Causality of Optimized Haskell

What is burning our cycles?

Peter M Wortmann [*] David Duke

University of Leeds

{scpmw,d.j.duke}@leeds.ac.uk

Abstract

Profiling real-world Haskell programs is hard, as compiler optimizations make it tricky to establish causality between the source code and program behavior. In this paper we attack the root issue by performing a causality analysis of functional programs under optimization. We apply our findings to build a novel profiling infrastructure on top of the Glasgow Haskell Compiler, allowing for performance analysis even of aggressively optimized programs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging Aids; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

Keywords Profiling; Optimization; Haskell; Causality

1. Introduction

A major selling point of functional languages is their clean model of computation, prioritizing ease of composition over efficient mapping to hardware. By strictly dividing up responsibilities, aggressive compiler optimizations can then often close the gap again. Unfortunately, this also makes it hard to reason about compilation process: Where a performance problem ends up falling through the cracks, it often becomes exceptionally hard to explain. Therefore we need specialized profiling tools to assist the programmer in pin-pointing the performance problems and explaining the root causes behind them. These tools must bridge the full abstraction gulf, accurately measuring low level performance while relating it all the way back to the original source code.

1.1 Context

The ecosystem of real world Haskell development currently centers mostly around the Glasgow Haskell Compiler GHC [19]. And this for good reason, as it is has managed to make extensive optimization techniques such as heavy in-lining [18], rules [4, 22] and lax code-reordering [17] relevant to everyday programming tasks. This has enabled Haskell programs to perform competitively even while taking full advantage of the language's abstraction mechanisms.

However, there have been common complaints that the performance of Haskell programs can be quite unpredictable [28]. And while there has been a robust and well-tested profiling framework for the GHC compiler for some years now [25], it has to impose significant restrictions on what optimizations the compiler is allowed to perform. Furthermore, newer work [14] suggests that its cost attribution scheme still has room for improvement.

1.2 Motivation

Consider the following popular example program in Haskell:

```
fac :: Int → Int
fac n = foldr (∗) 1 [1..n]
```

We compute the factorial of a number n by multiplying out the numbers from 1 to n. What performance would we expect?

The answer is surprisingly complicated, even for this simple program using common library functions. If we compile with GHC, we have to account amongst others for specialization, inlining, short-cut deforestration [7], strictness analysis as well as the worker-wrapper transformation [5]. After many steps, these transformations strip our function down to its core loop:

```
go !w !n | w == n      = w
         | otherwise   = w ∗ go (w+1) n
```

This is remarkable – yet for large inputs the optimized code *still* runs out of stack space needlessly, as it was not constructed to be tail-recursive! To make matters worse, conventional profiling would be useless here, as it would either prevent key optimizations from happening – or present information so coarsely that we would have to guess at what is happening in the inner loop.

1.3 Contribution

We have built a profiling framework that is able to handle profiling even for such tricky interactions between various code pieces and compiler optimizations. This is fundamentally about balancing two concerns: Accurate tracking of causes for performance problems, and feasibility of actual instrumentation and profile analysis.

We will attack these separately: In Section 2 we will explain our causality model and apply it to derive exact cause-effect relations for the evaluation of a simple lazy functional language. In Section 3 we will then show how we can extend the approach to reason about compiler optimizations as well.

Starting with Section 5 we will consider how to use the annotations for real-world profiling. We explain necessary simplifications and analyze their consequences, with the existing cost-centres framework [25] as our reference. Finally Section 6 will introduce our own profiling framework, which offers high-accuracy profiling of fully optimized programs at minimum overhead.

[*] This work was kindly supported by Microsoft Research through its PhD Scholarship Programme.

Haskell '13, September 23–24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2383-3/13/09…$15.00.
http://dx.doi.org/10.1145/2503778.2503788

2. Tracking Causality

In profiling, we are ultimately looking for explanations: *Why* did this performance problem occur? This invariably leads to deeper questions: Why did the machine behave like that? Where exactly was that call coming from? What made the compiler generate the code in that way? The answers to these questions make up the actual *causal* story behind the performance of the program.

In this section we will start out by showing how we can find such answers for a simplified lazy functional language. The idea is that we track the *cause* for every piece of code, value or run-time cost. As long as we make sure that we identify causality correctly at every step along the data flow, this promises us profiles that are correct by construction.

2.1 Counter-Factual Causation

In order to formally reason about causality, we need a causality model. Our goal is to be able to support claims such as:

Using **foldr** here causes excessive stack usage.

How would we show such a statement to be true? Let us negate both sides to get *counter-factual* conditionals [13]:

Not using **foldr** here can cause less stack usage.

This we can actually somewhat test: If we could show that similar programs *without* **foldr** lack the original problem, we have indeed demonstrated a causal connection. This approach forms the basis of Lewis' classic theory of counter-factual causality [13].

To put this formally, let us call the original program and its behavior a "world" W. Then if we have a cause α (using **foldr**) and effect ω (stack usage) that are true in W, we are allowed to derive the counter-factual causation relation

$$\neg\alpha \;\Box\!\!\rightarrow\; \neg\omega$$

if and only if the *closest* world W' where α would be false would *also* see β becoming false.

Perhaps surprisingly, Lewis theory leaves it almost completely open what the "closest" world should be – it is a parameter that has to be set intuitively based on the problem at hand. In our case, this means we first need to define how to reason about programs and their execution! In the following we will continue to notate causes as α, β, γ and δ, and use $\alpha\beta = \alpha \wedge \beta$ for a cause conjunction.

2.2 Definitions

Let us define a simple cause-annotated lazy functional language to continue our discussion. We will notate variables as x or y, constructors as C or D, values as v and expressions as e:

$$\begin{array}{lcl}
v & ::= & C\,x_1\,x_2 ... \\
 & | & \lambda y.\underline{e} \\
 & | & \bot \\
e & ::= & v \\
 & | & \underline{e}\,x \\
 & | & x \\
 & | & \texttt{let}\,\{x_1 = \underline{e}_1,\, x_2 = \underline{e}_2,\, \ldots\}\,\texttt{in}\,\underline{e} \\
 & | & \texttt{case}\,\underline{e}\,\texttt{of}\,\{C\,x_1\,x_2 ... \rightarrow \underline{e}_1;\; D\,y_1\,y_2 ... \rightarrow \underline{e}_2;\; \ldots\}
\end{array}$$

where $\underline{e} = {}_{\langle\alpha\rangle}e$ is an expression *annotated* with its cause α. By this we mean that if we have β as the event of encountering ${}_{\langle\alpha\rangle}e$, we have $\neg\alpha \;\Box\!\!\rightarrow\; \neg\beta$: if α becomes false, it might cause e to change.

To reason about performance, we introduce cost terms $\mathcal{O}$:

$$\mathcal{O} \quad ::= \quad \mathcal{L}\,|\,\mathcal{C}\,|\,\mathcal{A}\,|\,\mathcal{V}\,|\,\mathcal{T}\,|\,\mathcal{E}\,|\,\bot$$

with each unit of abstract cost corresponding to an application of a rule from the language semantics we are about to define. For example each evaluation of a `let` expression should produce one unit of $\mathcal{T}$ cost. Cost annotations will have the same meaning as on expressions, but use the slightly more compact notation $\underline{\mathcal{O}} = \alpha\mathcal{O}$. We define profiles $\theta = \alpha\mathcal{O}_1 + \beta\mathcal{O}_2 + \cdots$ as bags of such cause-annotated costs.

At this point we have enough to consider evaluation semantics. We will use Launchbury's natural semantics for lazy evaluation [12] as our starting point, but extend it with annotations and profiles. Our judgments will take the following form:

$$\Gamma : {}_{\langle\beta\rangle}e \Downarrow_\theta \Gamma' : {}_{\langle\gamma\rangle}v$$

With Γ standing for the heap (a variable map $x \mapsto \underline{e}; ...$) before evaluation and ${}_{\langle\beta\rangle}e$ the annotated expression to evaluate. On the right side the judgment yields the profile θ, a new heap Γ' and an annotated result value ${}_{\langle\gamma\rangle}v$. The profile notation was inspired by Launchbury as well as Sansom et al [25]. As usual, where a judgment can not be derived (finitely) from the semantics, the judgment result are $\bot$ terms.

2.3 Deriving Annotations

Consider the application rule:

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : \boxed{{}_{\langle\delta\rangle}\lambda y.\underline{e}_2} \qquad \Gamma_2 : \underline{e}_2[x/y] \Downarrow_{\theta_3} \Gamma_3 : \underline{v}}{\Gamma_1 : \boxed{{}_{\langle\beta\rangle}(\underline{e}\,x)} \Downarrow_{?\mathcal{A}+?\theta_2+?\theta_3} \Gamma_3 : {}_{\langle ? \rangle}\underline{v}}$$

So far this is just Launchbury's rule with annotations and profiles added where syntactically required. Unfortunately it is not quite immediately obvious how we should annotate: Do we need extra annotations on v, and what about the various costs in the generated profile? We must consult our causality model and derive causes from the terms the rule match depends on (highlighted).

First consider the application expression $(\underline{e}\,x)$. To find its effects, our causality model says that we need to consider an alternate world W' where $(\underline{e}\,x)$ was "miraculously" replaced with a different e' just as we were about to apply the rule. Let us pick $e' = \bot$ as our replacement. As no rule is allowed to match $\bot$, this would lead to the following W' judgment:

$$\Gamma_1 : \bot \Downarrow_\bot \bot$$

We observe that in W' the judgment does not generate v or any of the costs in question. We can therefore conclude that they were caused by encountering $(\underline{e}\,x)$ and therefore – by transitivity – by β. Hence we must annotate β on all costs in the profile as well as on the return value.

Let us repeat this procedure with the result of the evaluation of $\underline{e}$, which the rule expects to be ${}_{\langle\delta\rangle}\lambda y.\underline{e}_2$. Again we ask: What if in an alternate world W'' we switched out the returned value by $e'' = \bot$? By matching as far as possible[1] we obtain the W'' judgment

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : \bot}{\Gamma_1 : {}_{\langle\beta\rangle}(\underline{e}\,x) \Downarrow_{\beta\mathcal{A}+\beta\theta_2+\beta\bot} {}_{\langle\beta\rangle}\bot}$$

Accordingly, it follows that we need to annotate the profile θ_3 as well as the return value v. We arrive at the fully annotated application rule:

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : {}_{\langle\delta\rangle}\lambda y.\underline{e}_2 \qquad \Gamma_2 : \underline{e}_2[x/y] \Downarrow_{\theta_3} \Gamma_3 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}(\underline{e}\,x) \Downarrow_{\beta\mathcal{A}+\beta\theta_2+\beta\delta\theta_3} \Gamma_3 : {}_{\langle\beta\delta\rangle}\underline{v}}$$

Note that we completely ignore heaps where the result was $\bot$, even though alternate worlds would probably also update the heap differently! This is because we have to think of expressions and values as graphs and our rules as graph reductions: a free variable is actually a place-holder for a shared expression or value from the heap. Consequently, the heap state does not matter without a reference to it.

[1] We are essentially using small-step semantics for a moment.

$$\Gamma : {}_{\langle\beta\rangle}C\ x_1\ x_2... \Downarrow_{\beta\mathcal{C}} \Gamma : {}_{\langle\beta\rangle}C\ x_1\ x_2... \qquad \textit{(Con)}$$

$$\Gamma : {}_{\langle\beta\rangle}\lambda y.\underline{e} \Downarrow_{\beta\mathcal{L}} \Gamma : {}_{\langle\beta\rangle}\lambda y.\underline{e} \qquad \textit{(Lam)}$$

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : {}_{\langle\delta\rangle}\lambda y.\underline{e}_2 \qquad \Gamma_2 : \underline{e}_2[x/y] \Downarrow_{\theta_3} \Gamma_3 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}(\underline{e}\ x) \Downarrow_{\beta\mathcal{A}+\beta\theta_2+\beta\delta\theta_3} \Gamma_3 : {}_{\langle\beta\delta\rangle}\underline{v}} \qquad \textit{(App)}$$

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta} \Gamma_2 : \underline{v}}{\Gamma_1; x \mapsto \underline{e} : {}_{\langle\beta\rangle}x \Downarrow_{\beta\mathcal{V}+\beta\theta} \Gamma_2; x \mapsto \underline{v} : {}_{\langle\beta\rangle}\underline{v}} \qquad \textit{(Var)}$$

$$\frac{\Gamma_1; y_i \mapsto \underline{e}_i[{}_{\langle\beta\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...]; ... : \underline{e}[{}_{\langle\beta\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...] \Downarrow_{\theta} \Gamma_2 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}\texttt{let}\,\{x_i = \underline{e}_i, ...\}\,\texttt{in}\,\underline{e} \Downarrow_{\beta\mathcal{T}+\theta} \Gamma_2 : \underline{v}} \qquad \textit{(Let)}$$

(with y fresh variables)

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : {}_{\langle\delta\rangle}C_i\ y_j... \qquad \Gamma_2 : \underline{e}_i[y_j/x_j, ...] \Downarrow_{\theta_3} \Gamma_3 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C_i\ x_j... \rightarrow \underline{e}_i, ...\} \Downarrow_{\beta\mathcal{E}+\beta\theta_2+\beta\delta\theta_3} \Gamma_3 : {}_{\langle\beta\delta\rangle}\underline{v}} \qquad \textit{(Case)}$$

$$\frac{\Gamma_1 : \underline{e} \Downarrow_{\theta_2} \Gamma_2 : {}_{\langle\delta\rangle}C\ y_j... \qquad \Gamma_2 : \underline{e}_0[{}_{\langle\beta\delta\gamma\rangle}y_j/{}_{\langle\gamma\rangle}x_j, ...] \Downarrow_{\theta_3} \Gamma_3 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}\texttt{case}\ \underline{e}\ \texttt{of}\ \{C\ x_j... \rightarrow \underline{e}_0\} \Downarrow_{\beta\mathcal{E}+\beta\theta_2+\theta_3} \Gamma_3 : \underline{v}} \qquad \textit{(Case')}$$

Figure 1. Annotated evaluation rules

2.4 Advanced Annotations

In the last section, our recipe for removing expressions in the alternate world was to replace them by $e' = \bot$. On one hand, this is very effective, as it clearly removes any trace of the original expression. On the other hand, it always forced an immediate rule mismatch, which is rather crude. After all, Lewis' theory suggests comparing to a world that is as *close* as possible to the original.

Consider for example the Sestoft-style `let` rule [26]:

$$\frac{\Gamma_1; y_i \mapsto \underline{e}_i[y_i/x_i, ...]; ... : \underline{e}^B[y_i/x_i, ...] \Downarrow_{\theta} \Gamma_2 : \underline{v}}{\Gamma_1 : \boxed{{}_{\langle\beta\rangle}\texttt{let}\,\{x_i = \underline{e}_i, ...\}\,\texttt{in}\,\underline{e}^B} \Downarrow_{?\mathcal{T}+?\theta} \Gamma_2 : {}_{\langle ?\rangle}\underline{v}}$$

Here the evaluated expression is the only interesting input term. If we again set $e' = \bot$ we would obtain the following annotated rule:

$$\frac{\Gamma_1; y_i \mapsto \underline{e}_i[y_i/x_i, ...]; ... : \underline{e}^B[y_i/x_i, ...] \Downarrow_{\theta} \Gamma_2 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}\texttt{let}\,\{x_i = \underline{e}_i, ...\}\,\texttt{in}\,\underline{e}^B \Downarrow_{\beta\mathcal{T}+\beta\theta} \Gamma_2 : {}_{\langle\beta\rangle}\underline{v}}$$

with annotations both on all costs as well as the returned value. However, this result might sound a bit suspicious: The role of a `let` expression is to add bindings, not influence control flow or the result of the evaluation of the body. From this point of view, the annotations on θ and v look a bit shaky.

Can this intuition be supported formally? If we were right, we would have to find another world W'' where the `let` expression was gone, but the costs as well as the result of the body's evaluation would still appear. Well, let us try to simply set $e'' = e^B$! In the unlikely case that evaluation of the naked body happens to never use the now-undefined x_i variables, this would allow the following W'' judgment:

$$\Gamma_1 : \underline{e}^B \Downarrow_{\theta} \Gamma_2 : \underline{v}$$

Which would indeed be the same θ and $\underline{v}$ that we originally obtained. This in turn would mean that we would not have to annotate either, matching the intuition of not seeing them as directly influenced by the `let` expression!

2.5 Binding Effects

We seem to be going into the right direction: Simply by changing the alternate expression under consideration we have identified a case where we can safely reduce the amount of annotation. On the other hand, we can obviously not simply ignore the possibility that `let`-bound variables might be in use. Launchbury defines three rules where bindings influence program evaluation:

1. The variable rule, which would fail to look up x_i on the heap.
2. The constructor rule, which might produce a constructor mentioning x_i instead of y_i.
3. The application rule, which could sub-evaluate an expression using x_i instead of y_i

We see that all three rules can be expected to produce different results where they use the eliminated x_i bindings. Therefore we want their result terms to get annotated.

We can achieve this using a trick: If we look ahead to Figure 1, we see that all three rules add their expression annotation to their results. Therefore, we can simply have the annotated `let`-rule add the annotations on the affected *expressions*:

$$\frac{\Gamma_1; y_i \mapsto \underline{e}_i[{}_{\langle\beta\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...]; ... : \underline{e}[{}_{\langle\beta\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i, ...] \Downarrow_{\theta} \Gamma_2 : \underline{v}}{\Gamma_1 : {}_{\langle\beta\rangle}\texttt{let}\,\{x_i = \underline{e}_i, ...\}\,\texttt{in}\,\underline{e} \Downarrow_{\beta\mathcal{T}+\theta} \Gamma_2 : \underline{v}}$$

where $e_i[{}_{\langle\beta\gamma\rangle}y_i/{}_{\langle\gamma\rangle}x_i]$ is a replacement rule updating the variable name as well as the annotation on the expression directly containing it. This way ensures indirectly that any (potentially) differing result will be annotated with the cause of the `let` expression.

2.6 Assembling

We see that a smart choice for the "closest world" can lead to more specific annotations. However, as the last subsection should have demonstrated, actually finding ways of getting the right annotations is not always trivial. As a compromise between accuracy and resulting annotations rule complexity, we will from here on use the following alternate world rule for removing an expression e:

$$e' = \begin{cases} e^B & \text{if } e = \texttt{let}\,\{...\}\,\texttt{in}\,e^B \\ e^B & \text{if } e = \texttt{case}\ ...\ \texttt{of}\ \{... \rightarrow e^B\} \\ & \quad \text{with } e^B \text{ sole branch} \\ \bot & \text{otherwise} \end{cases}$$

Note that we added a new special case for one-branch `case` statements here, as they work very similar to `let` expressions. This comes with just one extra subtlety: The removal might skip a non-terminating scrutinee computation, a change that strictly speaking would require an annotation. However, note that the object of annotation would be a profile or value that in W would never be produced in the first place, therefore this is inconsequential for profiling. Annotating all remaining rules accordingly, we finally arrive at the rule set laid out in Figure 1.

3. Optimizations

At this point we can derive program behavior and profiles for annotated programs in our toy language. With this we could, for example, already de-sugar a higher level language into our representation and derive meaningful profiles.

However in real-world situations we will rarely want to execute a program exactly as it was written. Compiler optimizations allow more flexibility for the programmer in how to write the program, by taking advantage of functionality-preserving transformations that are expected to improve program performance.

Optimizations pose a special challenge for tracking causality: While such transformations will often radically change how the program looks and executes at the point of optimization, in the end we are still bound to arrive at the same values and cost profiles with only a few local changes. For our analysis this means that just looking at the transformation in isolation would be short-sighted. Instead we must look all the way through evaluation and consider what effect the transformation has on the costs of the evaluation of the *full program*. Then we can again derive annotations, this time using the behavior of the unoptimized program as the "closest world" point of comparison.

3.1 Beta Reduction

Assume we have a situation where the function expression of an application is a lambda. We can then optimize the expression by eliminating both the application and the lambda expression using beta reduction. The rule could look like follows:

$$_{\langle\beta_1\rangle}\left(_{\langle\beta_2\rangle}\lambda x.\underline{e}\right)\ y \quad\Longrightarrow\quad _{\langle ?\rangle}\underline{e}\,[y/x]$$

To derive the annotation, we compare the behavior of the unoptimized version (W) with the optimized one (W'). Plugging the original expression into the semantics from Figure 1, we derive for W:

$$\frac{\Gamma : {}_{\langle\beta_2\rangle}\lambda x.\underline{e} \Downarrow_{\beta_2\mathcal{L}} \Gamma : {}_{\langle\beta_2\rangle}\lambda x.\underline{e} \qquad \Gamma : \underline{e}[y/x] \Downarrow_{\theta} \Gamma' : \underline{v}}{\Gamma : {}_{\langle\beta_1\rangle}\left(_{\langle\beta_2\rangle}\lambda x.\underline{e}\right)\ y \Downarrow_{\beta_1\mathcal{A}+\beta_1\beta_2\mathcal{L}+\beta_1\beta_2\theta} \Gamma' : {}_{\langle\beta_1\beta_2\rangle}\underline{v}}$$

This looks promising: In both worlds we end up evaluating $\underline{e}[x/y]$. Moreover, both the result value $\underline{v}$ and the profile θ gain just an $\beta_1\beta_2$ annotation in W. So we just have to make sure all costs generated by $\underline{e}[x/y]$ get a $\beta_1\beta_2$ annotation in W' and the difference of the profiles will be exactly the optimized-out costs $\mathcal{A} + \mathcal{L}$!

Unfortunately, we can not quite achieve this by adding a simple $\beta_1\beta_2$ annotation on the right side of the optimization rule. After all we have to account for `let` and one-branch `case` expressions, which as explained in Section 2.4 do not add their expression cause to all their results. Yet we can still achieve the desired effect by "pushing" the annotation inwards on these occasions:

$$_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e} = \begin{cases} _{\langle\beta\rangle}\texttt{let}\{...\}\texttt{in}\ _{\langle\!\langle\alpha\rangle\!\rangle}\underline{e'} \\ \qquad\qquad \text{if } \underline{e} = {}_{\langle\beta\rangle}\texttt{let}\{...\}\texttt{in}\ \underline{e'} \\ _{\langle\beta\rangle}\texttt{case}\ ...\ \texttt{of}\ \left\{C_0\ ... \rightarrow {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e'}\right\} \\ \qquad\qquad \text{if } \underline{e} = {}_{\langle\beta\rangle}\texttt{case}\ ...\ \texttt{of}\ \{C_0\ ... \rightarrow \underline{e'}\} \\ _{\langle\alpha\rangle}\underline{e} \qquad \text{otherwise} \end{cases}$$

For which we now can easily see that:

$$\Gamma : \underline{e} \Downarrow_{\theta} \Gamma' : \underline{v} \quad\Rightarrow\quad \Gamma : {}_{\langle\!\langle\alpha\rangle\!\rangle}\underline{e} \Downarrow_{\alpha\theta} \Gamma' : {}_{\langle\alpha\rangle}\underline{v}$$

And therefore re-state the beta reduction rule as:

$$_{\langle\beta_1\rangle}\left(_{\langle\beta_2\rangle}\lambda x.\underline{e}\right)\ y \quad\Longrightarrow\quad _{\langle\!\langle\beta_1\beta_2\rangle\!\rangle}\underline{e}\,[y/x]$$

In the end, let us reiterate that our approach lead to the *exact* same profile – minus the cost for the sub-expressions that was optimized out. Clearly it is not impossible to rework the program significantly without reducing the profile quality.

3.2 General Case

Consider the evaluation of an arbitrary expression e that was transformed to $e' = opt(e)$ by an optimization:

$$\Gamma_1 : \underline{e} \Downarrow_{\theta} \Gamma_2 : {}_{\langle\beta\rangle}v \quad\Rightarrow\quad \Gamma_1 : opt(\underline{e}) \Downarrow_{\theta'} \Gamma_2' : {}_{\langle\beta'\rangle}v$$

As long as the optimization is correct, we know that we have to arrive at the same result value v once we have evaluated it to normal form by resolving all referenced thunks. Given that most optimizations are local improvements, we would also expect a lot of costs to appear in both θ as well as θ'. As we only have to account for effects that actually change, this means that we do not have to reconsider their causes: The annotation rule should attempt to match them and ideally arrange for the annotations to stay the same after transformation. On the other hand, there are some cases where we have to allow changes to the profile. We will explain the interesting cases in the following subsections.

3.2.1 Preemption

In some situations, preemption might make it impossible to actually retain the same annotations. Consider the example:

$$\texttt{let}\{x = ...\,, y = {}_{\langle\alpha\rangle}\texttt{case}\ x\ \texttt{of}\ ...\}\texttt{in}\ ...\ x\ ...\ y\ ...$$

Will the cost for evaluating x be annotated with α? Obviously, this depends on whether x or y will end up getting evaluated first. Now suppose we have an optimization that changes the order in which x and y get evaluated: Should we require that it somehow retains or suppresses the α annotation on the costs for evaluating x?

This however would be very hard to achieve in this case. Luckily, we can argue that it is not actually required: We can see the different possible causes for the evaluation of x as *preempting* each other. Either of them would be a sufficient cause, it just so happens that only one of them can actually occur in any given program execution. Therefore causes that preempt each other are interchangeable in our analysis, and we can allow optimizations to freely substitute them with each other.

3.2.2 Overheads

Furthermore, sometimes optimizations might not just remove or reorganize costs, but actually introduce *new* costs. Such overheads might for example happen where an optimization rule replaces one cost type by another: We would see it as optimizing out one cost and introducing the new one as overhead.

This constitutes an actual alternate world change, with the transformation as its cause. Therefore our causality model demands annotating such effects with the transformation cause. For typical transformation rules, this will be the annotations of the sub-expressions that the transformation matches on, such as $\beta_1\beta_2$ in Section 3.1.

3.2.3 Annotation Overreach

The only way for optimization rules to influence annotations on the final profile is to feed carefully chosen expression annotations to the evaluator. However, annotating even just one expression can have far-reaching effects in the resulting profile. In this situation we might often be forced to produce extra annotations on one term in order to reach enough annotations on another.

$$
\begin{array}{l}
\Gamma : {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \left\{C\ x \to {}_{\langle\alpha_2\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\right\}\\
\left[\begin{array}{l}\Gamma : \underline{e}_1\\ \Downarrow_{\theta_1}\ \Gamma_1 : {}_{\langle\delta\rangle}C\ \hat{x}\end{array}\right.\\
\left[\begin{array}{l}\Gamma_1 : {}_{\langle\alpha_2\rangle}\texttt{let}\ \{y=\underline{e}_2\}\ \texttt{in}\ \underline{e}_3[{}_{\langle\alpha_1\delta\gamma\rangle}\hat{x}/{}_{\langle\gamma\rangle}x]\\
\left[\begin{array}{l}\Gamma_1;\hat{y}\mapsto\underline{\hat{e}}_2 : \underline{\hat{e}}_3[{}_{\langle\alpha_1\delta\gamma\rangle}\hat{x}/{}_{\langle\gamma\rangle}x]\\ \Downarrow_{\theta_2}\ \Gamma_2 : \underline{v}\end{array}\right.\\
\Downarrow_{\alpha_2\mathcal{T}+\theta_2}\ \Gamma_2 : \underline{v}\end{array}\right.\\
\Downarrow_{\alpha_1\mathcal{E}+\alpha_1\delta\alpha_2\mathcal{T}+\alpha_1\theta_1+\alpha_1\delta\theta_2}\ \Gamma_2 : {}_{\langle\alpha_1\rangle}\underline{v}\\
\left(\text{with } \underline{\hat{e}} = \underline{e}[{}_{\langle\alpha_2\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y]\right)
\end{array}
$$

$$
\begin{array}{l}
\Gamma : {}_{\langle\beta_2\rangle}\texttt{let}\ \{y = \underline{e}_2\}\ \texttt{in}\ \left({}_{\langle\beta_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x \to \underline{e}_3, ...\}\right)\\
\left[\begin{array}{l}\Gamma;\hat{y}\mapsto\underline{\hat{e}}_2 : {}_{\langle\beta_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x\to\underline{\hat{e}}_3, ...\}\\
\left[\begin{array}{l}\Gamma;\hat{y}\mapsto\underline{\hat{e}}_2 : \underline{e}_1\\ \Downarrow_{\theta_1}\ \Gamma_1;\hat{y}\mapsto\underline{\hat{e}}_2 : {}_{\langle\delta\rangle}C\ \hat{x}\end{array}\right.\\
\left[\begin{array}{l}\Gamma_1;\hat{y}\mapsto\underline{\hat{e}}_2 : \underline{e}_3[{}_{\langle\beta_2\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y][{}_{\langle\beta_1\delta\gamma\rangle}\hat{x}/{}_{\langle\gamma\rangle}x]\\ \Downarrow_{\theta_2'}\ \Gamma_2 : \underline{v}'\end{array}\right.\\
\Downarrow_{\beta_1\mathcal{E}+\beta_1\theta_1+\beta_1\delta\theta_2'}\ \Gamma_2 : {}_{\langle\beta_1\rangle}\underline{v}'\end{array}\right.\\
\Downarrow_{\beta_1\mathcal{E}+\beta_2\mathcal{T}+\beta_1\theta_1+\beta_1\delta\theta_2'}\ \Gamma_2 : {}_{\langle\beta_1\rangle}\underline{v}'\\
\left(\text{with } \underline{\hat{e}} = \underline{e}[{}_{\langle\beta_2\gamma\rangle}\hat{y}/{}_{\langle\gamma\rangle}y]\right)
\end{array}
$$

Figure 2. Evaluation before and after floating optimization

When handling tricky cases, this will end up as our main source for profile degradation, and we will see an example in the next section. Note however that overestimating effects is a common and relatively benign mistake – essentially introducing circumstantial information of little direct relevance into the profile. As long as do not overload the profile with such annotations we can be reasonably sure that it will still have value in performance analysis.

3.3 Floating

Let us test these rules to the "floating" optimization, which relocates bindings within an expression [20]:

$$
\begin{array}{l}
\quad {}_{\langle\alpha_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \left\{C\ x \to {}_{\langle\alpha_2\rangle}\texttt{let}\ \{y=\underline{e}_2\}\ \texttt{in}\ \underline{e}_3, ...\right\}\\
\Longrightarrow\ {}_{\langle\beta_2\rangle}\texttt{let}\ \{y=\underline{e}_2\}\ \texttt{in}\ \left({}_{\langle\beta_1\rangle}\texttt{case}\ \underline{e}_1\ \texttt{of}\ \{C\ x\to\underline{e}_3, ...\}\right)
\end{array}
$$

Here we are floating the y binding out – for example because we are in the process of promoting it to a statically allocated top-level constant. This is a very common optimization, hence we would like it to have little impact on profile quality. Can we set β_1 and β_2 in a way that achieves this?

Figure 2 shows the evaluation of both versions under the assumption that the $\underline{e}_3$ branch gets actually taken and all sub-evaluations terminate. Note that the profiles look very similar if we assume $\beta_1 = \alpha_1$ and $\beta_2 = \alpha_2$, mostly thanks to the special `let` rules we derived in Section 2.4. However there is one problem with setting β_2 this way: The $\mathcal{T}$ cost would lose the $\alpha_1\delta$ annotation.

This is hardly surprising: Clearly the execution of the `let` expression used to depend on what `case` branch was taken, whereas now it gets evaluated unconditionally. How do we handle this? No preemption takes place, and due to δ getting only determined at run-time, we have no way to include it in β_2. The only option left is therefore to see $\mathcal{T}$ as first optimized out, then re-introduced again as an *overhead*. This shifts its cause to the transformation, and makes it valid to annotate it with $\beta_2 = \alpha_1\alpha_2$:

$$
\begin{array}{lllll}
\theta = \alpha_1\mathcal{E} & +\alpha_1\delta\alpha_2\mathcal{T} & & +\alpha_1\theta_1 & +\alpha_1\delta\theta_2\\
\quad\Downarrow & \text{(remove)} & \text{(new)} & \quad\Downarrow & \quad\Downarrow\\
\theta' = \alpha_1\mathcal{E} & & +\alpha_1\alpha_2\mathcal{T} & +\alpha_1\theta_1 & +\alpha_1\delta\theta_2'
\end{array}
$$

However, we are still not quite finished: Note that β_2 gets inserted into $\underline{e}_3$ via substitution, therefore potentially propagating the extra α_1 annotation into θ' and $\underline{v}'$ – an instance of annotation overreach. Yet note that this does no actual harm, as both terms get annotated with $\beta_1 = \alpha_1$ by the `case` rule anyway, and we actually know that $\alpha_1\theta_2 = \alpha_1\theta_2'$. So in the end the only difference between θ and θ' will be that the $\alpha_1\delta$ annotation on $\mathcal{T}$ gets replaced by $\alpha_1\alpha_2$, truthfully reflecting exactly the relocation of the `let` expression.

3.4 Arbitrary Optimizations

At this point we have shown that our system is flexible enough to accommodate two transformations commonly used in optimization. But what can we find a way to satisfy the requirements from Section 3.2 no matter what the optimization rule is? This is interesting as we would ideally like to support programmer-supplied optimization rules [22] that are allowed to make almost arbitrary program modifications.

Consider the basic short-cut list fusion rule from the cited paper:

$$
\begin{array}{l}
\{-\#\ \texttt{RULES}\ \text{“}\texttt{foldr/build}\text{”}\\
\quad\forall kz(g :: \forall b.(a\to b\to b)\to b\to b).\\
\quad {}_{\langle\alpha\rangle}\texttt{foldr}\ k\ z\ ({}_{\langle\beta\rangle}\texttt{build}\ g) = {}_{\langle\alpha\beta\rangle}g\ k\ z\ \#-\}
\end{array}
$$

The idea here is that k and z encode a way to consume a list, while g describes how the list should be built. The rule says that whenever we find them combined in this way, we can just eliminate the immediate list and construct the result directly.

For the profile, this is a dangerous change: Just from the rule, we know very little about its cost implications. We can just estimate them to be substantial, given that we eliminate calls to two non-trivial functions. Saying more would require deep analysis of the reasoning behind the rule, which is a task that the compiler cannot perform automatically. Therefore we have to either ask the rule writer to provide the proper annotation method for us, or find a way to annotate that requires no further knowledge about the rule.

In the last section we have already seen how we can handle cost changes that we fail to map cleanly: We simply declare them as being removed, then being re-created by the transformation as overheads. In our case, the most conservative annotation strategy is therefore to see this as happening for *all* costs touched by the optimization. Fortunately, in a pure language we know these to be exactly the returned profile and result value, therefore forcing annotations on these two terms should yield a valid program profile.

As shown in Section 3.1, we can achieve this using a push annotation. In general, if we have an optimization *opt* that gets triggered by an event $\alpha\beta$ (as above), it is always correct to annotate:

$$
{}_{\langle\alpha\rangle}\cdots\underline{e}_1\cdots{}_{\langle\beta\rangle}\cdots\underline{e}_2 \quad\Longrightarrow\quad {}_{\langle\!\langle\alpha\beta\ldots\rangle\!\rangle}\mathit{opt}(\underline{e}_1,\underline{e}_2)
$$

Note that for unmatched sub-expressions $\underline{e}_i$ we do not need to touch annotations, as the rule application does not depend on them. As a result, while this is the most conservative choice, we still end up with fairly non-intrusive annotations. In the example of the `foldr/build` rule, it is actually easy to see that this is the only valid choice. Paired with the fact that such specialized rules generally are not applied very often compared with, say, `let` floating, this treatment can be expected to retain acceptable profile quality.

$$\begin{aligned}&{}_{\langle\alpha\rangle}\texttt{let}\,\{f = ...,\; g = ...,\\ &\qquad h = {}_{\langle\gamma\rangle}\texttt{case}\; y\;\texttt{of}\{C_1\,x \to f, C_2\,x \to g\}\}\\ &\quad\texttt{in}\;\boxed{{}_{\langle\beta\rangle} h\; x}\end{aligned}$$

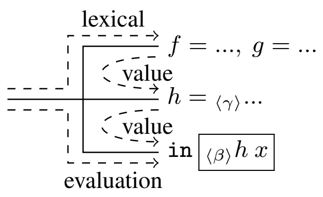

lexical
$f = ...,\; g = ...$
value
$h = {}_{\langle\gamma\rangle}\cdots$
value
$\texttt{in}\;\boxed{{}_{\langle\beta\rangle} h\; x}$
evaluation

Figure 3. Scopes for an application expression

4. Profiling Design

Up to this point, we have only considered what cost causes would be if assigned by a perfect outside observer with limitless memory. This is the appropriate viewpoint to take while compiling and optimizing source code, as there is no good reason to sacrifice diagnostic power at that point. However once we deal with an actual running program, cost analysis and attribution has a tricky problem: If measurements consume significant resources itself, it will skew the very results we are interested in! In practice, this means that we *must* reduce the amount of causes we track if we want to produce useful results.

This section will investigate the design space, outlining different ways we can cut down on the amount of information gathered. We will use this occasion to discuss other profiling frameworks and evaluate their design decisions.

4.1 Prioritization

If we have to drop annotations, which ones should we choose? Observe that cause annotations vary in usefulness: For example, *every* cost will causally depend on the evaluation of the entry point "main". Therefore unless the performance problem resided exactly in this function, this lead would be pretty useless while tracking down a performance problem.

We therefore postulate that some causes are more closely coupled to an effect than others: The fewer other effects a cause has, the more likely it is to be the root of the problem. We even have a way to reason about this from our semantics: Note that outer rule applications generally add annotations to *all* costs that are produced by nested rule applications. Therefore outer rules annotate strictly *more* effects and are therefore less specific.

This "ranking" of causes is very commonly exploited by profiling frameworks to cut down the amount of information to track, while at the same time structuring it for easier analysis. For example the original profiling solution proposed by Sansom et al [25] only ever tracks the upper-most cost-centre. Other implementations track cause terms as stacks, as for example `gprof` [8] or newer implementations of cost-centre profiling [24]. This practice is especially prevalent in profiling tools for imperative languages, as call stacks are relatively stable there.

Note however that this is generally not the case for optimized purely functional programs. In fact we made no attempts whatsoever so far to ensure stack consistency or annotation order, in contrast to Sansom's work. This is a deliberate, as we see no way to maintain stack consistency while allowing enough freedom for optimizations.

4.2 Value Annotations

Our semantics annotate every value with its full causal history. As there is no way to predict the annotations statically, a direct implementation would have to store and dynamically compose causes at run-time. This however would come at considerable cost, as programs typically process large amounts of data. Tracking every value would increase memory access and consumption significantly. Figure 1 also tells us that there are merely two rules where value annotations make their way into the profile: *(App)* and *(Case)* – unsurprisingly the two points where control flow depends on values.

The most common simplification is to skip value annotations in the *(Case)* rule, which allows ignoring annotations on all non-lambda values at run-time. This helps a lot, as non-lambda values can be expected to make up most of the run-time data. And while it is entirely possible to encode program behavior as data, primitive or constructor values rarely end up being the most interesting influence on program performance.

4.3 Lambda Annotations

Lambda values, on the other hand, need more consideration. It is characteristic for the functional style to not only control execution by calling functions, but also by *passing* functions to be called by other code, for example when using monads [29]. Therefore, both the origin of the lambda as well as the application site might turn out to be a fruitful lines of inquiry for solving a performance problem. For example in Figure 3 we should get the full $\alpha\beta\gamma$ annotation on costs coming from f or g. However implementing this would be expensive, as there is no cheap way for h to update the annotation of the closure as it gets passed through.

To solve this, Sansom et al track only the cause of the lambda allocation, the *lexical scope* α. The advantage of this approach is that the allocation site is constant for any closure object on the heap, and therefore does not need to be updated. This design ignores the cause for the evaluation of the application, a deliberate design choice [24]. However recent work by Marlow [14] concludes that this often leads to misleading cause stacks, and extends Samson's work by merging lexical and evaluation scope in applications. This would lead to an annotation of $\alpha\beta$ in the above example.

Either solution would miss γ, which is arguably acceptable for the same reasons that we regard value annotations as less important. Note that cost-centre profiling already has to pay significant overheads just for tracking evaluation and lexical scopes in this fashion. Most critically, it requires a non-trivial heap layout change, which not only changes the space complexity of the program, but also makes it impossible to use normally compiled code in a program that uses profiling.

4.4 Static Lexical Scope

To get around this, note that we can actually know a good portion of the lexical scope just by looking at the closure identity. Suppose we find out that $\underline{e}_1$ in the following example was evaluated:

$$_{\langle\alpha\rangle}\texttt{let}\,\left\{f = {}_{\langle\beta\rangle}\lambda x.\underline{e}_1\right\}\,\texttt{in}\,\underline{e}_2$$

Here we *know* that $\alpha\beta$ must be part of the cause of costs, as the evaluation of $\underline{e}_1$ clearly causally depends on the parent `let` and the lambda expression. This technique can be quite powerful as optimizations such as in-lining or specialization work in our favor by making the control flow more predictable. This means that as long as we maintain information about the "static context" of a given intermediate code piece, we can often extrapolate significant cause terms without the need for complex instrumentation.

5. Implementation

At this point we have a solid idea of how we expect the compiler to attribute code with cause terms, and in the last section we outlined the general trade-offs involved in an implementation. We will now describe our implementation of a new profiling framework for Haskell. In line with the theoretical work, we will focus on allowing profiling of fully optimized programs with high accuracy and minimum overhead. We will especially use zero code instrumentation, basing everything off maps of the generated machine code and non-intrusive run-time system changes.

The reasons for this focus are two-fold: Firstly, the framework by Sansom et al [25] is already well-established for big-picture profiling. But more importantly, modern Haskell usage has shown its great potential for heavy optimization, leading to tight inner loops that require careful performance analysis. We aim to provides useful results even under these demanding circumstances.

5.1 Cause Tracking in Core

Our work is based on the Glasgow Haskell Compiler [19]. The first step in the implementation of a proper profiling framework is to make sure that we can track causes throughout its compilation according to the rules we explained in the first sections.

To make this work, we have to represent cause annotations in every representation the program takes throughout the compilation process. The first stage in the process is that GHC will "de-sugar" full Haskell into a significantly reduced functional language called "Core" [17]. We can use existing facilities [6] at this point to ensure that every interesting program expression will be annotated with a "src" tick expression. For example, the program from the introduction could look like follows after de-sugaring:

```
fac n :: Int → Int
fac n =
   src<fac.hs:2,1−2,27>
   foldr (src<fac.hs:2,15−2,18> (∗))
         (src<fac.hs:2,19>       1)
         (src<fac.hs:2,21−2,27> [1..n])
```

Note that in contrast to our syntax from Section 2.2 we now represent causes as an *expression* type. Additionally, we will actually see them as applying to every profile and value that gets passed through, making their semantics that of the see them as corresponding to the "push" annotations from Section 3.1. These choices make annotations fit naturally with the compiler design. For example in-lining "**foldr**" right away would yield:

```
fac n :: Int → Int
fac n =
   src<fac.hs:2,1−2,27>
   src<Base.lhs:14,1−17,36>
   let go ys =
           src<16,13−17,36> case ys of
              []       → src<Base.lhs:16,25>
                           src<fac.hs:2,19> 1
              (y:ys) → [...]
   in go
```

This is consistent with our rules. Consider how we would achieve this result: We would first let-bind the parameters to "**foldr**", then unfold its definition and β-reduce it according to Section 3.1. This leads to the double-annotation on the top-level expression.

Second, we would let-float the parameters inwards as described in Section 3.3 until we have reached the usage site. On the way, we do not have to add additional annotations on the expressions we pass, as we established. Only when we unfold the values in question at the usage site we take over the respective annotation, leading to the "inner" annotation on the "1" expression in the example.

5.2 Core Issues

Unfortunately, this mapping is not a perfect implementation of our semantics. While giving annotations "push" semantics makes it more straight-forward to relocate expressions, this means that there is no way of annotating `let` or one-branch `case` expressions themselves. For a more complete solution, we should probably allow attaching ticks individually to these sub-expressions.

On the other hand, note that the interaction between push annotations and binding expressions allows us to "float" **src** ticks easily. This means that we are allowed to optimize ticks like follows:

$$\begin{aligned}&\textbf{src}{<}1{>}\ \textbf{let}\ \ \ldots\ \ \textbf{in}\ \textbf{src}{<}2{>}\ \ldots\\ &\qquad\Longrightarrow\quad \textbf{src}{<}1{>}\,\textbf{src}{<}2{>}\ \textbf{let}\ \ \ldots\ \ \textbf{in}\ \ \ldots\end{aligned}$$

By repeating this process, we can limit the places in the expression that we can encounter source ticks, which makes it easier to make sure that we they do not get into the way of GHC's optimizations.

5.3 Cause Tracking in Cmm

After the Core optimizations have taken place, GHC will translate the Core program into a high-level assembly language called Cmm (related to C-- [21]). The inner loop of our factorial function would be implemented by the following annotated Cmm code:

```
Main.$wgo_entry()
{
A:  src<fac.hs:5:15−17>, ...
    ctx<B> ctx<C>
    if (Sp−16 < SpLim) goto B; else goto C;
B:  R1 = Main.$wgo_closure;
    call stg_gc_fun(R2, R1);
C:  ctx<D> ctx<E>
    _tmp = R2;
    if (_tmp != 20) goto D; else goto E;
    [...]
}
```

While Cmm borrows conventions from C, it is quite a low-level language. We can see that all code has been broken down into blocks that are jumped to directly – and that the stack check that might eventually cause our program to fail is now spelled out explicitly.

To track source code annotations, we again insert meta-statements that "color" the code following it. However note that in contrast to functional code, Cmm does not have a recursive structure that we can exploit: Without further annotations, it would be unclear whether the source annotation in block A should apply to, for example, block C as well. This is a special instance of the problem mentioned in Section 4.4: We want to be able to follow control flow backwards in order to maximize the annotations we are able to "see". We solve this by introducing special "context" annotations that declares the containing block to be the static context of the second block, which is therefore allowed to inherit all annotations.

5.4 DWARF

The final step for the compiler is to translate the Cmm code into native machine code. As our goal is to minimize profiling overhead, we want to generate the same code as if our annotations had not been present in the first place. To achieve this, we generate DWARF debugging information [1], which is a standard for object file meta-data explaining structure and functionality of machine code. Adhering to a known industry standard has the big advantage that we can use existing infrastructure and tools to retain source code relations even after linking and relocation. At the point of writing we support generation of DWARF for both GHC's own native code back-end as well as the LLVM back-end [11, 27].

5.5 Sampling

At this point, we have enough information to track any piece of executable code back to at least one position in the source or library code – and typically many more due to inlining, as explained in Section 4.4. This means that all we require in order to profile a program is an approximate map of positions in the executable to costs. The standard process of obtaining such a map is called sampling: We stop the program at periods proportional to its resource usage, and inspect the program state to find clues as to what lead to the resource hunger of the program.

Currently we implement the following sampling methods:

1. By hardware counters: Such counters exist in all modern CPUs and allow interrupting the program whenever a certain processor action has been executed a certain number of times. Allows sampling by cycles, branch mis-predictions or cache misses.
2. By heap allocation: Programs compiled with GHC request heap memory from the run time system in certain "chunks" of moderate size. By sampling program locations as they requested blocks, we can easily identify hot allocation spots.
3. By heap retention: On each major garbage collection, the GHC run-time system can build a map of all closure types that remain on the heap [25]. As closure types correspond directly with code pointers, we can simply use these as sampling data. The advantage of this approach over standard memory profiles is that we can not only track space usage of raw data, but also of delayed calculations waiting for execution (thunks).

Even though it is normally offered by profiling frameworks, we do not yet offer simple timer-based sampling. This is because hardware counter profiling is strictly more flexible where available. We plan to correct this in future to improve portability.

Collected sampling data gets streamed to the hard disk using GHC's event-log format [9]. To keep run-time overheads low we perform only minimal pre-processing of the samples, leaving most of the heavy-lifting to the analysis tools.

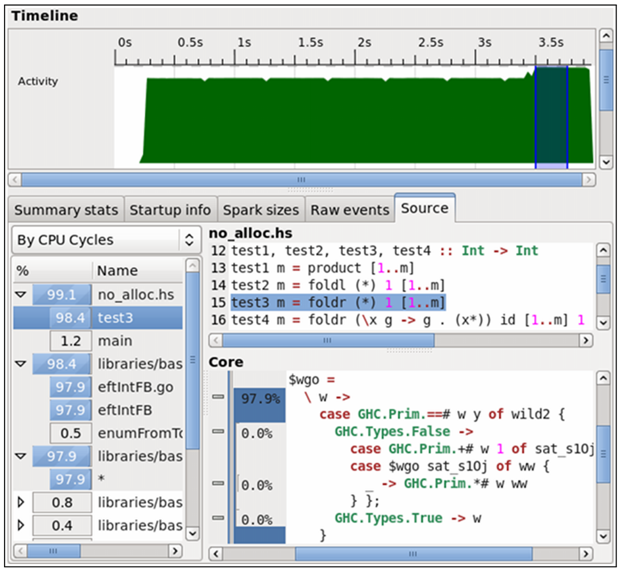
5.6 Presentation

The complicated and sometimes unpredictable nature of profiles makes it especially important to communicate profiling data well to the user. We have extended ThreadScope [9] to allow analysis of our profiles within the existing graphical user interface (see Figure 4). Apart from its flexible event-log back-end, this approach allows us to re-use user interface elements such as the CPU activity time-line.

A full explanation of the user interface design is beyond the scope of this paper, however our main idea is to focus on one source code element at a time, which can be selected from a hierarchical list on the left. We show a rough estimate of how much cost was associated with the cause in question. Note that due to optimizations, significant portions of the cost might be associated with *multiple* source code elements at the same time.

Such overlap might be slightly surprising to users of classic profiling tools. Yet note that it represents the situation *truthfully*, as program behavior is rarely just determined by a singular program element. On the contrary, by exposing such overlap we can identify which program elements interacted at the critical points of the program's execution. Notice that in Figure 4, the user interface highlights multiple entries to indicate source interaction. This is meant to suggest visiting their definitions in order to learn more about how the code was translated.

Finally, the main view is dedicated to understanding the selected source code element and its translation. We show both the original source code for reference, as well as the fully-optimized Core [17] code to give advanced users a way to follow the actions of the optimization passes.



%	Name
99.1	no_alloc.hs
98.4	test3
1.2	main
98.4	libraries/bas
97.9	eftIntFB.go
97.9	eftIntFB
0.5	enumFromTc
97.9	libraries/bas
97.9	*
0.8	libraries/bas
0.4	libraries/bas

```
test1, test2, test3, test4 :: Int -> Int
test1 m = product [1..m]
test2 m = foldl (*) 1 [1..m]
test3 m = foldr (*) 1 [1..m]
test4 m = foldr (\x g -> g . (x*)) id [1..m] 1
```

```
        $wgo =
97.9%     \ w ->
            case GHC.Prim.==# w y of wild2 {
0.0%          GHC.Types.False ->
                case GHC.Prim.+# w 1 of sat_s10j
                case $wgo sat_s10j of ww {
0.0%              _ -> GHC.Prim.*# w ww
                } };
0.0%          GHC.Types.True -> w
            }
```

Figure 4. Profile analysis in ThreadScope

6. Evaluation

We have tested out profiling solution on a variety of programs, ranging from small example programs with obvious performance problems to large applications with well-hidden inefficiencies. We generally compile as much code as possible with annotations present, which means that we end up with full debug information for not only the application, but also libraries as well as parts of the run-time system.

Making claims about accuracy and usefulness of profiling results is a bit harder, as we can not directly verify them. In this paper we will split the evaluation into three parts: Overheads and side-effects of debug information generation, overheads and possible inaccuracies of the different sampling methods, and finally usefulness of the resulting profile.

6.1 Base Overhead

First let us investigate "set-up" costs that we even have to pay when not doing any actual profiling. We will for example use the event-logging framework, which contributes some cost due to tracking the program state and garbage collections alone [9]. Furthermore, we have to verify that compiling while tracking source code links does not result in significantly different object code and therefore performance.

In Figure 5 we have collected benchmark results for running the `nofib` benchmark suite [15] using different configurations. We use the GHC master development version as of 30-05-2013 as our baseline. "Ev" will mark builds with event-logging enabled, and "Ann" builds use our modified annotation-tracking compiler. All benchmarks were using the "slow" data sets on a single core of an Intel Xeon® CPU at 3.1 GHz. In order to focus on significant results and make the presentation more compact, we discarded results under 0.4 seconds run-time, and omitted them from the table if under 1 second (except notable outliers).

First consider the overhead of event-logging: Clearly the overhead is minor, showing +0.7% and -0,1% mean change respectively. The amount of spread among the benchmarks is a bit surprising though, with especially the relatively simple "tak" benchmark producing large outliers for all three configurations.

More importantly, we see that compiling with annotations clearly has side-effects as well, with for example the "reverse-

	Base [s]	Ev [%]	Ann [%]	Ann+Ev [%]
atom	1.24	+0.6	+2.1	+2.3
binary-trees	11.52	+1.7	-0.3	+4.2
constraints	2.25	-0.7	+4.2	+4.9
exp3-8	1.84	+2.6	+0.2	+1.3
fannkuch-redux	47.59	+0.1	+2.2	+3.2
fasta	3.81	+1.2	+2.1	+1.3
integer	1.53	+2.1	-0.5	+3.0
k-nucleotide	51.88	+0.4	+3.8	-0.2
kahan	0.86	-1.6	-6.1	-5.8
lcss	1.35	+0.9	+0.4	+0.6
n-body	11.31	-0.4	-0.7	-0.4
pidigits	1.15	+0.9	-0.2	+1.6
reverse-complement	0.77	+3.6	+14.3	+13.0
solid	0.41	+4.9	+4.9	+0.0
tak	0.60	-8.0	+22.6	+13.0
Min		-8.0	-6.1	-5.8
Max		+4.9	+22.6	+13.0
Geometric Mean		+0.7	+3.2	+3.1

Figure 5. Runtime overhead of event-logging and annotations

	Ann+Ev [s]	Cycle [%]	Alloc [%]	Heap [%]
atom	1.27	+4.4	+0.6	-0.3
binary-trees	12.01	+5.7	-0.6	+79.1
constraints	2.36	+5.4	+0.2	+15.3
exp3-8	1.87	+8.0	+0.7	+0.2
fannkuch-redux	49.13	+8.4	-0.4	+0.0
fasta	3.86	+11.5	+0.0	+0.4
integer	1.57	+9.7	-0.8	+0.0
k-nucleotide	51.78	+10.2	-0.5	-4.1
lcss	1.36	+3.5	+0.4	+4.3
n-body	11.26	+10.7	+0.0	+0.6
paraffins	0.70	+3.1	+2.0	+10.8
pidigits	1.17	+14.6	+0.0	+0.5
primes	0.69	+7.2	-2.9	-0.6
reverse-complement	0.87	+8.3	+5.1	+0.7
spectral-norm	8.26	+9.0	-0.4	+0.0
Min		+3.1	-2.9	-4.1
Max		+14.6	+5.1	+79.1
Geometric Mean		+8.0	+0.3	+3.6

Figure 6. Runtime overhead of sampling

complement" benchmark losing about 14% of performance. Sadly, this means that our aim to be as unobtrusive to the compilation process as possible has not been completely successful, as we have to conclude that some program transformations produced different results with out annotations present. On the other hand, 3.2% mean overhead means that no important optimization has gone missing, and that the annotated program is still a valid target for profiling.

Note that we have ignored one source of overhead at this point: The cost for relocating and copying the contents of the debug sections to the event log. This often is a considerable amount of data, as it maps every single piece of machine code to both the source code as well as intermediate Core code. For example, the meta-data for profiling GHC weights around 350MB (about 5.2 million records covering 1.8 million code blocks) which take several seconds to be copied to the hard disk. However as start-up cost does not impact the validity of profiling results, we have performed all benchmarks with "stripped" executables.

6.2 Sampling Overhead

Next we consider the cost for doing actual performance measurements – taking samples at points of interest and writing them out to the hard disk. As mentioned before, we support three separate ways of sample collection: Hardware counter profiling, heap retention profiling and heap allocation profiling. For the measurements in Figure 6, we configured all three to yield about 1000 sample values per second, which is enough data to allow meaningful analysis down to time-slices of about a second.

The "Cycle" configuration uses Linux' `perf_events` hardware counter kernel interface to take samples at an interval of 100,000 cycles. This method of sample collection is extremely robust, as we can delegate the actual profiling task to the operating system. From the program's point of view, we just have to flush a memory map at regular intervals. The overhead will therefore be paid mostly in context switches to the operating system, which means that outside of timings, our program executes exactly as it would have without profiling. This means that while 8% mean overhead is not a stellar result, it is enough to support meaningful profiling.

The "Alloc" sampling method takes one sample every 4096 bytes of allocation, which we achieve through a relatively cheap run-time system change. As a result, we see that the overhead is almost not noticeable across all tests.

The "Heap" configuration on the other hand works quite differently from the first two – we are basically using GHC's standard mechanism for heap retention profiling [25] for our purposes. This will take snapshots of the heap at major garbage collection a minimum of 0.1 seconds apart. As a result, we can see in Figure 6 that the overhead varies a lot, depend on the heap size and the frequency of garbage collection. Note that the overhead does not reduce the quality of the memory profile, as the amount of heap retention will not change due to our analysis.

6.3 In Practice

The profiling work-flow with our tool will generally consist first of finding points in the profile that do not match expectations. It is in fact quite common to see obvious "hot spots" which provide obvious starting points. For example, we can see in Figure 4 that 98% of the time is spent in one block of Core code.

Due to optimizations, these blocks might often have quite complicated relations to the source code, so the second step will consist of investigating what caused the program behavior around the focal point. Solutions to performance problems then often come from considering how library code and optimizations interacted with the application's source code.

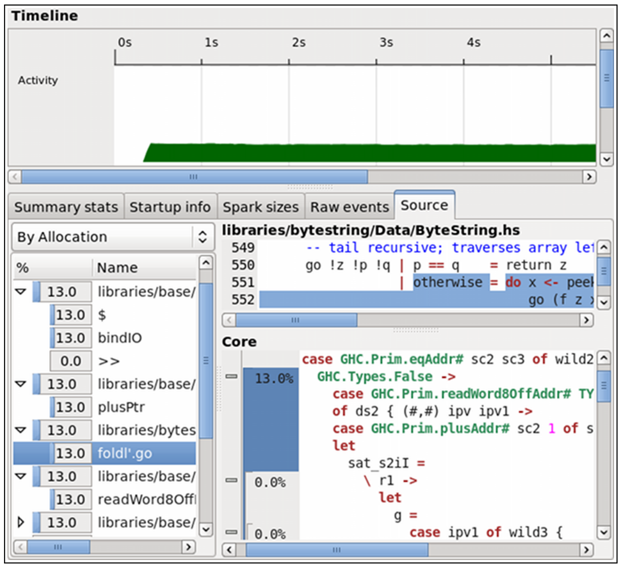
6.3.1 Example 1: String Escaping

To illustrate how this process would look like for real-life programs, let us walk through the performance analysis of some example programs. First, let us look at a program that escapes a byte-string using the blaze-builder library[2]:

```
escape :: ByteString → Builder
escape = B.foldl' f mempty
    where f b c = b `mappend` escape1 c
```

The idea is clear: With escape1 taking care of escaping an individual character, we use **foldl**' to apply it to every byte in the string and combine results using mappend. All functions and libraries used have a great reputation for speed, so it seems reasonable to expect this approach to yield a fast program.

[2] This program was submitted to StackExchange's code review section by Joey Adams (http://codereview.stackexchange.com/questions/9998). The blaze-builder library was authored by Jaspar Van der Jeugt and Simon Meier.



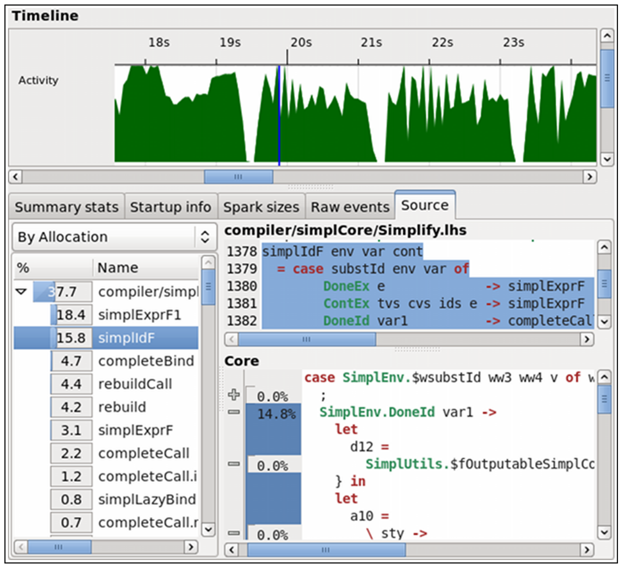
%	Name
13.0	libraries/base/
13.0	$
13.0	bindIO
0.0	>>
13.0	libraries/base/
13.0	plusPtr
13.0	libraries/bytes
13.0	foldl'.go
13.0	libraries/base/
13.0	readWord8Offl
13.0	libraries/base/

libraries/bytestring/Data/ByteString.hs

```
      -- tail recursive; traverses array le
      go !z !p !q | p == q     = return z
                  | otherwise = do x <- peel
                                  go (f z
```

```
13.0%  case GHC.Prim.eqAddr# sc2 sc3 of wild2
         GHC.Types.False ->
           case GHC.Prim.readWord8OffAddr# TY
           of ds2 { (#,#) ipv ipv1 ->
           case GHC.Prim.plusAddr# sc2 1 of s
           let
             sat_s2iI =
0.0%           \ r1 ->
                 let
                   g =
0.0%                 case ipv1 of wild3 {
```

Figure 7. Profile of string escaping



%	Name
37.7	compiler/simpl
18.4	simplExprF1
15.8	simplIdF
4.7	completeBind
4.4	rebuildCall
4.2	rebuild
3.1	simplExprF
2.2	completeCall
1.2	completeCall.i
0.8	simplLazyBind
0.7	completeCall.r

compiler/simplCore/Simplify.lhs

```
simplIdF env var cont
  = case substId env var of
        DoneEx e                -> simplExprF
        ContEx tvs cvs ids e -> simplExprF
        DoneId var1             -> completeCal
```

```
        case SimplEnv.$wsubstId ww3 ww4 v of w
0.0%      ;
14.8%     SimplEnv.DoneId var1 ->
            let
              d12 =
0.0%            SimplUtils.$fOutputableSimplCc
            } in
            let
              a10 =
0.0%            \ sty ->
```

Figure 8. Profile of the Glasgow Haskell Compiler

However in practice we find quite the opposite: The activity timeline in Figure 7 shows that the program is actually really slow, spending just 20% of time on productive work and the rest on garbage collection! Searching for hot spots, we unsurprisingly find a specialized version of the worker of the **foldl**' function busily reading bytes from the input byte-string. However, it does not seem to be writing any output right away, instead constructing lambda closures that we can identify as having blaze-builders's "Builder" type. Using **foldl**' we have merely forced the construction of a chain of builder closures instead of the result byte-string itself!

Unfortunately, this is an instance where forcing strictness does not help. We need to give blaze-builder control of the outer loop:

```
escape = mconcat $ map escape1 $ B.unpack c
```

By using lists, we can now take advantage of the list fusion optimization [22], which eliminates intermediate closures and indeed yields a much faster loop for our example.

6.3.2 Example 2: The Glasgow Haskell Compiler

The Glasgow Haskell Compiler itself is written in Haskell, and has been under active development by Haskell experts for a long time. Therefore it simultaneously represents one of the largest and best-optimized Haskell programs available today.

This is clearly reflected in the profile (Figure 8): In contrast to previous examples, cost is spread evenly across hundreds of functions and dozens of modules. However, using our tool we can still pick out inefficiencies: For example we see a surprising amount of allocation in the simplification phase between the functions "simplExprF1", "simplIdF" and "completeCall". Going further, we find that the code is actually allocating a number of "Outputable.SDoc" thunks, a data structure used for pretty-printing code. By selecting the thunks, we can trace them back to the definition of "completeCall", which looks roughly as follows:

```
completeCall :: [...] → SimplCont → SimplM [...]
completeCall [...] cont = do
  dflags ← getDynFlags
  [...]
  when (dopt Opt_D_dump_inlinings dflags) $
    [...] (ppr cont) [...]
```

Clearly the "(ppr cont)" part only gets used in the unlikely event that we request it on the command line using `-ddump-inlinings`. So why does GHC allocate thunks unconditionally?

Another look into the Core code reveals that the innocent-looking call to "getDynFlags" has the effect of wrapping the whole function body into a lambda, as "SimplM" is a reader monad. This in turn makes GHC think that it would be a good idea to apply the full-laziness transformation: Float out thunks for every sub-expressions that does not depend on the "dflags". This would be a very good idea if we were to call "completeCall" multiple times using different "dflags" every time – yet unfortunately this particular lambda will actually be evaluated exactly once.

One way to fix this is to force an earlier dependency on "dflags" using **seq**, for example in "simplExprF1". This indeed eliminates the allocations in question, but unfortunately does not affect program performance enough to produce measurable speed-ups.

7. Literature

The problem of debugging and profiling optimized code has been addressed in a number of ways before, with Appel et al [2] offering early thoughts on how to do it in functional languages. The later works by Sansom et al [24, 25] as well as Röjemo and Runciman [23] built properly fleshed out profiling frameworks for Haskell, but still dealt with causality and optimizations in a relatively informal manner. Furthermore, a recent paper by Perera et al [16] tracks effects throughout a functional program in a similar manner. However, they seem to take a formal approach in deriving cause and effects, and do not consider optimizations nor applications in profiling.

In the world of imperative languages, most early work has been on allowing meaningful debugging of optimized code. The work of Brooks et al [3] already introduces the idea that we might have to give up a one-to-one correspondence between source and output code, as well as outlining user interface principles that have parallels with the ones we are employing. Modern compilers such as LLVM [11] follow their tradition when tracking source code relations for optimized low-level code. Kaneshiro et al [10] address profiling of optimized code, employing similar ideas by having statements gather annotations when getting moved out of the context of loops.

8. Future Work

While we are quite confident that our profiling framework is already useful in its current state, we had to compromise on a number of issues. Firstly, while this work was inspired by the design decisions involved in building our profiler as an extension of the Glasgow Haskell Compiler, we have not formally checked that the translation fully adheres to what we can derive as the optimal annotation strategy. Especially when dealing with user rules, we probably end up with more annotations than needed.

On the technical side, we would really like to support stack tracing in some form, as it would allow us to say more about "big picture" profiling issues. Additionally, we believe that we could increase the usefulness of our profiling even more by not only tracking the effects of source code, but also of individual compiler optimizations. Cross-checking this data against both source code relations and performance data could give deeper insights into the effects of the decisions of the compiler.

9. Conclusion

Our work aims at the heart of the supposed divide between high-level languages and high-performance programming: The notion that we cannot possibly reason about both at the same time. While optimizations indeed add complexity to the task of following causality, we believe that careful analysis of compiler passes can yield annotations that are still valid after aggressive transformations. With good profiler and user interface design, we can use these to uncover performance problems that would otherwise seem impenetrable.

Acknowledgments

We would like to thank Simon Marlow as well as the anonymous reviewers from for their helpful feedback on various versions of this paper. We would also like to thank Microsoft Research for their sponsorship through the PhD Scholarship Programme.

References

[1] DWARF debugging information format, version 2. Technical report, UNIX International Programming Languages SIG, 1993.

[2] A. W. Appel, B. F. Duba, and D. B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, Nov 1988.

[3] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *Proceedings of the SIGPLAN conference on programming language design and implementation*, PLDI '92, pages 1–11, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on functional programming*, pages 315–326, New York, NY, USA, 2007. ACM.

[5] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, Mar. 2009.

[6] A. Gill and C. Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[7] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the symposium on compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[9] D. Jones, Jr., S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM.

[10] S. Kaneshiro and T. Shindo. Profiling optimized code: A profiling system for an hpf compiler. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 469–473, Washington, DC, USA, 1996. IEEE Computer Society.

[11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on code generation and optimization*, CGO '04, pages 75–86, Washington, DC, USA, March 2004. IEEE Computer Society.

[12] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM.

[13] D. Lewis. *Counterfactuals*. Blackwell Publishers, Oxford, 1973. ISBN 978-0-631-22425-9.

[14] S. Marlow. Why can't I get a stack trace? Sept. 2012. URL `haskell.org/haskellwiki/HaskellImplementorsWorkshop/2012`.

[15] W. Partain. The `nofib` benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

[16] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 365–376, New York, NY, USA, 2012. ACM.

[17] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3): 3 – 47, Sep 1998. 6th European Symposium on Programming.

[18] S. L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell compiler inliner. *Journal of Functional Programming*, 12(5):393–434, July 2002.

[19] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, JFIT '93, March 1993.

[20] S. L. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on functional programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM.

[21] S. L. Peyton Jones, T. Nordin, and D. Oliva. C–: A portable assembly language. In *Implementation of Functional Languages*, pages 1–19. Springer, 1998.

[22] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, Haskell '01, pages 203–233, Sept. 2001.

[23] N. Röjemo and C. Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 34–41, New York, NY, USA, 1996. ACM.

[24] P. M. Sansom. *Execution Profiling for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1994.

[25] P. M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, Mar. 1997.

[26] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997.

[27] D. A. Terei and M. M. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.

[28] J. Tibell. State of Haskell, 2011 Survey, Aug. 2011. URL `http://blog.johantibell.com/2011/08/results-from-state-of-haskell-2011.html`.

[29] P. Wadler. The essence of functional programming. In *Proceedings of the 19th symposium on principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.