Lucrative Late Lambda Lifting

Sebastian Graf

January 23, 2019

1 Introduction

Lambda lifting is a well-known transformation [5], traditionally employed for compiling functional programs to supercombinators [11]. 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 lambda lifting in this context as an optimising code generation strategy. Take this code in a Haskell-like language with explicit free variables as an example:

$$\begin{aligned}&\textbf{let } f = [x\ y]\lambda a\ b \to x * y + a * b\\&\quad g = [f\ x]\lambda d \to f\ d\ d + x\\&\textbf{in } g\ 5\end{aligned}$$

Closure conversion of f and g would allocate an environment with two entries for both. Now imagine we lambda lift f before that happens:

$$\begin{aligned}&f_{\uparrow}\ x\ y\ a\ b = x * y + a * b;\\&\textbf{let } f = [x\ y]\lambda \to f_{\uparrow}\ x\ y\\&\quad g = [f\ x]\lambda d \to f\ d\ d + x\\&\textbf{in } g\ 5\end{aligned}$$

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

$$\begin{aligned}&f_{\uparrow}\ x\ y\ a\ b = x * y + a * b;\\&\textbf{let } g = [x\ y]\lambda d \to f_{\uparrow}\ x\ y\ d\ d + x\\&\textbf{in } g\ 5\end{aligned}$$

The closure for f and the associated allocations completely vanished in favor of a few more arguments at its call site! The result looks much simpler.

But wait. Assume for the sake of the argument that we subtly change the body of f:

$$\begin{aligned}&\textbf{let } f = [x\ y]\lambda a\ b \to y\ (x + a * b)\\&\quad g = [f\ x]\lambda d \to f\ d\ d + x\\&\textbf{in } g\ 5\end{aligned}$$

Now f closes over a function y, occuring in an (assumed) early bound call in f's body. After the proposed transformation we would get this:

$f_\uparrow\ x\ y\ a\ b = y\ (x + a * b);$
$\textbf{let}\ g = [x\ y]\lambda d \to f_\uparrow\ x\ y\ d\ d + x$
$\textbf{in}\ g\ 5$

But now the call to the formal parameter y in $f_\uparrow$ is late bound, incurring a substantial slowdown.

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 [10] (section 2).
- 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 (GHC) 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 are impossible on GHC's more high-level Core language. We evaluate our pass against the `nofib` benchmark suite (section 4) and find that our static analysis works as advertised.
- Our approach builds and is similar to many previous works, which we compare to in section 5.

2 Transformation

The extension of Johnsson's formulation [5] 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 [8] of a binding.

2.1 Syntax

Although the STG language is tiny compared to typical surface languages such as Haskell, its definition [7] still contains much detail irrelevant to lambda lifting. As can be seen in fig. 1, we therefore adopt a simple untyped lambda calculus with **let** bindings as in Johnsson [5], with a few 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)

Additionally, we assume that **let** bindings are annotated with the non-top-level free variables of the right-hand side (RHS) they bind.

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.

$$
\begin{array}{rrcll}
\text{Variables} & f, x, y \in \mathsf{Var} & & & \\
\text{Expressions} & e \in \mathsf{Expr} & ::= & x & \text{Variable} \\
 & & \mid & f\ x_1 \ldots x_n & \text{Saturated function call} \\
 & & \mid & \textbf{let}\ b\ \textbf{in}\ e & \text{Recursive } \textbf{let} \\
\text{Bindings} & b \in \mathsf{Bind} & ::= & \overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\, r_i} & \\
\text{Right-hand sides} & r \in \mathsf{Rhs} & ::= & \lambda y_1 \ldots y_m \to e &
\end{array}
$$

Figure 1: An STG-like untyped lambda calculus

2.2 Algorithm

With the notation settled, we can begin to recap our variant of the lambda lifting transformation. It is assumed that all variables have unique names and that there is a sufficient supply of fresh names from which to draw.

We'll define a side-effecting function, lift, recursively over the term structure. This is its signature:

Take inspiration in "Implementing functional languages: a tutorial" and collect super-combinators afterwards for better separation of concerns. Is that possible? I think not, the hardest part probably is the subsequent inlining pass and the associated substitution. Separating out the decision logic won't really help much. On the other hand, we already lean on a hypothetical inlining pass in some places, so we could just delegate some more inlining work to it. I still don't think this would meaningfully simplify things.

$$\mathsf{lift}_{\_}(\_) \colon \mathsf{Expander} \to \mathsf{Expr} \to \mathcal{W}_{\mathsf{Bind}}\, \mathsf{Expr}$$

As its first argument, lift takes an $\mathsf{Expander}$, which is a partial function from lifted binders to their sets of required variables. 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 [5]. We write $\mathsf{dom}\,\alpha$ for the domain of the expander α and $\alpha[\mathsf{x} \mapsto S]$ to denote extension of the expander function, so that the result maps x to S and all other identifiers by delegating to α.

I think the occurrences of body expression etc. need to be meta-variables.

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 *writer monad*, denoted by $\mathcal{W}_{\mathsf{Bind}}\ \_$.

2.2.1 Side-effects

The following syntax, inspired by *idiom brackets* [6] and *bang notation*[1], will allow concise notation while hiding sprawling state threading:

Properly define the structure? Or is this 'obvious'?

$$[\![E[\langle e_1\rangle, ..., \langle e_n\rangle]]\!]$$

This denotes a side-effecting computation that, when executed, will perform the side-effecting subcomputations e_i in order (any fixed order will do for us). After that, it will lift the otherwise pure context E over the results of the subcomputations.

In addition, we make use of the monadic bind operators $\ggg=$ and $\gg$, defined in the usual way. The primitive operation note takes as argument a binding group and merges its bindings into the contextual binding group tracked by the writer monad.

2.2.2 Variables

Let's begin with the variable case.

$$\mathsf{lift}_\alpha(x) = \begin{cases} [\![x]\!], & x \notin \mathsf{dom}\,\alpha \\ [\![x\ y_1 \ldots y_n]\!], & \alpha(x) = \{y_1, \ldots, y_n\} \end{cases}$$

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 variables. There are no bindings occuring that could be lambda lifted, hence the function performs no actual side-effects.

2.2.3 Applications

Handling function application correctly is a little subtle, because only variables are allowed in argument position. When such an argument variable's binding is lifted to top-level, it turns into a non-atomic application expression, violating the STG invariants. Each such application must be bound to an enclosing **let** binding [2]:

The application rule is unnecessarily complicated because we support occurrences of lifted binders in argument position. Lifting such binders isn't worthwhile

[1] `http://docs.idris-lang.org/en/v1.3.0/tutorial/interfaces.html`

[2] To keep the specification reasonably simple, we also do so for non-lifted identifiers and assume that the compiler can do the trivial rewrite **let** $y = [x]\lambda \to x$ **in** $E[y] \Longrightarrow E[x]$ for us.

$$\mathsf{lift}_\alpha(f\ x_1 \ldots x_n) = [\![(\mathsf{wrap}_\alpha(x_n) \circ \ldots \circ \mathsf{wrap}_\alpha(x_1))(\langle \mathsf{lift}_\alpha(f)\rangle\ \ x'_1 \ldots x'_n)]\!]$$

The notation x' chooses a fresh name for x in a consistent fashion. The application head f is handled by an effectful recursive call to lift. Syntactically heavy **let** wrapping is outsourced into a helper function wrap:

$$\mathsf{wrap}_\alpha(x)(e) = \begin{cases} \textbf{let}\ x' = [x]\lambda \to x\ \textbf{in}\ e, & x \notin \mathsf{dom}\,\alpha \\ \textbf{let}\ x' = [\,]\lambda y_1 \ldots y_n \to x\ y_1 \ldots y_n\ \textbf{in}\ e, & \alpha(x) = \{y_1, \ldots, y_n\} \end{cases}$$

2.2.4 Let Bindings

Hardly surprising, the meat of the transformation hides in the handling of **let** bindings. This can be broken down into three separate functions:

$$\mathsf{lift}_\alpha(\textbf{let}\ bs\ \textbf{in}\ e) = (\mathsf{recurse}(e) \circ \mathsf{decide\text{-}lift}_\alpha \circ \mathsf{expand\text{-}envs}_\alpha)(bs)$$

The first step is to expand closure environments mentioned in bs with the help of α. Then a heuristic (that of section 3, for example) decides whether to lift the binding group bs to top-level or not. Depending on that decision, the binding group is noted to be lifted to top-level and syntactic subentities of the **let** binding are traversed with the updated expander.

$$\mathsf{expand\text{-}envs}_\alpha(\overline{f_i = [x_{i,1} \ldots x_{i,n_i}]r_i}) = \overline{f_i = [y_{i,1} \ldots y_{i,n'_i}]r_i}$$

where

$$\{y_{i,1} \ldots y_{i,n'_i}\} = \bigcup_{j=1}^{n_i} \begin{cases} x_{i,j}, & x_{i,j} \notin \mathsf{dom}\,\alpha \\ \alpha(x_{i,j}), & \text{otherwise} \end{cases}$$

$\mathsf{expand\text{-}envs}$ substitutes all occurences of lifted binders (those that are in $\mathsf{dom}\,\alpha$) in closure environments of a given binding group by their required set.

$$\mathsf{decide\text{-}lift}_\alpha(bs) = \begin{cases} (\varepsilon, \alpha', \mathsf{abstract}_{\alpha'}(bs)), & \text{if } bs \text{ should be lifted} \\ (bs, \alpha, \varepsilon), & \text{otherwise} \end{cases}$$

where

$$\alpha' = \alpha\left[\overline{f_i \mapsto \mathsf{rqs}(bs)}\right] \text{ for } \overline{f_i = [\_]\_} = bs$$

$\mathsf{decide\text{-}lift}$ returns a triple of a binding group that remains with the local **let** binding, an updated expander and a binding group prepared to be lifted to top-level. Depending on whether the argument bs is decided to be lifted or not, either the returned local binding group or the $\mathsf{abstract}$ed binding group is empty. In case the binding is to be lifted, the expander is updated to map the newly lifted bindings to their required set.

$$\mathsf{rqs}(\overline{f_i = [x_1 \ldots x_{n_i}]\_}) = \bigcup_i \{x_1, \ldots, x_{n_i}\} \setminus \{\overline{f_i}\}$$

The required set consists of the free variables of each binding's RHS, conveniently available in syntax, minus the defined binders themselves. Note that the required set of each binder of the same binding group will be identical. See section 5 for an argument about minimality of the resulting required sets.

$$\mathsf{abstract}_\alpha(\overline{f_i = [x_1 \ldots x_{n_i}]\lambda y_1 \ldots y_{m_i} \to e_i}) = \overline{f_i = [\,]\lambda \alpha(f_i)\ y_1 \ldots y_{m_i} \to e_i}$$

The abstraction step is performed in abstract, where closure variables are removed in favor of additional parameters, one for each element of the respective binding's required set.

$$\mathsf{recurse}(e)(bs, \alpha, lbs) = \mathsf{lift\text{-}bind}_\alpha(lbs) \ggg= \mathsf{note} \gg [\![\textbf{let}\ \langle \mathsf{lift\text{-}bind}_\alpha(bs)\rangle\ \textbf{in}\ \langle \mathsf{lift}_\alpha(e)\rangle]\!]$$

In the final step of the **let** "pipeline", the algorithm recurses into every subexpression of the **let** binding. The binding group to be lifted is transformed first, after which it is added to the contextual top-level binding group of the writer monad. Finally, the binding group that remains locally bound is traversed, as well as the original **let** body. The result is again wrapped up in a **let** and returned[3].

What remains is the trivial, but noisy definition of the lift-bind traversal:

Horizontal overflows. Argh

$$\mathsf{lift\text{-}bind}_\alpha(\overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\lambda y_{i,1} \ldots y_{i,m_i} \to e_i}) = [\![\overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\lambda y_{i,1} \ldots y_{i,m_i} \to \langle \mathsf{lift}_\alpha(e_i)\rangle}]\!]$$

3 When to lift

Lambda lifting a binding to top-level is always a sound transformation. The challenge is in identifying *when* it is beneficial to do so. This section will discuss operational consequences of lambda lifting, introducing multiple criteria based on 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 lift a binding **let** $f = [x\ y\ z]\lambda a\ b\ c \to e_1$ **in** e_2 to top-level has the following consequences:

[3] Similar to the application case, we assume that the compiler performs the obvious rewrite **let** ε **in** $e \Longrightarrow e$.

(S1) It eliminates the **let** binding.

(S2) It creates a new top-level definition.

(S3) It replaces all occurrences of f in e_2 by an application of the lifted top-level binding to its former free variables, replacing the whole **let** binding by the term $[f \mapsto f_\uparrow\ x\ y\ z]\ e_2$.[4]

(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_2 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.

The transformation from section 2 will immediately wrap the application in a **let** binding for the complex argument expression: $g\ 5\ x\ f \Longrightarrow \textbf{let}\ f' = f_\uparrow\ x\ y\ z\ \textbf{in}\ g\ 5\ x\ f'$. 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 2.2.3 becomes much simpler: The complicated wrap business 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*. Consider this example:

$$\begin{array}{l} \textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\ \quad g = [f\ x]\lambda d \to f\ d\ d + x \\ \textbf{in}\ g\ 5 \end{array}$$

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

[4] Actually, this will also need to give a name to new non-atomic argument expressions (cf. section 2.2.3). We'll argue shortly that there is hardly any benefit in allowing these cases.

$f_\uparrow = \lambda x\ y\ a\ b \rightarrow ...;$
let $g = [x\ y]\lambda d \rightarrow f_\uparrow\ x\ y\ d\ d + x$
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 $f_\uparrow$), 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.

Estimation of closure growth is crucial to identifying beneficial lifting opportunities. 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 x86_64)

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:

let $f = [\,]\lambda x \rightarrow 2 * x$
  $mapF = [f]\lambda xs \rightarrow ...f\ x\ ...$
in $mapF\ [1, 2, 3]$

Here, there is a *known call* to f in $mapF$ that can be lowered as a direct jump to a static address [7]. Lifting $mapF$ (but not f) yields the following program:

$mapF_\uparrow = \lambda f\ xs \rightarrow ...f\ x...;$
let $f = [\,]\lambda x \rightarrow 2 * x$
in $mapF_\uparrow\ f\ [1, 2, 3]$

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

Code size. (S2) (and, to a lesser extent, all other consequences) have the potential to increase or decrease code size. We regard this a secondary concern, but will have a look at it in section 4.

Sharing. Let's finish with a no-brainer: Lambda lifting updatable bindings (e.g., thunks) or constructor bindings is a bad idea, because it 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 criterions above, (C2) is the most important for reliable 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:

$$
\begin{array}{l}
\textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\
\quad g = [f\ x]\lambda d \to f\ d\ d + x \\
\textbf{in}\ g\ 5
\end{array}
$$

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) always leads to a one-time benefit. Simultaneously, each closure occurrence of f would be replaced by its referenced free variables. Removing f leads to a saving of one slot per closure, but the free variables x and y each occupy a closure slots in turn. Of these, only y really contributes to closure growth, because x already occurred in the single remaining closure of g.

This phenomenon is amplified whenever allocation happens under a multi-shot lambda, as the following example demonstrates:

$$
\begin{array}{l}
\textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\
\quad g = [f\ x]\lambda d \to \\
\qquad \textbf{let}\ h = [f]\lambda e \to f\ e\ e \\
\qquad \textbf{in}\ h\ d \\
\textbf{in}\ g\ 1 + g\ 2 + g\ 3
\end{array}
$$

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 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 occuring 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 called inside a closure under such a multi-shot lambda is conservative, but rules out worthwhile cases like this:

$$
\begin{array}{l}
\textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\
\quad g = [f\ x\ y]\lambda d \to \\
\qquad \textbf{let}\ h_1 = [f]\lambda e \to f\ e\ e \\
\qquad\quad h_2 = [f\ x\ y]\lambda e \to f\ e\ e + x + y
\end{array}
$$

$\textbf{in}\ h_1\ d + h_2\ d$
$\textbf{in}\ g\ 1 + g\ 2 + g\ 3$

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 denote positive closure growth under a multi-shot lambda by ∞.

3.3.2 Design

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

Maybe add the syntactic sort we operate on as a superscript?

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

Given two sets of variables for added and removed 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 2, 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 $\overline{f_i}$ out of an expression $\textbf{let}\ \overline{f_i = [x_1 \ldots x_{n_i}]\lambda y_1 \ldots y_{m_i} \to e_i}\ \textbf{in}\ e$, the following expression conservatively estimates the effects on heap allocation for performing the lift:

$$\mathsf{cl\text{-}gr}_{\alpha'(f_1)\,\{\overline{f_i}\}}(\textbf{let}\ \overline{f_i = [\,]\lambda x_1 \ldots x_{n_i} y_1 \ldots y_{m_i} \to e_i}\ \textbf{in}\ e) - \sum_i n_i$$

With the required set $\alpha'(f_1)$ passed as the first argument and with $\{\overline{f_i}\}$ for the second set (i.e. the binders for which lifting is to be decided).

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). 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 cases for variables and applications are trivial, because they don't allocate:

$$\mathsf{cl\text{-}gr}_{\varphi^+\varphi^-}(x) = 0$$
$$\mathsf{cl\text{-}gr}_{\varphi^+\varphi^-}(f\ x_1 \ldots x_n) = 0$$

As before, the complexity hides in **let** bindings and its syntactic components. We'll break them down one layer at a time. This makes the **let** rule itself nicely compositional, because it delegates most of its logic to cl-gr-bind:

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

Next, we look at how binding groups are measured:

$$\begin{aligned}
\mathsf{cl\text{-}gr\text{-}bind}_{\varphi^+\varphi^-}(\overline{f_i = [x_1 \ldots x_{n_i}]r_i}) &= \sum_i \mathsf{growth}_i + \sum_i \mathsf{cl\text{-}gr\text{-}rhs}_{\varphi^+\varphi^-}(r_i) \\
\mathsf{growth}_i &= \begin{cases} |\varphi^+ \setminus \{x_1, \ldots, x_{n_i}\}| - \nu_i, & \text{if } \nu_i > 0 \\ 0, & \text{otherwise} \end{cases} \\
\nu_i &= \left|\{x_1, \ldots, x_{n_i}\} \cap \varphi^-\right|
\end{aligned}$$

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:

$$\begin{aligned}
\mathsf{cl\text{-}gr\text{-}rhs}_{\varphi^+\varphi^-}(\lambda \ldots \to e) &= \mathsf{cl\text{-}gr}_{\varphi^+\varphi^-}(e) * [\sigma, \tau] \\
\sigma &= \begin{cases} 1, & e \text{ is entered at least once} \\ 0, & \text{otherwise} \end{cases} \\
\tau &= \begin{cases} 0, & e \text{ is never entered} \\ 1, & e \text{ is entered at most once} \\ 1, & \text{the RHS is bound to a thunk} \\ \infty, & \text{otherwise} \end{cases} \\
n * [\sigma, \tau] &= \begin{cases} n * \sigma, & n < 0 \\ n * \tau, & \text{otherwise} \end{cases}
\end{aligned}$$

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 [12] . Demand analysis estimates lower and upper bounds (σ and τ above) on how many times a RHS is entered relative to its defining expression.

What to cite? Progress on the new demand analysis paper seemed to have stalled. The cardinality paper? The old demand analysis paper from 2006? Both?

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 Evaluation

In order to assess effectiveness of our new optimisation, we measured performance on the `nofib` benchmark suite [9] against a GHC 8.6.1 release[5].

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.

4.1 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.

[5] `https://github.com/ghc/ghc/tree/0d2cdec78471728a0f2c487581d36acda68bb941`

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%

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

It's more informative to look at runtime measurements, where a total reduction of 0.6% 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.

4.2 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. 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`.

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%

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

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 [15] 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.

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.

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.

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.

5 Related and Future Work

5.1 Related Work

Johnsson [5] was the first to conceive lambda lifting as a code generation scheme for functional languages. Construction of the required set of free variables for each binding is formulated as the smallest solution of a system of set inequalities.

Although Johnsson's algorithm runs in $\mathcal{O}(n^3)$ 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 [8] were the first to present an algorithm that simultaneously has optimal runtime in $\mathcal{O}(n^2)$ and computes minimal required sets. They also give a nice overview over previous approaches and highlight their shortcomings.

That begs the question whether the somewhat careless transformation in section 2 has one or both of the desirable optimality properties of the algorithm by Morazán and Schultz [8].

As a separate theorem in section 2 or the appendix?

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.

At least 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 $\overline{f_i}$.

Suppose there exists j such that $x \in \mathsf{rqs}(f_j)$, in which case x must be part of the minimal set: Note that lexical scoping prevents coalescing a recursive group with their dominators in the call graph if they define variables that occur in the group. Morazán and Schultz [8] gave a convincing example that this was indeed what makes the quadratic time approach from Danvy and Schultz [3] non-optimal with respect to the size of the required sets.

When $x \notin \mathsf{rqs}(f_j)$ for any j, x must have been the result of expanding some function $g \in \mathsf{rqs}(f_j)$, 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.

Regarding runtime: Morazán and Schultz [8] 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 $\mathcal{O}(n^2)$ set operations. Ignoring needless expansions, the transformation performs $\mathcal{O}(n)$ set operations when merging free variable sets.

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 more like an *unpacking* of the closure record into multiple arguments, similar to performing Scalar Replacement [1] on the `this` parameter or what the worker-wrapper transformation [4] 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 [10] anticipates the effects of lambda-lifting in the context of the STG machine. Without the subsequent step which inlines the partial application, he comes to the correct 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 [14] 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 [2].

Lambda dropping, or more specifically parameter dropping, has a close sibling in GHC in the form of the static argument transformation [13] (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.

5.2 Future Work

In section 4 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 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 [15] could help here. Such static profiles would be convenient in a number of places, to determine viability of exploiting a costly optimisation opportunity, for example.

Currently, the decision of whether to lift a binding or not is all or nothing. There might be a middle-ground worthwhile to be explored: Don't abstract over *all* free variables, but only those with the most beneficial effects. For example, we might be able to float a binding out of a hot loop when we would just abstract over the most recently defined free variable.

6 Conclusion

We presented selective lambda lifting as an optimisation on STG terms and 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.

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 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.

7 Acknowledgments

acknowledgements

Todo list

References

[1] Steve Carr and Ken Kennedy. "Scalar replacement in the presence of conditional control flow". In: *Software: Practice and Experience* (1994). ISSN: 1097024X. DOI: 10.1002/spe.4380240104.

[2] Olivier Danvy and Ulrik P. Schultz. "Lambda-dropping: Transforming recursive equations into programs with block structure". In: *Theoretical Computer Science* (2000). ISSN: 03043975. DOI: 10.1016/S0304-3975(00)00054-2.

[3] Olivier Danvy and Ulrik P. Schultz. "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. 2002, pp. 134–151. ISBN: 3540442332. DOI: 10.1007/3-540-45788-7.

[4] Andy Gill and Graham Hutton. "The worker / wrapper transformation". In: 19.2 (2009), pp. 227–251. DOI: 10.1017/S0956796809007175.

[5] Thomas Johnsson. "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. 1985, pp. 190–203. ISBN: 9783540159759. DOI: 10.1007/3-540-15975-4_37.

[6] CONOR MCBRIDE and ROSS PATERSON. *Applicative programming with effects*. 2008. DOI: 10.1017/S0956796807006326.

[7] Simon Marlow and Simon Peyton Jones. "Making a fast curry". In: *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming - ICFP '04*. 2004, p. 4. ISBN: 1581139055. DOI: 10.1145/1016850.1016856. URL: http://portal.acm.org/citation.cfm?doid=1016850.1016856.

[8] Marco T. Morazán and Ulrik P. Schultz. "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. 2008, pp. 37–56. ISBN: 3540853723. DOI: 10.1007/978-3-540-85373-2_3.

[9] Will Partain and Others. "The nofib benchmark suite of Haskell programs". In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming* (1992), pp. 195–202.

[10] Simon L. Peyton Jones. "Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine". In: *Journal of Functional Programming* (1992). ISSN: 14697653. DOI: 10.1017/S0956796800000319.

[11] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/.

[12] Simon Peyton Jones, Peter Sestoft, and John Hughes. *Demand analysis*. Tech. rep.

[13] Adré Luís De Medeiros Santos. "Compilation by Transformation in Non-Strict Functional Languages". PhD thesis. 1995, p. 218. ISBN: 0520239601 (alk. paper).

[14] Tanel Tammet. "Lambda-lifting as an optimization for compiling Scheme to C". In: (1996).

[15] Youfeng Wu and James R. Larus. "Static branch frequency and program profile analysis". In: *Professional Engineering* (1994). ISSN: 09536639. DOI: 10.1109/MICRO.1994.717399.