Selective Lambda Lifting

ANONYMOUS AUTHOR(S)

Lambda lifting is a well-known transformation, traditionally employed for compiling functional programs to supercombinators. However, more recent abstract machines for functional languages like OCaml and Haskell tend to do closure conversion instead for direct access to the environment, so lambda lifting is no longer necessary to generate machine code.

We propose to revisit selective lambda lifting in this context as an optimising code generation strategy and conceive heuristics to identify beneficial lifting opportunities. We give a static analysis for estimating impact on heap allocations of a lifting decision. Performance measurements of our implementation within the Glasgow Haskell Compiler on a large corpus of Haskell benchmarks suggest modest speedups.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Functional languages*; *Procedures, functions and subroutines*;

Additional Key Words and Phrases: Haskell, Lambda Lifting, Spineless Tagless G-machine, Compiler Optimization

ACM Reference Format:
Anonymous Author(s). 2019. Selective Lambda Lifting. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2019), 16 pages.

1 INTRODUCTION

The ability to define nested auxiliary functions referencing variables from outer scopes is essential when programming in functional languages. Take this Haskell function as an example:

```
f a 0 = a
f a n = f (g (n `mod` 2)) (n - 1)
  where
    g 0 = a
    g n = 1 + g (n - 1)
```

To generate code for nested functions like g, a typical compiler either applies lambda lifting or closure conversion. The Glasgow Haskell Compiler (GHC) chooses to do closure conversion [Peyton Jones 1992]. In doing so, it allocates a closure for g on the heap, with an environment containing an entry for a. Now imagine we lambda lifted g before closure conversion:

```
g↑ a 0 = a
g↑ a n = 1 + g↑ a (n - 1)

f a 0 = a
f a n = f (g↑ a (n `mod` 2)) (n - 1)
```

The closure for g and the associated heap allocation completely vanished in favour of a few more arguments at the call site! The result looks much simpler. And indeed, in concert with the other optimisations within GHC, the above transformation makes f effectively non-allocating, resulting in a speedup of 50%.

So should we just perform this transformation on any candidate? We have to disagree. Consider what would happen to the following program:

```
f :: [Int] → [Int] → Int → Int
f a b 0 = a
```

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

```
f a b 1 = b
f a b n = f (g n) a (n `mod` 2)
  where
    g 0 = a
    g 1 = b
    g n = n : h
      where
        h = g (n - 1)
```

Because of laziness, this will allocate a thunk for h. Closure conversion will then allocate an environment for h on the heap, closing over g. Lambda lifting yields:

```
g↑ a b 0 = a
g↑ a b 1 = b
g↑ a b n = n : h
  where
    h = g↑ a b (n - 1)

f a b 0 = a
f a b 1 = b
f a b n = f (g↑ a b n) a (n `mod` 2)
```

The closure for g is gone, but h now closes over n, a and b instead of n and g. Worse, for a single allocation of g's closure, we get two additional allocations of h's closure on the recursive code path! Apart from making f allocate 10% more, this also incurs a slowdown of more than 10%.

So lambda lifting is sometimes beneficial, and sometimes harmful: we should do it selectively. This work is concerned with identifying exactly *when* lambda lifting improves performance, providing a new angle on the interaction between lambda lifting and closure conversion. These are our contributions:

- We derive a number of heuristics fueling the lambda lifting decision from concrete operational deficiencies in section 3.
- Integral to one of the heuristics, in section 4 we provide a static analysis estimating *closure growth*, conservatively approximating the effects of a lifting decision on the total allocations of the program.
- We implemented our lambda lifting pass in the Glasgow Haskell Compiler as part of its optimisation pipeline, operating on its Spineless Tagless G-machine (STG) language. The decision to do lambda lifting this late in the compilation pipeline is a natural one, given that accurate allocation estimates aren't easily possible on GHC's more high-level Core language.
- We evaluate our pass against the `nofib` benchmark suite (section 6) and find that our static analysis soundly predicts changes in heap allocations. The measurements confirm the reasoning behind our heuristics in section 3.

Our approach builds on and is similar to many previous works, which we compare to in section 7.

2 OPERATIONAL BACKGROUND

Typically, the choice between lambda lifting and closure conversion for code generation is mutually exclusive and is dictated by the targeted abstract machine, like the G-machine [Kieburtz 1985] or the Spineless Tagless G-machine [Peyton Jones 1992], as is the case for GHC.

Let's clear up what we mean by doing lambda lifting before closure conversion and the operational effect of doing so.

$$
\begin{array}{rlll}
\text{Variables} & f, g, x, y \in \mathrm{Var} & & \\
\text{Expressions} & e \in \mathrm{Expr} ::= & x & \text{Variable} \\
& & \mid\ f\ \overline{x} & \text{Function call} \\
& & \mid\ \textbf{let}\ b\ \textbf{in}\ e & \text{Recursive } \textbf{let} \\
\text{Bindings} & b \in \mathrm{Bind} ::= & \overline{f = r} & \\
\text{Right-hand sides} & r \in \mathrm{Rhs} ::= & \lambda \overline{x} \to e & \\
\text{Programs} & p \in \mathrm{Prog} ::= & \overline{f\ \overline{x} = e};\ e' &
\end{array}
$$

Fig. 1. An STG-like untyped lambda calculus

2.1 Language

Although the STG language is tiny compared to typical surface languages such as Haskell, its definition [Marlow and Jones 2004] still contains much detail irrelevant to lambda lifting. This section will therefore introduce an untyped lambda calculus that will serve as the subject of optimisation in the rest of the paper.

2.1.1 Syntax. As can be seen in fig. 1, we extended untyped lambda calculus with **let** bindings, just as in Johnsson [1985]. Inspired by STG, we also assume A-normal form (ANF) [Sabry and Felleisen 1993]:

- Every lambda abstraction is the right-hand side of a **let** binding
- Arguments and heads in an application expression are all atomic (e.g., variable references)

Throughout this paper, we assume that variable names are globally unique. Similar to Johnsson [1985], programs are represented by a group of top-level bindings and an expression to evaluate.

Whenever there's an example in which the expression to evaluate is not closed, assume that free variables are bound in some outer context omitted for brevity. Examples may also compromise on adhering to ANF for readability (regarding giving all complex subexpressions a name, in particular), but we will point out the details if need be.

2.1.2 Semantics. Since our calculus is a subset of the STG language, its semantics follows directly from Marlow and Jones [2004].

An informal treatment of operational behavior is still in order to express the consequences of lambda lifting. Since every application only has trivial arguments, all complex expressions had to be bound to a **let** in a prior compilation step. Consequently, heap allocation happens almost entirely at **let** bindings closing over free variables of their RHSs, with the exception of intermediate partial applications resulting from over- or undersaturated calls.

Put plainly: If we manage to get rid of a **let** binding, we get rid of one source of heap allocation since there is no closure to allocate during closure conversion.

2.2 Lambda Lifting vs. Closure Conversion

The trouble with nested functions is that nobody has come up with concrete, efficient computing architectures that can cope with them natively. Compilers therefore need to rewrite local functions in terms of global definitions and auxiliary heap allocations.

One way of doing so is in performing *closure conversion*, where references to free variables are lowered as field accesses on a record containing all free variables of the function, the *closure environment*. The environment is passed as an implicit parameter to the function body, which in turn is insensitive to lexical scope and can be floated to top-level. After this lowering, all functions are then regarded as *closures*: A pair of a code pointer and an environment.

$$\begin{array}{l}\textbf{let } f = \lambda a\ b \to \ldots x \ldots y \ldots \\ \textbf{in } f\ 4\ 2\end{array} \quad \overset{\text{CC } f}{\Longrightarrow} \quad \begin{array}{l}\textbf{data } EnvF = EnvF\ \{x :: Int, y :: Int\} \\ f_\star\ env\ a\ b = \ldots x\ env \ldots y\ env \ldots; \\ \textbf{let } f = (f_\star, EnvF\ x\ y) \\ \textbf{in } (fst\ f)\ (snd\ f)\ 4\ 2\end{array}$$

Closure conversion leaves behind a heap-allocated **let** binding for the closure[1].

Compare this to how *lambda lifting* gets rid of local functions. Johnsson [1985] introduced it for efficient code generation of lazy functional languages to G-machine code [Kieburtz 1985]. Lambda lifting converts all free variables of a function body into parameters. The resulting function body can be floated to top-level, but all call sites must be fixed up to include its former free variables.

$$\begin{array}{l}\textbf{let } f = \lambda a\ b \to \ldots x \ldots y \ldots \\ \textbf{in } f\ 4\ 2\end{array} \quad \overset{\text{LL } f}{\Longrightarrow} \quad \begin{array}{l}f_\uparrow\ x\ y\ a\ b = \ldots x \ldots y \ldots; \\ f_\uparrow\ x\ y\ 4\ 2\end{array}$$

The key difference to closure conversion is that there is no heap allocation at f's former definition site anymore. But earlier we saw examples where doing this transformation does more harm than good, so the plan is to transform worthwhile cases with lambda lifting and leave the rest to closure conversion.

3 WHEN TO LIFT

Lambda lifting is always a sound transformation. The challenge is in identifying *when* it is beneficial to apply. This section will discuss operational consequences of our lambda lifting pass, clearing up the requirements for our transformation defined in section 5. Operational considerations will lead to the introduction of multiple criteria for rejecting a lift, motivating a cost model for estimating impact on heap allocations.

3.1 Syntactic Consequences

Deciding to lambda lift a binding **let** $f = \lambda a\ b\ c \to e$ **in** e' where x and y occur free in e, has the following consequences:

- **(S1)** It replaces the **let** expression by its body.
- **(S2)** It creates a new top-level definition $f_\uparrow$.
- **(S3)** It replaces all occurrences of f in e' and e by an application of $f_\uparrow$ to its former free variables x and y[2].
- **(S4)** The former free variables x and y become parameters of $f_\uparrow$.

3.2 Operational Consequences

We now ascribe operational symptoms to combinations of syntactic effects. These symptoms justify the derivation of heuristics which will decide when *not* to lift.

Argument occurrences. Consider what happens if f occurred in the **let** body e' as an argument in an application, as in $g\ 5\ x\ f$. (S3) demands that the argument occurrence of f is replaced

[1]Note that the pair and the *EnvF* can and will be combined into a single heap object in practice.

[2]This will also need to give a name to new non-atomic argument expressions mentioning f. We'll argue in section 3.2 that there is hardly any benefit in allowing these cases.

by an application expression. This, however, would yield the syntactically invalid expression $g\ 5\ x\ (f_\uparrow\ x\ y)$. ANF only allows trivial arguments in an application!

Thus, our transformation would have to immediately wrap the application in a partial application: $g\ 5\ x\ (f_\uparrow\ x\ y) \Longrightarrow \textbf{let}\ f' = f_\uparrow\ x\ y\ \textbf{in}\ g\ 5\ x\ f'$. But this just reintroduces at every call site the very allocation we wanted to eliminate through lambda lifting! Therefore, we can identify a first criterion for non-beneficial lambda lifts:

(C1) Don't lift binders that occur as arguments

A welcome side-effect is that the application case of the transformation in section 5 becomes much simpler: The complicated **let** wrapping becomes unnecessary.

Closure growth. (S1) means we don't allocate a closure on the heap for the **let** binding. On the other hand, (S3) might increase or decrease heap allocation, which can be captured by a metric we call *closure growth*. This is the essence of what guided our examples from the introduction. We'll look into a simpler example:

$$\begin{array}{l}\textbf{let}\ f = \lambda a\ b \to \ldots x \ldots y \ldots \\ \quad g = \lambda d \to f\ d\ d + x \\ \textbf{in}\ g\ 5\end{array} \xrightarrow{\text{lift } f} \begin{array}{l} f_\uparrow\ x\ y\ a\ b = \ldots; \\ \textbf{let}\ g = \lambda d \to f_\uparrow\ x\ y\ d\ d + x \\ \textbf{in}\ g\ 5\end{array}$$

Should f be lifted? Just counting the number of variables occurring in closures, the effect of (S1) saved us two slots. At the same time, (S3) removes f from g's closure (no need to close over the top-level constant $f_\uparrow$), while simultaneously enlarging it with f's former free variable y. The new occurrence of x doesn't contribute to closure growth, because it already occurred in g prior to lifting. The net result is a reduction of two slots, so lifting f seems worthwhile. In general:

(C2) Don't lift a binding when doing so would increase closure allocation

Note that this also includes handling of **let** bindings for partial applications that are allocated when GHC spots an undersaturated call to a known function.

Estimation of closure growth is crucial to achieving predictable results. We discuss this further in section 4.

Calling convention. (S4) means that more arguments have to be passed. Depending on the target architecture, this entails more stack accesses and/or higher register pressure. Thus

(C3) Don't lift a binding when the arity of the resulting top-level definition exceeds the number of available argument registers of the employed calling convention (e.g., 5 arguments for GHC on AMD64)

One could argue that we can still lift a function when its arity won't change. But in that case, the function would not have any free variables to begin with and could just be floated to top-level. As is the case with GHC's full laziness transformation, we assume that this already happened in a prior pass.

Turning known calls into unknown calls. There's another aspect related to (S4), relevant in programs with higher-order functions:

$$\begin{array}{l}\textbf{let}\ f = \lambda x \to 2 * x \\ \quad mapF = \lambda xs \to \textbf{case}\ xs\ \textbf{of} \\ \quad\quad (x : xs') \to \ldots f\ x \ldots mapF\ xs' \ldots \\ \quad\quad [\,] \to \ldots \\ \textbf{in}\ mapF\ [1\,..\,n]\end{array} \xrightarrow{\text{lift } mapF} \begin{array}{l} mapF_\uparrow\ f\ xs = \textbf{case}\ xs\ \textbf{of} \\ \quad (x : xs') \to \ldots f\ x \ldots mapF_\uparrow\ f\ xs' \ldots \\ \quad [\,] \to \ldots; \\ \textbf{let}\ f = \lambda x \to 2 * x \\ \textbf{in}\ mapF_\uparrow\ f\ [1\,..\,n]\end{array}$$

Here, there is a *known call* to f in $mapF$ that can be lowered as a direct jump to a static address [Marlow and Jones 2004]. This is similar to an early bound call in an object-oriented language.

After lifting *mapF*, f is passed as an argument to $mapF_\uparrow$ and its address is unknown within the body of $mapF_\uparrow$. For lack of a global points-to analysis, this unknown (i.e. late bound) call would need to go through a generic apply function [Marlow and Jones 2004], incurring a major slow-down.

(C4) Don't lift a binding when doing so would turn known calls into unknown calls

Sharing. Consider what happens when we lambda lift an updatable binding, like a thunk[3]:

$$\begin{array}{l}\textbf{let } t = \lambda \to x + y \\ \quad addT = \lambda z \to z + t \\ \textbf{in } map\ addT\ [1\,..\,n]\end{array} \quad \overset{\text{lift } t}{\Longrightarrow} \quad \begin{array}{l}t\ x\ y = x + y; \\ \textbf{let } addT = \lambda z \to z + t\ x\ y \\ \textbf{in } map\ addT\ [1\,..\,n]\end{array}$$

The addition within t prior to lifting will be computed only once for each complete evaluation of the expression. Compare this to the lambda lifted version, which will re-evaluate $t\ n$ times!

In general, lambda lifting updatable bindings or constructor bindings destroys sharing, thus possibly duplicating work in each call to the lifted binding.

(C5) Don't lift a binding that is updatable or a constructor application

4 ESTIMATING CLOSURE GROWTH

Of the criteria above, (C2) is quite important for predictable performance gains. It's also the most sophisticated, because it entails estimating closure growth.

4.1 Motivation

Let's revisit the example from above:

$$\begin{array}{l}\textbf{let } f = \lambda a\ b \to \ldots x \ldots y \ldots \\ \quad g = \lambda d \to f\ d\ d + x \\ \textbf{in } g\ 5\end{array} \quad \overset{\text{lift } f}{\Longrightarrow} \quad \begin{array}{l}f_\uparrow\ x\ y\ a\ b = \ldots x \ldots y \ldots; \\ \textbf{let } g = \lambda d \to f_\uparrow\ x\ y\ d\ d + x \\ \textbf{in } g\ 5\end{array}$$

We concluded that lifting f would be beneficial, saving us allocation of two free variable slots. There are two effects at play here. Not having to allocate the closure of f due to (S1) leads to a benefit once per activation. Simultaneously, each occurrence of f in a closure environment would be replaced by the free variables of its RHS. Replacing f by the top-level $f_\uparrow$ leads to a saving of one slot per closure, but the free variables x and y each occupy a closure slot in turn. Of these, only y really contributes to closure growth, because x was already free in g before.

This phenomenon is amplified whenever allocation happens under a lambda that is called multiple times (a *multi-shot* lambda [Sergey et al. 2014]), as the following example demonstrates:

$$\begin{array}{l}\textbf{let } f = \lambda a\ b \to \ldots x \ldots y \ldots \\ \quad g = \lambda d \to \\ \quad\quad \textbf{let } h = \lambda e \to f\ e\ e \\ \quad\quad \textbf{in } h\ x \\ \textbf{in } g\ 1 + g\ 2 + g\ 3\end{array} \quad \overset{\text{lift } f}{\Longrightarrow} \quad \begin{array}{l}f_\uparrow\ x\ y\ a\ b = \ldots x \ldots y \ldots; \\ \textbf{let } g = \lambda d \to \\ \quad \textbf{let } h = \lambda e \to f_\uparrow\ x\ y\ e\ e \\ \quad \textbf{in } h\ x \\ \textbf{in } g\ 1 + g\ 2 + g\ 3\end{array}$$

Is it still beneficial to lift f? Following our reasoning, we still save two slots from f's closure, the closure of g doesn't grow and the closure of h grows by one. We conclude that lifting f saves us one closure slot. But that's nonsense! Since g is called thrice, the closure for h also gets allocated three times relative to single allocations for the closures of f and g.

In general, h might be defined inside a recursive function, for which we can't reliably estimate how many times its closure will be allocated. Disallowing to lift any binding which is closed over under such a multi-shot lambda is conservative, but rules out worthwhile cases like this:

[3] Assume that all nullary bindings are memoised.

```
let f = λa b → ...x ... y ...                    f↑ x y a b = ...x ... y...;
    g = λd →                                     let g = λd →
      let h1 = λe → f e e                          let h1 = λe → f↑ x y e e
          h2 = λe → f e e + x + y                      h2 = λe → f↑ x y e e + x + y
      in h1 d + h2 d                               in h1 d + h2 d
in g 1 + g 2 + g 3                               in g 1 + g 2 + g 3
```

lift f ⟹

Here, the closure of h_1 grows by one, whereas that of h_2 shrinks by one, cancelling each other out. Hence there is no actual closure growth happening under the multi-shot binding g and f is good to lift.

The solution is to denote closure growth in $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ and account for positive closure growth under a multi-shot lambda by ∞.

4.2 Design

Applied to our simple STG language, we can define a function cl-gr (short for closure growth) with the following signature:

$$\text{cl-gr}^{\_}_{\_}(\_) : \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Expr} \to \mathbb{Z}_\infty$$

Given two sets of variables for added (superscript) and removed (subscript) closure variables, respectively, it maps expressions to the closure growth resulting from

- adding variables from the first set everywhere a variable from the second set is referenced
- and removing all closure variables mentioned in the second set.

There's an additional invariant: We require that added and removed sets never overlap.

In the lifting algorithm from section 5, cl-gr would be consulted as part of the lifting decision to estimate the total effect on allocations. Assuming we were to decide whether to lift the binding group $\overline{g}$ out of an expression $\textbf{let } \overline{g = \lambda \overline{x} \to e} \textbf{ in } e'$[4], the following expression conservatively estimates the effect on heap allocation of performing the lift:

$$\text{cl-gr}^{\alpha'(g_1)}_{\{\overline{g}\}}(\textbf{let } \overline{g = \lambda \alpha'(g_1)\, \overline{x} \to e} \textbf{ in } e') - \sum_i 1 + \left|\text{fvs}(g_i) \setminus \{\overline{g}\}\right|$$

The *required set* of extraneous parameters [Morazán and Schultz 2008] $\alpha'(g_1)$ for the binding group contains the additional parameters of the binding group after lambda lifting. The details of how to obtain it shall concern us in section 5. These variables would need to be available anywhere a binder from the binding group occurs, which justifies the choice of $\{\overline{g}\}$ as the subscript argument to cl-gr.

Note that we logically lambda lifted the binding group in question without fixing up call sites, leading to a semantically broken program. The reasons for that are twofold: Firstly, the reductions in closure allocation resulting from that lift are accounted separately in the trailing sum expression, capturing the effects of (S1): We save closure allocation for each binding, consisting of the code pointer plus its free variables, excluding potential recursive occurrences. Secondly, the lifted binding group isn't affected by closure growth (where there are no free variables, nothing can grow or shrink), which is entirely a symptom of (S3). Hence, we capture any free variables of the binding group in lambdas.

Following (C2), we require that this metric is non-positive to allow the lambda lift.

$$\boxed{\text{cl-gr}^{-}_{\_}(\_) : \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Expr} \to \mathbb{Z}_\infty}$$

$$\text{cl-gr}^{\varphi^+}_{\varphi^-}(x) = 0 \qquad \text{cl-gr}^{\varphi^+}_{\varphi^-}(f\ \overline{x}) = 0$$

$$\text{cl-gr}^{\varphi^+}_{\varphi^-}(\textbf{let}\ bs\ \textbf{in}\ e) = \text{cl-gr-bind}^{\varphi^+}_{\varphi^-}(bs) + \text{cl-gr}^{\varphi^+}_{\varphi^-}(e)$$

$$\boxed{\text{cl-gr-bind}^{-}_{\_}(\_) : \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Bind} \to \mathbb{Z}_\infty}$$

$$\text{cl-gr-bind}^{\varphi^+}_{\varphi^-}(\overline{f = r}) = \sum_i \text{growth}_i + \text{cl-gr-rhs}^{\varphi^+}_{\varphi^-}(r_i) \qquad \nu_i = \left|\text{fvs}(f_i) \cap \varphi^-\right|$$

$$\text{growth}_i = \begin{cases} \left|\varphi^+ \setminus \text{fvs}(f_i)\right| - \nu_i, & \text{if } \nu_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\boxed{\text{cl-gr-rhs}^{-}_{\_}(\_) : \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Rhs} \to \mathbb{Z}_\infty}$$

$$\text{cl-gr-rhs}^{\varphi^+}_{\varphi^-}(\lambda\overline{x} \to e) = \text{cl-gr}^{\varphi^+}_{\varphi^-}(e) * [\sigma, \tau] \qquad n * [\sigma, \tau] = \begin{cases} n * \sigma, & n < 0 \\ n * \tau, & \text{otherwise} \end{cases}$$

$$\sigma = \begin{cases} 1, & e \text{ entered at least once} \\ 0, & \text{otherwise} \end{cases} \qquad \tau = \begin{cases} 0, & e \text{ never entered} \\ 1, & e \text{ entered at most once} \\ 1, & \text{RHS bound to a thunk} \\ \infty, & \text{otherwise} \end{cases}$$

Fig. 2. Closure growth estimation

4.3 Implementation

The definition for cl-gr is depicted in fig. 2. The cases for variables and applications are trivial, because they don't allocate. As usual, the complexity hides in **let** bindings and its syntactic components. We'll break them down one layer at a time by delegating to one helper function per syntactic sort. This makes the **let** rule itself nicely compositional, because it delegates most of its logic to cl-gr-bind.

cl-gr-bind is concerned with measuring binding groups. Recall that the added and removed set never overlap. The growth component then accounts for allocating each closure of the binding group. Whenever a closure mentions one of the variables to be removed (i.e. φ^-, the binding group $\{\overline{g}\}$ to be lifted), we count the number of variables that are removed in ν and subtract them from the number of variables in φ^+ (i.e. the required set of the binding group to lift $\alpha'(g_1)$) that didn't occur in the closure before.

The call to cl-gr-rhs accounts for closure growth of right-hand sides. The right-hand sides of a **let** binding might or might not be entered, so we cannot rely on a beneficial negative closure growth to occur in all cases. Likewise, without any further analysis information, we can't say if a right-hand side is entered multiple times. Hence, the uninformed conservative approximation would be to return ∞ whenever there is positive closure growth in a RHS and 0 otherwise.

That would be disastrous for analysis precision! Fortunately, GHC has access to cardinality information from its demand analyser [Sergey et al. 2014]. Demand analysis estimates lower

[4]We only ever lift a binding group wholly or not at all, due to (C4) and (C1).

and upper bounds (σ and τ above) on how many times a RHS is entered relative to its defining expression.

Most importantly, this identifies one-shot lambdas ($\tau = 1$), under which case a positive closure growth doesn't lead to an infinite closure growth for the whole RHS. But there's also the beneficial case of negative closure growth under a strictly called lambda ($\sigma = 1$), where we gain precision by not having to fall back to returning 0.

One final remark regarding analysis performance: cl-gr operates directly on STG expressions. This means the cost function has to traverse whole syntax trees *for every lifting decision*.

We remedy this by first abstracting the syntax tree into a *skeleton*, retaining only the information necessary for our analysis. In particular, this includes allocated closures and their free variables, but also occurrences of multi-shot lambda abstractions. Additionally, there are the usual "glue operators", such as sequence (e.g., the case scrutinee is evaluated whenever one of the case alternatives is), choice (e.g., one of the case alternatives is evaluated mutually exclusively) and an identity (i.e. literals don't allocate). This also helps to split the complex **let** case into more manageable chunks.

5 TRANSFORMATION

The extension of Johnsson's formulation [Johnsson 1985] to STG terms is straight-forward, but it's still worth showing how the transformation integrates the decision logic for which bindings are going to be lambda lifted.

Central to the transformation is the construction of the minimal *required set* of extraneous parameters $\alpha(f)$ [Morazán and Schultz 2008] of a binding f.

It is assumed that all variables have unique names and that there is a sufficient supply of fresh names from which to draw. In fig. 3 we define a side-effecting function, lift, recursively over the term structure.

As its first argument, lift takes an Expander α, which is a partial function from lifted binders to their required sets. These are the additional variables we have to pass at call sites after lifting. The expander is extended every time we decide to lambda lift a binding, its role is similar to the E_f set in Johnsson [1985]. We write $\mathrm{dom}\,\alpha$ for the domain of α and $\alpha[x \mapsto S]$ to denote extension of the expander function, so that the result maps x to S and all other identifiers by delegating to α.

The second argument is the expression that is to be lambda lifted. A call to lift results in an expression that no longer contains any bindings that were lifted. The lifted bindings are emitted as a side-effect of the **let** case, which merges the binding group into the top-level recursive binding group representing the program. In a real implementation, this would be handled by carrying around a *Writer* effect. We refrained from making this explicit in order to keep the definition simple.

5.1 Variables

In the variable case, we check if the variable was lifted to top-level by looking it up in the supplied expander mapping α and if so, we apply it to its newly required extraneous parameters.

5.2 Applications

As discussed in section 3.2 when motivating (C1), handling function application correctly is a little subtle. Consider what happens when we try to lambda lift f in an application like $g\ f\ x$: Changing the variable occurrence of f to an application would be invalid because the first argument in the application to g would no longer be a variable.

Our transformation enjoys a great deal of simplicity because it crucially relies on the adherence to (C1), meaning we never have to think about wrapping call sites in partial applications binding the complex arguments.

$$\boxed{\text{lift\_(\_)}\colon \text{Expander} \to \text{Expr} \to \text{Expr}}$$

$$\text{lift}_\alpha(x) = \begin{cases} x, & x \notin \text{dom}\,\alpha \\ x\ \alpha(x), & \text{otherwise} \end{cases} \qquad \text{lift}_\alpha(f\ \overline{x}) = \text{lift}_\alpha(f)\ \overline{x}$$

$$\text{lift}_\alpha(\textbf{let}\ bs\ \textbf{in}\ e) = \begin{cases} \text{lift}_{\alpha'}(e), & bs \text{ is to be lifted as lift-bind}_{\alpha'}(bs) \\ \textbf{let}\ \text{lift-bind}_\alpha(bs)\ \textbf{in}\ \text{lift}_\alpha(e) & \text{otherwise} \end{cases}$$

where

$$\alpha' = \text{add-rqs}(bs, \alpha)$$

$$\boxed{\text{add-rqs}(\_, \_)\colon \text{Bind} \to \text{Expander} \to \text{Expander}}$$

$$\text{add-rqs}(\overline{f = r}, \alpha) = \alpha\left[\overline{f \mapsto \text{rqs}}\right]$$

where

$$\text{rqs} = \bigcup_i \text{expand}_\alpha(\text{fvs}(r_i)) \setminus \{\overline{f}\}$$

$$\boxed{\text{expand\_(\_)}\colon \text{Expander} \to \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var})}$$

$$\text{expand}_\alpha(V) = \bigcup_{x \in V} \begin{cases} \{x\}, & x \notin \text{dom}\,\alpha \\ \alpha(x), & \text{otherwise} \end{cases}$$

$$\boxed{\text{lift-bind\_(\_)}\colon \text{Expander} \to \text{Bind} \to \text{Bind}}$$

$$\text{lift-bind}_\alpha(\overline{f = \lambda\overline{x} \to e}) = \begin{cases} \overline{f = \lambda\overline{x} \to \text{lift}_\alpha(e)} & f_1 \notin \text{dom}\,\alpha \\ \overline{f = \lambda\alpha(f)\,\overline{x} \to \text{lift}_\alpha(e)} & \text{otherwise} \end{cases}$$

Fig. 3. Lambda lifting

5.3 Let Bindings

Hardly surprisingly, the meat of the transformation hides in the handling of **let** bindings. It is at this point that some heuristic (that of section 3, for example) decides whether to lambda lift the binding group bs wholly or not. For this decision, it has access to the extended expander α', but not to the binding group that would result from a positive lifting decision $\text{lift-bind}_{\alpha'}(bs)$. This makes sure that each syntactic element is only traversed once.

How does α' extend α? By calling out to add-rqs in its definition, it will also map every binding of the current binding group bs to its required set. Note that all bindings in the same binding group share their required set. The required set is the union of the free variables of all bindings, where lifted binders are expanded by looking into α, minus binders of the binding group itself. This is a conservative choice for the required set, but we argue for the minimality of this approach in the context of GHC in section 5.4.

With the domain of α' containing bs, every definition looking into that map implicitly assumes that bs is to be lifted. So it makes sense that all calls to lift and lift-bind take α' when bs should be lifted and α otherwise.

This is useful information when looking at the definition of lift-bind, which is responsible for abstracting the RHS e over its set of extraneous parameters when the given binding group should be lifted. Which is exactly the case when *any* binding of the binding group, like f_1, is in the domain of the passed α. In any case, lift-bind recurses via lift into the right-hand sides of the bindings.

5.4 Regarding Optimality

Johnsson [1985] constructed the set of extraneous parameters for each binding by computing the smallest solution of a system of set inequalities. Although this runs in $O(n^3)$ time, there were several attempts to achieve its optimality wrt. the minimal size of the required sets with better asymptotics. As such, Morazán and Schultz [2008] were the first to present an algorithm that simultaneously has optimal runtime in $O(n^2)$ and computes minimal required sets.

That begs the question whether the somewhat careless transformation in section 5 has one or both of the desirable optimality properties of the algorithm by Morazán and Schultz [2008].

For the situation within GHC, we loosely argue that the constructed required sets are minimal: Because by the time our lambda lifter runs, the occurrence analyser will have rearranged recursive groups into strongly connected components with respect to the call graph, up to lexical scoping. Now consider a variable $x \in \alpha(f_i)$ in the required set of a **let** binding for the binding group $\overline{f}$. We'll look into two cases, depending on whether x occurs free in any of the binding group's RHSs or not.

Assume that $x \notin \mathrm{fvs}(f_j)$ for every j. Then x must have been the result of expanding some function $g \in \mathrm{fvs}(f_j)$, with $x \in \alpha(g)$. Lexical scoping dictates that g is defined in an outer binding, an ancestor in the syntax tree, that is. So, by induction over the pre-order traversal of the syntax tree employed by the transformation, we can assume that $\alpha(g)$ must already have been minimal and therefore that x is part of the minimal set of f_i if g would have been prior to lifting g. Since $g \in \mathrm{fvs}(f_j)$ by definition, this is handled by the next case.

Otherwise there exists j such that $x \in \mathrm{fvs}(f_j)$. When $i = j$, f_i uses x directly, so x is part of the minimal set.

Hence assume $i \neq j$. Still, f_i needs x to call the current activation of f_j, directly or indirectly. Otherwise there is a lexically enclosing function on every path in the call graph between f_i and f_j that defines x and creates a new activation of the binding group. But this kind of call relationship implies that f_i and f_j don't need to be part of the same binding group to begin with! Indeed, GHC would have split the binding group into separate binding groups. So, x is part of the minimal set.

An instance of the last case is depicted in fig. 4. h and g are in the indirect call relationship of f_i and f_j above. Every path in the call graph between g and h goes through f, so g and h don't actually need to be part of the same binding group, even though they are part of the same strongly-connected component of the call graph. The only truly recursive function in that program is f. All other functions would be nested **let** bindings (cf. the right column of the fig. 4) after GHC's middleend transformations, possibly in lexically separate subtrees. The example is due to Morazán and Schultz and served as a prime example in showing the non-optimality of the call graph-based algorithm in Danvy and Schultz [2002].

Generally, lexical scoping prevents coalescing a recursive group with their dominators in the call graph if the dominators define variables that occur in the group. Morazán and Schultz gave convincing arguments that this was indeed what makes the quadratic time approach from Danvy and Schultz [2002] non-optimal with respect to the size of the required sets.

Regarding runtime: Morazán and Schultz made sure that they only need to expand the free variables of at most one dominator that is transitively reachable in the call graph. We think it's possible to find this *lowest upward vertical dependence* in a separate pass over the syntax tree, but we found the transformation to be sufficiently fast even in the presence of unnecessary variable

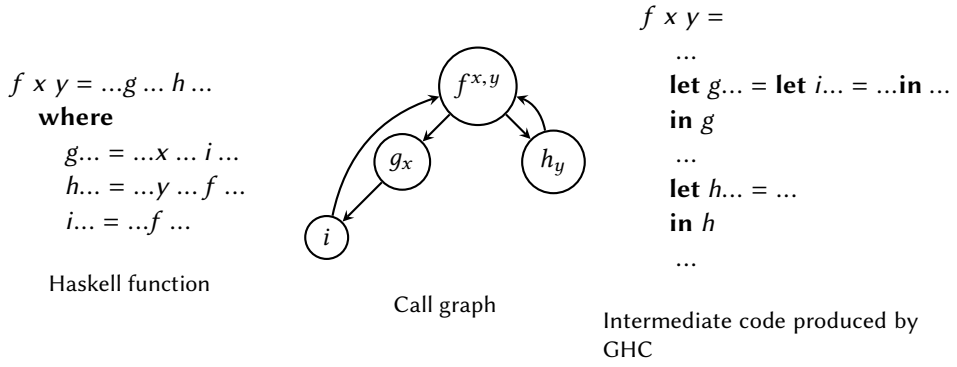

Fig. 4. Example from Morazán and Schultz [2008]

expansions for a total of $O(n^2)$ set operations, or $O(n^3)$ time. Ignoring needless expansions, which seem to happen rather infrequently in practice, the transformation performs $O(n)$ set operations when merging free variable sets.

6 EVALUATION

In order to assess the effectiveness of our new optimisation, we measured the performance on the `nofib` benchmark suite [Partain and Others 1992] against a GHC 8.6.1 release[5][6].

We will first look at how our chosen parameterisation (i.e. the optimisation with all heuristics activated as advertised) performs in comparison to the baseline. Subsequently, we will justify the choice by comparing with other parameterisations that selectively drop or vary the heuristics of section 3.

6.1 Effectiveness

The results of comparing our chosen configuration with the baseline can be seen in fig. 5.

We remark that our optimisation did not increase heap allocations in any benchmark, for a total reduction of 0.9%. This proves we succeeded in designing our analysis to be conservative with respect to allocations: Our transformation turns heap allocation into possible register and stack usage without a single regression.

Turning our attention to runtime measurements, we see that a total reduction of 0.7% was achieved. Although exploiting the correlation with closure growth payed off, it seems that the biggest wins in allocations don't necessarily lead to big wins in runtime: Allocations of `n-body` were reduced by 20.2% while runtime was barely affected. However, at a few hundred kilobytes, `n-body` is effectively non-allocating anyway. The reductions seem to hide somewhere in the `base` library. Conversely, allocations of `lambda` hardly changed, yet it sped up considerably.

In `queens`, 18% fewer allocations did only lead to a mediocre 0.5%. Here, a local function closing over three variables was lifted out of a hot loop to great effect on allocations, barely affecting runtime. We believe this is due to the native code generator of GHC, because when compiling with the LLVM backend we measured speedups of roughly 5%.

The same goes for `minimax`: We couldn't reproduce the runtime regressions with the LLVM backend.

[5] https://github.com/ghc/ghc/tree/0d2cdec78471728a0f2c487581d36acda68bb941
[6] Measurements were conducted on an Intel Core i7-6700 machine running Ubuntu 16.04.

Program	Bytes allocated	Runtime
awards	-0.2%	+2.4%
cryptarithm1	-2.8%	-8.0%
eliza	-0.1%	-5.2%
grep	-6.7%	-4.3%
knights	-0.0%	-4.5%
lambda	-0.0%	-13.5%
mate	-8.4%	-3.1%
minimax	-1.1%	+3.8%
n-body	-20.2%	-0.0%
nucleic2	-1.3%	+2.2%
queens	-18.0%	-0.5%
... and 94 more		
Min	-20.2%	-13.5%
Max	0.0%	+3.8%
Geometric Mean	-0.9%	-0.7%

Fig. 5. GHC baseline vs. late lambda lifting

Program	Bytes allocated	Runtime
bspt	-0.0%	+3.8%
eliza	-2.6%	+2.4%
gen_regexps	+10.0%	+0.1%
grep	-7.2%	-3.1%
integrate	+0.4%	+4.1%
knights	+0.1%	+4.8%
lift	-4.1%	-2.5%
listcopy	-0.4%	+2.5%
maillist	+0.0%	+2.8%
paraffins	+17.0%	+3.7%
prolog	-5.1%	-2.8%
wheel-sieve1	+31.4%	+3.2%
wheel-sieve2	+13.9%	+1.6%
... and 92 more		
Min	-7.2%	-3.1%
Max	+31.4%	+4.8%
Geometric Mean	+0.4%	-0.0%

Fig. 6. Late lambda lifting with vs. without (C2)

6.2 Exploring the design space

Now that we have established the effectiveness of late lambda lifting, it's time to justify our particular variant of the analysis by looking at different parameterisations.

Referring back to the five heuristics from section 3.2, it makes sense to turn the following knobs in isolation:

- Do or do not consider closure growth in the lifting decision (C2).
- Do or do not allow turning known calls into unknown calls (C4).
- Vary the maximum number of parameters of a lifted recursive or non-recursive function (C3).

Ignoring closure growth. Figure 6 shows the impact of deactivating the conservative checks for closure growth. This leads to big increases in allocation for benchmarks like `wheel-sieve1`, while it also shows that our analysis was too conservative to detect worthwhile lifting opportunities in `grep` or `prolog`. Cursory digging reveals that in the case of `grep`, an inner loop of a list comprehension gets lambda lifted, where allocation only happens on the cold path for the particular input data of the benchmark. Weighing closure growth by an estimate of execution frequency [Wu and Larus 1994] could help here, but GHC does not currently offer such information.

The mean difference in runtime results is surprisingly insignificant. That raises the question whether closure growth estimation is actually worth the additional complexity. We argue that unpredictable increases in allocations like in `wheel-sieve1` are to be avoided: It's only a matter of time until some program would trigger exponential worst-case behavior.

It's also worth noting that the arbitrary increases in total allocations didn't significantly influence runtime. That's because, by default, GHC's runtime system employs a copying garbage collector, where the time of each collection scales with the residency, which stayed about the same. A typical marking-based collector scales with total allocations and consequently would be punished by giving up closure growth checks, rendering future experiments in that direction infeasible.

Program	Runtime
`digits-of-e1`	+1.2%
`gcd`	+1.3%
`infer`	+1.2%
`mandel`	+2.7%
`mkhprog`	+1.1%
`nucleic2`	-1.3%
... and 99 more	
Min	-1.3%
Max	+2.7%
Geometric Mean	+0.1%

Fig. 7. Late lambda lifting with vs. without (C4)

Program	Runtime			
	4–4	5–6	6–5	8–8
`digits-of-e1`	+0.2%	-2.2%	-3.2%	+0.5%
`hidden`	-0.1%	+3.3%	+0.9%	+4.2%
`integer`	+2.7%	+3.7%	+2.1%	+3.1%
`knights`	+5.0%	-0.3%	+0.2%	-0.1%
`lambda`	+7.1%	-0.8%	-1.5%	-1.6%
`maillist`	+3.3%	+2.7%	+0.9%	+1.8%
`minimax`	-1.9%	+0.6%	+3.1%	+0.7%
`rewrite`	+1.9%	-1.0%	+3.2%	-1.6%
`wheel-sieve1`	+3.1%	+3.2%	+3.2%	-0.1%
... and 96 more				
Min	-2.8%	-2.2%	-3.2%	-1.6%
Max	+7.1%	+3.7%	+3.2%	+4.2%
Geometric Mean	+0.2%	+0.2%	+0.1%	+0.1%

Fig. 8. Late lambda lifting 5–5 vs. n–m (C3)

Turning known calls into unknown calls. In fig. 7 we see that turning known into unknown calls generally has a negative effect on runtime. By analogy to turning statically bound to dynamically bound calls in the object-oriented world this outcome is hardly surprising. There is `nucleic2`, but we suspect that its improvements are due to non-deterministic code layout changes in GHC's backend.

Varying the maximum arity of lifted functions. Figure 8 shows the effects of allowing different maximum arities of lifted functions. Regardless whether we allow less lifts due to arity (4–4) or more lifts (8–8), performance seems to degrade. Even allowing only slightly more recursive (5–6) or non-recursive (6–5) lifts doesn't seem to pay off.

Taking inspiration in the number of argument registers dictated by the calling convention on AMD64 was a good call.

7 RELATED AND FUTURE WORK

7.1 Related Work

Johnsson [1985] was the first to conceive lambda lifting as a code generation scheme for functional languages. We deviate from the original transformation in that we regard it as an optimisation pass by only applying it selectively and default to closure conversion for code generation.

Johnsson constructed the required set of free variables for each binding by computing the smallest solution of a system of set inequalities. Although this runs in $O(n^3)$ time, there were several attempts to achieve its optimality (wrt. the minimal size of the required sets) with better asymptotics. As such, Morazán and Schultz [2008] were the first to present an algorithm that simultaneously has optimal runtime in $O(n^2)$ and computes minimal required sets. In section 5.4 we compare to their approach. They also give a nice overview over previous approaches and highlight their shortcomings.

Operationally, an STG function is supplied a pointer to its closure as the first argument. This closure pointer is similar to how object-oriented languages tend to implement the `this` pointer. From this perspective, every function in the program already is a supercombinator, taking an

implicit first parameter. In this world, lambda lifting STG terms looks more like an *unpacking* of the closure record into multiple arguments, similar to performing Scalar Replacement [Carr and Kennedy 1994] on the `this` parameter or what the worker-wrapper transformation [Gill and Hutton 2009] achieves. The situation is a little different to performing the worker-wrapper split in that there's no need for strictness or usage analysis to be involved. Similar to type class dictionaries, there's no divergence hiding in closure records. At the same time, closure records are defined with the sole purpose of carrying all free variables for a particular function, hence a prior free variable analysis guarantees that the closure record will only contain free variables that are actually used in the body of the function.

Peyton Jones [1992] anticipates the effects of lambda lifting in the context of the STG machine, which performs closure conversion for code generation. He comes to the conclusion that direct accesses into the environment from the function body result in less movement of values from heap to stack.

The idea of regarding lambda lifting as an optimisation is not novel. Tammet [1996] motivates selective lambda lifting in the context of compiling Scheme to C. Many of his liftability criteria are specific to Scheme and necessitated by the fact that lambda lifting is performed *after* closure conversion, in contrast to our work, where lambda lifting happens prior to closure conversion.

Our selective lambda lifting scheme follows an all or nothing approach: Either the binding is lifted to top-level or it is left untouched. The obvious extension to this approach is to only abstract out *some* free variables. If this would be combined with a subsequent float out pass, abstracting out the right variables (i.e. those defined at the deepest level) could make for significantly fewer allocations when a binding can be floated out of a hot loop. This is very similar to performing lambda lifting and then cautiously performing block sinking as long as it leads to beneficial opportunities to drop parameters, implementing a flexible lambda dropping pass [Danvy and Schultz 2000].

Lambda dropping [Danvy and Schultz 2000], or more specifically parameter dropping, has a close sibling in GHC in the form of the static argument transformation [Santos 1995] (SAT). As such, the new lambda lifter is pretty much undoing SAT. We believe that SAT is mostly an enabling transformation for the middleend and by the time our lambda lifter runs, these opportunities will have been exploited. Other than that, SAT turns unknown into known calls, but in (C4) we make sure that we don't undo that.

7.2 Future Work

In section 6 we concluded that our closure growth heuristic was too conservative. In general, lambda lifting STG terms pushes allocations from definition sites into any closures of **let** bindings that nest around call sites. If only closures on cold code paths grow, doing the lift could be beneficial. Weighting closure growth by an estimate of execution frequency [Wu and Larus 1994] could help here. Such static profiles would be convenient in a number of places, for example in the inliner or to determine viability of exploiting a costly optimisation opportunity.

We find there's a lack of substantiated performance comparisons of closure conversion to lambda lifting for code generation on modern machine architectures. It seems lambda lifting has fallen out of fashion: GHC and the OCaml compiler both seem to do closure conversion. The recent backend of the Lean compiler makes use of lambda lifting for its conceptual simplicity.

8 CONCLUSION

We presented selective lambda lifting as an optimisation on STG terms and provided an implementation in the Glasgow Haskell Compiler. The heuristics that decide when to reject a lifting opportunity were derived from concrete operational considerations. We assessed the effectiveness of this evidence-based approach on a large corpus of Haskell benchmarks to conclude that our

optimisation sped up average Haskell programs by 0.7% in the geometric mean and reliably reduced the number of allocations.

One of our main contributions was a conservative estimate of closure growth resulting from a lifting decision. Although prohibiting any closure growth proved to be a little too restrictive, it still prevents arbitrary and unpredictable regressions in allocations. We believe that in the future, closure growth estimation could take static profiling information into account for more realistic and less conservative estimates.

REFERENCES

Steve Carr and Ken Kennedy. 1994. Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience* (1994). https://doi.org/10.1002/spe.4380240104

Olivier Danvy and Ulrik P. Schultz. 2000. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science* (2000). https://doi.org/10.1016/S0304-3975(00)00054-2

Olivier Danvy and Ulrik P. Schultz. 2002. Lambda-lifting in quadratic time. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 2441. 134–151. https://doi.org/10.1007/3-540-45788-7

Andy Gill and Graham Hutton. 2009. The worker / wrapper transformation. 19, 2 (2009), 227–251. https://doi.org/10.1017/S0956796809007175

Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 201 LNCS. 190–203. https://doi.org/10.1007/3-540-15975-4_37

Richard B. Kieburtz. 1985. The G-machine: A fast, graph-reduction evaluator. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/3-540-15975-4_50

Simon Marlow and Simon Peyton Jones. 2004. Making a fast curry. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming - ICFP '04*. 4. https://doi.org/10.1145/1016850.1016856

Marco T. Morazán and Ulrik P. Schultz. 2008. Optimal lambda lifting in quadratic time. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 5083 LNCS. 37–56. https://doi.org/10.1007/978-3-540-85373-2_3

Will Partain and Others. 1992. The nofib benchmark suite of Haskell programs. *Proceedings of the 1992 Glasgow Workshop on Functional Programming* (1992), 195–202.

Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* (1992). https://doi.org/10.1017/S0956796800000319

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. (1993).

Adré Luís De Medeiros Santos. 1995. *Compilation by Transformation in Non-Strict Functional Languages*. Ph.D. Dissertation.

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, Higher-order Cardinality Analysis in Theory and Practice. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014), 335–347. https://doi.org/10.1145/2535838.2535861

Tanel Tammet. 1996. Lambda-lifting as an optimization for compiling Scheme to C. (1996).

Youfeng Wu and James R. Larus. 1994. Static branch frequency and program profile analysis. *Professional Engineering* (1994). https://doi.org/10.1109/MICRO.1994.717399