Type-Based Resource Analysis on Haskell

Franz Siglmüller
Ludwig Maximilian University
Munich, Germany
frsiglmu@web.de

We propose an amortized analysis that approximates the resource usage of a Haskell expression. Using the plugin API of GHC, we convert the Haskell code into a simplified representation called GHC Core. We then apply a type-based system which derives linear upper bounds on the resource usage. This setup allows us to analyze real Haskell code, whereas previous implementations of similar analyses do not support any commonly used lazy functional programming languages.

1 Introduction

It can be very difficult to estimate the runtime costs of a program written in a lazy programming language. For example, consider the following two versions of a "repeat" function, which returns an infinite list with a given element:

```
repeat x = let xs = x : xs in xs
it = repeat 1 :: [Int]
repeat' x = x : repeat' x
it' = repeat' 1 :: [Int]
```

To a novice, it may not be obvious that the memory usage of this function is constant for the first version and linear for the second version. To assist in detecting these differences, we want to automatically generate annotated types such as the following:

$$\vdash^{4}_{0} \; it : \mu X.\ \{[] : (0, []) \mid (:) : (0, [\mathsf{T}^0(\mathsf{Int}), \mathsf{T}^0(X)])\}$$
$$\vdash^{6}_{0} \; it' : \mu X.\ \{[] : (0, []) \mid (:) : (0, [\mathsf{T}^0(\mathsf{Int}), \mathsf{T}^2(X)])\}$$

In these types, the second field of the $(:)$ constructor is wrapped in a thunk type that is annotated with costs 0 and 2, respectively. These values represent additional costs – in this case, memory allocations – that arise from accessing subsequent nodes of the linked list. As this cost is 0 in the first version, this means that no additional allocations will take place and the list therefore requires constant space. In the second version, the value is greater than 0, indicating linear costs.

There already are analyses that can assist in finding upper bounds for resource costs [1, 2]. However, these do not apply to any commonly used functional languages with laziness.

Our system is based on previous work described in "Type-Based Cost Analysis for Lazy Functional Languages" by Steffen Jost, Pedro Vasconcelos, Mário Florido and Kevin Hammond [2]. Their system works on an artificial language which we dubbed "JVFH" which was specifically designed for this analysis and is too unwieldy for common use. For example, consider the following expression, which is the JVFH translation of the Haskell definitions `repeat'` and `it'` above:

Submitted to:
DICE-FOPARA 2019

```
let repeat' = \x -> let xs = repeat' x
                    in letcons xt = Cons (x,xs)
                       in xt
in let one = 1
   in let it' = repeat' one
      in it'
```

Just like Haskell, JVFH is a lazy functional programming language; therefore, this work is a suitable inspiration for our analysis on Haskell.

2 Methodology

The architecture of our analysis is outlined in Figure 1. In order to analyze a given Haskell program, we first need to parse the text file and convert it into a suitable format. To avoid redundant work, we make use of the plugin API of the Haskell compiler GHC. This allows us to utilize the existing GHC compilation pipeline to convert the original Haskell code into any of the intermediate languages used within the compiler.

The analysis is implemented as a type-based system on GHC Core. Our approach is almost identical to the one used by Jost et al. for JVFH: A type system is used to derive a type annotated with mathematical variables, as well as a linear program over these variables. We then feed this linear program to an LP solver. If a valid variable assignment is found, we can then replace the type annotation with actual values.

Our system uses exactly the same syntax for its types as the original JVFH system does. For this reason, we can reuse most of the definitions and inference rules from the original paper by Jost et al. Modifications were only necessary for most of the syntax-driven type rules, to accommodate for any differences between JVFH and GHC Core. Most expressions in GHC Core have a similar counterpart in JVFH, requiring only minor adaptions. For example, pattern matches in GHC Core may have a default case which is used when none of the other cases apply, while JVFH requires the user to enumerate every possible case explicitly. However, some Core expressions – namely, type abstraction, type application, coercion and cast – do not have any equivalent in JVFH. Adding support for these expressions is deferred to future work.

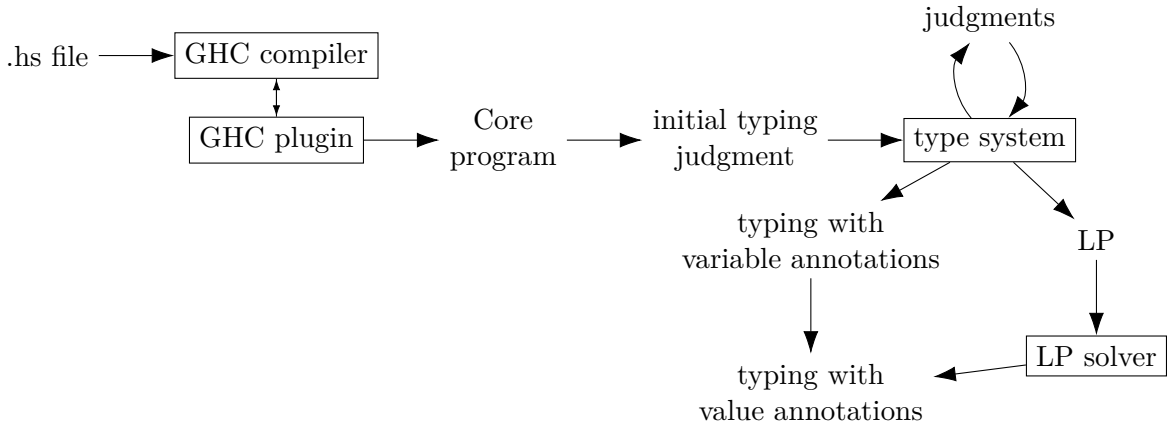

Figure 1: Outline of the architecture of our analysis

3 Evaluation

Our implementation can analyze a substantial subset of Haskell. For example, consider the following simple Haskell module, which contains examples of several concepts that are common in Haskell, but not available in JVFH; such as list comprehension and function applications with non-variable arguments:

```
module Example where
import Prelude hiding (map, repeat)
repeat x = xs where xs = x : xs
map f xs = [f x | x <- xs]
it = map (+1) $ repeat 1 :: [Int]
```

When applied to this module, our analysis will successfully generate the following typing for the variable `it`:

$\vdash_0^9 it : \mu X.\ \{[] : (0, []) \mid (:) : (0, [\mathsf{T}^1(\mathsf{Int}), \mathsf{T}^3(X)])\}$

This means that evaluating this expression to weak head normal form will induce a onetime cost of 9 allocations at most, and up to 3 more allocations for each list node accessed. Furthermore, accessing any of the list elements will also evoke one additional allocation.

For comparison, consider the following translation of the previous code to JVFH:

```
let repeat = \x -> letcons xs = Cons(x,xs) in xs
in let map = \f xt -> match xt with
           Nil() -> letcons r = Nil() in r
         | Cons(x,xs) -> let y = f x
                         in let ys = map f xs
                            in letcons r = Cons(y,ys)
                               in r
   in let one = 1
      in let ones = repeat one
         in let inc = \x -> let r = one + x in r
            in let it = map inc ones
               in it
```

From this expression, the original JVFH analysis will infer the following typing:

$\vdash_0^{10} it : \mu X.\ \{[] : (0, []) \mid (:) : (0, [\mathsf{T}^1(\mathsf{Int}), \mathsf{T}^3(X)])\}$

Note that the typing is very similar, but not identical. The specific values used as type annotations may differ, due to differences in how the code is represented in JVFH and GHC Core, respectively; and the LP solver also has some freedom in how values are assigned to variables.

However, we do acknowledge that there are some flaws that inhibit the usefulness of our analysis. Most conspicuously, polymorphism is not supported. In GHC Core, polymorphism is implemented via type abstraction and type application, which do not exist in JVFH. For convenience, we do have a workaround in place to support polymorphic functions that are used in a monomorphic way. For example, note that in the code above, we do not explicitly specify a type for the function `repeat`; Therefore, GHC will automatically infer the polymorphic type "$\forall a.\ a \to [a]$". However, as the function is used only once, we can still successfully analyze the code by simply ignoring type abstraction and application, and treating the function as "$\mathsf{Int} \to [\mathsf{Int}]$".

Additionally, `newtype`s and several Haskell language extensions such as GADT are not supported at all. Again, these features depend on GHC Core expressions that do not have any

equivalent in JVFH, namely, coercion and cast.

We also have not introduced support for multi-module programs yet; Therefore, we cannot analyze any expressions that contain variables imported from different modules, including the `Prelude`. In some cases, however, the compiler may automatically inline function calls, circumventing this limitation. This is the reason why we can use the operators `+` and `$` in the example code above.

4 Conclusion

We have presented an automated amortized analysis on Haskell. Our system uses the GHC plugin API to convert Haskell code to GHC Core, and then statically derives a type annotated with upper bounds for the resource usage. We have observed that our type-based system generates results similar to those of the original system it was based on. But unlike the previous work, our system accepts real Haskell code as input, which constitutes the main advantage of our work.

References

[1] Jan Hoffmann, Ankush Das & Shu-Chun Weng (2017): Towards automatic resource bound analysis for OCaml. In: ACM SIGPLAN Notices, 52, ACM, pp. 359–373, doi:10.1145/3093333.3009842. Available at `http://www.cs.cmu.edu/~janh/papers/HoffmannW15.pdf`.

[2] Steffen Jost, Pedro Vasconcelos, Mário Florido & Kevin Hammond (2017): Type-Based Cost Analysis for Lazy Functional Languages. Journal of Automated Reasoning 59(1), pp. 87–120, doi:10.1007/s10817-016-9398-9. Available at `http://www.dcc.fc.up.pt/~pbv/research/JAR2016-draft.pdf`.