Linear Tree Constraints

Sabine Bauer and Martin Hofmann, University of Munich

June 27, 2018

Abstract

Linear tree constraints were introduced by Hofmann and Rodriguez in the context of amortized resource analysis for object oriented programs. More precisely, they gave a reduction from inference of resource types to constraint solving. Thus, once we have found an algorithm to solve the constraints generated from a program, we can read off the resource consumption from their solutions.

These constraints have the form of pointwise linear inequalities between infinite trees labeled with nonnegative rational numbers. We are interested in the question if a system of such constraints is simultaneously satisfiable. Bauer and Hofmann have recently identified a fragment of the tree constraint problem (UTC) that is still sufficient for program analysis and they proved that the list case of UTC is decidable (which was presented at LPAR-21), whereas the case with trees of degree at least two remained open. In this paper, we solve this problem. We give a decision procedure that covers the entire range of constraints needed for resource analysis.

1 Introduction and Related Work

We start with a short overview of related work in amortized resource analysis, because that is where the constraint problem originates from. The idea of amortized analysis goes back to the 1980s [28]. 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 (cf. [12]). 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 a better bound than the worst case bound will hold. One well known example is copying a FIFO queue that is modeled with two stacks [13]. There one starts with pushing the elements on the first stack and when the first POP operation 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 has been done by moving the previous entries to the other stack. Another example are the so called self-adjusting data structures [28].

Hofmann and Jost first applied amortized analysis by the potential method to first-order functional programs in [20]. 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 was required to be linear. The approach was later generalized to multivariate polynomial potential by Hoffmann [16, 17, 13, 12]. This was the starting point for many other investigations in this direction. Hoffmann and Moser applied amortized analysis to term rewriting [22, 21]. Hoffmann refined his work, made it fully automatic and carried over the analysis to concurrent programs and programs in C and OCaml [15, 5, 6, 18, 19]. Rodriguez introduced an amortized analysis for a fragment of Java, which features object oriented programming, polymorphic functions and monomorphic recursion [25, 23, 20, 24] and is called RAJA (Resource Aware JAva).

Among other related work, mainly on resource analysis, are [9, 4, 11, 1, 10], which use different methods than our approach.

The analysis system for RAJA by Rodriguez and Hofmann is the motivation for our tree constraints. There the resource-type inference algorithm outputs conditions that must hold for the potential of the objects (represented as trees) in form of linear tree constraints. One can determine the resource consumption of a RAJA program if one has a solution to its constraints. Recently, the list constraint satisfiability problem for RAJA was proven decidable [3]. In this paper, we generalize that argument to trees. This decidability result enables us to analyze arbitrary RAJA programs with respect to their resource consumption. Until now this was only possible for a subset of programs that need linear resources.

In the prototype implementation of RAJA, one of the examples is sorting a list using merge-sort. There 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 [29, 2], 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 make our analysis independent of this construction and thus closer to real Java.

In addition to that, especially for bigger programs (like bank account models) or programs with auxiliary functions (like the sieve of Eratosthenes) or nested data structures, the constraint generation is very involved and often leads to superlinear potential. The same is true for cascades of recursive calls.

This paper is organized as follows: in the next section 2 we formulate the tree constraint satisfiability problem and recall existing results. Then we prove decidability of the tree case in section 3 and conclude in section 4.

## 2 Syntax and Semantics

A tree constraint system is a set of pointwise linear inequalities between tree variables, as for example $x \geq r(x) + l(x) + y$, where $x, y$ are binary trees with labels $l$ and $r$ such that $r(x)$ is the

\[ \text{raja.tcs.lif.lmu.de} \]
right subtree of \( x \) (and \( l(x) \) the left subtree.) In our setting, these tree variables can be instantiated with infinite trees that contain a nonnegative rational number in each node. The degree of the trees is arbitrary but finite.

Let \( L \) be a finite set of tree labels, \( \lambda \) a variable, \( n \) a number and let \( \Diamond(x) \) denote the root of tree \( x \). The formal syntax for the linear tree constraints is shown in Figure 1.

Figure 1: Linear Tree Constraint Syntax

\[
\begin{align*}
    t &::= x | l(t), \text{ where } l \in L \text{ with } |L| < \infty \quad \text{(Atomic tree)} \\
    te &::= t|te + te \quad \text{(Tree term)} \\
    c &::= te \geq te \quad \text{(Tree constraint)}
\end{align*}
\]

In addition to the tree constraints, we have arithmetic constraints given for the numbers in selected nodes of the trees that take the form of an arbitrary linear program with integer coefficients. 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 (where the root symbol \( \Diamond \) becomes \( \text{head} \) and the only label is \( \text{tail} \)) is \( \text{head}(x) + \text{head}(y) \geq 2 + \text{head}(\text{tail}(z)) \). Figure 2 gives the syntax for them.

Figure 2: Arithmetic Constraint Syntax

\[
\begin{align*}
    v &::= n|\lambda|\Diamond(t) \quad \text{(Atomic arithmetic expression)} \\
    h &::= v|h + h \quad \text{(Arithmetic term)} \\
    c &::= h \geq h \quad \text{(Arithmetic constraint)}
\end{align*}
\]

Each inequality over tree variables corresponds to infinitely many inequalities over arithmetic variables (i.e. variables for the numbers in the nodes.) Thus the problem to decide whether a set of tree constraints is simultaneously satisfiable can not directly be reduced to feasibility of a (finite) linear program.

A solution of the tree constraints is a set of infinite trees for which the constraints hold pointwise for each number in the nodes. More precisely, a constraint \( x \geq y \) holds for concrete trees \( t_1, t_2 \), if \( \Diamond(t_1) \geq \Diamond(t_2) \) and for all labels \( l \) (denoting the immediate subtrees) holds \( l(t_1) \geq l(t_2) \) (cf. rule (Label) in Figure 5).

**Example 1.** The system consisting of the arithmetic constraint \( \Diamond(t_1) = 1 \) and the tree constraints

\[
r(t_1) \geq 2l(t_1), l(t_1) \geq 3t_1, t_1 \geq lr(t_1),
\]

where the \( \Diamond(\cdot) \) symbol denotes the variable in the root of a tree, is unsatisfiable, because it implies \( 1 = \Diamond(t_1) \geq \Diamond(lr(t_1)) \geq 2\Diamond(l(t_1)) \geq 6\Diamond(t_1) = 6 \). The system \( \Diamond(t_2) = 1, r(t_2) = 2t_2, l(t_2) = 3t_2 \) has the solution in Figure 3. The subtrees can be computed by duplicating the value in the root of the
subtree when going right and multiplying by three when going left:
\[
\forall w \in (l|r)^* : \mathcal{O}(w(t_2)) = 2^i3^j, \ i = \text{number of } r's \text{ in } w, \ j = \text{number of } l's \text{ in } w.
\]

The following problems are closely related \[3\].

- **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
  
  \[
  x_n = a_1x_{n-1} + \cdots + a_kx_{n-k}, \ n > k
  \]
  
  \[
  x_1 = b_1, \ldots, x_k = b_k
  \]
  
  \[
  a_i, b_i \in \mathbb{Q}, b_i \geq 0 \text{ for all } i = 1, \ldots, k, a_k \neq 0,
  \]
  
  Asked: Is there an index \( n \) such that \( x_n = 0 \)?

- **List Constraint Satisfiability Problem (LC)**
  
  Given: A finite system of list constraints (constraints over trees of degree 1 with label tail) and arithmetic constraints
  
  Asked: Is there a set of lists, which simultaneously satisfies all constraints in of the system in \( \mathbb{D} = \mathbb{Q}_0^+ \cup \{\infty\} \)?

- **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 \( \mathbb{D} \)?

In \[3\], it is shown that SML can be reduced to LC. Thus LC and TC are very hard and probably undecidable problems; at least the decidability status of the famous and NP-hard SML problem is still unknown. This led to the consideration of \textit{unilateral} constraints, that can be shown to be
sufficient for our purposes and that are considerably easier to solve.

**Definition 1.** A unilateral tree constraint is a constraint with only one summand on the greater side of the inequality (i.e. of the form \( t \geq te \) according to Figure 1, or equivalently \( t \geq t_1 + \cdots + t_n \)).

We call unilateral tree constraints UTC.

For instance, the satisfiable tree constraints in Example 1 are not unilateral.

It follows from the nonnegativity of the coefficients on the right hand side, that UTC is a proper fragment of TC. Indeed, it was shown that for the list case there exists a polynomial decision procedure by reduction to linear programming [3]. This paper shows that UTC is also decidable.

In contrast to ULC (Unilateral List Constraints), our decision procedure is not polynomial in the size of the input.

We also remark that according to the recurrence-like syntax nearly all constraints have only nonlinear solutions. For instance, in the case of lists, the linear system by Hofmann and Rodriguez can only (partially) treat periodic lists. In the tree case, we have analogous growth rates as for lists in [3]. This means, as soon as we have a tree constraint with sums like e.g. \( lrlrx \geq lr + lrx \), the tree exhibits exponential growth.

## 3 Decidability

In this section we establish our main theorem, namely that satisfiability of unilateral linear tree constraints is decidable. The proof is structured as follows: We observe that unsatisfiability is semi-decidable. We 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 following from a set of constraints using a sound and complete proof system,
- characterize 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. Both together imply that satisfiability is decidable.

### 3.1 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 4 to obtain the constraints for the next step.
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 1.** 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 [7].

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.

### 3.2 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 5. 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.

Figure 5: Proof system for unilateral tree constraints

- \(TC \vdash ux \geq ux\) (Reflexivity)
- \(TC \vdash x \geq y\) \(\rightarrow\) \(TC \vdash l(x) \geq l(y)\) (Label)
- \(x \geq y_1 + \ldots + y_n \in TC\) \(\rightarrow\) \(TC \vdash uy_i \geq z\) (Transitivity)

Tree expressions are of the form \(ux\) where \(u : \Sigma^*\) and \(\Sigma\) 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 judgment $TC \vdash x \geq y$, 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 5.

**Theorem 1.** The proof system in Figure 5 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 $E = \{ (x,y) \mid$ there is a constraint $x' \geq y_1 + \cdots + y_n \in TC$ and $u : \Sigma^*.x = ux'$ and $y = uy_i$ for some $i \}$.

Now fix a tree expression $x_0$ and define a valuation $\eta$ 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 $\eta$ 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 $u : \Sigma^*$. Now if $ux$ is unreachable from $x_0$ then $\eta(\Diamond(ux)) = \infty$ and the inequality holds. On the other hand, 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 $\eta$ 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 $TC$.

As a next step, we are interested in the set $L^\geq_{x,y}$ 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 $L_{x,y} := \{ u \mid TC \vdash ux \geq y \}$ with $x$ and $y$ fixed tree variables as an auxiliary step to compute $L^\geq_{x,y}$.

**Theorem 2.** The language $L_{x,y}$ is regular.

**Proof.** We construct a finite automaton that accepts a word $w$, if and only if $TC \vdash ux \geq y$.

With the proof system in Figure 5 we can first build a stack automaton $A$ from $TC$, that reads no input and such that $u : L_{x,y}$ if and only if $A$ accepts beginning from stack $ux$.

We give the idea for the construction of a slightly more general stack automaton, namely a stack automaton that accepts a word $uxyw^r$ 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 it 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 2. Consider the constraints

\[ lr(x) \geq rr(x), \]
\[ lr(x) \geq l(y). \]

The stack automaton \( M = (Z, \Sigma, \Gamma, \delta, z_0, \#) \) 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 \( \delta \) is defined as depicted in Figure 6. Note that the lower case input symbols in \( \Sigma \) correspond to the according upper case letters in the stack alphabet \( \Gamma \). Here \( z_0 \) is the write-state, \( z_{\infty} \) the compare-state and \( z', z'' \) are auxiliary states. For instance, in Figure 6 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 and the stack content, that are then mapped to the next state and the new stack content by \( \delta \). In the picture, the triples on the arrows mean the input symbol, the stack before and the stack after the transition. The symbol \( a \in \Sigma \)

The language \( L \) corresponding to this generalization 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 \( lr(x) \geq lrlrx \) and consider the word \( (lr)^i xx (rl)^i+1 \), which is implied by the constraints and thus in \( L \). Then the word \( \alpha = (lr)^p \) is obviously longer than \( p \) and for each division of \( \alpha = uww \) in three words \( u, v, w \), for instance with \( v = lr \), then holds e.g. \( \forall k. (lr)^i+k xx (rl)^i+1 \in L \), 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.

But we can use $L$, however, to show that the language $L_{x,y}$ with the second variable $y$ and its label word and $x$ fixed is always regular. For that, we build the above mentioned stack automaton $A$ similar to the construction above. We can assume w.l.o.g. (possibly by introducing new states), that $A$ has only transitions of the form $x \to y, x \xrightarrow{\text{pop}(a)} y$, or $x \xrightarrow{\text{push}(a)} y$, 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 7. Now we obtain for $L_{x,y}$, with $x, y$ both variables, the representation in Figure 8. From this we can read off a finite automaton $B$ for $L_{x,y}$ directly: the states are the states of $A$, for each pair $(x, y) \in Q$ we introduce an $\epsilon$-move from state $x$ to state $y$, and the $A$-transitions $x' \xrightarrow{\text{pop}(a)} x''$ are the nontrivial moves that consume the letter $a$.

We then also have $L_{ux,y} = \{w | uw : L_{x,y}\}, L_{x,vy}$ and $L_{ux,vy}$ regular. The disjoint union over 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 3.** Let the constraints be

$$lx \geq x, x \geq rz, lrz \geq ll, ly \geq y.$$  

\footnote{In the list case, where we have only one label, this is a direct consequence of Parikh’s theorem.}

**Figure 8: Rules for $L_{x,y}$**

$$\begin{align*}
(x, y) : Q \\
\epsilon : L_{x,y}
\end{align*}$$

$$\begin{align*}
u : L_{x''', y} \\
(x'', x''') : Q \\
x'' \xrightarrow{\text{pop}(a)} x''' \\
(x, x') : Q
\end{align*}$$
Then $L_{x,y} = L_{x,ly} = l^+$ and $L_{lx,y} = l^*$. The language $L_{lrz}^\leq = \{lrz, ll_y, l_y, \epsilon_y\}$, 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 $L_a^\geq \cap L_b^\leq := \{x \mid a \leq \Diamond(x) \leq b\}$ can be computed using the languages $L_a^\geq$ and $L_b^\leq$ 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 $a = \Diamond(lrzz)$, then the subtree $x$ is $lrzz$. Thus we can write $L_a^\geq \cap L_b^\leq = \{x \mid TC \vdash y_a \leq x \leq y_b\}$, where $y_a$ (resp. $y_b$) is the tree with root $\Diamond(y_a) = a$ (resp. $b$).

**Example 4.** Consider the constraints $\Diamond(x) = 1, lrx \geq x, lx \geq x, mx \geq x, x \geq rlx, x \geq mlx$.

The language of trees greater with root greater than the root of $x$, less than the root of $x$, and equal to it are:

$$L_{\Diamond(x)}^\geq = (lr | l | m)*x, L_{\Diamond(x)}^\leq = (ml | rl)^*x,$$

$$L_{\Diamond(x)}^\neq = L_{\Diamond(x)}^\leq \cap L_{\Diamond(x)}^\geq = m(lr)^*lx = ml(rl)^*x.$$

This is obtained by iteratively applying the constraints and transitivity.

### 3.3 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 then there is 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 4 to all thus 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 \(wx \geq w_1y_1 + \cdots + w_iy_i\), with all \(w_i\) shorter than \(w\).
- **upper bounds** have the shape \(wx \leq w'y - w_1y_1 - \cdots - w_iy_i\), with all \(w_i\) and \(w'\) shorter than \(w\).
- **undirected constraints** have two label words of the same length on both sides, 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 label application, which makes it immediate to decide), 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 5.** 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 9. Then we 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. Why we need this, will become clearer in Example 6.

### 3.4 Idea and Examples

Before moving to Example 6, we will briefly explain the idea behind our procedure. Afterwards, we will cover the technical details. The intuition is that we label all nodes in the trees which are in a set \(L^a \cap L^b\) for arithmetic variables \(a\) and \(b\) with sets of intervals, in which the number in the node has to lie. These intervals are derived from the constraints. Nodes with the same set of intervals are defined to be in the same **class**. Then we show that in a subset of all constraint systems that contains the satisfiable ones, there are only finitely many different classes. 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 \(\hat{\phi}(x)\) in the interval assigned to \(\hat{\phi}(x)\)). 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 1 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 6. Consider the (list-) constraints

\[ \Diamond(x) = \Diamond(y) = 1, x \geq y, lx \leq x, ly \geq y. \]

They are equivalent to the system \( \Diamond(x) = \Diamond(y) = 1 \) in conjunction with \( C = \{c_1, c_2, c_3, c_4\} \), where

\[ c_1 := ly \leq lx, c_2 := lx \geq ly, c_3 := lx \leq x, c_4 := ly \geq y, \]

which is in normal form (with the constraint \( c_1 \) redundant in this case). The constraints \( c_3 \) (resp. \( c_4 \)) deliver the intervals \([0,1]\) for \( \Diamond(lx) \) (resp. \([1,\infty]\) for \( \Diamond(ly) \)). Then \( c_1 \) gives us the interval \([1,1]\) for \( \Diamond(lx) \) 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 6 is satisfiable by the trees consisting only of 1s. If we modify it slightly, it becomes unsatisfiable:

Example 7. 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

\[ d_1 := ly \leq lx, d_2 := lx \geq ly, d_3 := lx \leq 0.5x, d_4 := ly \geq 2y. \]

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 \( \Diamond(lx) \) 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 1 applies.

3.5 Satisfiability is Decidable

Before we prove the central fact (Theorem 1) of this paper, 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 [8].

Lemma 2. 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 3. If for word \( x, y, z \) holds \( x^ny^m = z^k \), with \( n, m, k \geq 2 \), then exists \( t \) such that \( x, y, z \in t^* \).

Lemma 4. Let \( c \) be a unilateral tree constraint. If it is of the form

\[ a_1a_2 \ldots a_nx \geq c_1 \cdot a_2 \ldots a_nx + \cdots + c_n \cdot a_nx + c_{n+1} \cdot x \]

with all \( a_i \neq c, c_j \in \mathbb{N}_0 \) and label words and \( x \) a tree, then it can be transformed (by 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.
We apply Lemma 2 to Theorem 3. Satisfiability of linear tree constraints is semi-decidable.

We have already an interval for the starting nodes of the DAG constructed above. We traverse with all constraints, we observe the following. The membership of all nodes in interval \([w, \infty]\), which have the form \(a_1 \ldots a_n\), we return this as a result.

Proof. We assume that there is a label word \(t \in \Sigma^*\) such that all summands stay dependent of \(a_1 a_2 \ldots a_n x\). (If such a \(t\) does not exist, then all summands are already independent.) This is,

\[ ta_1 a_2 \ldots a_n \text{ has the suffixes } ta_2 \ldots a_n, ta_3 \ldots a_n, \ldots, ta_n, t. \tag{2} \]

Thus \( ta_2 \ldots a_n = q(a_1 \ldots a_n)^i a_1 \ldots a_n\). According to (2), \( ta_2 \ldots a_n\) is a suffix of \( ta_1 a_2 \ldots a_n = q(a_1 \ldots a_n)^{i+1}\). This means that \(a_1\) and \(a_2 \ldots a_n\) commute. Thus there is \(p_1\) such that both are in \(p_1^+\).

Similarly, \(ta_1 a_2 \ldots a_n\) has the suffix \(ta_3 \ldots a_n = q(a_1 \ldots a_n)^i a_3 \ldots a_n\), and thus \(a_1 a_2\) and \(a_3 \ldots a_n\) commute (see Figure 10, where words of the same length are written in boxes). We can thus conclude that there is \(p_2\) with \(a_1 a_2\) and \(a_3 \ldots a_n\) in \(p_2^+\). We proceed the same way until we obtain in the last step that \(a_1 \ldots a_{n-1}\) and \(a_n\) commute. We now write \(a_1 = p_1^{i_1}, a_2 \ldots a_n = p_1^{j_1}\) and \(a_1 a_2 = p_2^{i_2}, a_3 \ldots a_n = p_2^{j_2}\), etc.

Application of Lemma 3 allows us to conclude from \((a_1 \ldots a_n)^2 = p_1^{i_1+j_1} p_2^{i_2+j_2} = p_3^{2(i_1+j_3)}\) 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 3.** Satisfiability of linear tree constraints is semi-decidable.

Proof. We assume that all constraints are in the normal form described above. Then we 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 if the constraints have label word of length exactly \(n\). We now calculate the sets \(L_a^e \cap L_b^e\) for all pairs of arithmetic variables \(a, b\). W.l.o.g. we can further assume that all \(a, b\) are nonzero and not infinity: for instance, we could try all variants of the constraints with additional \(A \ni a_i = 0\) or \(B \ni a_i = \infty\) for all pairs of disjoint subsets \(A, B\) of the set of arithmetic variables and such that all other \(a_i\) are neither zero nor infinity. If one of them is satisfiable, we return this as a 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 \(wx \geq w_1 y_1 + \cdots + w_m y_m\), with all \(w_i\) shorter than \(w\), we add the interval \([\sum w_i \hat{w} y_i, \infty]\). For the upper bounds, that have the shape \(wx \leq w' y - w_1 y_1 - \cdots - w_m y_m\), with all \(w_i\) and \(w'\) shorter than \(w\), we add the interval \([0, \hat{w} y - \sum w_i y_i]\). For the undirected constraints, we observe the following. The membership of all nodes in \(L_a^e \cap L_b^e\) 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 \(x \geq y_1 + \cdots + y_m\) (resp. \(x \leq y - z_1 - \cdots -\)
We claim that after a finite amount of steps, no new intervals are derived any more. This is, if
the new set of intervals \{\sum_i a_i, \infty \mid \{a_i, b_i\} \text{ is an interval for } \diamond(y_i)\} (\text{resp. } \{0, b - \sum_i c_i \mid \{a, b\} \text{ is an interval for } \diamond(y) \text{ and } \{c_i, d_i\} \text{ is an interval for } \diamond(z_i)\}).

Further, we set all nodes that have bounds in only one direction to 0, [0, \infty] 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 formalizes how we compute new interval sets.

\[ I(x) + I(y) = \{[a + c, \infty] \mid [a, b] \in I(x), [c, d] \in I(y)\}, \]
\[ I(x) - I(y) = \{[0, b - c] \mid [a, b] \in I(x), [c, d] \in I(y)\}. \]

**Observation 1.** The order of evaluation does not play any role for sums of interval sets (i.e. \(I(x) - I(y) - I(z) = I(x) - (I(y) + I(z))\)).

To prove this, let w.l.o.g. be \(I(x) = [a, b], I(y) = [c, d], I(z) = [e, f]\). Then

\[
[a, b] - [c, d] - [e, f] = [0, b - c - e] = [0, b - (c + e)] = [a, b] - [e + \infty] = [a, b] - ([c, d] + [e, f]).
\]

In step \(n+1\), we apply the rule (LabelSum) in Figure 3 to the constraints to make their left sides have a label word of length increased 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 in 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. 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. This is, 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). But since all considered nodes are bounded from above and below, we would at some point obtain a contradiction (see Example 7).

Thus it is enough to give a criterion ensuring that we need no longer search for new classes because we found all. Having this, 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 = lcm(4, 3). Note that in the list case, \(S\) is a bound on the period length of the solution lists (cf. 3).

The criterion looks as follows: If in \(S\) iterations no new interval sets for the nodes in \(L = \cup_{a,b}(L_a^x \cap L_b^x)\) 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 \(x\),

\[
z_m)
\]

\[
\text{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). But since all considered nodes are bounded from above and below, we would at some point obtain a contradiction (see Example 7).}

Thus it is enough to give a criterion ensuring that we need no longer search for new classes because we found all. Having this, 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 = lcm(4, 3). Note that in the list case, \(S\) is a bound on the period length of the solution lists (cf. 3).

The criterion looks as follows: If in \(S\) iterations no new interval sets for the nodes in \(L = \cup_{a,b}(L_a^x \cap L_b^x)\) 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 \(x\),

\[
z_m}
\]
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 1** (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 \( I = \bigcup_{j \in \mathbb{N}} I_j = I_{n+S} \).

**Claim 2** (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 \( I_{n+k+l:S} = I_{n+k} \). We consider three cases. If we have a lower bound constraint \( px \geq \sum_i y_i \), we know that for all label words \( p \) with \( px \in L \) and \( |p| = S \), 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 \((*)\). Similarly, in case of upper bounds \( px \leq y - \sum_i z_i \), the upper bounds for \( qy_i \) are not stronger than those for \( y_i \) and for all label \( px \in L \) on level \( n \), this is a consequence of the assumption, and if \( qx \in L \) and \( qx_i \notin L \), then \( qz_i \) has interval \([0,0]\). Last, if \( qx \notin L \) then \( qx \) has interval \([\infty, \infty]\) and delivers no new bounds. This property is called \((**)\). In these two cases, the right hand side in \((*)\) and \((**)\) is on level less or equal to the level on the left.

Thus we may assume that all \( py_i, qy_i, qz_i \in L \). For all label words \( q, p \) with \( qpx \in L \) and \( |q|, |p| = S \), we have \( I(qy_i) = I(y_i), I(q) = I(y), I(z_i) = I(z) \) and the set of intervals for \( qpx \), which is the sum of the intervals for \( qx, qz_i \) is (after intersection) not smaller than the set of intervals for \( qpx \) (according to \((*)\) and \((**)\)), and also not smaller than the interval for \( x \). More precisely, we have

\[
I(qpx) = I(qy_i) + \cdots + I(qy_m) = I(y_i) + \cdots + I(y_m) = I(px) = I(x), \quad \text{or resp.}
I(qpx) = I(qy) - I(qz_i) - \cdots - I(qz_m) = I(y) - I(z_i) - \cdots - I(z_m) = I(px) = I(x).
\]

The last case is if we have an undirected constraint \( x \geq y + z \). This implies \( px \geq py + pz \), and with \( I(px) = I(x), I(py) = I(y), I(pz) = I(z) \), we have 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 \( I(px) \neq I(x) \). 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. \( qx \leq z - R \)) with \( |q| = S \). For the lower bounds, we have to consider all possibilities for the choice of \( y \), whereas for the upper bounds there is only one positive summand. We can therefore assume that \( I(px) \subset I(x) \) because of the choice of \( S \); else, if \( pqy \) had strictly weaker lower bounds (resp. strictly weaker upper bounds) than \( px \), the interval \( I(pqy) \) would contain \( I(px) \) and thus \( qy \) could not have \( z \) as a bound. Then 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 $z = y$ hold\footnote{If this is not the case for the initial $y$, the next candidate for $x$ is $z$.}. So we have a constraint $px \geq x + R$ or $px \leq x - R$ derivable just by unfolding using the (LabelSum) rule in Figure 4. We just treat the first since both are similar.

According to Lemma\footnote{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\footnote{ppx}} 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 $qR := qr_1 + \cdots + qr_m$ are independent of $qpx$ for all $q$ of length $S$. If the second happens, if no tree $t'x$ is reachable\footnote{ppx} 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 $p p x = t' p x = t'^2 x \geq t' x + t^k z + R'$, and that means $p' x \geq p'^{-1} x + p^k x$ 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 $(\Diamond(p'))_i$.

In both cases, this is a contradiction, since only finitely many $p' x$ 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 are satisfiable.

This completes our proof. \hfill \Box

Combined with the semi-decidable unsatisfiability, we can decide UTC.

**Example 8.** Let the constraints be

$$\Diamond(y) = 1, y \geq l y, y \geq r y \text{ and } l x \geq x + y, r x \geq x + y, x \geq l r x.$$ 

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 $l r^* x$ and $\Diamond(y)$. Thus we only need to compute intervals for $l r^* x$. The root $\Diamond(x)$ gets the interval $[1, 1]$. On level two, there is no node in $L$. Then, on level three, $\Diamond(l r x)$ 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 $\forall w \in (l r)^*.\Diamond(wx) = 1$ is not possible.

### 4 Conclusion

We have proven that linear constraints over infinite trees, as generated by an automatic resource type inference for the language RAJA, are decidable. Our approach uses finite automata to generalize the list constraint theory to trees. For the latter, satisfiability was previously proven
decidable in polynomial time. In contrast to that, 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.

With this result, we can now analyze arbitrary object oriented (RAJA-) programs with respect to 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 more concrete description of minimal constraint solutions as closed formulas and an implementation based on the existing RAJA tool.

We also will further investigate possibilities to increase the efficiency by optimizing the decision procedure. In order to develop a powerful tool for analyzing real programs, we plan to add Java features (as exceptions, garbage collection, concurrent threads) to RAJA. Another approach would be implementing an automatic translation from Java code to an equivalent RAJA program (w.r.t. the resource consumption), which can then be analyzed using our results.

References

[1] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-sensitive cost analysis for concurrent objects. Software Testing, Verification and Reliability, 25(3):218–271, 2015.

[2] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In Jan Vitek and Doug Lea, editors, Proceedings of the 2010 International Symposium on Memory Management (ISMM’10), pages 121–130. ACM, 2010. doi:10.1145/1806651.1806671.

[3] Sabine Bauer and Martin Hofmann. Decidable linear list constraints. In Thomas Eiter and David Sands, editors, LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, volume 46 of EPiC Series in Computing, pages 181–199. EasyChair, 2017.

[4] Achim Blumensath and Erich Grädel. Automatic Structures. In Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000), pages 51–62. IEEE Computer Society Press, June 2000.

[5] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional Certified Resource Bounds. pages 467–478. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015.

[6] Quentin Carbonneaux, Jan Hoffmann, Zhong Shao, and Tahina Ramananand. End-to-end verification of stack-space bounds for C programs. pages 270–281. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014.

[7] Vijay Chandru and John Hooker. Optimization Methods for Logical Inference. Wiley, 1999.
[8] Christian Choffrut and Juhani Karhumäki. *Combinatorics of Words*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[9] Stefan Dantchev and Frank D Valencia. On Infinite CSPs. *Modelling and Reformulating Constraint Satisfaction Problems*, 2009.

[10] Florian Frohn and Juergen Giesl. Analyzing Runtime Complexity via Innermost Runtime Complexity. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 249–268. EasyChair, 2017.

[11] Reinhold Heckmann and Christian Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. AbsInt Angewandte Informatik GmbH. http://www.absint.com.

[12] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, University of Munich, 2011.

[13] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, November 2012.

[14] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. *Resource Aware ML*, pages 781–786. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 359–373, 2017.

[16] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *APLAS*, pages 172–187, 2010.

[17] Jan Hoffmann and Martin Hofmann. *Amortized Resource Analysis with Polynomial Potential*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[18] Jan Hoffmann and Zhong Shao. *Type-Based Amortized Resource Analysis with Integers and Arrays*, pages 152–168. Springer International Publishing, Cham, 2014.

[19] Jan Hoffmann and Zhong Shao. *Automatic Static Cost Analysis for Parallel Programs*, pages 132–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[20] Martin Hofmann and Steffen Jost. 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 ’03*, pages 185–197, New York, NY, USA, 2003. ACM.

[21] Martin Hofmann and Georg Moser. Amortised Resource Analysis and Typed Polynomial Interpretations (extended version). *CoRR*, abs/1402.1922, 2014. http://arxiv.org/abs/1402.1922.

[22] Martin Hofmann and Georg Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In Thorsten Altenkirch, editor, *13th International Conference on Typed
[23] Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In CSL: 18th EACSL Annual Conference on Computer Science Logic. LNCS, Springer-Verlag, 2009.

[24] Martin Hofmann and Dulma Rodriguez. Linear Constraints over Infinite Trees. In Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR’12, pages 343–358, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] Martin Hofmann and Dulma Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In ESOP: 22nd European Symposium on Programming, 2013.

[26] Dulma Rodriguez. Amortized Analysis for Object Oriented Programs. PhD thesis, University of Munich, 2012.

[27] Alexander Schrijver. Theory of Linear and Integer Programming. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[28] Robert Endre Tarjan. Amortized Computational Complexity. SIAM Journal on Algebraic Discrete Methods, 6(2):306–318, 1985.

[29] Leena Unnikrishnan and Scott D. Stoller. Parametric heap usage analysis for functional programs. In Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009, pages 139–148, 2009.