

Lucrative Late Lambda Lifting

ANONYMOUS AUTHOR(S)

Lambda lifting is a well-known transformation [Johnsson 1985], traditionally employed for compiling functional programs to supercombinators [Peyton Jones 1987]. 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 suggest reliable 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. Lucrative Late Lambda Lifting. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2019), 17 pages.

1 INTRODUCTION

The ability to define nested auxiliary functions referencing free variables is essential when programming in functional languages. Compilers had to generate code for such functions since the dawn of Lisp.

A compiler compiling down to G-machine code [Peyton Jones 1987] would generate code by converting all free variables into parameters in a process called *lambda lifting* [Johnsson 1985]. The resulting functions are insensitive to lexical scope and can be floated to top-level.

An alternative to lambda lifting is *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*, passed as an implicit parameter to the function. All functions are then regarded as *closures*: A pair of a code pointer and an environment.

Current abstract machines for functional programming languages such as the spineless tagless G-machine [Peyton Jones 1992] choose to do closure conversion instead of lambda lifting for code generation. Although lambda lifting seems to have fallen out of fashion, we argue that it bears potential as an optimisation pass prior to closure conversion. Take this Haskell code as an example:

$$\begin{aligned} f\ a\ 0 &= a \\ f\ a\ n &= f\ (g\ (n \text{ 'mod' } 2))\ (n - 1) \\ \text{where} \\ g\ 0 &= a \\ g\ n &= 1 + g\ (n - 1) \end{aligned}$$

Closure conversion of g would allocate an environment with an entry for a . Now imagine we lambda lift g before that happens:

$$\begin{aligned} g_{\uparrow}\ a\ 0 &= a \\ g_{\uparrow}\ a\ n &= 1 + g_{\uparrow}\ a\ (n - 1) \\ f\ a\ 0 &= a \end{aligned}$$

$f\ a\ n = f\ (g\ (n\ \text{'mod'}\ 2))\ (n - 1)$

where

$g = g_{\uparrow}\ a$

Note that closure conversion would still allocate the same environments. Lambda lifting just separated closure allocation from the code pointer of g_{\uparrow} . Suppose now that the partial application g gets inlined:

$g_{\uparrow}\ a\ 0 = a$

$g_{\uparrow}\ a\ n = 1 + g_{\uparrow}\ a\ (n - 1)$

$f\ a\ 0 = a$

$f\ a\ n = f\ (g_{\uparrow}\ a\ (n\ \text{'mod'}\ 2))\ (n - 1)$

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

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

$f\ a\ b\ 0 = a$

$f\ a\ b\ 1 = b$

$f\ a\ b\ n = f\ (g\ n)\ a\ (n\ \text{'mod'}\ 2)$

where

$g\ 0 = a$

$g\ 1 = b$

$g\ n = n : g\ (n - 1)$

Because of laziness, this will allocate a thunk for the recursive call to g in the tail of the cons cell. Lambda lifting yields:

$g_{\uparrow}\ a\ b\ 0 = a$

$g_{\uparrow}\ a\ b\ 1 = b$

$g_{\uparrow}\ a\ b\ n = n : g_{\uparrow}\ a\ b\ (n - 1)$

$f\ a\ b\ 0 = a$

$f\ a\ b\ 1 = b$

$f\ a\ b\ n = f\ (g_{\uparrow}\ a\ b\ n)\ a\ (n\ \text{'mod'}\ 2)$

The closure for g has vanished, but the thunk in g_{\uparrow} 's body now closes over two additional variables. Worse, for a single allocation of g 's closure environment, we get n allocations on the recursive code path! Apart from making f allocate 10% more, this also incurs a slowdown of more than 10%.

Unsurprisingly, there are a number of subtleties to keep in mind. This work is concerned with finding out when doing this transformation is beneficial to performance, providing an interesting angle on the interaction between lambda lifting and closure conversion. These are our contributions:

- We describe a selective lambda lifting pass that maintains the invariants associated with the STG language [Peyton Jones 1992] (section 4).
- A number of heuristics fueling the lifting decision are derived from concrete operational deficiencies in section 3. 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 STG pipeline. 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 5) and find that our static analysis works as advertised.

- Our approach builds on and is similar to many previous works, which we compare to in section 6.

2 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 Syntax

As can be seen in fig. 1, we extended untyped lambda calculus with **let** bindings, just as in Johnsson [1985]. There are a few additional STG-inspired characteristics:

- 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)

We decomposed **let** expressions into smaller syntactic forms for the simple reason that it allows the analysis and transformation to be defined in more granular (and thus more easily understood) steps. Throughout this paper, we assume that variable names are globally unique.

Variables	$f, x, y \in \text{Var}$	
Expressions	$e \in \text{Expr} ::= x$	Variable
	$ f \bar{x}$	Function call
	$ \text{let } b \text{ in } e$	Recursive let
Bindings	$b \in \text{Bind} ::= \overline{f = r}$	
Right-hand sides	$r \in \text{Rhs} ::= \lambda \bar{x} \rightarrow e$	

Fig. 1. An STG-like untyped lambda calculus

2.2 Semantics

Giving a full operational semantics for the calculus in fig. 1 is out of scope for this paper, but since it's 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 by a **let** in a prior compilation step. Consequently, allocation happens almost entirely at **let** bindings closing over free variables of their RHSs, with the exception of 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.

3 WHEN TO LIFT

Lambda lifting and inlining are always sound transformations. The challenge is in identifying *when* it is beneficial to apply them. This section will discuss operational consequences of our lambda

lifting pass, clearing up the requirements for our transformation defined in section 4. Operational considerations will lead to the introduction of multiple criteria for rejecting a lift, motivating a cost model for estimating impact on heap allocations.

We'll take a somewhat loose approach to following the STG invariants in our examples (regarding giving all complex subexpressions a name, in particular), but will point out the details if need be.

3.1 Syntactic consequences

Deciding to lambda lift a binding **let** $f = \lambda a\ b\ c \rightarrow e$ **in** e' where x, y and z occur free in e , followed by inlining the residual partial application, has the following consequences:

- (S1) It eliminates the **let** binding.
- (S2) It creates a new top-level definition.
- (S3) It replaces all occurrences of f in e' and e (now part of the new top-level definition) by an application of the lifted top-level binding to its former free variables, replacing the whole **let** binding by the term $[f \mapsto f_{\uparrow} x\ y\ z]\ e'$.¹
- (S4) All non-top-level variables that occurred in the **let** binding's right-hand side become parameter occurrences.

Naming seemingly obvious things this way means we can precisely talk about *why* we are suffering from one of the operational symptoms discussed next.

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 by an application expression. This, however, would yield a syntactically invalid expression because the STG language only allows trivial arguments in an application!

Thus, our transformation would have to immediately wrap the application in a partial application: $g\ 5\ x\ f \Rightarrow \mathbf{let}\ f' = f_{\uparrow} x\ y\ z\ \mathbf{in}\ g\ 5\ x\ f'$. 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 fig. 3 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, occurring in some expression defining x and y :

```

let  $f = \lambda a\ b \rightarrow \dots x \dots y \dots$ 
     $g = \lambda d \rightarrow f\ d\ d + x$ 
in  $g\ 5$ 

```

Should f be lifted? It's hard to tell without actually seeing the lifted version:

¹Actually, this will also need to give a name to new non-atomic argument expressions mentioning f . We'll argue shortly that there is hardly any benefit in allowing these cases.

```

197    $f_{\uparrow} x y a b = \dots;$ 
198   let  $g = \lambda d \rightarrow f_{\uparrow} x y d d + x$ 
199   in  $g\ 5$ 

```

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 3.3.

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:

```

224   let  $f = \lambda x \rightarrow 2 * x$ 
225    $mapF = \lambda xs \rightarrow \dots f\ x \dots$ 
226   in  $mapF\ [1..n]$ 

```

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. Lifting $mapF$ (but not f) yields the following program:

```

230    $mapF_{\uparrow} f\ xs = \dots f\ x \dots;$ 
231   let  $f = \lambda x \rightarrow 2 * x$ 
232   in  $mapF_{\uparrow} f\ [1..n]$ 

```

Now, 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 a updatable binding, like a thunk:

```

240   let  $t = \lambda \rightarrow x + y$   -- Assume all nullary bindings are memoised
241    $addT = \lambda z \rightarrow z + t$ 
242   in  $map\ addT\ [1..n]$ 

```

The addition within t will be computed only once for each complete evaluation of the expression. This is after lambda lifting:

```

246   t x y = x + y;
247   let addT = λz → z + t x y
248   in map addT [1..n]

```

This 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

3.3 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.

3.3.1 *Motivation.* Let's revisit the example from above:

```

258   let f = λa b → ...x ... y ...
259       g = λd → f d d + x
260   in g 5

```

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) always leads to a one-time benefit. Simultaneously, each occurrence of f in a closure environment would be replaced by the free variables of its RHS. Removing f 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 multi-shot lambda, as the following example demonstrates:

```

269   let f = λa b → ...x ... y ...
270       g = λd →
271           let h = λe → f e e
272           in h x
273   in g 1 + g 2 + g 3

```

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 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:

```

282   let f = λa b → ...x ... y ...
283       g = λd →
284           let h1 = [f]λe → f e e
285               h2 = [f x y]λe → f e e + x + y
286           in h1 d + h2 d
287   in g 1 + g 2 + g 3

```

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 the (not quite max-plus) algebra $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ and account for positive closure growth under a multi-shot lambda by ∞ .

3.3.2 Design. Go over this once more

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

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

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.

In the lifting algorithm from section 4, `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 \bar{f} out of an expression **let** $f = \lambda \bar{x} \rightarrow e$ **in** e' , the following expression conservatively estimates the effect on heap allocation of performing the lift²:

$$\text{cl-gr}_{\{\bar{f}\}}^{\alpha'(f_1)}(\text{let } \overline{f = \lambda \alpha'(f_1) \bar{x} \rightarrow e} \text{ in } e') - \sum_i 1 + |\text{fvs}(f_i) \setminus \{\bar{f}\}|$$

The *required set* of extraneous parameters [Morazán and Schultz 2008] $\alpha'(f_1)$ 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 4. These variables would need to be available anywhere a binder from the binding group occurs, which justifies the choice of $\{\bar{f}\}$ as the subscript argument to `cl-gr`.

Note that we logically lambda lifted the binding group in question without actually floating out the binding. 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).

In practice, we require that this metric is non-positive to allow the lambda lift.

3.3.3 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 measuring binding groups. The growth component accounts for allocating each closure of the binding group. Whenever a closure mentions one of the variables to be removed (i.e. φ^- , the bindings 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) 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.

²The effect of inlining the partial applications resulting from vanilla lambda lifting, to be precise.

$$\begin{aligned}
& \boxed{\text{cl-gr}_{-}(\cdot): \mathcal{P}(\text{Var}) \rightarrow \mathcal{P}(\text{Var}) \rightarrow \text{Expr} \rightarrow \mathbb{Z}_{\infty}} \\
& \text{cl-gr}_{\varphi^{-}}^{\varphi^{+}}(x) = 0 \quad \text{cl-gr}_{\varphi^{-}}^{\varphi^{+}}(f \ \bar{x}) = 0 \\
& \text{cl-gr}_{\varphi^{-}}^{\varphi^{+}}(\text{let } bs \text{ in } e) = \text{cl-gr-bind}_{\varphi^{-}}^{\varphi^{+}}(bs) + \text{cl-gr}_{\varphi^{-}}^{\varphi^{+}}(e) \\
& \boxed{\text{cl-gr-bind}_{-}(\cdot): \mathcal{P}(\text{Var}) \rightarrow \mathcal{P}(\text{Var}) \rightarrow \text{Bind} \rightarrow \mathbb{Z}_{\infty}} \\
& \text{cl-gr-bind}_{\varphi^{-}}^{\varphi^{+}}(\bar{f} = \bar{r}) = \sum_i \text{growth}_i + \text{cl-gr-rhs}_{\varphi^{-}}^{\varphi^{+}}(r_i) \quad v_i = |\text{fvs}(f_i) \cap \varphi^{-}| \\
& \text{growth}_i = \begin{cases} |\varphi^{+} \setminus \text{fvs}(f_i)| - v_i, & \text{if } v_i > 0 \\ 0, & \text{otherwise} \end{cases} \\
& \boxed{\text{cl-gr-rhs}_{-}(\cdot): \mathcal{P}(\text{Var}) \rightarrow \mathcal{P}(\text{Var}) \rightarrow \text{Rhs} \rightarrow \mathbb{Z}_{\infty}} \\
& \text{cl-gr-rhs}_{\varphi^{-}}^{\varphi^{+}}(\lambda \bar{x} \rightarrow e) = \text{cl-gr}_{\varphi^{-}}^{\varphi^{+}}(e) * [\sigma, \tau] \quad n * [\sigma, \tau] = \begin{cases} n * \sigma, & n < 0 \\ n * \tau, & \text{otherwise} \end{cases} \\
& \sigma = \begin{cases} 1, & e \text{ entered at least once} \\ 0, & \text{otherwise} \end{cases} \quad \tau = \begin{cases} 0, & e \text{ never entered} \\ 1, & e \text{ entered at most once} \\ 1, & \text{RHS bound to a thunk} \\ \infty, & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 2. Closure growth estimation

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

4 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 [Morazán and Schultz 2008] of a binding.

As suggested in the introduction, we interleave pure lambda lifting with an inlining pass that immediately inlines the resulting partial applications.

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*, 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, it plays a similar role as the E_f set in Johnsson [1985]. We write $\text{dom } \alpha$ for the domain of the expander 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.

4.1 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. Notice that this has the effect of inlining the partial application that would arise in vanilla lambda lifting.

4.2 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. Inlining the partial application fails, so lambda lifting f is hardly of any use.

Our transformation enjoys a great deal of simplicity because it crucially relies on the adherence to (C1), so that inlining the partial application of f will always succeed.

4.3 Let Bindings

Hardly surprising, 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 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

$$\begin{aligned}
& \boxed{\text{lift_}(_): \text{Expander} \rightarrow \text{Expr} \rightarrow \text{Expr}} \\
& \text{lift}_\alpha(x) = \begin{cases} x, & x \notin \text{dom } \alpha \\ x \ \alpha(x), & \text{otherwise} \end{cases} \quad \text{lift}_\alpha(f \ \bar{x}) = \text{lift}_\alpha(f) \ \bar{x} \\
& \text{lift}_\alpha(\text{let } bs \text{ in } e) = \begin{cases} \text{lift}_{\alpha'}(e), & bs \text{ is to be lifted as lift-bind}_{\alpha'}(bs) \\ \text{let lift-bind}_\alpha(bs) \text{ in lift}_\alpha(e) & \text{otherwise} \end{cases} \\
& \text{where} \\
& \alpha' = \text{add-rqs}(bs, \alpha) \\
& \boxed{\text{add-rqs}(_, _): \text{Bind} \rightarrow \text{Expander} \rightarrow \text{Expander}} \\
& \text{add-rqs}(\overline{f = r}, \alpha) = \alpha \left[\overline{f \mapsto \text{rqs}} \right] \\
& \text{where} \\
& \text{rqs} = \bigcup_i \text{expand}_\alpha(\text{fvs}(r_i)) \setminus \{\bar{f}\} \\
& \boxed{\text{expand_}(_): \text{Expander} \rightarrow \mathcal{P}(\text{Var}) \rightarrow \mathcal{P}(\text{Var})} \\
& \text{expand}_\alpha(V) = \bigcup_{x \in V} \begin{cases} \{x\}, & x \notin \text{dom } \alpha \\ \alpha(x), & \text{otherwise} \end{cases} \\
& \boxed{\text{lift-bind_}(_): \text{Expander} \rightarrow \text{Bind} \rightarrow \text{Bind}} \\
& \text{lift-bind}_\alpha(\overline{f = \lambda \bar{x} \rightarrow e}) = \begin{cases} \overline{f = \lambda \bar{x} \rightarrow \text{lift}_\alpha(e)} & f_1 \notin \text{dom } \alpha \\ \overline{f = \lambda \alpha(f) \bar{x} \rightarrow \text{lift}_\alpha(e)} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 3. Lambda lifting

choice for the required set, but we argue for the minimality of this approach in the context of GHC in section 4.4.

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.

4.4 Regarding Optimality

Johnsson [1985] 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.

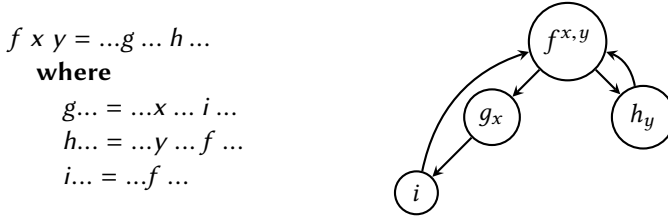


Fig. 4. Example call graph from Morazán and Schultz [2008]

That begs the question whether the somewhat careless transformation in section 4 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 dependency 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 f_i .

Suppose $x \notin \text{fvs}(f_j)$ for any j . Then x must have been the result of expanding some function $g \in \text{fvs}(f_j)$, 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 .

Otherwise there exists j such that $x \in \text{fvs}(f_j)$. 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 redefines 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.

A similar situation is depicted in fig. 4. h and g are in a call relationship similar to 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. The only truly recursive function in that program is f . All other functions would be nested **let** bindings after GHC's occurrence analysis, possibly in lexically separate subtrees. The example is of Morazán and Schultz [2008] and served as a prime example in showing the non-optimality of 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 [2008] 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 [2008] 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 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.

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%
<i>... and 94 more</i>		
Min	-20.2%	-13.5%
Max	0.0%	+3.8%
Geometric Mean	-0.9%	-0.7%

Table 1. Interesting benchmark changes compared to the GHC 8.6.1 baseline.

5 EVALUATION

In order to assess effectiveness of our new optimisation, we measured performance on the `nofib` benchmark suite [Partain and Others 1992] against a GHC 8.6.1 release³⁴.

We will first look at how our chosen parameterisation (e.g., 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.

5.1 Effectiveness

The results of comparing our chosen configuration with the baseline can be seen in table 1.

It shows that there was no benchmark that increased in heap allocations, for a total reduction of 0.9%. On the other hand that's hardly surprising, since we designed our analysis to be conservative with respect to allocations and the transformation turns heap allocation into possible register and stack allocation, which is not reflected in any numbers.

It's more informative to look at runtime measurements, where 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. Conversely, allocations of `lambda` hardly changed, yet it sped up considerably.

5.2 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.

³<https://github.com/ghc/ghc/tree/0d2cdec78471728a0f2c487581d36acda68bb941>

⁴Measurements were conducted on an Intel Core i7-6700 machine running Ubuntu 16.04.

Program	Bytes allocated	Runtime
bspt	-0.0%	+3.8%
eliza	-2.6%	+2.4%
gen_regexp	+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%

Table 2. Comparison of our chosen parameterisation with one where we allow arbitrary increases in allocations.

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. Table 2 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 rises 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%

Table 3. Runtime comparison of our chosen parameterisation with one where we allow turning known into unknown calls.

Turning known calls into unknown calls. In table 3 we see that turning known into unknown calls generally has a negative effect on runtime. There is nucleic2, but we suspect that its improvements are due to non-deterministic code layout changes.

By analogy to turning statically bound to dynamically bound calls in the object-oriented world this outcome is hardly surprising.

Varying the maximum arity of lifted functions. Table 4 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.

6 RELATED AND FUTURE WORK

6.1 Related Work

Johnsson [1985] was the first to conceive lambda lifting as a code generation scheme for functional languages. As explained in section 4, we deviate from the original transformation in that we interleave an inlining pass for the residual partial applications and regard this interleaving as an optimisation pass by only applying it selectively.

Johnsson's 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. We compare in section 4.4 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. References to free variables are resolved indirectly through the closure pointer, mimicing access of a heap-allocated record. 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

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%

Table 4. Runtime comparison of our chosen parameterisation 5-5 with one where we allow more or less maximum arity of lifted functions. A parameterisation $n-m$ means maximum non-recursive arity was n and maximum recursive arity was m .

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 to carry all free variables for a particular function and 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. Without the subsequent step which inlines the partial application, 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. Our approach however inlines the partial application to get rid of heap accesses altogether.

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.

Our selective lambda lifting scheme proposed 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 less 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 argue that SAT is mostly an enabling transformation for the middleend and by the time our lambda lifter runs, these opportunities will have been exploited.

6.2 Future Work

In section 5 we concluded that our closure growth heuristic was too conservative. 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.

In general, lambda lifting STG terms and then inlining residual partial applications pushes allocations from definition sites into any closures 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.

Our selective lambda lifter 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 less 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].

7 CONCLUSION

We presented the combination of lambda lifting with an inlining pass for residual partial applications 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 deficiencies. We assessed the effectiveness of this evidence-based approach on a large corpus of Haskell benchmarks and concluded that average Haskell programs sped up by 0.7% in the geometric mean.

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.

acknowledgements

TODO LIST

Go over this once more 6

acknowledgements 16

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](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
- 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 Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall. <https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/>
- Simon Peyton Jones, Peter Sestoft, and John Hughes. [n. d.]. *Demand analysis*. Technical Report.
- 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>
- Adré Luis De Medeiros Santos. 1995. *Compilation by Transformation in Non-Strict Functional Languages*. Ph.D. Dissertation.
- 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>