Automated Micro-analysis of Haskell

Adam Crume
University of California, Santa Cruz
adamcrume@soe.ucsc.edu

March 13, 2010

Abstract

A tool was created to statically analyze very simple Haskell programs and report their predicted runtime in an arbitrary unit. Analysis is limited to programs whose runtime can be expressed exactly as a polynomial.

1 Introduction

An important question in computer science is "How fast is this program?" One can, of course, run the program to find out, but that may not always be feasible. Instead, static analysis may be performed to yield theoretical runtimes in terms of input size. Manual analysis is tedious and error-prone, so automated analysis would be preferable when possible. Run time analysis in the general case reduces to the halting problem and is therefore not solvable, but a large subset of useful programs can still be analyzed.

2 Related Work

Andrew Shewmaker worked on a project performing micro-analysis of C[3]. His approach was to generate formulas that rely on probabilities. He mentions that solving recurrence equations ("recursive time formulas") would be a better approach, which is exactly how the new tool works.

3 Assumptions

To make the problem tractable, a number of assumptions have been made.

1. Integer arithmetic takes constant time. For the Int type in Haskell, this should be true. Note that for the Integer type, arithmetic takes $\log n$ time, although it should be relatively constant over the range supported by Int.

2. Function arguments are integers or lists of integers. If an argument is a list, run time depends only on the length of the list.

3. Function run times can be expressed exactly as a polynomial with rational coefficients.

4. Inputs to the function being analyzed have already been completely evaluated.

5. The result of the function being analyzed is completely evaluated, and time to do so is included in the run time.

6. Run times are meaningful only if the program would halt for the given input.

7. The compiler is generally non-optimizing. Specifically:

   (a) Common subexpression elimination is not performed.

(b) Function applications are not memoized, so evaluating `f 0` twice takes twice as long as evaluating it once.

(c) Function inlining is not performed.

4 Implementation

The tool itself was written in Haskell and uses GHC for parsing and type checking. Code for dealing with equations was partially ported from a Java library previously written by the author.

Recurrence equations are generated from the source code being analyzed. The solutions to the recurrence equations are assumed to be polynomials which evaluate to the total number of function applications that occur when running the code with a given input. The polynomials are substituted for the recurrence functions, a linear system is extracted with the polynomial coefficients as variables, and the system is solved.

Since Haskell uses curried functions, each argument counts as a function application. `f x y` is really `(f x) y`, so it has two function applications. These are counted in the function definition instead of when it is applied so that time formulas for base cases (e.g. $T_{fac}(0)$) yield non-zero values. Arithmetic operators such as "+" are assumed to use three function applications, one per argument and one for the actual addition.

For example, consider a factorial function:

```
fac 0 = 1
fac x = x * fac (x - 1)
```

The recurrence equations would be:

$$T_{fac}(0) = 1$$
$$T_{fac}(x) = 7 + T_{fac}(x-1)$$

Assuming $T_{fac}(x) = ax^2 + bx + c$, we get

$$a0^2 + b0 + c = 1$$
$$ax^2 + bx + c = 7 + a(x-1)^2 + b(x-1) + c$$

which simplifies to

$$c = 1$$
$$0 = 7 - 2ax + a - b$$

The linear system extracted from this is:

$$c = 1$$
$$0 = -2a$$
$$0 = 7 + a - b$$

Solving this yields $a = 0$, $b = 7$, and $c = 1$, so the final solution is $T_{fac}(x) = 7x + 1$. Note that if the assumed polynomial has a higher degree than the time function's actual polynomial, it still works correctly. The higher coefficients will simply be zero.

5 Results

The linear system extracted from the code may be under-constrained. This may be caused by lack of a base case in a recursive function. In this case, the tool returns an error.

The linear system may be over-constrained. This may be caused by:

- The time function may not be a polynomial
- The time function may have multiple cases
- The time function's polynomial may have a higher degree than the polynomial assumed

In each of these cases, the tool returns an error. However, the third case is relatively easy to fix. A polynomial of higher degree may be assumed and the analysis re-run. The disadvantage to this is that it requires more memory and processor usage.

Lazy evaluation in Haskell complicates analysis because, to handle it properly, one must keep track of what has been evaluated. However, the lack of side effects eases analysis because values may be reasoned about equationally without a time variable. It also means that the analysis should be easier to prove correct in a multithreaded environment, since other threads cannot modify variables.

Generating test cases is slightly harder than originally assumed. It seems that in Haskell, single-integer-argument functions that do not call other functions cannot have greater than linear complexity. It is also surprisingly easy to create functions whose runtime is not an exact polynomial (such as $\lfloor \frac{x}{2} \rfloor + 1$).

The task of converting Java code to Haskell is quite difficult. Moving from an imperative language to a functional one is much harder than, say, moving from Java to Python. Only a portion of the equation library was ported due to the time involved. Thankfully, the code was sufficiently functional that it did not need to be entirely rewritten.

Time tests show that the expected times are accurate (within a constant factor) for functions which make use of tail call recursion (see figs. 1 and 2). However, a recursive function which does not use tail recursive calls has an unusual time function (see fig. 3). Its constant factor was roughly an order of magnitude larger than the others, suggesting a lot of overhead in creating and evaluating “thunks.” (Note that the constant factor was adjusted independently for each function to best-fit the time function against the data. Constant factors are not a focal point of this project; that is why they are not estimated from the code.)

6 Future Work

The tool could easily be extended to support more syntax. Support could also be added for more types. Asymptotic times (big-O notation) could be provided in many cases even if exact formulas cannot be derived. For example, if the time is known to be $ax^2 + bx + c$ where $a = 1$ but b and c are undetermined, the time is still known to be $O(x^2)$. It should be possible to solve recurrence equations that involve polynomial-like functions, but it would probably be much more difficult. The converted equation library is quite slow and needs much optimization.

The tool currently uses the TypecheckedModule[2], but it should be changed to use the DesugaredModule. There was not enough time to figure out how to use the desugared source. Type parameters are represented as arguments to functions (including, for example, the “+” operator). Integer literals are represented as constructors called with unboxed integers. Desugared source has fewer expression types and a more regular structure, but it is much less intuitive.

A practical use of this tool would be to verify expected function runtimes. For example, a sorting function could be annotated[1] with:

```
{-# ANN sort RunTime
     "O[T[n]] == Length[n] ^ 2" #-}
```

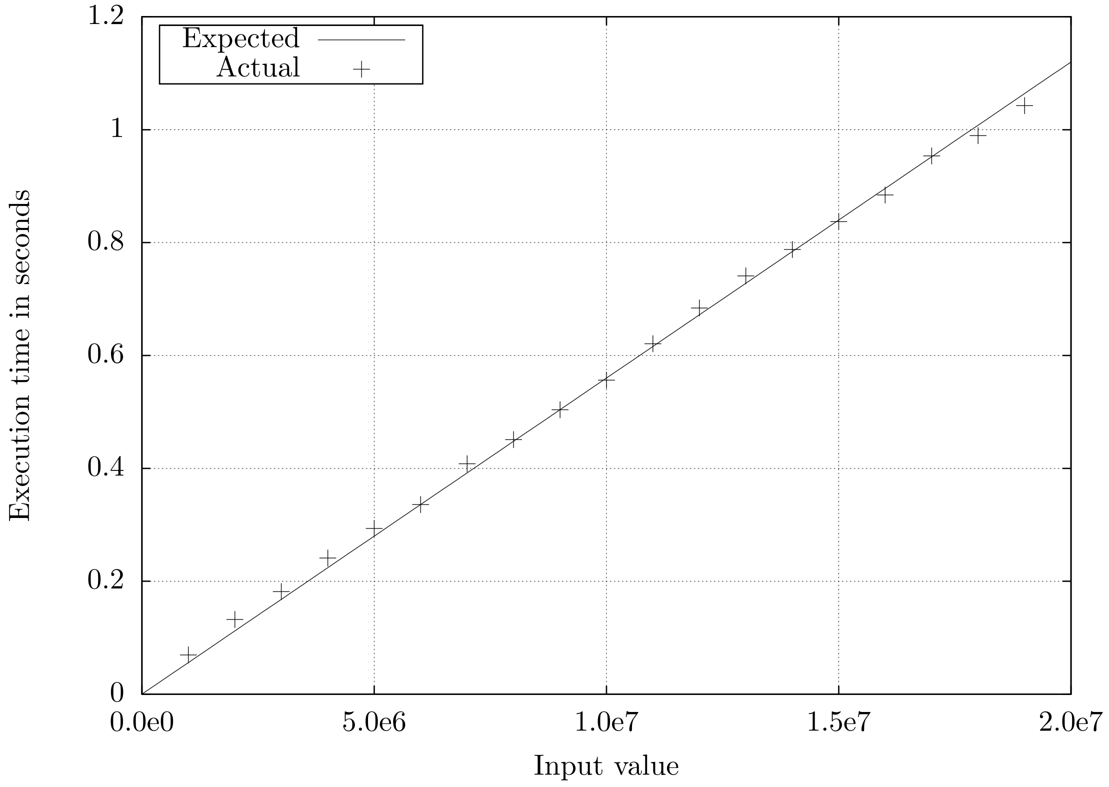
If a bug caused the sort function to run in n^3 time, the tool would issue a warning.

References

[1] batterseapower and simonpj. Annotations - GHC. http://hackage.haskell.org/trac/ghc/wiki/Annotations.

[2] ghc-6.12.1: The GHC API. http://www.haskell.org/ghc/docs/latest/html/libraries/ghc/GHC.html.

[3] Andrew Shewmaker. Micro-analysis of C project report. http://users.soe.ucsc.edu/~cormac/wiki/lib/exe/fetch.php?id=projects&cache=cache&media=micro-analysis-of-c-report.pdf, Dec 2007.

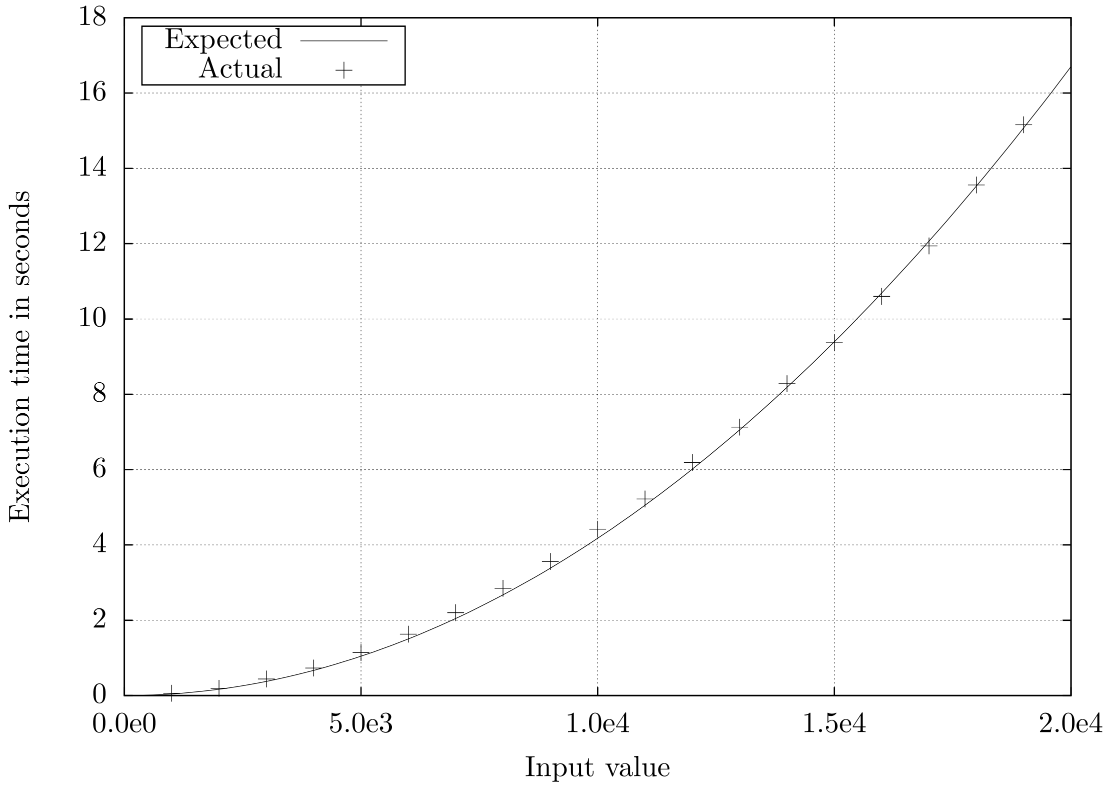


(a) Execution time, constant factor is 1.12e-8

```
-- Expected time: 5x + 2
f 0 0 = 0
f x y = f (x-1) 0
```

(b) Code

Figure 1: Linear tail call recursive function

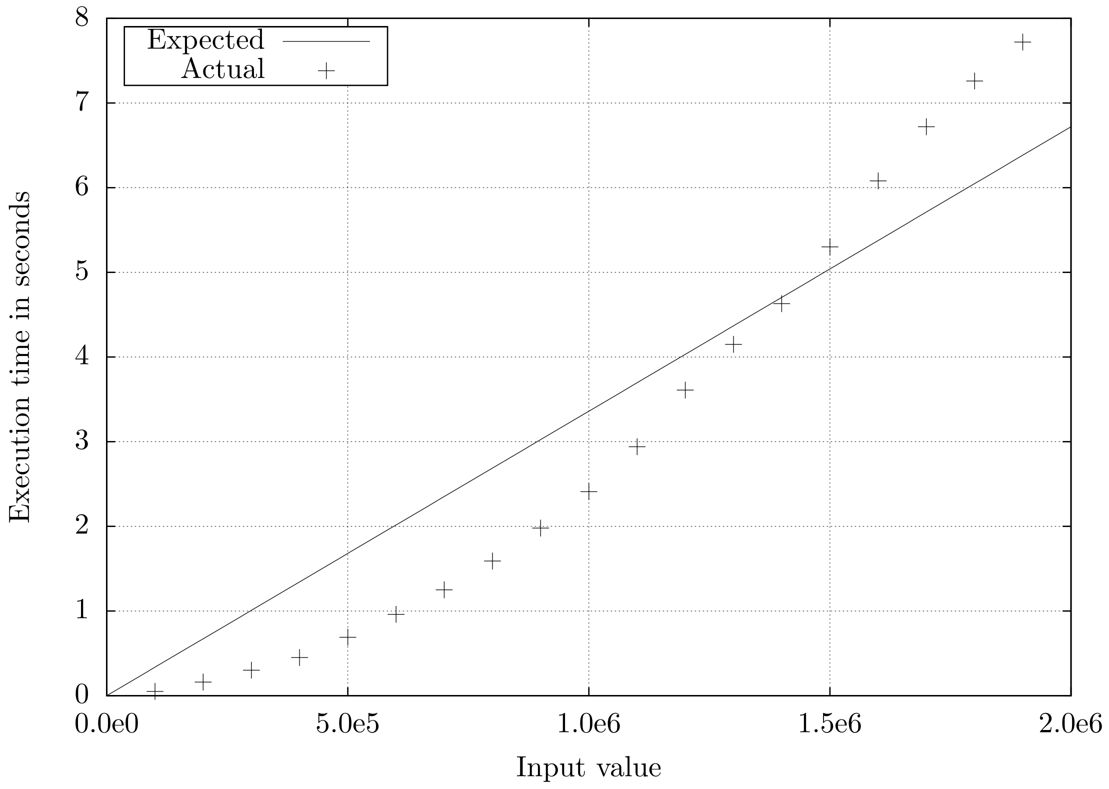


(a) Execution time, constant factor is 1.67e-8, y is 0

```
-- Expected time: 5/2*x^2 + 15/2*x + 5y + 2
f 0 0 = 0
f x 0 = f (x-1) x
f x y = f x (y-1)
```

(b) Code

Figure 2: Quadratic tail call recursive function



(a) Execution time, constant factor is 4.8e-7

```
-- Expected time: 7x + 1
f 0 = 0
f x = 1 + f (x - 1)
```

(b) Code

Figure 3: Non-tail call recursive function