master thesis in computer science

by

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

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Assoc. Prof. Dr. Georg Moser,
insert the institute of your supervisor

Innsbruck, 29 January 2018

Master Thesis

Amortized Resource Analysis for Term Rewrite Systems

Manuel Schneckenreither (1117198)
manuel.schneckenreither@student.uibk.ac.at

29 January 2018

Supervisor: Assoc. Prof. Dr. Georg Moser

Eidesstattliche Erklärung

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

Datum	Unterschrift

Abstract

Amortized resource analysis yields highly accurate resource complexity bounds which are commonly much better than the ones by other resource analysis techniques, like the worst-case analysis. During the last decades techniques for amortized resource analysis were developed and transformed to be compatible with arbitrary data structures and (typed) Term Rewrite Systems. This seminar report briefly introduces these developments and shows transformation of the analysis methods to TRSs. Further, the current status of the implementation for the univariate polynomial analysis is presented.

Contents

1 Introduction

Quantitative analysis of algorithms, especially according to their execution time, are important when comparing algorithms, designing efficient programs and to identify performance bottlenecks in software [5]. In recent years amortized resource analysis techniques have been developed and advanced in several publications.

In 2000 Hofmann presented in [6] the basic ideas for the amortized resource analysis: A resource type was introduced that controls the number of constructor symbols in recursive functions and ensures linear heap space. However, this heap-space analysis is restricted to data structures composed of lists, binary trees and pairs. The result of the paper [6] is a compilation process of a linear typed first-order functional programming language (LFPL) into malloc()-free C code. They show that the program will run in linear heap-size according to the input paramters. Similarly in [7] and [8] Hofmann and Jost convert the concepts for the heap-size analysis of [6] to a first-order typed functional language and respectively Featherweight Java programs [11] using resource annotations. In 2006 in [8] they first used the notation of potential, which was utilized in the later works and thus forge the bridge to amortized resource analysis as it was introduced by Tarjan in 1985 [13].

In 2009 Jost et al. improved the previous version for functional programs such that arbitrary (recursive) data structures can be used and also lifted the method to incorporate worst-case runtime bounds [12]. The experiment results show astonishing accurate predictions when compared to actual resource usage but the analysis method are still limited to linear bounded input programs. Later, Hoffmann et al. reduce the limitations by presenting an analysis which is capable of handling multivariate bounds [4]. Nonetheless, the input programs are still restricted to a first-order fragment of OCaml featuring integers, lists, binary trees, and recursion. This OCaml fragment is named Resource Aware ML (RaML).

In 2014 Hofmann and Moser converted the techniques of the univariate analysis to typed Term Rewrite Systems (typed TRS) resulting in polynomial bounds on the innermost runtime complexity of the given typed TRS [9], before lifting the analysis to gain multivariate bounds for typed TRS in [10]. The most important improvement to [12] and [4] is that arbitrary data types can be analyzed with the newly demonstrated methods.

This seminar report focuses on the transformation of the techniques developed for RaML, especially those presented in [12], such that they can be applied to TRS as explained in [9]. Furthermore, we describe the current implementation of the univariate analysis for typed TRS from [9] and point out differences to the implementation of the RaML prototypes as described in [3].

For the rest of the seminar paper we assume familiarity with the notations and analysis

methods presented in the publications [12] and [9], as well as basic knowledge of Term Rewrite Systems (TRS).

The rest of the seminar report is structured as follows. Section 2 explains the most important transformations and differences of the methods presented in [12] and [9]. Section 3 shows, by an example, the different input formats to the tools and explains the output of the implementations. Then, Section 4 summarizes the main concepts of the tool that implements the theory of [9]. Section 5 introduces some preliminary results to get a feeling for the execution times of the analyses. Next, Section 6 displays an outlook for the future, before we conclude in Section 7.

2 Theory

This section explains the transformation and improvements of the univariate linear bound analysis presented in [12] to gain the univariate polynomial bound analysis method for typed TRS introduced in [9]. Thus, this section heavily depends on content, and adapts concepts and ideas from [12] and [9].

2.1 Typing Rules

The heart of the analysis are the typing rules. This subsection is concerned with the evolution of the typing rules.

As a recap, the typing judgment $\Gamma \vdash^{p}_{q} t\colon A$ specifies that for all valuations ν the expression t has type $\nu(A)$ under the typing context Γ (which maps identifiers to types). Moreover, evaluating t under the context Γ requires at most a potential $p + \Phi(\sigma\colon\Gamma)$ and leaves at least a potential $q+\Phi(\sigma\colon\Gamma)$. Whereas, σ is a substitution, such that for all $x \in \mathsf{dom}(\Gamma)$ $x\sigma$ is of type Γ(x). Thus, the costs for evaluating the expression $t\colon A$ under context Γ is bounded from above by $p - q$.

As an example Figure 2.1 shows the typing rule for a function call as it was presented in [12]. The typing context Γ is the set $\{y_1\colon A_1, \ldots, y_k\colon A_k\}$, fid is the function call identifier, further the function is called with arguments $y_1, \ldots, y_k$. The type of the expression is C which already includes the annotations that specify the potential of this type. The constraints over resource variables specified in ψ must hold when the rule is applied. Further, the function signature $\Sigma(\mathsf{fid})$ must equal the quadruple containing the expression e_{fid} that needs to be proven well-typed, the variables $y_1, \ldots, y_k$, the signature of this function call, and the given constraints ψ. The application of the rule is restricted to functions with at least one argument. The costs for the function call are $p - p'$, then $Kcall(k)$ and $KCall'(k)$ represent the absolute costs of setting up before the call and clearing up after the call.

$$\frac{\Sigma(\mathit{fid}) = \left(e_{\mathit{fid}}; y_1, \ldots, y_k; \langle A_1, \ldots, A_k\rangle \xrightarrow[p']{p} C; \psi\right) \qquad k \geq 1}{y_1\colon\!A_1, \ldots, y_k\colon\!A_k \vdash^{p+\mathtt{Kcall}(k)}_{p'-\mathtt{Kcall}'(k)} \mathit{fid}\ \langle y_1, \cdots, y_k\rangle : C \mid \psi}$$

Figure 2.1: Typing rule for a function call from [12].

The typing rules of [12] can be transformed to be compatible with typed TRSs. Figure 2.2 shows the function application rule for typed TRSs. In this representation the constraints over the resource variables are left out, further every defined function symbol has only one allowed function signature. Thus, the signature map does not store

quadruples anymore, but annotated signatures. Further, the fixed costs $Kcall(k)$ and $Kcall'(k)$ have been dropped as for typed TRSs only evaluation steps are analyzed, but no memory structures.

$$\frac{\mathcal{F}(f) = A_1 \times \cdots \times A_n \xrightarrow[q]{p} C}{x_1\colon A_1, \ldots, x_n\colon A_n \left|\frac{p}{q}\right. f(x_1, \ldots, x_n)\colon C}$$

Figure 2.2: Tying rule for a function call transformed to typed TRSs.

This direct transformation was done for all of the typing rules. Gaining the typing system for a runtime analysis of linear bounded typed TRSs as given in Figure 2.3. This system differentiates between function calls of defined function symbols f and constructor symbols c. Furthermore, the CASE and LET rules were replaced by a function composition rule, which handles function calls to arbitrary terms.

$$\frac{}{x\colon A \left|\frac{0}{0}\right. x\colon A} \qquad \frac{\Gamma \left|\frac{p}{q}\right. t\colon C \quad p' \geqslant p \quad p' - p \geqslant q' - q}{\Gamma \left|\frac{p'}{q'}\right. t\colon C}$$

$$\frac{\Gamma \left|\frac{p}{q}\right. t\colon C}{\Gamma, x\colon A \left|\frac{p}{q}\right. t\colon C} \qquad \frac{\Gamma, x\colon A_1, y\colon A_2 \left|\frac{p}{q}\right. t[x, y]\colon C \quad \Upsilon(A \,|\, A_1, A_2)}{\Gamma, z\colon A \left|\frac{p}{q}\right. t[z, z]\colon C}$$

$$\frac{C = \mu X.\langle \cdots, c(B_1, \ldots, B_m)\colon k, \cdots \rangle \quad \forall i\colon A_i = B_i \vee (A_i = C \wedge B_i = X)}{x_1\colon A_1, \ldots, x_m\colon A_n \left|\frac{k}{0}\right. c(x_1, \ldots, x_m)\colon C}$$

$$\frac{\mathcal{F}(f) = A_1 \times \cdots \times A_n \xrightarrow[q]{p} C}{x_1\colon A_1, \ldots, x_n\colon A_n \left|\frac{p}{q}\right. f(x_1, \ldots, x_n)\colon C}$$

$$\frac{\Gamma_1 \left|\frac{p}{p_1}\right. t_1\colon A_1 \quad \cdots \quad \Gamma_n \left|\frac{p_{n-1}}{p_n}\right. t_n\colon A_n \quad x_1\colon A_1, \ldots, x_n\colon A_n \left|\frac{p_n}{q}\right. f(x_1, \ldots, x_n)\colon C}{\Gamma_1, \ldots, \Gamma_n \left|\frac{p}{q}\right. f(t_1, \ldots, t_n)\colon C} \quad \text{all } x_i \text{ are fresh}$$

$$\frac{\Gamma, x\colon B \left|\frac{p}{q}\right. t\colon C \quad A <\colon B}{\Gamma, x\colon A \left|\frac{p}{q}\right. t\colon C} \qquad \frac{\Gamma \left|\frac{p}{q}\right. t\colon D \quad D <\colon C}{\Gamma \left|\frac{p}{q}\right. t\colon C}$$

Figure 2.3: Type system for linear bounded typed TRSs.

In further developments the attention was put onto the analysis of runtime of typed TRSs. Therefore, the potential below the judge, that is the potential that is given back after an evaluation of the respective expression, was removed, as runtime cannot be passed back when once utilized. These improved typing rules are shown in Figure 2.4 and were presented in [9]. The obvious change is that the constructor rule was removed and merged with the function call rule. This reduces complexity and removes the need of explicit data constructors for TRSs. Thus, these rule might further be simplified by just replacing the types (which include the annotations) with the annotations themselves. This does not change the runtime complexity of the given TRS [1]. However, heuristics that may utilize these types, e.g. by detecting the recursive parts of functions cannot be used when the types are fully erased from the analysis. An example on how to perform this transformation is given in Figure 2.5, showing the function application rule without data types. Another important improvement of the types system is that the scalar annotations were replaced by vectors. The length of the vectors that are needed to type the system then represent the degree of the polynomial that is needed to bound the runtime of the input typed TRS, cf. Theorem 3.2 of [9].

$$\frac{f \in \mathcal{C} \cup \mathcal{D} \quad [A_1^{\vec{u_1}} \times \cdots \times A_n^{\vec{u_n}}] \xrightarrow{p} C^{\vec{v}} \in \mathcal{F}(f)}{x_1 \colon A_1^{\vec{u_1}}, \ldots, x_n \colon A_n^{\vec{u_n}} \mid\!\!\vdash^{p} f(x_1, \ldots, x_n) \colon C^{\vec{v}}} \qquad \frac{\Gamma \mid\!\!\vdash^{p} t \colon C \quad p' \geqslant p}{\Gamma \mid\!\!\vdash^{p'} t \colon C}$$

$$\frac{\begin{array}{ll} \text{all } x_i \text{ are fresh} & p = \sum_{i=0}^{n} p_i \\ x_1 \colon A_1, \ldots, x_n \colon A_n \mid\!\!\vdash^{p_0} f(x_1, \ldots, x_n) \colon C & \Gamma_1 \mid\!\!\vdash^{p_1} t_1 \colon A_1 \ \cdots \ \Gamma_n \mid\!\!\vdash^{p_n} t_n \colon A_n \end{array}}{\Gamma_1, \ldots, \Gamma_n \mid\!\!\vdash^{p} f(t_1, \ldots, t_n) \colon C}$$

$$\frac{\Gamma \mid\!\!\vdash^{p} t \colon C}{\Gamma, x \colon A \mid\!\!\vdash^{p} t \colon C} \qquad \frac{\Gamma, x \colon A_1, y \colon A_2 \mid\!\!\vdash^{p} t[x, y] \colon C \quad \Upsilon(A \,|\, A_1, A_2) \quad x, y \text{ are fresh}}{\Gamma, z \colon A \mid\!\!\vdash^{p} t[z, z] \colon C}$$

$$\frac{\Gamma, x \colon B \mid\!\!\vdash^{p} t \colon C \quad A <: B}{\Gamma, x \colon A \mid\!\!\vdash^{p} t \colon C} \qquad \frac{}{x \colon A \mid\!\!\vdash^{0} x \colon A} \qquad \frac{\Gamma \mid\!\!\vdash^{p} t \colon D \quad D <: C}{\Gamma \mid\!\!\vdash^{p} t \colon C}$$

Figure 2.4: Type system for univariate polynomial bounded TRSs [9].

LaTeX

$$\frac{f \in \mathcal{C} \cup \mathcal{D} \quad \vec{u_1} \times \cdots \times \vec{u_n} \xrightarrow{p} \vec{v} \in \mathcal{F}(f)}{x_1 \colon \vec{u_1}, \ldots, x_n \colon \vec{u_n} \mid\!\!\overset{p}{-} f(x_1, \ldots, x_n) \colon \vec{v}}$$

Figure 2.5: Further development of function call type rule by removing the need of types.

2.2 Well-Typedness

Clearly as the type system evolved also the specification for well-typedness changed. Figure 2.6 shows the well-typedness constraint as it was given in [12]. It means, if a signature as defined by Σ is used for the function fid, then there must be a derivation using the type system from above proving the well-typedness of this signature. Note the reversed signs for the calls to $Kcall(a)$ and $Kcall'(a)$ as opposed to the function application typing rule from before. Here, p and p' must be large enough to be able to evaluate the function call, which needs $Kcall(a)$ resources when executed. All functions defined in the input program have to be derived by the type system and must coincide with the well-typedness constraints.

$$\Sigma(\mathit{fid}) = (e_{\mathit{fid}}; y_1, \ldots, y_a; \langle A_1, \ldots, A_a \rangle \xrightarrow[p']{p} C; \psi) \qquad \Longrightarrow$$
$$y_1 \colon A_1, \ldots, y_a \colon A_a \vdash^{p - \mathtt{Kcall}(a)}_{p' + \mathtt{Kcall'}(a)} e_{\mathit{fid}} : C \mid \psi$$

Figure 2.6: Well-typedness as defined in [12].

These basic ideas were again transformed to fit for TRSs. Figure 2.7 shows the well-typedness constraints as presented in [9]. All defined function symbols of the input TRS must admit these constraints to ensure the system is well-typed, whereas for all rules $f(l_1, \ldots, l_n) \to r$ the variables of the left-hand side (lhs) are $\mathcal{V}\mathsf{ar}(f(l_1, \ldots, l_n)) = \{y_1, \ldots, y_l\}$. These constraints must be derivable for all signatures $[A_1 \times \cdots \times A_n] \xrightarrow{p} C \in \mathcal{F}(f)$, for all annotated types B_j ($j \in \{1, \ldots, l\}$), and costs k_i, such that $y_1 \colon B_1, \ldots, y_l \colon B_l \mid\!\!\overset{k_i}{-} l_i \colon A_i$ is derivable without using the relax rule, which could otherwise decrease the costs k_i. Here, the potentials of the constructors of the lhs are used to sum up the actual potential of the lhs of the rewrite rule. Further, the -1 corresponds to the $Kcall(a)$ in [12] and specifies the application of the given rewrite rule, gaining the right-hand side (rhs) r. Informally, the -1 specifies that one step in the derivation is happening, whenever the rewrite rule is applied. Thus, the potential must at least decrease by 1, leaving the rhs with less potential than the lhs.

$$y_1 \colon B_1, \ldots, y_l \colon B_l \mid\!\!\overset{p-1+\sum_{i=1}^{n} k_i}{-} r \colon C$$

Figure 2.7: These are the well-typedness constraints as presented in [9].

2.3 Potential Function

As keeping track of the potential (or credits as named in [12]) is the main concept of this amortized analysis the function was kept as it was. Figure 2.8 shows the main design of the function: The potential is recursively summed up for the given type with the corresponding annotated data type. It is easy to see that again the data types cannot bring an advantage for the analysis over simple vectors.

LaTeX

$$\Phi(t\colon C) = p + \Phi(t_1\colon A_1) + \cdots + \Phi(t_n\colon A_n).$$

Figure 2.8: The potential function recursively sums up the potential of an expression with the associated type. Here the term t is supposed to consist of a function (maybe without arguments) of the shape $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C} \cup \mathcal{D})$ and admits the signature $[A_1 \times \cdots \times A_n] \xrightarrow{p} C$.

To be able to correlate the length of the vectors used with the runtime of the given TRS Hofmann and Moser introduced Theorem 3.2 [9]. This Theorem bounds the growth of the potential of the constructors, such that the runtime complexity $\mathsf{rc}_{\mathcal{R}}(n) \in O(n^k)$, where $n = |v|$ and k is the vector length used. Thus, if this theorem is satisfied, the length of the vector specifies the degree of the polynomial needed to bound the runtime complexity of the input system from above [9].

2.4 Superposition and Uniqueness

Problems arouse by merging the constructor rule with the function rule. This merging would allow different annotations for the same constructor terms. That means, the same terms could have different potential values. This is obviously not desired at it would allow ambiguity of the potential. Thus, the so called superposition and uniqueness properties were introduced in [9].

The *uniqueness property* is defined over the set of annotated signatures for a constructor $f \in \mathcal{C}$. If f has the result type C, then for each annotation $C^{\boldsymbol{q}}$ there should exist exactly one declaration of the form $[A_1^{\boldsymbol{p_1}} \times \cdots \times A_n^{\boldsymbol{p_n}}] \xrightarrow{p} C^{\boldsymbol{q}}$ in $\mathcal{F}(f)$.[1]

The *superposition principle* states that if a constructor c admits the annotations $[A_1^{\boldsymbol{p_1}} \times \cdots \times A_n^{\boldsymbol{p_n}}] \xrightarrow{p} C^{\boldsymbol{q}}$ and $[A_1^{\boldsymbol{p'_1}} \times \cdots \times A_n^{\boldsymbol{p'_n}}] \xrightarrow{p'} C^{\boldsymbol{q'}}$ then it also has the annotations $[A_1^{\lambda \boldsymbol{p_1}} \times \cdots \times A_n^{\lambda \boldsymbol{p_n}}] \xrightarrow{\lambda p} C^{\lambda \boldsymbol{q}}$ with $\lambda \geqslant 0$ and $[A_1^{\boldsymbol{p_1}+\boldsymbol{p'_1}} \times \cdots \times A_n^{\boldsymbol{p_n}+\boldsymbol{p'_n}}] \xrightarrow{p+p'} C^{\boldsymbol{q}+\boldsymbol{q'}}$.

[1] Note that in [9] the uniqueness property is defined for functions $f \in \mathcal{C} \cup \mathcal{D}$. However, as uniqueness for functions $f \in \mathcal{D}$ is only needed for the polynomial interpretations, this definition is simplified by removing the set of defined function symbols $\mathcal{D}$ from the definition.

3 Example

This section shows the differences of the tools and input programs by an example. Listing 3 represents the reverse function in RaML code. The append function appends two lists together. This is done by recursively calling append on the tail of the first list. This implementation is used in real world code, for instance in Haskell, append (++) is implemented exactly like this[1]. The second function is reverse, which takes as input parameter a list and returns another list with the elements in opposite order. To reverse the given list it first calls itself on the tail of the input list, before calling append for each element of the input list once. As append is recursive on the same parameter as reverse is, the runtime complexity of this program can be bound at best by a 2^{nd}-degree polynomial. The web-interface of RaML[2], which is by now also capable of analyzing non-linear programs as well, cf. [4], gives for the append function an upper bound of $3.00 + 9.00 \cdot M$ where M is the number of ::-nodes of the 1^{st} component of the argument. Thus, the append function itself has linear runtime complexity. The function reverse returns an upper bound of $3.00 + 9.50 \cdot M + 4.50 \cdot M^2$. As expected reverse cannot be bounded by a linear function, as its runtime complexity growth in the order of a 2^{nd} degree polynomial[3].

Therefore, the method introduced in [12] is only capable of analyzing the append function. Only using the improvements for RaML from [4] the input program can be fully analyzed.

{language=,numbers=none,caption={The reverse function implemented in RaML.}}

```
1 let rec append xs ys =
2   match xs with
3     | [] → ys
4     | (x'::xs') → x'::(append xs' ys)
5
6 let rec reverse xs =
7   match xs with
8     | [] → []
9     | x'::xs' → append (reverse xs') (x'::[])
```

These functions can easily be converted to a typed TRSs. However, the list has to be defined as data type itself. Therefore, there is a recursive data type shown in Listing 3 named L that consist of a constructor nil for the empty list and a constructor cons that concatenates an element to a given list. The constructor cons coincides with the

[1] See `http://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html`.

[2] Available at `http://www.raml.co/`.

[3] The reverse function can also be implemented using an accumulator. This implementation can then be bound by a linear function. For such an implementation see, for instance `http://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.List.html`.

infix function :: of the RaML program. Then there is a second data type defined which specifies what kind of elements this list holds. As it is not important for the analysis what kind of data types the list holds, cf. the upper bounds depend only on the number of cons (i.e., ::), it has no constructors. The two functions do exactly the same as the ones specified in the RaML program: append merges two lists together and reverse takes a list and returns a new list with the same elements but in the opposite order. So the last element of the input list will become the first element in the output list, etc.

{language=,numbers=none,caption={The reverse function implemented as typed TRS.}}

```
(VAR x xs ys)

(DATATYPES
   Elem =      < >
   L    = μX. < nil , cons(Elem,X) >
 )

(SIGNATURES
   append  :: L x L → L
   reverse ::      L → L
 )

(RULES
  append(nil , ys)         → ys
  append(cons(x,xs), ys) → cons(x,append(xs, ys))

  reverse(nil)          → nil
  reverse(cons(x, xs)) → append(reverse(xs),cons(x, nil))
)
```

We have a running prototype[4] implemented in Haskell of the methods presented in [9]. When the program is run with the input typed TRSs from Listing 3 it prints as a solution the content shown in Listing 4.4. In this case the analysis was setup to look for a solution with vector length of 2. Therefore, the append function has also vectors with length 2, as the current prototype does not analyze the function one after the other. However, when only append is fed to the analyzer it finds a solution with vectors of length 1 also. Thus, append itself has linear runtime complexity, whereas reverse has quadratic polynomial runtime complexity. The polynomials have the same degrees as the RaML tool upper bound polynomials, however, the polynomial bounds given by the RaML implementation are closer to the real growth. The base constructors are used to check the uniqueness and superposition properties, cf. Section 2.4, how this is done will be explained in detail in the next Section.

LaTeX {language=,numbers=none,caption={The output of the prototype implementing the methods as presented in [9].}}

```
Solution:
---------
```

[4]The source code can be accessed and downloaded at `https://bitbucket.org/schnecki/amortized-cost-analysis/`

```
3
4    append :: [L(0, 15) x L(1, 15)] -(2)-> L(0, 15)
5    append :: [L(0, 15) x L(1, 15)] -(2)-> L(1, 15)
6
7    cons :: [Elem(14, 5) x L(13, 15)] -(13)-> L(0, 15)
8    cons :: [Elem(0, 1) x L(1, 0)]     -(2)->  L(1, 15)
9
10   nil :: [] -(0)-> L(0, 15)
11   nil :: [] -(0)-> L(7, 15)
12   nil :: [] -(0)-> L(7, 9)
13   nil :: [] -(0)-> L(14, 15)
14
15   reverse :: [L(0, 15)] -(5)-> L(0, 15)
16
17 Base Constructors:
18 ------------------
19   cons :: [Elem(0, 1) x L(1, 0)]     -(2)->  L(1, 15)
20   cons :: [Elem(14, 5) x L(13, 15)] -(13)-> L(0, 15)
21   nil :: [] -(0)-> L(0, 3)
22   nil :: [] -(0)-> L(7, 6)
```

4 Implementation

We have a prototype implementing the analysis methods from [9][1]. This section explains the details of the current implementation.

4.1 Parsing

First of all the system is parsed by a self-made extension of the *term-rewriting* library[2], called *term-rewriting-ext*[3]. It extends the *term-rewriting* library such that data types and function signatures can be parsed and are saved in corresponding data structures which were added for this purpose. As stated before, the information of data types are not crucial for the analysis itself [1], except for heuristics that may be integrated in the analysis. Therefore, at a later point the analysis implementation might be changed to use the *term-rewriting* library, if experiments show that the heuristics are not needed for a scaleable analysis with this method.

4.2 Well-Typedness Constraints

Each rule then is used once to generate the desired leafs of the proof derivation trees. So for example the rule $\mathsf{append}(\mathsf{nil}, \mathsf{ys}) \to \mathsf{ys}$ from Listing 3 will create $ys\colon L^{p(0,1)} \frac{k(0)-1+k(1)}{} ys\colon L^{r(0)}$, where the functions p, k and r can be thought of as references to a global table of signatures. In this example two signatures would be added to the global table. The functions p, k and r in the global table represent names of the vectors and will later be replaced by the actual vectors of the solution.

Idx	Signature		
0	append	$:: \mathrm{L}^{\mathrm{p}(0,0)} \times \mathrm{L}^{\mathrm{p}(0,1)}$	$\xrightarrow{k(0)} L^{\mathrm{r}(0)}$
1	nil	$:: []$	$\xrightarrow{k(1)} L^{\mathrm{r}(1)}$
	...		

The signatures of the global table are indexed starting with 0. The first parameter of the functions p, k and r are the index of the signature in the global table, and for the references of shape $p(\cdot,\cdot)$ the second parameter stands for index of the input parameter to the defined function symbol. Thus, all vector-names in the global signature table are unique. These names will be used to generate constraints. As stated above (see

[1] Available at `https://bitbucket.org/schnecki/amortized-cost-analysis`
[2] `https://github.com/haskell-rewriting/term-rewriting`
[3] `https://github.com/schnecki/term-rewriting-ext`

Section 2.2), for each constructor with costs k_i of the lhs $y_1\colon B_1, \ldots, y_l\colon B_l \mid\!\!\overset{k_i}{\vdash} l_i\colon A_i$ must be derivable without using the relax rule. Therefore, each constructor of the lhs will add one more proof leaf to the list of derivable clauses.

4.3 Type System

Using the inference rule of the type system from Figure 2.4 the proofs are generated using the rules bottom-up. First the *share* rule is applied if applicable. Then recursively the rules *function application*, *identity*, *composition*, and *weake* are applied. The rules *relax*, *supertype* and *subtype* are integrated in the other rules. Thus the generated constraints will often include $\geqslant$ -signs instead of $=$ -signs. The function application rule adds an entry to the global signature table, whereas the other inference rules do not. So for instance the rewrite rule $\mathsf{append}(\mathsf{nil}, \mathsf{ys}) \to \mathsf{ys}$ uses the identity inference rule and does not alter the global signature table:

LaTeX

$$\frac{}{ys\colon \mathsf{L}^{p(0,1)} \mid\!\!\overset{k(0)-1+k(1)}{\vdash} ys\colon \mathsf{L}^{r(0)}}$$

However, some constraints follow by applying the inference rules. In this case, according to the identity rule from the type system, the types from the lhs and the rhs of the judge have to be equal. When the *super-* and *subtype* rules are integrated, this yields constraint (1). The costs $k(0) - 1 + k(1)$ must be equal to 0. However, when the relax rule is integrated constraint (2) is gained. This is done for all clauses that need to be derived, coming up with a set of constraints and an extended global signature table.

LaTeX

$$p(0,1) \geqslant r(0) \tag{4.1}$$
$$k(0) - 1 + k(2) \geqslant 0 \tag{4.2}$$

4.4 Other Constraints

Several other constraints need to be added to ensure that the derivations are correct, superposition and uniqueness are satisfied, and the potential of the constructors does not grow to steep.

First for every signature of a defined function symbol in the global signature table that was generated from the well-typedness proof leaf generation, explained in Section 4.2, constraints are added such that all vectors are equal. In other words, the signatures of the same defined function symbol as root which are directly gained from the lhs of the rewrite rules must be equal. These signatures will in the following be called base signatures of the defined function symbols. Where Hoffmann generated a set of possible signatures [3, p. 93], we use simple $\geqslant$ -constraints to ensure that all signatures are compatible with

the corresponding base signatures (and thus their derivations), cf. Section 2.2. Hence, any occurrence in the rhs of a rewrite rule must have (a) parameter types with at least the same annotation as the base signatures do, (b) costs with at least the same as the base signatures do and (c) the return type must not be greater than the one of the base signatures. These constraints follow by applying the *super-* and *subtype* inference rules to the signatures gained by applying the type system.

To ensure superposition and uniqueness (cf., Section 2.4) so called base constructor signatures are added. For each constructor and every dimension of the vectors a base signature is added. So for instance, in the reverse example for the constructor cons for a maximum vector length of 2, the following base constructors were added to the constraint system:

LaTeX {language=,numbers=none,caption={Base signatures for cons-constructor of the reverse example.}}

```
1  cons_0 :: [Elem(0, 1) x L(1, 0)]     -(2)-> L(1, 15)
2  cons_1 :: [Elem(14, 5) x L(13, 15)] -(13)-> L(0, 15)
```

Each cons-signature in the global signature table must now be a linear combination of these two base signatures: $c_1 \cdot$cons_0$+c_2 \cdot$cons_1, where c_1, c_2 are arbitrary constants and cons_0 and cons_1 are the base signatures. This ensures that the superposition principle is satisfied. Further constraints are added, such that, whenever the rhs annotations of two constructor signatures of the same identifier are equal, then all other annotation must be equal as well. This ensures that the uniqueness property holds.

Finally, constraints are added to bound the growth of the potential of the constructor symbols, as it is defined in Theorem 3.2 in [9].

4.5 Solving

Just before the constraint problem is written to a file, the variables are lifted to vectors of the desired length. Currently the system is not able to use different vector lengths for different function signatures. So for example, if the maximum vector length is set to 2 the system will convert all constraints to vectors of size 2, e.g., $p(0,1) \geqslant r(0)$ become the constraints $p_0(0,1) \geqslant r_0(0) \wedge p_1(0,1) \geqslant r_1(0)$. These constraints are then written to a file and a SMT solver is called. In our case this is the SMT solver called *z3*[4] which is developed by Microsoft.

The result is parsed and the variables in the global signature table are replaced with the actual solution vectors. Finally, the solution is printed on the screen. It is possible to display the inference trees computed by the system[5].

[4] See `https://github.com/Z3Prover/z3/wiki`.

[5] See `-h` for the usage info.

5 Preliminary Experiments

This section shows first experimental results of the implementation of [9]. During the experiment executions a bug was detected: The TRS given in Listing 5, which calculates the value of 2^x where x is the input parameter, was typed with a vector length of 2, thus inferring quadratic runtime. However, the runtime complexity of this calculation cannot be bound by a polynomial. Therefore, the prototype in the current state is unsound. Thus, the execution times of the experiments have to be considered with caution and further should not be compared to similar analysis tools yet. Nonetheless, we expect to have similar execution times after the problem was located and fixed.

LaTeX {language=,numbers=none,caption={TRS calculating the 2^x, where x is the input paramter.}}

```
1  (VAR x)
2
3  (DATATYPES
4    Nat = μX.< 0, s(X) >
5  )
6
7  (SIGNATURES
8    d   :: Nat → Nat
9    exp :: Nat → Nat
10 )
11
12 (RULES
13   d(0)     → 0
14   d(s(x)) → s(s(d(x)))
15   exp(0)    → s(0)
16   exp(s(x)) → d(exp(x))
17 )
```

The program was run on a small set of test TRSs on a Laptop with a Intel® Mobile Core™ i7–3840QM Processor with 32 GB DDR3 RAM running ArchLinux (64-bit).

Table 5.1 shows problems which could be shown to have linear runtime complexity. The execution times are an average of 10 independent executions. Table 5.2 displays problems that have, according to the (unsound) tool, quadratic runtime complexity. In sum there are 15 solvable linear problems, 22 solvable quadratic problems, 14 problems were infeasible (the SAT solver returned 'unsat') and 22 problems ran into the timeout of 30 seconds. Further, the problems which were infeasible or ran into a timeout, could also not be solved when the program was asked for a cubic or even higher degree polynomial as a runtime complexity bound.

Adding to the obvious need of making the tool sound, the results also demonstrate the need for improvements in the sense of scaleability as 22 problems raised a timeout.

This shows clearly, that the tool needs to be optimized, to also work for more complex examples. All test TRSs are also available on the git-repository[1].

Example	t [ms]	RetVal
addition.trs	52.4	Success
append.trs	53.3	Success
id.trs	40.3	Success
list.trs	177.5	Success
minus.trs	38.8	Success
queue.trs	761.7	Success
quotient.trs	136.4	Success
reverse.trs	49.0	Success
typed-div.trs	151.9	Success
typed-flatten.trs	89.9	Success
typed-jones1.trs	60.1	Success
typed-jones2.trs	74.9	Success
typed-jones4.trs	97.3	Success
typed-jones6.trs	113.7	Success
typed-rationalPotential.raml.trs	386.4	Success

Table 5.1: Execution times of the analysis of the TRSs with linear runtime complexity. The execution time is displayed in milliseconds and is an average over 10 independent executions.

[1] See `https://bitbucket.org/schnecki/amortized-cost-analysis`.

Example	t [ms]	RetVal
exp.trs	320.7	Success
infeasible.trs	160.7	Success
mult3.trs	197.3	Success
reverse.trs	321.3	Success
typed-appendAll.raml.trs	723.7	Success
typed-bits.trs	1151.7	Success
typed-dcquad.trs	772.3	Success
typed-exp.trs	360.9	Success
typed-isort.trs	980.7	Success
typed-jones5.trs	277.6	Success
typed-lcs-safe.trs	1900.9	Success
typed-lcs.trs	1611.5	Success
typed-minsort.raml.trs	5114.5	Success
typed-pairs.trs	640.3	Success
typed-qbf.trs	2199.2	Success
typed-quad.trs	278.7	Success
typed-quicksort-nat.trs	3522.5	Success
typed-reverse.trs	315.8	Success
typed-sat.trs	2695.7	Success
typed-shuffleshuffle.trs	959.5	Success
typed-shuffle.trs	772.0	Success
typed-subtrees.raml.trs	495.4	Success

Table 5.2: Execution times of the analysis of the TRSs with quadratic runtime complexity. The execution time is displayed in milliseconds and is an average over 10 independent executions.

6 Future Work

First the detected bug needs to be fixed. Then for the next couple of weeks we will be running experiments to get an idea how stable the prototype is. Afterwards, still in the near future we will be implementing a graph analysis to identify strongly connected components of the input system. This will allow separation of analyses of the different functions of the input program. The gained constraint problems should be simpler and therefore faster solved by the SMT solver. This ought to increase the scaleability of the method. Further, we will try out heuristics, like the additive shift as presented in [9]. According to the results, either the heuristics will be integrated into the system, or the need for data types will be removed. Adding to this, once stable we plan to integrate the tool into TcT [2]. Further, it is planned to lift this univariate implementation to the multivariate analysis as presented in [10]. However, to this point unknown problems will most likely emerge by this lifting, and therefore this is planned for later.

7 Conclusion

This seminar report presented the evolution of amortized resource analysis techniques over the past decades with a special focus on the developments for TRS analyses. We showed how the ideas used in RaML were converted to be compatible with TRS and how these methods than were improved to be usable with generic data types and to gain an analysis of univariate polynomial bounds for (typed) TRSs. This includes the most important ideas, the type system, the potential function, and superposition and uniqueness concepts. Further, we have explained the current status of the prototype. The system is able to type programs which also admit non-linear runtime complexity. However the tool is not yet stable enough as it was not extensively tested until today. The implementation was explained and illustrated. Using well-typedness the desired leafs of the inference tree proofs are generated. This includes constructor proof derivations. Then the type system is used to create the proofs using the inference rules bottom-up. By doing so constraints are simultaneously generated and collected. A global signature table is used to be able to refer and uniquely identify each signature that is used during the proof creation. Several other constraints are added to ensure the correctness of the derivations. And finally a SMT solver is called to solve the constraint problem.

In the future the system will be tested, heuristics tried out, and improvements will be made, before the tool can be integrated into TcT, a runtime complexity analyzer built by the Computational Logic group at the University of Innsbruck[1]. We hope to be able to add value to TcT during the upcoming months and believe that runtime complexity analysis will be an important field of study for the future, for instance when supporting programmers at constructing efficient algorithms using functional programming languages.

[1] See `http://cl-informatik.uibk.ac.at`

Bibliography

[1] Martin Avanzini and Bertram Felgenhauer. Type introduction for runtime complexity analysis. In *14th International Workshop on Termination (WST 2014)*, page 1, 2014.

[2] Martin Avanzini, Georg Moser, and Michael Schaper. Tct: Tyrolean complexity tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–423. Springer, 2016.

[3] Jan Hoffmann. *Types with potential: Polynomial resource bounds via automatic amortized analysis*. epubli, 2011.

[4] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):14, 2012.

[5] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *International Conference on Computer Aided Verification*, pages 781–786. Springer, 2012.

[6] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming*, pages 165–179. Springer, 2000.

[7] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, volume 38, pages 185–197. ACM, 2003.

[8] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, pages 22–37. Springer Berlin Heidelberg, 2006.

[9] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *International Conference on Rewriting Techniques and Applications*, pages 272–286. Springer International Publishing, 2014.

[10] Martin Hofmann and Georg Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 241–256, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-87-3. doi: http://dx.doi.org/10.4230/LIPIcs.TLCA.2015.241. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5167`.

[11] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM SIGPLAN Notices*, volume 34, pages 132–146. ACM, 1999.

[12] Steffen Jost, Hans wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. Carbon credits for resource-bounded computations using amortised analysis. In *In Formal Methods (FM '09), LNCS 5850*, pages 354–369. Springer, 2009.

[13] Robert Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi: 10.1137/0606031. URL `http://epubs.siam.org/doi/abs/10.1137/0606031`.