Uniform Resource Analysis by Rewriting: Strengths 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 build 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, i.e. 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 $R$ we can gather the additional information that the length of any derivation over $R$ 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 $P$ is abstract and represented as a TRS. Suppose further termination of $P$ 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 $P$ will be bounded by a polynomial in the size of the input to $P$. 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...
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 \( R \), 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 \( O(n) \) 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], CoFlCo [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].

```ocaml
let rec fold_left f acc =
  | [] → 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.

```
main(x0) → m1(x0) @ f
m1(x0) @ x1 → m2(x0) @ r(x1)
m2(x0) @ x1 → x1 @ x0
f @ x0 → f1 @ x0
f1 @ x1 → f2(x1)
f2(x1) @ x2 → f3(x1,x2)
f3(x1,x2) @ x2 → f4(x2,x0,x1)
f4(x0 :: x1,x2,x3) @ x2 → f @ x1 @ (x2 @ x3 @ x0) @ x2
```

(c) Simplified first-order term rewrite system.

```
main(x0) → f([],x0)          f(x0,[]) → x0          f(x0,x1 :: x2) → f(x1 :: x0,x2)
```

---

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

**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 example 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...
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 $\text{TCT}$ 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. $\text{TCT}$ is open source, released under the BSD3 license. All components of $\text{TCT}$ are written in Haskell. $\text{TCT}$ 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 $\text{TCT}$, 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 $\text{TCT}$ and the underlying abstract resource analysis framework giving rise to a uniform resource analysis indicated above can be found in [6].

I want to emphasise that $\text{TCT}$ 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 $\text{tct-hrs}$ in the figure). To exemplify this, the actual strategy code to analyse object-oriented bytecode

![Figure 4 Complexity Analyser $\text{TCT}$](image_url)
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 $\text{TCT}$, 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 \( T_\mathcal{C} \) 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. \( \text{Ara} \)
denotes our standard implementation of amortised analysis in \( T_\mathcal{C} \), which employs non-linear
constraints encoded as SMT problems so that either MiniSMT (\( M \)) or Z3 (\( Z \)) can be employed.
On the other hand \( \text{H} \) 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 \( T_\mathcal{C} \). One can correctly argue that we should not
strip information from the input language, if we observe that this information is crucial for
<table>
<thead>
<tr>
<th>Result</th>
<th>Number of TRSs</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(n¹)</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>O(n²)</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>O(n³)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Timeout</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 7 Experimental Evidence.

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 \( B_s x > b_s \) and \( B_w x \geq b_w \) represent conjunctions of linear strict or weak inequalities over the state variables \( x \). Furthermore \( Ax + c \) 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 \( b_s, b_w \) and \( c \) 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

2:10  Uniform Resource Analysis by Rewriting: Strenghs and Weaknesses


