Lucrative Late Lambda Lifting

Sebastian Graf

September 19, 2018

1 Introduction

2 Transformation

Lambda lifting is a well-known technique [2]. Although Johnsson's original algorithm runs in wort-case cubic time relative to the size of the input program, Morazán and Schultz [4] 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 free variables they close over
2. Arguments in an application expression are all atomic (e.g. variable references)
3. Every lambda abstraction is the right-hand side of a **let** binding
4. All applications are fully saturated

$$
\begin{array}{rlll}
\text{Variables} & f, x, y & & \\
\text{Expressions} & e ::= & x & \text{Variable} \\
 & \mid & f\ x_1 \ldots x_n & \text{Saturated function call} \\
 & \mid & \textsf{let}\ b\ \textsf{in}\ e & \text{Recursive } \textsf{let} \\
\text{Bindings} & b ::= & \overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\lambda y_{i,1} \ldots y_{i,m_i} \to e_i} &
\end{array}
$$

Figure 1: An STG-like untyped lambda calculus

2.2 Algorithm

Our implementation extends the original formulation of Johnsson [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 function lift recursively over the term structure. This is its signature:

$$\mathsf{lift}_{\_}(\_) \colon \mathsf{Expander} \to \mathsf{Expr} \to \mathsf{Expr} \times \mathsf{Bind}$$

As first argument lift takes an Expander, which is a partial function from lifted binders to the set 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 [2]. We write $\mathsf{dom}\,\alpha$ for the domain of the expander α and the notation $\alpha[\mathsf{x} \mapsto S]$ to extend 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 a pair of

1. An expression that no longer contains any bindings that were lifted to the top-level
2. A binding group of the bindings that were lifted to the top-level

2.2.1 Variables

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

$$\begin{aligned}
\mathsf{lift}(x) &= \begin{cases} x, & x \notin \mathsf{dom}\,\alpha \\ x\ y_1 \ldots y_n, & \alpha(x) = \{y_1, \ldots, y_n\} \end{cases} \\
\mathsf{lift}(f\ x_1 \ldots x_n) &= (\mathsf{wrap}(x_n) \circ \ldots \circ \mathsf{wrap}(x_1) \circ \mathsf{lift})(f\ \mathsf{subst}(x_1) \ldots \mathsf{subst}(x_n)) \\
\mathsf{wrap}(x)(e) &= \begin{cases} e, & x \notin \mathsf{dom}\,\alpha \\ \textbf{let}\ \mathsf{subst}(x) = [\,]\lambda y_1 \ldots y_n \to x\ y_1 \ldots y_n\ \textbf{in}\ e, & \alpha(x) = \{y \end{cases} \\
\mathsf{subst}(x) &= \begin{cases} x, & x \notin \mathsf{dom}\,\alpha \\ x', & x'\ \text{fresh} \end{cases} \\
\mathsf{lift}(\textbf{let}\ \overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\lambda y_{i,1} \ldots y_{i,m_i} \to e_i}\ \textbf{in}\ e) &= \begin{cases} x, & x \notin \mathsf{dom}\,\alpha \\ x\ y_1 \ldots y_n, & \alpha(x) = \{y_1, \ldots, y_n\} \end{cases}
\end{aligned}$$

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?

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** $f = [x\ y\ z]\lambda a\ b\ c \to e_1$ **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.

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 liftingTODO: 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:

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

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

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

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

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:

$$\begin{array}{l} \textbf{let}\ f = [\,]\lambda x \to 2 * x \\ \quad mapF = [f]\lambda xs \to \lambda ldots\ f\ x \lambda ldots \\ \textbf{in}\ mapF\ [1, 2, 3] \end{array}$$

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:

$\mathit{mapF}_{\uparrow} = \lambda f\ xs \to \lambda ldots\ f\ x \lambda ldots;$
let $f = [\,]\lambda x \to 2 * x$
in $\mathit{mapF}_{\uparrow}\ f\ [1, 2, 3]$

(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 $f = [x]\lambda y\ z \to x + y + z;$
in $\mathit{map}\ (f\ x)\ [1, 2, 3]$

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?
$f_{\uparrow} = \lambda x\ y\ z \to x + y + z;$
$\mathit{map}\ (f_{\uparrow}\ x\ x)\ [1, 2, 3]$

The call to $f_{\uparrow}$ 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 criterions 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 $f = [x\ y]\lambda a\ b \to \lambda ldots$
  $g = [f\ x]\lambda d \to f\ d\ d + x$
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 $f = [x\ y]\lambda a\ b \to \lambda ldots$
  $g = [f\ x]\lambda d \to$
    let $h = [f]\lambda e \to f\ e\ e$
    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 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:

```
let $f = [x\ y]\lambda a\ b \to \lambda ldots$
  $g = [f\ x\ y]\lambda d \to$
    let $h_1 = [f]\lambda e \to f\ e\ e$
      $h_2 = [f\ x\ y]\lambda e \to f\ e\ e + x + y$
    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] 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.