since we cannot assume that all trees are regular as in the list case, we cannot simply reduce to a finite linear program.

The complexity for the algorithm in the satisfiable case is at least exponential according to the number of linear programs used when assuming that all initial values are nonzero.

6.4 Optimization

Lemma 20. *Let x be a variable for an infinite (sub)tree and let T be a set of unilateral tree constraints on x. A statement of the form*

$$p^i x \geq p^{i-j} x + p^{i-k} x$$

with $\mathbb{N} \ni j, k < i \in \mathbb{N}$ or $\mathbb{N} \ni j, k > i \in \mathbb{N}$ and p a path as above, can be derived from the constraints, if and only if there are only solutions of exponential growth. In this case the list which has as entry number i the root label of the tree $p^i(x)$ is exponentially increasing.

Proof. We only treat the case of a lower bound. For an exponential lower bound we must have arbitrarily many equations with two lists on the right hand side that both depend on the left hand side. If there is at most one dependent list on the right hand side, we can apply a similar argument as in the previous section for the estimation of growth rates and obtain at most polynomial growth. In order to apply Lemma 17, we write the constraint with the x-dependent right hand side as $uvx \geq vx + x$, where $u \neq \epsilon$. We then apply the Lemma such that t is a suffix of tuv, and then such that tv is a suffix of tuv. Note that here t stands for a word of labels applied from the left. First, we get $uv = rq$ and $t = q(rq)^i$. Second, $t = q(uv)^i$ (w.l.o.g t is long enough that $i \geq 2$). We have $tv = q(uv)^i v$ and $tuv = q(uv)^{i+1}$. Thus, if tv is a suffix of tuv, then uvv must be a depending on vuv. But then, since $|uvv| = |vuv|$, we have $uv = vu$ and $u = p^k$ and $v = p^l$ for a word p by [63]. This proof is a special case of lemma 19. □

By a similar argument as in the list case, one can show the following.

Lemma 21. *In the case of lower and upper bounds along one path, the entries, considered as a list, become periodic.*

Remark 12. *If a unilateral tree constraint system is satisfiable, then it admits a solution where all paths become periodic over $\mathbb{D}$. This statement does not imply decidability itself, because there are infinitely many different paths.*

Proof. If there is a strictly increasing bound, a constraint of the form (see Theorem 20)

$$p^i qx \geq p^j qx + p^k qx$$

or two constraints

$$p^i qx \geq p^j qx + p^k q'y, \text{ and } p^k q'y \geq p^{k'} q'y, k' < k,$$

where either $j, k > i$ or $j, k < i$, can be derived from the given set of constraints. Thus we have an upper (resp. lower) bound. If the constraint is satisfiable, we can not have a bound in the other direction, since this would imply a periodic path. Thus we can set all entries along the path to zero or infinity. □

Sometimes we can bring constraint systems into this form by unfolding. While this can not be used to decide satisfiability, we can still use it for the estimation of growth rates.

Example 30. *Consider the constraint $lrx \geq 2rx + x$. It implies*

$$llr \geq 2lr + l, \tag{6.8}$$

$$rlr \geq 2rr + r. \tag{6.9}$$

After renaming of subtrees ($r = y, l = z$), we obtain $lly \geq 2ly + z$ and $rly \geq 2ry + y$, so that the second summand in inequality (6.8) is independent from lly and the first in inequality (6.9) is independent from rly. We end up with a set of arithmetic constraints (namely

$$\{\Diamond(lr) \geq \Diamond(r) + \Diamond(x),$$
$$\Diamond(rlr) \geq \Diamond(rr) + \Diamond(r),$$
$$\Diamond(llr) \geq 2\Diamond(lr) + \Diamond(l)\})$$

and the tree constraints $ly \geq 2y$ and $rly \geq y$, which cannot be simplified further.

Example 31. *We consider the constraint system*

$$l^4x \geq l^3x + l^3y,$$
$$l^3y \geq l^2y + l^2z,$$
$$l^2z \geq lz.$$

Then $\Diamond(l^{(3+k)}y) \geq \sum_{i=0}^{k} \Diamond(l^{(2+i)}z) + \Diamond(l^{(2+k)}y)$ *and* $\Diamond(l^{(4+k)}x) \geq \sum_{i=0}^{k} \Diamond(l^{(3+i)}y) + \Diamond(l^{(3+k)}x)$. *Since* $(l^iz)_i$ *is at least the constant list with all entries equal to one,* $(l_iy)_i$ *is growing at least linearly, and* $(l_ix)_i$ *is quadratic, since* $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$.

Remark 13. *The label* l *in this example could be replaced by an arbitrary path* $p \in \Sigma^*$ *and we then would have polynomial growth along this path.*

We summarize the discussion about growth rates in the tree case:

Theorem 8. *If the minimal solution tree* t *to a constraint system* (TC, AC) *has a growth rate* $f(n) = \Theta(g(\frac{1}{q}n))$*, where* n *is the level and* $f(n)$ *the maximal entry on level* n*, then there is an infinite path* $(p_i)_{i\geq 0}$ *of length* q *in* t*, which has (viewed as list of* $(\Diamond(p_ix))_{i\geq 0}$*) the growth rate* $\Theta(g(n))$*. Further, we can find this growth rate using the constraints that belong to the set* X_∞*.*

Proof. We assume w.l.o.g. that each path has length 1 as in the list case. This can be achieved by introducing a new tree as follows. Let m be the least common multiple of all label word lengths. Clearly, the length of all appearing paths divides m. We then introduce one child for each path. Note that the information needed for the languages in the decision procedure is lost here, in particular in the new tree the levels are not preserved. But for the growth rates, it is enough to keep track of the respective paths length k and multiply the argument of the growth rate function by k.

For instance, if we have constraints $llx \geq x + x, lrx \geq x, rrrx \geq rx + rrx$, we introduce a new tree with labels a, b, c and such that $ax = x + x, bx = x, cccx \geq cx + ccx$ and a has additional information in terms of a number, here 2, b has 2 and c has 1. Then we know that the growth rate along path a is 2^n, thus in the whole tree $2^{\frac{1}{2}n}$, for b it is constant, and for c it is $(\frac{\sqrt{5}+1}{2})^n$. In this case, path a has the maximal growth rate. Further, we cannot get greater growth rates by combining paths in this tree.

We now examine all lower bound constraints. For this, we argue by induction over the level n (with constants for $n = 0$) in this new tree and use so-called "back-references". A

back-reference is a jump back along the same path, for instance $\mathbf{tail^4}(\mathbf{x}) \geq \mathbf{tail}(\mathbf{x})$ is one from $\mathbf{tail^4}(\mathbf{x})$ to $\mathbf{tail}(\mathbf{x})$. It corresponds to the path **tail** and a 4×4 matrix for the growth rates (cf. section 5).

We consider tree constraint systems after application of the elimination procedure, similarly as for lists in Chapter 4 (i.e. all variables have a constraint that contains one or more back-references). Equivalently, we could check in each step whether the summands on the right lead to new back-references for the variable on the left[4].

We proceed the following way for all constraints separately.

For a lower bound, the level either decreases when following the back-references in a constraint, or we have no back-references but only inhomogeneities in form of other lists, like $x \geq y + z$. If we have no back-references, then the growth rate is equal to the maximal growth rate of the summand on the right (possibly times a constant if the summand appears several times) and we can continue with the induction hypothesis, if the level decreases. Else, we continue with all summands on the right. If we have lower bounds, the level must decrease at some point, because else we would only have cycles and could set all entries equal and thus constant. Thus we can again apply our induction hypothesis.

Otherwise, if we have one or more back-references along a path p, we consider the additional summands with different variables (if there are any). If there are *no* such other summands *with the same path*, we can directly set up the matrix system as in the list setting for our variable on the left, apply the estimation of growth rates for lists and scale the argument by the inverse of the paths length. We take the maximum of this growth rate and the growth rates of the remaining independent summands (again if there are any). This is one candidate for the maximal growth rate of the minimal solution.

If there exist other summands than x on the right that lie on the same path p, we proceed the same way with these summands. One has to notice here that we can assume there are no cycles by setting all variables that appear in cycles to be equal. Then we take the maximum of the obtained growth rates of these summands and continue as follows. By the induction hypothesis, there is one path q containing the maximal growing entries of the tree until level $n - 1$. If there is only one back-reference and $p \neq q$, we can calculate the growth rates as in Chapter 5 (cf. Example 31) for the path p. Otherwise, also in case of only one

[4]Since we have no upper bounds, this is computable without needing a finite automaton as in Section 6.2.

back-reference, and if $p = q$, we multiply this growth rate along q by n. Else, in case of several back-references, we have exponential growth, which we can estimate precisely using Rothblums results [45, 59]. Lastly, we take the maximum of all these growth rates.

□

We have seen that the growth rates of the list case directly apply also to the tree case, with the only difference, that we have to sum them over the exponential number of all tree entries, whereas for lists, in order to obtain a potential function for a list, one has to sum only linearly many elements (with the known growth). So the potential of a constant binary tree labeled with number a is $(2^n - 1) \cdot a$, but this is still linear in size of the input, namely a binary tree and thus subsumed by Rodriguez' algorithm. As soon as there is one at least linearly growing path (or an irregular structure) in all possible solution trees, the algorithm for linear bounds will no longer be sufficient. Then, we first check satisfiability and if the answer is "satisfiable", we estimate the growth using the result of this chapter.

Chapter 7

List and Tree Constraint Solving for Concrete Examples

We plan to integrate optimized versions of our algorithms into the existing RAJA tool. By now, they are not yet in a state where they can be implemented efficiently. This topic is subject to future investigations.

However, we have already implemented the list constraint solver independently from RAJA and included a set of interesting example constraints to be solved by it in a separate file. This can be used for testing the algorithm. We tried to keep the data as simple as possible such that the constraints can still be solved by hand, but they are intended to be representative in the sense that they cover all nontrivial cases of the agorithm.

For now, we focus on programs which deliver constraints that can be solved by hand with our new methods. We solved the constraints (generated by the existing RAJA tool) of the following examples. The first program constructs a degenerate tree from a list by writing the n'th list element on all nodes on level n in the tree. The class `List` is the superclass of classes `Nil` and `Cons`. This also means that the constraints that hold for `List` must also hold for the subclasses (i.e. when seen as `Cons`, an object must have greater or equal potential than when seen as a `List` object).

We first define the class `List` with methods that will be specified in the subclasses. The append method for lists is taken from [1].

```
class List{
  List listPlus(List l){return null;}
  List copy(){return null;}
  void appAux(List y, Cons dest){return null;}
  void printL(){return null;}
  List append(List y){
    let dest = new Cons in
    let _ = this.appAux(y, dest) in
    let result = dest.next in
    let _ = free(dest) in
    return result;
  }
  Tree toTree(){return null;}
}
```

Next, we redefine the methods in the class `Nil`.

```
class Nil extends List{
  void appAux(List y, Cons dest) {
    let _ = dest.next <- y in
    return null;
  }
  List copy() {
    return this;
  }
  void printL()  {
    return print("");
  }
  Tree toTree(){
    return new Empty;
  }
  Nil listPlus(List l){let m = new Nil in
    return m;
  }
}
```

The class `Cons` is defined as follows.

```
class Cons extends List{
  List next;
  int elem;

  void appAux(List y, Cons dest){
    let _ = dest.next <- this in
    return this.next.appAux(y, this);
    }
  void printL()  {
    let _ = print(this.elem) in
    let _ = print(", ") in
    return this.next.printL();
  }
  List copy() {
    let ress = new Cons in
    let next =  this.next.copy() in
    let _ = ress.next <- next in
    let _ = ress.elem <- this.elem in
    return ress;
  }
  Nil listPlus(Nil n){
    let  m = new Nil in
    return m;
  }
  Cons listPlus(Cons l){
    let l' = new Cons in
    let _  = l'.elem <- l.elem+ this.elem in
    let _ = l'.next <- this.next.listPlus(l.next) in
    return l';
  }
  Tree toTree(){
    let t = new NonemptyTree in
    let _ = t.elem <- this.elem in
    let _ = t.right <- this.next.toTree() in
    let _ = t.left <- this.next.toTree() in
```

```
    return t;
  }
}
```

We also define trees as `Tree` being a superclass of `NonemptyTree` and `Empty`.

```
class Tree{
  List toList(){return null;}
}
class Empty extends Tree{
  List toList(){let m= new Nil in
  return m;
  }
}

class NonemptyTree extends Tree{
  int elem;
  Tree left;
  Tree right;
  List toList(){
    let l= new Nil in
    let _= l.append((List) this.elem) in
    let _= l.append(this.left.toList().append(this.left.toList()))
    in return l;
  }
}
```

In order to run the analysis, we need a `Main` program.

```
class Main{
  List main(List l){
let t = l.toTree().toList() in return t;}
}
```

The most interesting constraint that appears there after running the RAJA tool and forcing

it to display the constraints as an intermediate step, is (in the syntax from [1]),

$$\begin{aligned}&\text{NonemptyTree.left.g}(\text{NonemptyTreetoListself}^{\text{NonemptyTree})}) \sqsubseteq \\ &\text{NonemptyTreetoListself}^{\text{NonemptyTree}} + \text{NonemptyTreetoListself}^{\text{NonemptyTree}}\end{aligned}$$

There the g stands for the get-view of the tree `this` in method `toList` of class `NonemptyTree`, which has to be a subview of twice `this` itself ("self"). The tool cannot solve this constraint and return that the program cannot be analyzed. After translating the constraint into our normal form by dividing the degree by two and eliminating variables it becomes

$$lx \geq x + x,$$

which means the left of the tree grows exponentially. All other constraints have constant solutions, and we can create a modified linear version of this program with another summand. If we remove the sharing of potential in the left subtrees, which is used twice in method `toList` of class `NonemptyTree`, the solution for the tree constraints becomes regular, but the list it is transformed to stays irregular.

We also analyzed a program that creates a degenerate tree out of a tree by replacing the left subtree by the sum of all its elements.

```
class Tree{
  Tree TreePlus(Tree t){return null;}
  int sums(){return null;}
  Tree toDegTree(){return null;}
}
class Empty extends Tree{
  Nil TreePlus(Tree t){
    let m= new Nil in
    return m;
  }
  int sums(){return 0;}
}
class NonemptyTree extends Tree{
  int elem;
  Tree left;
  Tree right;
```

```
  Tree TreePlus(NonemptyTree t){
    let t' = new NonemptyTree in
    let _ = t'.elem <- this.elem + t.elem in
    let _ = t'.left <- this.left.TreePlus(t.left) in
    let _ = t'.right <- this.right.TreePlus(t.right) in
    return t';
  }
  int sums(){
    let l = new List in
    return this.elem + this.left.sums()+this.right.sums();
  }
  Tree toDegTree(){
    let _ = this.left <- (NonemptyTree) this.right.sums() in
    let _ = this.right <- this.right.toDegTree()
    in return this;
  }
}
class Main{
  Tree main(NonemptyTree t'){
  return t'.toDegTree();}
}
```

This program contains (after rewriting and eliminating most of the initially 1398 constraints) the condition

$$rx \geq y + x + z,$$

where the inhomogeneities belong to constant lists and thus cause linear growth for the tree x, when going right.

As a last example, we have RAJA program that generates an exponential tree.

```
class Main{
  List main(NonemptyTree t'){
    let t = new NonemptyTree in
    let t'' = new NonemptyTree in
    let _ = t''.elem <- 2 in
    let _ = t.left <- t' in
```

```
    let _ = t.right <- t'' in
    let _ = t.elem <-4 in
    return t.toList();}
}
```

The constraints include

$$lx \geq x + x,$$
$$rx \geq x + x,$$

and are satisfiable.

As mentioned above, we have not yet integrated the list constraint solving procedure into the RAJA tool from previous work. For illustration how the procedure works on small example constraints, the code can be found on my homepage:

https://www.tcs.ifi.lmu.de/mitarbeiter/sabine-bauer.

Chapter 8

Summary and Future Work

We have studied a constraint satisfaction problem arising from the amortized resource analysis for object-oriented programs but which due to its simple and natural formulation may be worth exploring in its own right.

We show that this problem is computationally hard in general but identified a natural fragment which still covers the entire range of the resource analysis problems. Thus, the computational hardness can be seen as an artifact of an unnecessarily general formulation of the problem, which is still of general interest, however. For list programs, we first demonstrate feasibility in the sense that both satisfiability and estimation of growth rates can be decided in polynomial time. This special format encompasses systems whose minimal solutions exhibit polynomial or exponential growth, a case that was not within the scope of the heuristic method proposed earlier [1]. Our method uses a novel combination of ideas from constraint satisfaction, recurrence theory, linear programming, and matrix theory.

Our results can be used to extend the existing amortized analysis, in particular such that type inference becomes decidable for all kinds of list programs in RAJA and such that the tool can derive resource bounds from the minimal solutions.

The natural next step was the generalization of our method to decide satisfiability of more general constraint systems involving $\mathbb{D}$-labeled trees; investigations based on results of [62, 63] suggest that this problem cannot be reduced to the case of lists, since it involves complicated word combinatorics because of the different branchings.

However, we generalize our method to trees of arbitrary degree. We have proven that linear

constraints over infinite trees, again as generated by an automatic resource type inference for the language RAJA, are decidable. We have described an effective procedure for deciding satisfiability of constraint systems in the special format. The decision procedure uses finite automata to generalize the list constraint theory to trees. In contrast to the list case, our algorithm for trees needs exponential time, because the number of the linear programs that we reduce the problem to is exponential in the size of the input.

In the abstract, we mentioned that it is also an interesting question how to finitely represent concrete solution trees. We briefly explain this now. The decision procedure in Chapter 6 delivers a set of intervals for each node that contain symbolic variable expressions as bounds for the number in the node. The condition that the intervals are nonempty delivers an equisatisfiable linear program. If we have found a satisfying assignment for this linear program, we obtain concrete interval sets in which the node numbers must lie. Then we can not choose all numbers independently to obtain a solution to the tree constraints. For instance, if the root of x is in the interval $[1, 2]$ and the root of the $r(x)$ is in $[1.2, 1.7]$, and we have a constraint $rx \geq x$, then we cannot set x's root to be 2. But we can leave the interval sets symbolic and introduce new variables for the node numbers that have to lie between the two bounds and translate all constraints on these nodes by unfolding. In the example, we have variables a, b, c, d for the interval bounds and the additional variables e, f for the actual numbers in the nodes, and intervals $[a, b] \ni e, [c, d] \ni f$, together with the inequalities $a \leq e \leq b, c \leq f \leq d, f \geq e$. Since we have only a finite number of different interval sets, we get again a (possibly much bigger but) finite linear program. A solution of this extended program then gives a concrete solution of the constraints.

With our results, we have set the theoretical basis to now analyze arbitrary object oriented (RAJA-) programs with respect to the satisfiability question for their constraints and their resource consumption. We can read off upper bounds on the memory usage from the solutions of the constraints.

The next parts of our planned future work include a prototype implementation based on the existing RAJA tool. We also will further investigate possibilities to increase the efficiency by optimizing the decision procedure and then we will determine the exact complexity. Another interesting direction would be to find out which further advantages than the freedom of choice lie in the fact that the solutions are unrestricted regarding their general shape.

Once we have integrated the algorithm in the RAJA tool, we will have a prototype for a

fully automatic amortized analysis for Java-like programs. We will then try to prove bounds from the functional systems with RAJA and compare the results and examine if we can improve the functional systems using techniques from the object oriented setting. More precisely, we will try to adapt the concept of views to the functional world and investigate the benefits.

Bibliography

[1] Rodriguez, Dulma: *Amortized Analysis for Object Oriented Programs*, University of Munich, Diss., 2012

[2] Hofmann, Martin ; Rodriguez, Dulma: Efficient Typechecking for Amortised Heap-Space Analysis. In: *Proceedings of the 18th Annual Conference of the EACSL Computer Science Logic (CSL)*, 2009, S. 317–331

[3] Hofmann, Martin ; Rodriguez, Dulma: Linear Constraints over Infinite Trees. In: *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-18)*, 2012, S. 343–358

[4] Hofmann, Martin ; Rodriguez, Dulma: Automatic Type Inference for Amortised Heap-Space Analysis. In: *Proceedings of the 22nd Symposium on Programming (ESOP)*, 2013, S. 593–613

[5] Hofmann, Martin ; Jost, Steffen: Static Prediction of Heap Space Usage for First-order Functional Programs. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003, S. 185–197

[6] Hofmann, Martin ; Jost, Steffen: Type-Based Amortised Heap-Space Analysis. In: Sestoft, Peter (Hrsg.): *Proceedings of the 15th Symposium on Programming (ESOP)*, Springer Berlin Heidelberg, 2006, S. 22–37

[7] Hoffmann, Jan ; Aehlig, Klaus ; Hofmann, Martin: Resource Aware ML. In: Madhusudan, P. (Hrsg.) ; Seshia, Sanjit A. (Hrsg.): *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, Springer Berlin Heidelberg, 2012, S. 781–786

[8] Bauer, Sabine ; Hofmann, Martin: The Decision Problem for Linear Tree Constraints. In: *21st International Conference on Types for Proofs and Programs (TYPES), Collection of Abstracts* (2015)

[9] Bauer, Sabine ; Hofmann, Martin: Decidable linear list constraints. In: Eiter, Thomas (Hrsg.) ; Sands, David (Hrsg.): *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21)* Bd. 46, EasyChair, 2017 (EPiC Series in Computing), S. 181–199

[10] Bauer, Sabine ; Jost, Steffen ; Hofmann, Martin: Decidable Inequalities over Infinite Trees. In: Barthe, Gilles (Hrsg.) ; Sutcliffe, Geoff (Hrsg.) ; Veanes, Margus (Hrsg.): *Proceedings of the 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-22)* Bd. 57, EasyChair, 2018 (EPiC Series in Computing), S. 111–130

[11] Hoffmann, Jan: *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*, University of Munich, Diss., 2011

[12] Unnikrishnan, Leena ; Stoller, Scott D.: Parametric heap usage analysis for functional programs. In: *Proceedings of the 8th International Symposium on Memory Management (ISMM)*, 2009, S. 139–148

[13] Albert, Elvira ; Genaim, Samir ; Gómez-Zamalloa, Miguel: Parametric Inference of Memory Requirements for Garbage Collected Languages. In: *Proceedings of the 9th International Symposium on Memory Management (ISMM)*, ACM, 2010, S. 121–130

[14] Tarjan, Robert E.: Amortized Computational Complexity. In: *SIAM Journal on Algebraic Discrete Methods* 6 (1985), S. 306–318

[15] Hoffmann, Jan ; Aehlig, Klaus ; Hofmann, Martin: Multivariate amortized resource analysis. In: *Proceedings of the 38th ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011, S. 357–370

[16] Jost, Steffen ; Vasconcelos, Pedro B. ; Florido, Mário ; Hammond, Kevin: Type-Based Cost Analysis for Lazy Functional Languages. In: *Journal of Automated Reasoning* 59 (2017), S. 87–120

[17] Jost, Steffen ; Loidl, Hans-Wolfgang ; Hammond, Kevin ; Scaife, Norman ; Hofmann, Martin: "Carbon Credits" for Resource-Bounded Computations Using

Amortised Analysis. In: *Proceedings of the 2nd World Congress on Formal Methods (FM)*, 2009, S. 354–369

[18] OES, Hugo R. S. ; VASCONCELOS, Pedro B. ; FLORIDO, Mário ; JOST, Steffen ; HAMMOND, Kevin: Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In: *Proceedings of the 12th ACM SIGPLAN-SIGACT International Conference on Functional Programming (ICFP)*, 2012, S. 165–176

[19] HOFMANN, Martin ; MOSER, Georg: Multivariate Amortised Resource Analysis for Term Rewrite Systems. In: *13th International Conference on Typed Lambda Calculi and Applications (TLCA)*, 2015 (Leibniz International Proceedings in Informatics (LIPIcs)), S. 241–256

[20] HOFMANN, Martin ; MOSER, Georg: Amortised Resource Analysis and Typed Polynomial Interpretations. In: DOWEK, Gilles (Hrsg.): *Rewriting and Typed Lambda Calculi*, Springer International Publishing, 2014, S. 272–286

[21] HOFFMANN, Jan ; DAS, Ankush ; WENG, Shu-Chun: Towards Automatic Resource Bound Analysis for OCaml. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017, S. 359–373

[22] CARBONNEAUX, Quentin ; HOFFMANN, Jan ; SHAO, Zhong: Compositional Certified Resource Bounds. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015, S. 467–478

[23] CARBONNEAUX, Quentin ; HOFFMANN, Jan ; SHAO, Zhong ; RAMANANANDRO, Tahina: End-to-end verification of stack-space bounds for C programs. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, S. 270–281

[24] HOFFMANN, Jan ; SHAO, Zhong: Type-Based Amortized Resource Analysis with Integers and Arrays. In: CODISH, Michael (Hrsg.) ; SUMII, Eijiro (Hrsg.): *Proceedings of the Functional and Logic Programming: 12th International Symposium (FLOPS)*, 2014, S. 152–168

[25] HOFFMANN, Jan ; SHAO, Zhong: Automatic Static Cost Analysis for Parallel Programs. In: VITEK, Jan (Hrsg.): *Proceedings of the 24th Symposium on Programming (ESOP)*, Springer Berlin Heidelberg, 2015, S. 132–157

[26] Hoffmann, Jan ; Hofmann, Martin: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Ueda, Kazunori (Hrsg.): *Proceedings of the 24th Symposium on Programming (ESOP)*, Springer Berlin Heidelberg, 2010, S. 172–187

[27] Hoffmann, Jan ; Hofmann, Martin: Amortized Resource Analysis with Polynomial Potential: A Static Inference of Polynomial Bounds for Functional Programs. In: Gordon, Andrew D. (Hrsg.): *Proceedings of the 24th Symposium on Programming (ESOP)*, Springer Berlin Heidelberg, 2010, S. 287–306

[28] Vasconcellos, Cristiano D. ; Figueiredo, Lucília ; ao, Carlos C.: Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. In: *Journal of Universal Computer Science* 9 (2003), S. 873–890

[29] Sinn, Moritz ; Zuleger, Florian ; Veith, Helmut: A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In: Biere, Armin (Hrsg.) ; Bloem, Roderick (Hrsg.): *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, Springer International Publishing, 2014, S. 745–761

[30] Albert, Elvira ; Arenas, Puri ; Fernández, Jesús C. ; Genaim, Samir ; Gómez-Zamalloa, Miguel ; Puebla, Germán ; Román-Díez, Guillermo: Object-sensitive cost analysis for concurrent objects. In: *Software Testing, Verification and Reliability* 25 (2015), S. 218–271

[31] Albert, Elvira ; Arenas, Puri ; Genaim, Samir ; Puebla, German ; Zanardini, Damiano: COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In: Boer, Frank S. (Hrsg.) ; Bonsangue, Marcello M. (Hrsg.) ; Graf, Susanne (Hrsg.) ; Roever, Willem-Paul (Hrsg.): *Formal Methods for Components and Objects*. Springer-Verlag, 2008, S. 113–132

[32] Gulwani, Sumit ; Mehra, Krishna K. ; Chilimbi, Trishul: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, S. 127–139

[33] Ferdinand, Christian ; Heckmann, Reinhold: aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In: Jacquart, Renè (Hrsg.): *Building the*

Information Society. Springer US, 2004, S. 377–383

[34] Frohn, Florian ; Giesl, Juergen: Analyzing Runtime Complexity via Innermost Runtime Complexity. In: Eiter, Thomas (Hrsg.) ; Sands, David (Hrsg.): *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21)* Bd. 46, EasyChair, 2017 (EPiC Series in Computing), S. 249–268

[35] Frohn, Florian ; Giesl, Jürgen: Complexity Analysis for Java with AProVE. In: *Proceedings of the 13th International Conference on Integrated Formal Methods (IFM)*, 2017, S. 85–101

[36] Silva, Alexandra ; Rutten, Jan: A coinductive calculus of binary trees. In: *Information and Computation* 208 (2010), S. 578 – 593

[37] Dantchev, Stefan ; Valencia, Frank D.: On Infinite CSPs. In: *Modelling and Reformulating Constraint Satisfaction Problems* (2007), S. 48

[38] Blumensath, Achim ; Grädel, Erich: Automatic Structures. In: *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, 2000, S. 51–62

[39] Habermehl, Peter ; Iosif, Radu ; Vojnar, Tomáš: What Else Is Decidable about Integer Arrays? In: Amadio, Roberto (Hrsg.): *Proceedings of the 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, Springer Berlin Heidelberg, 2008, S. 474–489

[40] Bozga, Marius ; Iosif, Radu ; Konečný, Filip: Fast Acceleration of Ultimately Periodic Relations. In: Touili, Tayssir (Hrsg.) ; Cook, Byron (Hrsg.) ; Jackson, Paul (Hrsg.): *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, Springer Berlin Heidelberg, 2010, S. 227–242

[41] Schrijver, Alexander: *Theory of Linear and Integer Programming*. John Wiley and Sons, Inc., 1986

[42] Graham, Ronald L. ; Knuth, Donald E. ; Patashnik, Oren: *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1994

[43] Graham, Everest ; Poorten, Alf van d. ; Shparlinski, Igor ; Ward, Thomas: *Recurrence sequences*. American Mathematical Society, 2003 (Mathematical surveys and monographs)

[44] Wilf, Herbert S.: Chapter 1 - Introductory ideas and examples. In: Wilf, Herbert S. (Hrsg.): *Generatingfunctionology (Second Edition)*. Academic Press, 1994, S. 1 –29

[45] Denardo, Eric V. ; Rothblum, Uriel G.: Totally expanding multiplicative systems. In: *Linear Algebra and its Applications* 406 (2005), S. 142–158

[46] Withers, Christopher S. ; Nadarajah, Saralees: The nth power of a matrix and approximations for large n. In: *New Zealand Journal of Mathematics* 38 (2008), S. 171–178

[47] Ouaknine, Joël ; Worrell, James: Decision Problems for Linear Recurrence Sequences. In: Finkel, Alain (Hrsg.) ; Leroux, Jérôme (Hrsg.) ; Potapov, Igor (Hrsg.): *Proceedings of the 6th International Workshop on Reachability Problems (RP)*. Springer Berlin Heidelberg, 2012

[48] Ouaknine, Joël ; Worrell, James: Positivity Problems for Low-order Linear Recurrence Sequences. In: *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2014 (14), S. 366–379

[49] Birmajer, Daniel ; Gil, Juan B. ; Weiner, Michael D.: *Linear Recurrence Sequences with Indices in Arithmetic Progression and their Sums*. 2015. – http://arxiv.org/pdf/1505.06339v2.pdf

[50] Bell, Jason P. ; Gerhold, Stefan: On the positivity set of a linear recurrence sequence. In: *Israel Journal of Mathematics* 157 (2007), S. 333–345

[51] Halava, Vesa ; Harju, Tero ; Hirvensalo, Mika ; Karhumäki, Juhani: Skolem's Problem - On the Border between Decidability and Undecidability. In: *TUCS Technical Reports* (2005), Nr. 683

[52] Blondel, Vincent D. ; Portier, Natacha: The presence of a zero in an integer linear recurrent sequence is NP-hard to decide. In: *Linear Algebra and its Applications* 351 (2002), S. 91–98

[53] Fine, Nathan J. ; Wilf, Herbert S.: Uniqueness Theorems for Periodic Functions. In: *Proceedings of the American Mathematical Society* 16 (1965), S. 109–114

[54] Abu-Omar, Amer ; Kittaneh, Fuad: Estimates for the numerical radius and the spectral radius of the Frobenius companion matrix and bounds for the zeros of polynomials. In: *Annals of Functional Analysis* 5 (2014), S. 56–62

[55] Etessami, Kousha ; Stewart, Alistair ; Yannakakis, Mihalis: Greatest Fixed Points of Probabilistic Min/Max Polynomial Equations, and Reachability for Branching Markov Decision Processes. In: Halldórsson, Magnús M. (Hrsg.) ; Iwama, Kazuo (Hrsg.) ; Kobayashi, Naoki (Hrsg.) ; Speckmann, Bettina (Hrsg.): *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP)*, Springer Berlin Heidelberg, 2015, S. 184–196

[56] Pool, Roman ; Cáceres, Manuel: Effective Perron-Frobenius eigenvalue for a correlated random map. 82 (2010), S. 035203

[57] Goldberg, M. ; Zwas, G.: On matrices having equal spectral radius and spectral norm. In: *Linear Algebra and its Applications* 8 (1974), S. 427–434

[58] Béal, Marie-Pierre ; Berstel, Jean ; Eilers, Soren ; Perrin, Dominique: Symbolic dynamics. (2010)

[59] Denardo, Eric V. ; Rothblum, Uriel G.: A Turnpike Theorem For a Risk-Sensitive Markov Decision Process with Stopping. In: *SIAM Journal Control and Optimization* 45 (2006), Nr. 2, S. 414–431

[60] Droste, Manfred ; Kuich, Werner ; Vogler, Heiko: *Handbook of Weighted Automata, 1st edition*. Springer Publishing Company, Incorporated, 2009

[61] Chandru, Vijay ; Hooker, John: *Optimization Methods for Logical Inference*. Wiley, 1999

[62] Choffrut, Christian ; Karhumäki, Juhani: Combinatorics of Words. In: Rozenberg, Grzegorz (Hrsg.) ; Salomaa, Arto (Hrsg.): *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*. Springer Berlin Heidelberg, 1997, S. 329–438

[63] Dömösi, Päl ; Ito, Masami: *Context-free languages and primitive words*. World Scientific, 2014