Analyzing Sorting Algorithms in Resource Aware ML

Jan Hoffmann

Ludwig-Maximilians-Universität München
jan.hoffmann@ifi.lmu.de

Abstract. Software development sometimes requires to statically predict the quantity of resources—such as memory and time—that is needed to execute a program. The difficulties of manual resource analysis led to extensive research on automatic methods for quantitative resource analysis.
Recently we developed an automatic amortized analysis to compute polynomial resource bounds for first-order functional programs at compile time. Its basis is a type system that augments types with resource annotations. The analysis system is integrated in the programming language Resource Aware ML. Our experiments with a prototype implementation show that the analysis efficiently computes precise time and heap-space bounds for many example programs.
In this paper I demonstrate how a user can employ Resource Aware ML to analyze the worst-case time behavior of the sorting algorithms quick sort, insertion sort and merge sort. To illustrate pros and cons, I compare our automatic analysis to a manual analysis of the algorithms in a standard textbook. The paper is divided into three sections. Section 1 motivates the research on static resource analysis. Section 2 briefly introduces Resource Aware ML. Finally, Section 3 contains the actual case study on sorting algorithms.

Key words: Quantitative Analysis, Functional Programming, Resource Consumption, Amortized Analysis

1 Static Resource Analysis

The determination of the quantitative resource behavior of an algorithm is a classic problem of computer science. It is often referred to as the *analysis of algorithms* and elaborately discussed in many textbooks such as *The Art of Computer Programming* [1]. Quantities that are subject to the analyses include execution time and memory usage but also particular properties such as the amount of data passed through a channel.

Quantitative analysis of algorithms is a non-trivial problem. Often, software engineers are not only interested in the asymptotic behavior of an algorithm but rather in an exact determination of the resource costs of a concrete implementation. In fact, this aspect is present in *The Art of Computer Programming* where Knuth implements algorithms in an assembly language for the MIX architecture

to determine their exact use of clock cycles and memory cells. Such concrete bounds can be employed in various ways in software development.

Most notably, they can be used to determine the hardware requirements of embedded systems and to ensure the safety of hard real-time systems. In the former, one wants to use hardware that is *just good enough* to accomplish a task in order to produce a large number of units at lowest possible cost. In the latter, one needs to guarantee specific worst-case running times to ensure the safety of the system.

Even for basic programs, a manual analysis of the specific (non-asymptotic) costs is very cumbersome. Not everyone commands the mathematical ease of Knuth and even he would run out of steam if he had to do these calculations over and over again while going through the debugging loops of program development. In short, derivation of precise bounds by hand appears to be unfeasible in practice in all but the simplest cases.

As a result, *automatic methods for static resource analysis* are highly desirable and have been the subject of extensive research. Of course, one can not expect the full automation of a manual analysis that involves creativity and sophisticated mathematics. But in most resource analyses the greater part of the complexity arises from the glut of detail and the program size rather than from conceptual difficulty.

The state of the art in resource analysis research relies on various techniques of program analysis. The field of worst-case execution time (WCET) is mainly focused on the analysis of code with given inputs and deals especially with architectural features such as caches and instruction pipelines [2]. Complementary, there are methods to derive bounds on the number of loop iterations and recursive calls. For instance, the COSTA project has made recent progress in the automation of the classic approach of deriving and solving recurrence relations that describe the program behavior [3]. Another approach is to cleverly annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation [4]. The technique we developed is called *automatic amortized resource analysis*. It is described more detailed in the following sections.

2 Resource Aware ML

Recently, we developed the functional programming language *Resource Aware ML (RAML)* [5–7]. It implements a new type-based resource analysis system that automatically computes *polynomial bounds* on the (worst-case) resource behavior of (first-order) programs. The analysis works without any program annotations and is fully automatic if a maximal degree of the polynomials is given. Our system is parametric in the resource and can compute bounds for every quantity that can be associated with an atomic step of the computation. This includes clock cycles, heap space, and stack space.

The technique we use is an automation of the potential method of amortized analysis which was initially introduced by Sleator and Tarjan [8] to manually

analyze the efficiency of data structures. The automation of amortized analysis was pioneered by Hofmann and Jost [9] to infer *linear bounds* on the heap-space consumption of functional programs by integrating it into a type system. This linear system has been successfully used to compute memory and clock-cycle bounds for 32 MHz Renesas M32C/85U embedded micro-controllers using HUME, a functional language for real-time embedded systems [10].

Since the problem of deciding whether a given program admits a polynomial resource bound is undecidable in general there exist programs with polynomial bounds for which our analysis unsuccessfully terminates. However, an experimental evaluation showed that our method can compute time and space bounds for many interesting programs that appear in practice.

Compared to other approaches our system seems to better deal with recursion and nested inductive data structures. It is for example the only one that can fully automatically analyze functions such as quick sort. A prototype implementation along with the examples is available online[1]. It is easy to use, adequately documented, and can be run directly in a web browser.

3 Case Study: Sorting Algorithms in RAML

A classic way to demonstrate quantitative resource analysis is to analyze the runtime behavior of sorting algorithms. In the book *The Art of Computer Programming* [1], Knuth manually determines worst-case bounds for many well-known sorting algorithms that are implemented in an assembly language for the MIX architecture. Among the analyzed algorithms are quick sort, which uses at most $2n^2+37n+3$ MIX cycles, insertion sort, at most $9\binom{n}{2}+7n-6 = 4.5n^2+2.5n-6$ MIX cycles, and merge sort, roughly $10n\log n+4.92n$ MIX cycles[2] (n is the size of the input).

As a result of a careful and elaborate analysis, the bounds are tight in the sense that they exactly match the actual worst-case behavior of the functions.

To give you an impression of such an analysis, I included Knuths implementation of insertion sort below.

```
START ENT1  2-N              1            S1. Loop on j. j ←2.
2H    LDA   INPUT+N,1        N-1          S2. Set up i, K, R.
      ENT2  N-1,1            N-1          i ← j-1.
3H    CMPA  INPUT,2          B+N-1-A      S3. Compare K : Ki.
      JGE   5F               B+N-1-A      To S5 if K ≥ Ki.
4H    LDX   INPUT,2          B            S4. Move Ri, decrease i.
      STX   INPUT+1,2        B            Ri+1 ←Ri.
      DEC2  1                B            i ← i-1.
      J2P   3B               B            To S4 if i > 0.
5H    STA   INPUT+1,2        N-1          S5. R into Ri+1.
      INC1  1                N-1
      J1NP  2B               N-1          2 ≤ j ≤ N.
```

[1] http://raml.tcs.ifi.lmu.de

[2] The actual worst-case bound is more complicated and presented in a form that is only meaningful in combination with the source code.

The locations `INPUT+1` through `INPUT+N` are the array to be sorted. The first column contains the MIX program and the third column contains comments. Please refer to *The Art of Computer Programming* [1] for a detailed description of the program and the MIX architecture.

In the second column you find the number of times each instruction is executed, where N is the size of the input, A is the number of times i decreases to zero in step S4, and B is the number of moves. The running time of the program on the MIX machine is $9B + 10N - 3A - 9$ units. A thorough analysis shows that $A = N - 1$ and $B = \frac{N^2-N}{2}$ in the worst-case.

In the remainder of this section we implement the three sorting algorithms in RAML to automatically determine a bound on the number of evaluation steps they use. We then compare our automatic analysis with the manual analysis of Knuth.

Insertion Sort Below is the implementation of insertion sort in RAML. The same implementation may also be given in a textbook.

```
insert(x,l) = match l with | nil → [x]
              | (y::ys) → if y < x then y::insert(x,ys) else x::y::ys;

isort l = match l with | nil → nil
           | (x::xs) → insert (x,isort xs);
```

If we instantiate our analysis system with the evaluation-step metric then the prototype implementation automatically computes the following output.

```
insert: (int,L(int)) → L(int)
(*,0) → 5.0, (*,1) → 12.0

The number of evaluation steps consumed by insert is at most:
         12.0*n + 5.0
where
   n is the length of the second component of the input

-----------------------------------------------------------------------

isort: L(int) → L(int)
0 → 3.0, 1 → 12.0, 2 → 12.0

The number of evaluation steps consumed by isort is at most:
         6.0*n^2 + 6.0*n + 3.0
where
   n is the length of the input
```

For each function there is a line that states a usual ML-like type which states the shape of the arguments and the result. Below that type you find a mapping from type annotations to rational numbers which are computed by our analysis system (see [5] for details). This mapping is then transformed into a user-friendly presentation of the bound.

Quick Sort Quick sort can also be implemented in RAML in the usual way.

```
append(l,ys) = match l with | nil → ys | (x::xs) → x::append(xs,ys);

split(p,l) = match l with | nil → (nil,nil)
               | (x::xs) → let (ls,rs) = split (p,xs) in
                           if x > p then (ls,x::rs) else (x::ls,rs);

qsort l = match l with | nil → nil
           | (x::xs) → let (ls,rs) = split (x,xs) in
                       append(qsort ls, x::(qsort rs));
```

With the evaluation-step metric, the analysis automatically infers that *qsort* uses at most $12n^2 + 14n + 3$ evaluation steps if n is the length of the input list. As the computed bounds indicate, insertion sort indeed admits a better worst-case behavior than quick sort. The reason is that there is an (expansive) call of *append* at each recursive call of *qsort*. Below is a tail-recursive version of quick sort that does not use *append*.

```
q_aux(l,acc) = match l with | nil → acc
             | (x::xs) → let (ls,rs) = split (x,xs) in
                         let acc' = x::q_aux(rs,acc) in q_aux(ls,acc');

qsort2 l = q_aux(l,[]);
```

The computed bound for *qsort2* is $8n^2 + 18n + 7$ (n is the length of the input). It improves the bound of *qsort* in the quadratic part. The reduced potential in the second position of the type annotation of the argument corresponds directly to the costs for the calls of *append*. However, insertion sort has still a slightly better bound.

Merge Sort The last function we implement is merge sort.

```
msplit l = match l with | nil → (nil,nil)
            | (x1::xs) → match xs with | nil → ([x1],nil)
                         | (x2::xs') → let (l1,l2) = msplit xs' in
                                       (x1::l1,x2::l2);

merge (l1,l2) = match l1 with | nil → l2
                | (x::xs) → match l2 with | nil → (x::xs)
                            | (y::ys) → if x<y then x::merge(xs,y::ys)
                                        else y::merge(x::xs,ys);
msort l = match l with | nil → nil
           | (x1::xs) → match xs with | nil → l
                        | (x2::xs') → let (l1,l2) = msplit l in
                                      merge (msort l1, msort l2);
```

The evaluation-step bound for *msort* is $36.66n^2 - 29.33n + 3$ where n is the length of the input. Our system can only compute polynomial bounds. Thus it can not express an asymptotically tight $O(n \log n)$ bound for *msort*.

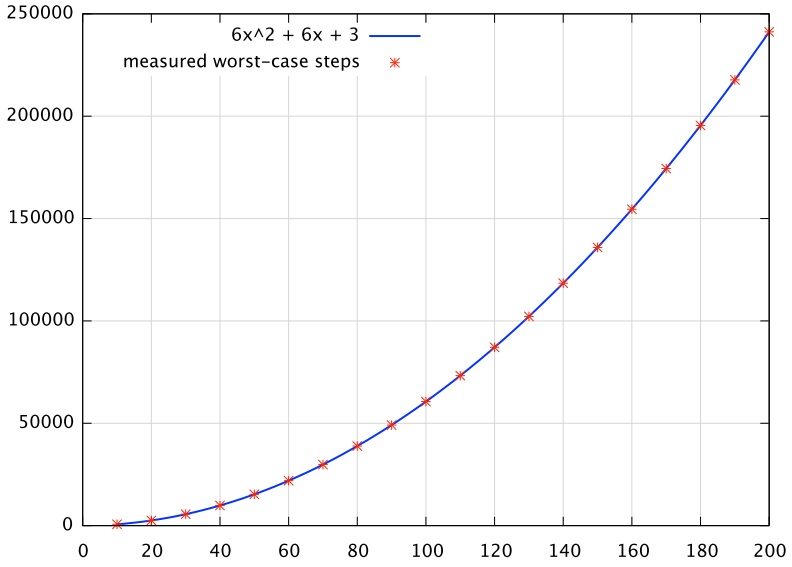

Fig. 1. The computed evaluation-step bound (blue line) compared to the actual worst-case number of evaluation steps for sample inputs of various sizes (red crosses) used by *isort*.

RAML vs. Knuth 1. Speed. Our prototype runs less then a second on a Macbook Pro with a 2.16 GHz Core 2 Duo to compute the bounds for all the above functions. Even though Knuth is known for his high productivity, such a performance seems to be hardly manually achievable. **2. Quality.** For sorting algorithms it is easy to identify inputs for which the worst-case run time behavior emerges. So we measured the actual worst-case behavior of the algorithms for several input sizes and compared it to the inferred bounds. Figure 1 shows the results of these experiments for *isort*. The measured worst-case behavior of the function matches exactly our computed bound. In fact, RAML computes tight evaluation-step bounds for *isort*, *qsort*, and *qsort2*. An automatic analysis can of course not always achieve the same accuracy as a careful manual analysis. In the case of RAML, we can only deal with polynomial bounds. Since the actual worst-case behavior of merge sort is $O(n \log n)$, the inferred quadratic bound is loose. **3. Scalability.** There is still a large number of polynomially bounded programs that cannot be analyzed in our system. An example is the sorting algorithm bubble sort. However, our experiments with the prototype indicate that the analysis scales well for larger programs such as the multiplication of a list of matrices with fitting (but arbitrary many) dimensions. It takes only a few seconds on standard desktop computers to analyze functional programs with several hundred lines of code. On the other hand, a manual analysis is nearly impossible for large programs that are written in a high-level language. **4. Practicability.** First, a manual analysis of assembly code is tedious and

error-prone. Secondly, it is time intensive and expensive. Thirdly, it has to be repeated after every change in the program which can lead to subtle errors due to false assumptions about the nature of the change. In contrast, the RAML analysis is proved to be sound and available at the touch of a button every time the program has changed.

References

1. Knuth, D.E.: The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
2. Wilhelm, R., et al.: The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. ACM Trans. Embedded Comput. Syst. **7**(3) (2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07). (2007) 157–172
4. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
5. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symp. on Principles of Prog. Langs. (POPL'11). (2011)
6. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: 8th Asian Symp. on Prog. Langs. (APLAS'10). (2010)
7. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010) 287–306
8. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Algebraic Discrete Methods **6**(2) (1985) 306–318
9. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). (2003) 185–197
10. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In: 16th Symp. on Form. Meth. (FM'09). (2009) 354–369