

Lucrative Late Lambda Lifting

Sebastian Graf

September 25, 2018

1 Introduction

2 Transformation

Lambda lifting is a well-known technique [2]. Although Johnsson’s original algorithm runs in worst-case cubic time relative to the size of the input program, Morazán and Schultz [5] gave an algorithm that runs in $\mathcal{O}(n^2)$.

Our lambda lifting transformation is unique in that it operates on terms of the *spineless tagless G-machine* (STG) [1] as currently implemented [3] in GHC. This means we can assume that the nesting structure of bindings corresponds to the condensation (the directed acyclic graph of strongly connected components) of the dependency graph. **TODO: less detail? less language?** Additionally, every binding in a (recursive) **let** expression is annotated with the free variables it closes over. The combination of both properties allows efficient construction of the set of *required* **TODO: any better names? former free variables, abstraction variables...** variables for a total complexity of $\mathcal{O}(n^2)$, as we shall see.

2.1 Syntax

Although STG is but a tiny language compared to typical surface languages such as Haskell, its definition [3] still contains much detail irrelevant to lambda lifting. As can be seen in fig. 1, we therefore adopt a simple lambda calculus with **let** bindings as in Johnsson [2], with a few STG-inspired features:

1. **let** bindings are annotated with the non-top-level free variables they close over
2. Every lambda abstraction is the right-hand side of a **let** binding
3. Arguments and heads in an application expression are all atomic (e.g. variable references)

Variables	f, x, y	
Expressions	$e ::= x$ $\quad \quad f \ x_1 \dots x_n$ $\quad \quad \mathbf{let} \ b \ \mathbf{in} \ e$	Variable Function call Recursive let
Bindings	$b ::= \frac{}{f_i = [x_{i,1} \dots x_{i,n_i}] \lambda y_{i,1} \dots y_{i,m_i} \rightarrow e_i}$	

Figure 1: An STG-like untyped lambda calculus

2.2 Algorithm

Our implementation extends the original formulation of Johnson [2] to STG terms, by exploiting and maintaining closure annotations. We will recap our variant of the algorithm in its whole here. 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:

TODO: Why not use plain Haskell? TODO: Why not formulate this as inference rules?

$\text{lift_}(-) : \text{Expander} \rightarrow \text{Expr} \rightarrow \mathcal{W}_{\text{Bind}} \text{Expr}$

As its first argument, `lift` takes an `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 Johnson [2]. 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 *writer monad*, denoted by $\mathcal{W}_{\text{Bind}}$.

2.2.1 Side-effects

TODO: Properly define the structure? Or is this 'obvious'? The following syntax, inspired by *idiom brackets* [4] and *bang notation*¹, will allow concise notation while hiding sprawling state threading:

$\llbracket E[\langle e_1 \rangle, \dots, \langle e_n \rangle] \rrbracket$

¹<http://docs.idris-lang.org/en/v1.3.0/tutorial/interfaces.html>

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 $\gg=$ 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.

$$\text{lift}_\alpha(x) = \begin{cases} \llbracket x \rrbracket, & x \notin \text{dom } \alpha \\ \llbracket x \ y_1 \dots y_n \rrbracket, & \alpha(x) = \{y_1, \dots, y_n\} \end{cases}$$

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 occurring 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** binding²:

TODO: The application rule is unnecessarily complicated because we support occurrences of lifted binders in argument position. Lifting such binders isn't worthwhile anyway (see section 3). Maybe just say that we don't allow it?

$$\text{lift}_\alpha(f \ x_1 \dots x_n) = \llbracket (\text{wrap}_\alpha(x_n) \circ \dots \circ \text{wrap}_\alpha(x_1))(\langle \text{lift}_\alpha(f) \rangle \ x'_1 \dots x'_n) \rrbracket$$

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**:

$$\text{wrap}_\alpha(x)(e) = \begin{cases} \text{let } x' = [x]\lambda \rightarrow x \text{ in } e, & x \notin \text{dom } \alpha \\ \text{let } x' = []\lambda y_1 \dots y_n \rightarrow x \ y_1 \dots y_n \text{ in } e, & \alpha(x) = \{y_1, \dots, y_n\} \end{cases}$$

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

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:

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

1. The first step is to expand closures mentioned in bs with the help of α .
2. Second, a heuristic (that of section 3, for example) decides whether to lift the binding group bs to top-level or not.
3. 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.

$$\text{expand-closures}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}] \lambda z_1 \dots z_{m_i} \rightarrow e_i}) = \overline{f_i = [y_1 \dots y_{n'_i}] \lambda z_1 \dots z_{m_i} \rightarrow e_i}$$

where

$$\{y_1 \dots y_{n'_i}\} = \bigcup_{j=1}^{n_i} \begin{cases} x_j, & x_j \notin \text{dom } \alpha \\ \alpha(x_j), & \text{otherwise} \end{cases}$$

expand-closures substitutes all occurrences of lifted binders (those that are in $\text{dom } \alpha$) in closures of a given binding group by their required set.

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

where

$$\alpha' = \alpha \left[\overline{f_i} \mapsto \text{fvs}(bs) \right] \text{ for } \overline{f_i} = [-] \lambda_- \rightarrow_- = bs$$

decide-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 **lambda-lifted** 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.

$$\text{fvs}(\overline{f_i = [x_1 \dots x_{n_i}] \lambda y_1 \dots y_{m_i} \rightarrow e_i}) = \bigcup_i \{x_1, \dots, x_{n_i}\} \setminus \overline{f_i}$$

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.

$$\text{lambda-lift}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}] \lambda y_1 \dots y_{m_i} \rightarrow e_i}) = \overline{f_i = [] \lambda \alpha(f_i) y_1 \dots y_{m_i} \rightarrow e_i}$$

The syntactic lambda lifting is performed in **lambda-lift**, where closure variables are removed in favor of a number of parameters, one for each element of the respective binding’s required set.

$$\text{recurse}(e)(bs, \alpha, lbs) = \text{lift-bind}_\alpha(lbs) \gg \text{note} \gg \llbracket \text{let } \langle \text{lift-bind}_\alpha(bs) \rangle \text{ in } \langle \text{lift}_\alpha(e) \rangle \rrbracket$$

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 returned³.

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

$$\text{lift-bind}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}] \lambda y_1 \dots y_{m_i} \rightarrow e_i}) = \llbracket \overline{f_i = [x_1 \dots x_{n_i}] \lambda y_1 \dots y_{m_i} \rightarrow \langle \text{lift}_\alpha(e_i) \rangle} \rrbracket$$

3 When to lift

Lambda lifting a binding to top-level is always **TODO: except when we would replace a parameter occurrence by an application** 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.

3.1 Syntactic consequences

Deciding to lift a binding **let** $f = [x \ y \ z] \lambda a \ b \ c \rightarrow e_1$ **in** e_2 to top-level 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_2 by an application of the lifted top-level binding to its former free variables, replacing the whole **let** binding by the term $[f \mapsto f_\uparrow x \ y \ z] \ e_2$. **TODO: Maybe less detail here**
- (S4) All non-top-level variables that occurred in the **let** binding’s right-hand side become parameter occurrences.

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

Consider what happens if f occurred in 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.

An easy fix would be to bind the complex expression to an auxiliary **let** binding, thereby re-introducing the very allocation we wanted to eliminate through lambda lifting **TODO: Move this further down?**. Therefore, we can identify a first criterion for non-beneficial lambda lifts:

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

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.

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. Consider this example:

```
let f = [x y]λa b → λdots
  g = [f x]λd → f d d + x
in g 5
```

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

```
f↑ = λx y a b → λdots;
let g = [x y]λd → f↑ 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 $f↑$), 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

Estimation of closure growth is crucial to identifying beneficial lifting opportunities. We discuss this further in 3.3.

Calling Convention. (S4) means that more arguments have to be passed. Depending on the target architecture, this means 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 hardware registers (e.g. 5 arguments on x86_64)

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

```
let f = []λx → 2 * x
    mapF = [f]λxs → λdots f xλdots
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 [3]. Lifting *mapF* (but not *f*) yields the following program:

```
mapF↑ = λf xs → λdots f xλdots;
let f = []λx → 2 * x
in mapF↑ f [1, 2, 3]
```

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

Undersaturated calls. When GHC spots an undersaturated call, it arranges allocation of a partial application that closes over the supplied arguments. Pay attention to the call to *f* in the following example:

```
let f = [x]λy z → x + y + z;
in map (f x) [1, 2, 3]
```

Here, the undersaturated (e.g. curried) call to *f* leads to the allocation of a partial application, carrying two pointers, to *f* and *x*, respectively. What happens when *f* is lambda lifted?

```
f↑ = λx y z → x + y + z;
map (f↑ x x) [1, 2, 3]
```

The call to *f_↑* will still allocate a partial application, with the only difference that it now also closes over *f*’s free variable *x*, canceling out the beneficial effects of (S1). Hence

(C5) Don’t lift a binding that has undersaturated calls

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.

(C6) Don’t lift a binding that is updatable or a constructor application

3.3 Estimating Closure Growth

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

Let’s revisit the example from above:

```
let f = [x y]λa b → λdots
    g = [f x]λd → f d d + x
in g 5
```

We concluded that lifting f would be beneficial, saving us allocation of one free variable slot. 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:

```

let  $f = [x\ y]\lambda a\ b \rightarrow \lambda \text{dots}$ 
 $g = [f\ x]\lambda d \rightarrow$ 
  let  $h = [f]\lambda e \rightarrow f\ e\ e$ 
  in  $h\ d$ 
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 occurring 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:

```

let  $f = [x\ y]\lambda a\ b \rightarrow \lambda \text{dots}$ 
 $g = [f\ x\ y]\lambda d \rightarrow$ 
  let  $h_1 = [f]\lambda e \rightarrow f\ e\ e$ 
   $h_2 = [f\ x\ y]\lambda e \rightarrow f\ e\ e + x + y$ 
  in  $h_1\ d + h_2\ d$ 
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. We express this in our cost model by an infinite closure growth whenever there was any positive closure growth under a multi-shot lambda.

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

Instead, our implementation first abstracts 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 (e.g. literals don't allocate).

References

- [1] 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.
- [2] 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.
- [3] 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>.
- [4] CONOR MCBRIDE and ROSS PATERSON. *Applicative programming with effects*. 2008. DOI: 10.1017/S0956796807006326.
- [5] 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.