# Uniform Resource Analysis by Rewriting: Strenghts and Weaknesses

Georg Moser

**Department of Computer Science, University of Innsbruck, Innsbruck, Austria**
georg.moser@uibk.ac.at

**Abstract**

In this talk, I'll describe how rewriting techniques can be successfully employed to built state-of-the-art automated resource analysis tools which favourably compare to other approaches. Furthermore I'll sketch the genesis of a uniform framework for resource analysis, emphasising success stories, without hiding intricate weaknesses. The talk ends with the discussion of open problems.

**1998 ACM Subject Classification** F.3.2 Semantics of Programming Languages, F.4.2 Grammars and Other Rewriting Systems

**Keywords and phrases** resource analysis, term rewriting, automation

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2017.2

**Category** Invited Talk

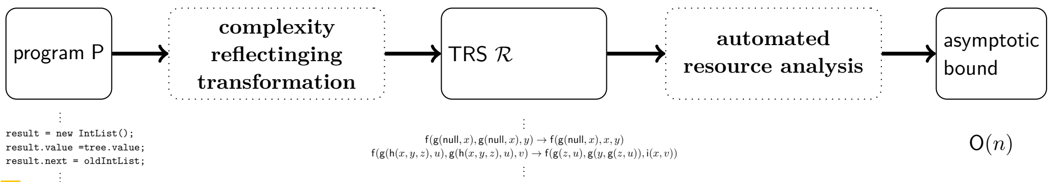
## 1 Prelude

In rewriting, complexity analysis traditionally played a minor role. Complexity analysis was mainly understood as *derivational complexity analysis*, ie. the analysis of the maximal derivation height of a given *first-order term rewrite system* (*TRS* for short), cf. [8, 51]. While deep results, emphasising the connection to proof theory, have been obtained in this direction by Hofbauer, Lautemann and Weiermann, these results were always conceived from the viewpoint of termination analysis, cf. [28, 27, 55]. I exemplarily mention Hofbauer's result that the multiset path order [27] implies primitive recursive derivational complexity, that is, from the proof of termination of TRS $\mathcal{R}$ we can gather the additional information that the length of any derivation over $\mathcal{R}$ starting in a term $t$ will be bounded by a primitive recursive function in the size of $t$. While in principle a result in runtime complexity analysis of TRSs, this result was proposed by Hofbauer as a classification result of termination analysis: the realm of multiset path orders is restricted to those TRSs of primitive recursive derivation length. See also [54, 16, 18, 53, 34, 36, 40, 35, 39] for further results and pointers into similar research.

Implicit computational complexity theory (ICC for short) provides machine independent characterisations of complexity classes, cf. [19]. While the mainstream in ICC was (and is) concerned with the delineation of suitable restrictions of linear logic, like *bounded linear logic* [24, 20] to characterise the class of polytime computable functions, Cichon and others, followed an approach more in line with related work in rewriting. Suppose program $\mathsf{P}$ is abstract and represented as a TRS. Suppose further termination of $\mathsf{P}$ follows by a suitable restricted ranking function, more precisely a *restricted polynomial interpretation*. Then in dependence on the precise restrictions enforced the *(worst-case) runtime complexity* of $\mathsf{P}$ will be bounded by a polynomial in the size of the input to $\mathsf{P}$. See for example [17, 11, 12, 13].

In conjunction, these traditions provide a wealth of techniques for the runtime complexity analysis of first-order rewrite systems. Due to the focus on automation of termination analysis

© Georg Moser;
licensed under Creative Commons License CC-BY


program P
complexity reflectinging transformation
TRS $\mathcal{R}$
automated resource analysis
asymptotic bound
$\mathsf{O}(n)$

```
⋮
result = new IntList();
result.value =tree.value;
result.next = oldIntList;
⋮
```

$$\mathsf{f}(\mathsf{g}(\mathsf{null},x),\mathsf{g}(\mathsf{null},x),y) \to \mathsf{f}(\mathsf{g}(\mathsf{null},x),x,y)$$
$$\mathsf{f}(\mathsf{g}(\mathsf{h}(x,y,z),u),\mathsf{g}(\mathsf{h}(x,y,z),u),v) \to \mathsf{f}(\mathsf{g}(z,u),\mathsf{g}(y,\mathsf{g}(z,u)),\mathsf{i}(x,v))$$

**Figure 1** Automated Resource Analysis via Transformations.

in rewriting it was easy to come up with fully automated methods for the derivational and runtime complexity analysis of first-order term rewrite systems, cf. [37, 5].

In this talk, rather than focusing on these techniques precisely, I want to focus on the bigger picture. That is, I will be concerned with application of these methods in the *resource analysis* of programs. Here I aim at a uniform analysis that allows to abstract away particular features of the given program P. In particular we will see ongoing success stories of such a general purpose analysis that can equally well handle *higher-order functional programs* as well as object-oriented bytecode like *Java Bytecode* (*JBC* for short). However, I also want to emphasise challenges and limitations of this approach in practise and discuss future work.

## 2 Transformation Based Runtime Complexity Analysis

Suppose P is an arbitrary given input program. The idea of resource analysis via *complexity reflecting* transformations is depicted in Figure 1. Here we show the process of transforming a JBC program P into a TRS $\mathcal{R}$, whose runtime complexity is subsequently analysed. Suppose the analysis yields a linear upper bound of the worst-case runtime complexity. Then, as the transformation is *complexity reflecting*, we reflect the analysis of the TRS to conclude $\mathsf{O}(n)$ as asymptotic bound on the runtime complexity of P. If the transformation is also *complexity preserving*, we can employ the same transformation based approach to analyse lower bounds, either for worst-case or best-case complexity. This approach has been successfully evaluated for *higher-order* pure `OCaml` programs, as well as for JBC programs, cf. [4, 38].

With respect to higher-order functional programs the transformation phase also provides a defunctionalisation of higher-order functions in order to represent the original program as an (applicative) term rewrite system. An example run is depicted in Figure 2. The induced whole program analysis, as implemented in our tool TCT [6], is comparable in strength to the RaML prototype [29, 30] and occasionally stronger. In particular our analysis is able to handle variable capture in closures properly. A noticeable difference between our analysis and RaML are the asymptotic bounds obtained by TCT, in comparison to the precise bounds by the RaML prototype. This, however is more a design choice, rather than a restriction. Asymptotic bounds allow for better and instant readability and usability, which is at the core of our concern: provide the working programmer with a tool that quickly and automatically provides resource bounds, which can be directly employed for the validation of design choices. Furthermore, it simplifies composability of the analysis.

With respect to imperative programs, our approach allows a more refined representation of the heap in contrast to tools like COSTA [1], CoFloCo [22], or AProVE [23]. This yields tight asymptotic bounds in some cases, which are outside of the realm of the competing methods. These often employ a path-length abstraction of the heap, which is practically more sensible in a variety of cases, but which falls short in programs that crucially rely on an analysis of the heap.

Still, the comparison of the transformational approach to methods rooted more classically in program analysis clarified a shortcoming of the approach. In Figure 1 a unique abstract

**(a)** Reversing a list, taken from Bird's textbook on functional programming [10].

```
let rec fold_left f acc = function
    [] -> acc
  | x::xs -> fold_left f (f acc x) xs ;;
let rev l = fold_left (fun xs x -> x::xs) [] l ;;
```

**(b)** Defunctionalised applicative rewrite system.

$$\begin{aligned}
\mathsf{main}(x_0) &\to \mathsf{m}_1(x_0) \mathbin{@} \mathsf{f} & \mathsf{r}(x_0) \mathbin{@} x_1 &\to x_0 \mathbin{@} \mathsf{r}_1 \mathbin{@} [\,] \mathbin{@} x_1\\
\mathsf{m}_1(x_0) \mathbin{@} x_1 &\to \mathsf{m}_2(x_0) \mathbin{@} \mathsf{r}(x_1) & \mathsf{r}_1 \mathbin{@} x_0 &\to \mathsf{r}_2(x_0)\\
\mathsf{m}_2(x_0) \mathbin{@} x_1 &\to x_1 \mathbin{@} x_0 & \mathsf{r}_2(x_0) \mathbin{@} x_1 &\to x_1 :: x_0\\
\mathsf{f} \mathbin{@} x_0 &\to \mathsf{f}_1 \mathbin{@} x_0 & \mathsf{f}_3(x_0, x_1) \mathbin{@} x_2 &\to \mathsf{f}_4(x_2, x_0, x_1)\\
\mathsf{f}_1 \mathbin{@} x_1 &\to \mathsf{f}_2(x_1) & \mathsf{f}_4([\,], x_0, x_1) &\to x_1\\
\mathsf{f}_2(x_1) \mathbin{@} x_2 &\to \mathsf{f}_3(x_1, x_2) & \mathsf{f}_4(x_0 :: x_1, x_2, x_3) &\to \mathsf{f} \mathbin{@} x_1 \mathbin{@} (x_2 \mathbin{@} x_3 \mathbin{@} x_0) \mathbin{@} x_2
\end{aligned}$$

**(c)** Simplified first-order term rewrite system.

$$\mathsf{main}(x_0) \to \mathsf{f}([\,], x_0) \qquad \mathsf{f}(x_0, [\,]) \to x_0 \qquad \mathsf{f}(x_0, x_1 :: x_2) \to \mathsf{f}(x_1 :: x_0, x_2)$$

**Figure 2** Example run of the HoCA prototype on a pure OCaml program.

```
public static void test(int n, int m){
  if (0 < n && n < m) {
    int j = n+1;
    while(j < n || j > n){
      if (j > m){
        j=0;
      } else {
        j=j+1;
      }
    }}}
```

**Figure 3** The Need for Disjunctive Bounds.

representation was conceived. Either TRSs or a straightforward extension thereof. However, in practise this turned out to be a weakness of the setup, which hinders a competitive resource analysis.

## 3 Uniform Resource Analysis by Rewriting

To clarify the weakness of assuming one specific form of intermediate language, we consider the Java program depicted in Figure 3. The examples is due to Gulwani et al. [25]. It is not difficult to see that the program is terminating. Given non-zero numbers $n$, $m$ as input, such that $n < m$, the while loop is executed as long as $j \neq n$ holds, that is, exactly $m$ times. While simple, the example is not trivial, for example, state-of-the-art tools like AProVE or COSTA cannot even show termination fully automatically. Furthermore the straightforward transformation of this program into a TRS, following the translation proposed in [21] is non-terminating.

In order to overcome this, a different flavour of rewrite systems need to be employed as abstract representations. More precisely, the crucial features of an imperative program can be better characterised by variants of transition systems [41, 45]. In particular, Figure 5 depicts

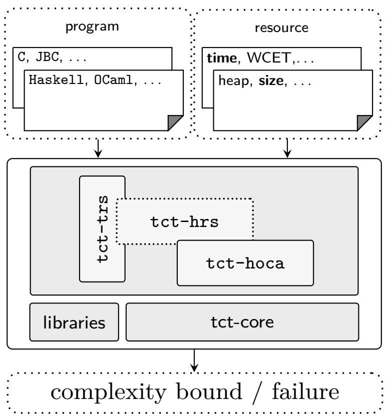


**Figure 4** Complexity Analyser T<sub>C</sub>T.

a suitable encoding of the example in Figure 3 into a so-called *integer transition system* (*ITS* for short). The transformation is complexity reflecting and the runtime complexity can thus be automatically assessed. Our implementation of the resource analysis of ITSs takes inspiration from [15]. However, while transition systems work well for imperative programs, the implicit size abstraction does not provide a suitable match for (higher-order) functional programs, like the list reversal algorithm shown in Figure 2.

We have cast these observations into a uniform resource analysis tool which is largely independent on the specifics of the programming language (and even programming paradigm) of the given input program P. The setup of our tool TCT is depicted in Figure 4. First, the input program in relation to the resource of interest is transformed to an abstract representation, generalising the transformational approach briefly described in Section 2. We refer to the result of applying such a transformation as *abstract program*. It has to be guaranteed that the employed transformations are *complexity reflecting* and if our interest are lower bound also *complexity preserving*. Note that the resource analysis deals with a general *resource analysis problem* that consists of a program together with the resource metric of interest as input. Second, we employ problem specific techniques to derive bounds on the given problem and finally, the result of the analysis, that is, a successful asymptotic resource analysis or the notice of failure, is relayed to the original program. TCT is open source, released under the BSD3 license. All components of TCT are written in *Haskell*. TCT is open with respect to the complexity problem under investigation and problem specific techniques. Moreover it provides an expressive problem independent strategy language that facilitates the proof search, extensibility and automation. The code of TCT, at least for the analysis of term rewrite systems is in part certified, that is, the analysis is guaranteed to provide a correct upper bound, see for example. [7]. A more detailed account of the implementation of TCT and the underlying abstract resource analysis framework giving rise to an uniform resource analysis indicated above can be found in [6].

I want to emphasise that TCT does not make use of a unique abstract representation, but is designed to employ a variety of different representations. Moreover, different representations may interact with each other. For now we make use of ITSs and various forms of rewrite systems, not necessarily first-order TRSs. Currently, we are in the process of developing dedicated techniques for the analysis of *higher-order rewrite systems* (*HRSs* for short) that once should become another abstraction subject to resource analysis (depicted as `tct-hrs` in the figure). To exemplify this, the actual strategy code to analyse object-oriented bytecode

```
start(m,n,j) -> while(m,n,n+1) :|: m > n && n > 0
while(m,n,j) -> while(m,n,0)   :|: m > n && n > 0 && j > n && j > m
while(m,n,j) -> while(m,n,j+1) :|: m > n && n > 0 && j > n && j <= m
while(m,n,j) -> while(m,n,j+1) :|: m > n && n > 0 && j < n && j <= m
```

**Figure 5** Integer Transition System.

```
jbc :: Strategy ITS () → Strategy TRS () → Strategy JBC ()
jbc its trs = toCTRS ⋙ Race (toIts ⋙ its) (toTrs ⋙ trs)
```

**Figure 6** `Java` Transformation Pipeline Modelled in `tct-jbc`.

programs is shown in Figure 6. Our transformation from JBC employs a term abstraction of the heap which gives rise to so-called *constraint term rewrite systems* (*cTRS*). However, in the actual implementation of a resource analysis for cTRS are handled by either conceiving this abstract representation as TRS or as ITS. Consequently we run the correponding method in parallel, as indicated by the keyword `Race` in the code.

The interaction in the resource analysis of these different abstract programs has not yet reached its full potential. But is envisioned in the future to work also on the level of program slices. Certain parts of a program P dealing for example with user-defined datatypes are best abstracted as terms. Other parts are more naturally abstracted in size, like an increasing counter. Based on these different abstractions, different forms of resource analysis are most effective. Combining the obtained different results effectively and in a precise manner is one of our current research agendas. This setup improves *modularity* of the approach and provides scalability and precision of the overall analysis.

Our current work mainly aims at improvements of the backend of $\mathsf{T_CT}$, that is, the improvement of the existing resource analysis for the provided abstract representation as well as the incorporation of effective analysis of a particular new representation, the above mentioned HRSs. With respect to TRSs we have recently finalised the incorporation of an amortised analysis for worst-case bounds on the runtime complexity (see also Section 4 below). Furthermore we are in the process of incorporating an amortised analysis providing lower bounds on the best-case complexity of TRSs. Moreover, we are actively developing a new backend for the resource analysis of ITSs. Here the emphasis is on scalability of the methods.

Still, all is not well in the realm of uniform resource analysis tools. Apart from the future work on a better integration of various forms of abstract representations, we noticed in our quest for the incorporation of amortised analysis that we cannot easily mimic simple methods in this generic setting. In particular we have implemented a simple heuristics of the RaML prototype [30] to yield a univariate amortised analysis. While it was not difficult to extend the sophisticated work of Hoffmann et al. from functional programs to TRSs (see [32, 33]) it is less straightforward to obtain the same analysis strength in practise in our uniform setting. I detail the observations from this case study in the next section.

## 4 Case Study: Amortised Resource Analysis

For clarity, I briefly sketch the potential method of amortised analysis as proposed by Sleator and Tarjan in [48, 50]. The motivation for this analysis technique were self-balancing data structures that sometimes need to perform costly operations that pay off later on, e.g. rebalancing operations on a search tree.

In brief, one assigns to the participating datastructures a nonnegative real-value, the *potential* in an a priori arbitrary fashion. One then defines the amortised cost of an operation as its actual cost, for example the runtime, plus the difference in potential of all datastructures before and after the operations. In this way, the amortised cost of a costly operation may be small if it results in a big decrease of potential. On the other hand some cheap operations that increase the potential will be overcharged. In this way, one can “save money” now to pay for costly operations later. By a simple telescoping argument the sum of all amortised costs in a sequence of operations plus the potential of the initial input data structure is also an upper bound on the *actual* cost of that sequence. Essentially by reversing the orders, the method can be used equally well for lower bound analysis of best-case complexities.

Thus amortised analysis may yield rigorous bounds on actual resource usage and not just approximate or average bounds. If the potential functions are chosen well then the amortised costs of operations are either constant or exhibit merely a very simple dependency on the maximum size of all intermediate results. A simple classical example is the implementation of a queue by two stacks, an in-tray and an out-tray. Incoming elements are added to the in-tray, outgoing elements are taken from the top of the out-tray. Only if the out-tray becomes empty the entire in-tray is reverse-copied into the out-tray. In this case, the length of the in-tray is clearly a suitable potential function. The costly operation of copying can entirely be paid from the big decrease in potential it causes.

For automated resource analysis, amortised cost analysis has been in particular pioneered by Hoffmann et al., whose RaML prototype has grown into a highly sophisticated analysis tool for (higher-order) functional programs. In a similar spirit, resource analysis tools for imperative programs like COSTA [2], CoFloCo [22] and LOOPUS [47] have integrated amortised reasoning. See also work by Atkey [3].

Let me sketch the implementation of the univariate amortised analysis as proposed in [32]. Naturally the choice of the correct potential functions is usually non-trivial. Following an approach proposed already in [31] we select for each datatype a suitable annotation to define the potential functions. Say these annotations are vectors of natural numbers. For a value $v$ its potential $\Phi(v)$ can then be estimated as a polynomial in the size of $v$, where the degree of the polynomial depends on the length of the resource annotations used. In automation the precise annotations are unknown and a symbolic amortised cost analysis is employed to fix these annotations. For this one makes use of a simple heuristics [31] taking into account the type signatures of the program considered. The heuristics guarantees that the constraints remain linear, which allows a very efficient analysis, cf. [30].

In the setup of $\mathsf{T_CT}$ such an analysis is performed on the abstract representation of the original (first-order) RaML programs. A natural and direct abstraction are TRSs. TRSs are blind on types and we loose in this transformation the seemingly neglectable type information in the input RaML program. Indeed the analysis presented in [32] does not depend on the type information and in [33] we naturally emphasised this generalisation as a step forward. However, without this type information the heuristics doesn’t perform well, see Figure 7.

The table presents the results of the amortised resource analysis compared to the analysis using the heuristics discussed above on an independent testbed comprised of a collection of direct encodings of functional programs as well as automatic translations thereof. Ara denotes our standard implementation of amortised analysis in $\mathsf{T_CT}$, which employs non-linear constraints encoded as SMT problems so that either MiniSMT (M) or Z3 (Z) can be employed. On the other hand H indicates the use of heuristics, where the TRSs were manually typed to have comprehensive type information required. As we see the heuristics is significantly faster, but shows significantly poorer behaviour in power. Note that this heuristics works perfectly well within the RaML prototype.

Arguably this is a bug or feature of $\mathsf{T_CT}$. One can correctly argue that we should not strip information from the input language, if we observe that this information is crucial for

| | Number of TRSs | | | | Execution Time (in seconds) | | | |
|---|---|---|---|---|---|---|---|---|
| Result | Ara M | Ara Z | Ara MH | Ara ZH | Ara M | Ara Z | Ara MH | Ara ZH |
| MAYBE | 15 | 14 | 30 | 31 | 13.39 | 10.27 | 4.65 | 6.37 |
| $O(n^1)$ | 17 | 17 | 12 | 12 | 0.29 | 0.31 | 0.23 | 0.20 |
| $O(n^2)$ | 9 | 9 | 4 | 4 | 5.89 | 6.40 | 0.55 | 0.51 |
| $O(n^3)$ | 1 | 1 | 1 | 1 | 1.37 | 1.58 | 0.63 | 0.53 |
| Timeout | 16 | 17 | 11 | 10 | 60.01 | 60.05 | 60.02 | 60.06 |
| Sum/Avg | 27 | 27 | 17 | 17 | 21.04 | 21.19 | 13.88 | 13.85 |

**Figure 7** Experimental Evidence.

```
while (B_s x > b_s) ∧ (B_w x ⩾ b_w)
{
  x' = Ax + c
}
```

**Figure 8** Simple Loop Program.

the effectivity of the resource analysis. However, by doing so we effectively instantiate RaML programs as yet another abstract representation thereby completely loosing the uniformity of our tool. As typed TRS are not standard we cannot employ the method for the analysis of TRS proper. In my understanding this is not a sensible approach but it clarifies that a uniform solver like TCT may easily become too powerful to be of any use, if not engineered well.

Similar cases can be made for imperative programs and the comparison to cost equations as abstract representations. I believe that such seemingly limitations of the here proposed methodology may act as stepping stone to a more intense coupling of different techniques in one solver so that to achieve the best of all worlds in resource analysis.

## 5 Open Problems

In the following I want to mention two open problems directly related to the effective resource analysis of programs, whose solutions appears to require some thought.

Our ongoing quest for automated resource analysis of term rewrite systems are essentially fuelled by the idea to automatically perform amortised resource analysis as proposed in standard textbooks [43]. All automated methods provided so far in the literature are restricted to functions with *constant* amortised costs [30], while a convincing argument can be made for the automated analysis of algorithms with *logarithmic* amortised costs. In particular a number of classical examples of amortised analysis for self-balancing datastructures have logarithmic amortised costs. It is currently very unclear how to tackle this problem, raised as a challenge by Nipkow last year, cf. [42].

Yet another area of related study are *simple loop programs*. Simple loop programs are typically represented as matrix programs of the following form depicted in Figure 8. Here $B_s\mathbf{x} > \mathbf{b_s}$ and $B_w\mathbf{x} \geqslant \mathbf{b_w}$ represent conjunctions of linear strict or weak inequalities over the state variables $\mathbf{x}$. Furthermore $A\mathbf{x} + \mathbf{c}$ represents an affine update of each state variable. Execution of the instruction in the loop body is in parallel, which is denoted as usual by the use of primed state variables. Simple loop programs have also be called *linear while loop programs* in the literature. If vectors $\mathbf{b_s}, \mathbf{b_w}$ and $\mathbf{c}$ are zero vectors then the program is called *homogeneous*, otherwise it is *inhomogeneous*.

State variables are either interpreted over the reals, the rationals, or over the integer and depending on the domain or further restrictions on the programs, termination is known to be decidable. More precisely over $\mathbb{R}$ and $\mathbb{Q}$ termination is decidable in polynomial time [52, 14]. For homogenous programs over $\mathbb{N}$ these results imply decidability. In the case that the update matrix A is diagonalisable, that is, if it is similar to a diagonal matrix, then termination for integer loop programs is decidable, cf. [44]. However the general case for integer loop programs is (wide) open and in practise are typically subject to ranking functions [46, 9].

Building upon these decidability results we focus on a resource analysis of simple loop programs. In particular we study runtime and size complexity of simple loop programs over $\mathbb{Q}$. Here runtime complexity measures the maximal number of symbolic executions, that is, the number of loops in relation to the value of the input. To date no a priori runtime complexity analysis of simple loop programs is known, although for homogenous programs over $\mathbb{N}$ we have decidability of termination. Due to the strong links of the problem to linear recurrence sequences in general and the Skolem Problem [26, 49] no simple solutions seem forthcoming.

## References

1 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *TCS*, 413(1):142–159, 2012.

2 E. Albert, S. Genaim, and A. N. Masud. On the inference of resource usage upper and lower bounds. *TOCL*, 14(3):22, 2013.

3 R. Atkey. Amortised Resource Analysis with Separation Logic. *LMCS*, 7(2), 2011.

4 M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. 20th ICFP*, pages 152–164. ACM, 2015.

5 M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.

6 M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. 22nd TACAS*, LNCS, pages 407–423, 2016.

7 M. Avanzini, C. Sternagel, and R. Thiemann. Certification of complexity proofs using ceta. In *Proc. 26th RTA*, volume 36 of *LIPIcs*, pages 23–39, 2015.

8 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

9 A. Ben-Amram and S. Genaim. Ranking functions for linear-constraint loops. *JACM*, 61(4):26:1–26:55, 2014. `doi:10.1145/2629488`.

10 R. Bird. *Introduction to Functional Programming using Haskell, Second Edition.* Prentice Hall, 1998.

11 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretations. In *Proc. 7th CSL*, volume 1584 of *LNCS*, pages 372–384, 1998.

12 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *JFP*, 11(1):33–53, 2001.

13 G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations a way to control resources. *TCS*, 412(25):2776–2796, 2011.

14 M. Braverman. Termination of integer linear programs. In *Proc. 18th CAV*, volume 4144 of *LNCS*, pages 372–385, 2006.

15 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. 20th TACAS*, volume 8413 of *LNCS*, pages 140–155, 2014.

16 W. Buchholz. Proof-theoretical analysis of termination proofs. *APAL*, 75:57–65, 1995.

17 E.-A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *Proc. 11th CADE*, volume 607 of *LNCS*, pages 139–147, 1992.

18 E.-A. Cichon and A. Weiermann. Term rewriting theory for the primitive recursive functions. *APAL*, 83(3):199–223, 1997.

19 U. Dal Lago. A short introduction to implicit computational complexity. In *Lectures on Logic and Computation – ESSLLI 2010*, volume 7388 of *LNCS*, pages 89–109, 2011.

20 U. Dal Lago and M. Hofmann. Bounded linear logic, revisited. *LMCS*, 6(4), 2010.

21 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. 22nd CADE*, volume 5663 of *LNCS*, pages 277–293, 2009.

22 A. Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *Proc. 21st FM*, volume 9995 of *LNCS*, 2016. doi:10.1007/978-3-319-48989-6_16.

23 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with aprove. *JAR*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

24 J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *TCS*, 97(1):1–66, 1992.

25 S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proc. PLDI'10*, pages 292–304. ACM, 2010.

26 V. Halava, T. Harju, M. Hirvensalo, and J. Karhumäki. *Skolem's Problem: On the Border Between Decidability and Undecidability.* TUCS technical report. Turku Centre for Computer Science, 2005.

27 D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *TCS*, 105:129–140, 1992.

28 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. 3rd RTA*, volume 355 of *LNCS*, pages 167–177, 1989.

29 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14, 2012.

30 J. Hoffmann, A. Das, and S. Weng. Towards automatic resource bound analysis for OCaml. In *Proc. 44th POPL*, pages 359–373. ACM, 2017.

31 J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *Proc. 19th ESOP*, volume 6012 of *LNCS*, pages 287–306, 2010.

32 M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Proc. of Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 272–286, 2014.

33 M. Hofmann and G. Moser. Multivariate amortised resource analysis for term rewrite systems. In *Proc. 13th TLCA*, volume 38 of *LIPIcs*, pages 241–256, 2015. doi:10.4230/LIPIcs.TLCA.2015.241.

34 I. Lepper. Derivation lengths and order types of Knuth-Bendix orders. *TCS*, 269:433–450, 2001.

35 A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, and H. Zankl. Joint spectral radius theory for automated complexity analysis of rewrite systems. In *Proc. 4th CAI*, volume 6742 of *LNCS*, pages 1–20, 2011.

36 G. Moser. The Hydra battle and Cichon's principle. *AAECC*, 20(2):133–158, 2009. doi:10.1007/s00200-009-0094-4.

37 G. Moser. Proof theory at work: Complexity analysis of term rewrite systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

38 G. Moser and M. Schaper. A complexity preserving transformation from Jinja bytecode to rewrite systems. *IC*, 2017. To appear.

**39** G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *LMCS*, 7(3), 2011.

**40** G. Moser and A. Schnabl. Termination proofs in the dependency pair framework may induce multiple recursive derivational complexity. In *Proc. 21st RTA*, volume 10 of *LIPIcs*, pages 235–250, 2011.

**41** F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 2005.

**42** T. Nipkow. Verified analysis of functional data structures. In *1st FSCD*, pages 4:1–4:2, 2016. `doi:10.4230/LIPIcs.FSCD.2016.4`.

**43** C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

**44** J. Ouaknine, J. S. Pinto, and J. Worrell. On termination of integer linear loops. In *Proc. 26th SODA*, pages 957–969. SIAM, 2015.

**45** A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. *TOPLAS*, 29(3), 2007.

**46** A. Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. 5th VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004.

**47** M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. Software Engineering*, volume 252 of *LNI*, pages 101–102, 2016.

**48** D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proc. of the 15th STOC*, pages 235–245. ACM, 1983. `doi:10.1145/800061.808752`.

**49** T. Tao. *Structure and randomness: pages from year one of a mathematical blog.* AMS, 2008.

**50** R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth*, 6(2):306–318, 1985.

**51** TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracks in Theoretical Computer Science*. Cambridge University Press, 2003.

**52** A. Tiwari. Termination of linear programs. In *Proc. 16th CAV*, volume 3114 of *LNCS*, pages 70–82, 2004.

**53** H. Touzet. Encoding the Hydra battle as a rewrite system. In *Proc. 23rd MFCS*, volume 1450 of *LNCS*, pages 267–276, 1998.

**54** A. Weiermann. Complexity bounds for some finite forms of Kruskal's theorem. *JSC*, 18(5):463–488, November 1994.

**55** A. Weiermann. Termination proofs for term rewriting systems with lexicographic path ordering imply multiply recursive derivation lengths. *TCS*, 139:355–362, 1995.