Lucrative Late Lambda Lifting

Sebastian Graf

September 7, 2018

1 Introduction

2 Transformation

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

```
let f = [x y]λa b → ...
    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 → ...;
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 $f_\uparrow$), 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:

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

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

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

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 $map\ (f\ x)\ [1,2,3]$

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?

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

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 $f = [x\ y]\lambda a\ b \to ...$
  $g = [f\ x]\lambda d \to f\ d\ d + x$
in $g\ 5$

Will lifting