Resource Aware ML

Jan Hoffmann[1], Klaus Aehlig[2], and Martin Hofmann[2]

[1] Yale University
[2] Ludwig-Maximilians-Universität München

Abstract. The automatic determination of the quantitative resource consumption of programs is a classic research topic which has many applications in software development. Recently, we developed a novel multivariate amortized resource analysis that automatically computes polynomial resource bounds for first-order functional programs.

In this tool paper, we describe Resource Aware ML (RAML), a functional programming language that implements our analysis. Other than in earlier articles, we focus on the practical aspects of the implementation. We describe the syntax of RAML, the code transformation prior to the analysis, the web interface, the output of the analysis, and the results of our experiments with the analysis of example programs.

Keywords: Functional Programming, Static Analysis, Resource Consumption, Quantitative Analysis, Amortized Analysis.

1 Introduction

A quantitative analysis of a program determines the amount of resources, such as memory and time, that the program consumes during its evaluation. Quantitative analyses are needed to compare different algorithms for the same task, to design efficient programs, and to identify performance bottlenecks in software.

Sometimes, it is sufficient to determine the *asymptotic* resource behavior of a program. However, many applications in embedded systems, hard real-time systems, and cloud computing require *concrete* (non-asymptotic) upper bounds for specific hardware. The manual determination of such bounds is not only cumbersome and time consuming but also prone to errors, especially if the analysis has to be repeated after an iteration of the development cycle. As a result, mechanical assistance for the determination of resource bounds is an important and active area of research.

Classic methods for obtaining bounds on the number of loop iterations and recursive calls are based on automatically extracting and solving recurrence relations [1,2,3]. However, both, extracting and solving recurrence relations is a difficult problem. As a result, alternative techniques for the inference of resource bounds have been studied recently. Gulwani et al. propose counter instrumentation and abstract-interpretation–based invariant generation to obtain bounds on loop iterations and function calls [4]. To obtain loop bounds from disjunctive invariants one can use size-change abstraction [5] or proof rules that employ

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 781–786, 2012.

SMT-solvers [6]. Type-based techniques for automatically inferring bounds on recursive functions are based on sized types [7,8] or amortized analysis [9,10], and often restricted to *linear* bounds.

We have recently developed the first type-based resource analysis system that automatically computes *polynomial* resource bounds [11,12,13]. It is inspired by automatic amortized analysis for linear bounds [9]. In a nutshell, we annotate function types with a priori unknown, non-negative rational numbers that represent coefficients of *multivariate resource polynomials*, a class of functions that generalizes non-negative linear combinations of binomial coefficients.[1] A syntax-directed static type analysis then derives linear inequalities for the unknown rational coefficients. Finally, a solution of the resulting linear program with an off-the-shelf LP solver yields a resource polynomial that bounds the resource consumption of the corresponding function. Such an automatic amortized analysis is favorable in the presence of (nested or intermediate) data structures and function composition. See [13] for a detailed comparison with related approaches.

We implemented our multivariate amortized resource analysis in Resource Aware ML (RAML), a first-order, functional language with an ML-like syntax. While one can formalize algorithms and functional programs directly in RAML, it can also be used as a target of resource-preserving translations from other programming languages. In particular we have experimented with a translation from C using the Frama-C framework[2].

In this tool paper, we describe the current state of development of RAML from a user's point of view. For a description of the analysis technique that we implemented, please refer to our previous papers [11,12,13,14]. Note that the prototype implementation has been used in a previous paper for an experimental evaluation [13]. However, we have never demonstrated the tool at a conference.

2 The Prototype Implementation

The prototype implementation of RAML is written in Haskell and consists of a parser (546 lines of code), a standard type checker (490 lines of code), an interpreter (333 lines of code), an LP solver interface (301 lines of code), and the multivariate analysis system [13] (1637 lines of code). Overall, we needed 4.5 man-months for the implementation of the analysis.

The implementation is well documented and publicly available. The source code of the latest RAML version can be downloaded on the web site of the project [15]. Additionally, there is a web form that can be used to evaluate RAML programs and to compute resource bounds directly on the web.

Extended Syntax. The RAML syntax in the prototype extends the syntax described in our previous papers [11,13]. For example, expressions are not restricted to let normal from. We also have more built-in operators and allow a destructive

[1] The user has to provide a maximal degree of the polynomials to limit the number of unknown coefficients.

[2] http://frama-c.com

pattern matching *matchD* that deallocates the memory cell associated with the matched node of the data structure.

Data types τ are binary trees ($T(\tau)$), lists ($L(\tau)$), integers, Booleans, units, and tuples as defined by the following grammar.

$$\tau ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid (\tau_1, \ldots, \tau_n) \mid L(\tau) \mid T(\tau)$$

The following EBNF grammar defines expressions e. The reserved function *tick* is used in the *tick metric* which is described later. The argument q of *tick* denotes a floating point literal. The operations *binop* and *unop* are the usual standard operations for integers and Booleans.

$$\begin{aligned} e ::= {} & () \mid \mathit{True} \mid \mathit{False} \mid n \mid x \mid \mathit{tick}(q) \mid e_1\ \mathit{binop}\ e_2 \mid \mathit{unop}\ e \mid f(e_1, \ldots, e_n) \\ & \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{if}\ e\ \mathit{then}\ e_t\ \mathit{else}\ e_f \mid [] \mid [e_1, \ldots, e_n] \mid (e_1, \ldots, e_n) \\ & \mid \mathit{match}\ e_1\ \mathit{with}\ (x_1, \ldots, x_n) \to e_2 \mid \mathit{let}\ (x_1, \ldots, x_n) = e_1\ \mathit{in}\ e_2 \\ & \mid \mathit{nil} \mid \mathit{cons}(e_h, e_t) \mid (\mathit{match} \mid \mathit{matchD})\ e\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \\ & \mid \mathit{leaf} \mid \mathit{node}(e_0, e_1, e_2) \mid (\mathit{match} \mid \mathit{matchD})\ e\ \mathit{with} \mid \mathit{leaf} \to e_1 \mid \mathit{node}(x_0, x_1, x_2) \to e_2 \end{aligned}$$

A RAML program consists of a (possibly empty) list of declarations followed by a main expression. A declaration is either a type declaration $f : \tau_1 \to \tau_2$ or a function definition $f(x_1, \ldots, x_n) = e$, where f is a function name, τ_i are data types, x_i are variables, and e is an expression. There must be exactly one type declaration for every function definition. For every identifier, at most one type declaration and at most one function definition is allowed. Note that one has to provide a monomorphic type for every function in a program. The reason why we avoid polymorphic functions is that the resource consumption of a function depends on its type. Alternatively, we could allow polymorphic functions and analyze a function for each concrete type it is used with in the program.

Destructive Pattern Match. A destructive pattern match—written using *matchD*—can be used to deallocate memory cells. For instance, in the evaluation of the expression $\mathit{matchD}\ x\ \mathit{with} \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x, xs) \to e_2$ the memory cell that is referenced in the variable x is deallocated. If memory cells are allocated during the evaluation of e_2 then the deallocated cell may be used to store a new value. So if a deallocated value is accessed during the evaluation of an expression then the behavior of the program is undefined. If used carefully, destructive pattern matches can help to develop and analyze programs that use memory very efficiently. A typical example is an in-place quick-sort algorithm which destructs the input list [15].

Transformation to Let Normal Form. To simplify the resource analysis, we transform the unrestricted RAML expressions of the prototype implementation into expressions in let normal form as defined in [13]. An expression is in let normal form if, whenever possible, term formers are applied to variables only. Furthermore, we make sharing of variables explicit to enable the use of a syntax-directed type rule for sharing of potential in the type inference.

The transformation to let normal from uses a special form of a let expression—called *freelet*—that does not consume any resources. For every expression that

occurs in a position where only variables are allowed, we introduce a new variable with a *freelet*. For technical reasons we also introduce a new variable if the expression in such a *variable only position* in the source program is a variable itself. In this way, it becomes easy to preserve the resource cost of the source program because we know that all variables in the *variable only positions* have been introduced by a *freelet*.

To make sharing explicit, we add an additional syntactic construct to the expression each time a variable occurs multiple times. If a free variable x occurs twice in an expression e, we replace the first occurrence of x with x_1 and the second occurrence of x with x_2, obtaining a new expression e'. We then replace e with $\mathit{share}(x, x_1, x_2)$ *in* e'. In this way, the sharing rule becomes a conventional syntax directed rule in the type inference.

Resource Metrics. Our analysis is parametric in the resource and can deal with every quantity whose consumption in an atomic evaluation step is bounded by a constant. We included three resource metrics in the prototype and it is easy to define more by instantiating the resource constants for the evaluation steps.

The first included metric is the evaluation-step metric that counts the number of evaluation steps in the big-step operational semantics described in [13].

The second metric we included is the heap-space metric. The heap-space used by a node of a data structure depends on the type of the elements of the data structure. That is why we allow the resource constants to depend on the types of the respective expressions in the prototype. For instance, we do not simply have K^{cons} which defines the resource usage of a *cons* but rather $K^{\mathrm{cons}}(A)$ where A is the type of the elements of the list. We define

$$\mathrm{size}(A) = \begin{cases} n & \text{if } A = (A_1, \ldots, A_n) \\ 1 & \text{otherwise} \end{cases}$$

Then $K^{\mathrm{cons}}(A) = \mathrm{size}(A)+1$ is the number of memory cells that are used to store a node of a list of type $L(A)$. Similarly, $K_1^{\mathrm{matCD}}(A) = \mathrm{size}(A) + 1$ memory cells become available in a destructive pattern match. Since the types $L(A)$ are known at compile time, it makes no difference for the analysis whether the constants depend on data types. In principle, the values of these constants could depend on anything that is statically known about the program. However, the current implementation limits this dependency to type information.

The third implemented metric measures the number of ticks that occur in an evaluation. To this end, a programmer can insert expressions such as *tick(3.5)* or *tick(−4)* into the code. Every time the expression *tick(q)* is evaluated, q resources are consumed, or $-q$ resources become available if q is negative. The tick metric can be used to manually model specific resource metrics and is helpful for testing.

A table with the values of the constants in the metrics can be found in [14].

Web Interface. The source code of the prototype is available for download on the RAML website [15]. Alternatively, programs can be executed directly on the web with input in a text field or selection of example files from a drop-down menu. A second text field contains the output of the RAML prototype.

One can use the web interface to compute resource bounds for a program or to evaluate the main expression. The following options are available.

1. The resource metric to be used in the analysis. It can either be heap-space consumption, evaluation steps or ticks.
2. An upper bound on the maximal degree that can occur in the resource bounds. If the degree is too low then the analysis reports that the linear program is infeasible.
3. Whether to have verbose output. The verbose output shows for instance the function definitions in let normal form.

Output of the Analysis. The result of a successful evaluation is the value of the main expression as well as the number of heap cells, the number of evaluation steps, and the number of ticks that have been used during the evaluation.

The output of a resource analysis is either a list of symbolic bounds along with refined typing information—one for each function in the program including the main expression—or an error message. If the program is type correct then the only error that can occur is the message *the linear program is infeasible*. It indicates that the LP solver finished unsuccessfully and that RAML was thus not able to compute a bound for the program. This often, but not necessarily, implies that the resource usage of the given program cannot be bounded by a polynomial of the given degree.

Of course, as with any static analysis, there also exist polynomially bounded programs for which RAML cannot compute bounds. For instance, the analysis often fails if recursion is guarded by a Boolean function as opposed to the constructors of a data structure. This is often the case in programs whose resource consumption depends on the values of integers. Nevertheless, the analysis works well for recursive functions that use inductive data types and pattern matching.

Below is the output of the analysis with the evaluation-step metric for the function *quicksort* in the file *quicksort.raml* which can be found online [15].

```
> raml analyse eval-steps 3 quicksort.raml
quicksort: L(int) -> L(int)
Positive annotations of the argument   Positive annotations of the result
0 -> 3.0    1 -> 26.0    2 -> 24.0

The number of evaluation steps consumed by quicksort is at most:
         12.0*n^2 + 14.0*n + 3.0
where n is the length of the input
```

It contains the type of the function and the potential annotations of the argument type and the result type. Finally, the potential annotations are converted into a usual polynomial for the convenience of the user. This transformation is a combination of a change of basis from binomial coefficients to the common basis and the abstraction from sizes of individual inner data structures to their maximal size. The exact meaning of the type annotations is described in our earlier work [13]. Note, however, that they may carry more detailed information then the symbolic bound. Also note that only non-zero annotations are shown in the output and that the resource annotations of the output type are all zero.

3 Experiments

We successfully applied the analysis to a wide range of examples from functional programming such as sorting algorithms, matrix multiplication, breadth-first search, and longest common subsequence via dynamic programming.

In most cases, the derived evaluation-step and heap-space bounds were asymptotically tight. The analysis works efficiently and only needs a few seconds, even on larger programs. We also compared our computed bounds with the measured worst-case resource consumption of the programs and found that the constants factors are often close or even identical to the optimal ones.

The analyzed programs, tables with running times and computed bounds, and plots that show the bounds and the measured costs are available online [15] and in the first author's dissertation [14].

References

1. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM 18(9), 528–539 (1975)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning, 161–203 (2011)
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 2009), pp. 127–139 (2009)
5. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound Analysis of Imperative Programs with the Size-Change Abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)
6. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: Conf. on Prog. Lang. Design and Impl. (PLDI 2010), pp. 292–304 (2010)
7. Chin, W.-N., Khoo, S.-C.: Calculating Sized Types. High.-Ord. and Symb. Comp. 14(2-3), 261–300 (2001)
8. Vasconcelos, P.: Space Cost Analysis Using Sized Types. PhD thesis, School of Computer Science, University of St Andrews (2008)
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 2003), pp. 185–197 (2003)
10. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
11. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
12. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 172–187. Springer, Heidelberg (2010)
13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symp. on Principles of Prog. Langs. (POPL 2011) (2011)
14. Hoffmann, J.: Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis. PhD thesis, Ludwig-Maximilians-Universiät München (2011)
15. Hoffmann, J., et al.: RAML Web Site, `http://raml.tcs.ifi.lmu.de`