Selective Lambda Lifting

Sebastian Graf, sebastian.graf@kit.edu

24 June 2019

Karlsruhe Institute of Technology

Motivation

```
f a 0 = a
f a n = f (g (n `mod` 2)) (n-1)
  where
    g 0 = a
    g n = 1 + g (n-1)
```

Motivation

```
f a 0 = a
f a n = f (g' a (n `mod` 2)) (n-1)

g' a 0 = a
g' a n = 1 + g' a (n-1)
```

Motivation

```haskell
f :: [Int] -> [Int] -> Int -> Int
f a b 0 = a
f a b 1 = b
f a b n = f (g n) a (n `mod` 2)
  where
    g 0 = a
    g 1 = b
    g n = n : h
      where
        h = g (n-1)
```

Motivation

```haskell
f :: [Int] -> [Int] -> Int -> Int
f a b 0 = a
f a b 1 = b
f a b n = f (g' a b n) a (n `mod` 2)

g' a b 0 = a
g' a b 1 = b
g' a b n = n : h
  where
    h = g' a b (n-1)
```

Closure Conversion vs. Lambda Lifting

- Codegen strategies: turn local functions into global functions and auxiliary heap allocations
- Closure Conversion: References to free variables lowered as fields accesses on a closure record containing all FVs
- Lambda Lifting: Convert free variables into parameters, supplied as additional arguments at call sites

```
let f = \a b -> a*x+b*y
in f 4 2
```

$\xrightarrow{\text{CC } f}$

```
data EnvF = EnvF
  { x :: Int
  , y :: Int }
f' env a b =
  a*x env + b*y env;
let f = (f', EnvF x y)
in (fst f) (snd f) 4 2
```

Closure Conversion vs. Lambda Lifting

- Codegen strategies: turn local functions into global functions and auxiliary heap allocations
- Closure Conversion: References to free variables lowered as fields accesses on a closure record containing all FVs
- Lambda Lifting: Convert free variables into parameters, supplied as additional arguments at call sites

```
let f = \a b -> a*x+b*y
in f 4 2
```

$\xrightarrow{\text{LL } f}$

```
f' x y a b = a*x + b*y;
f' x y 4 2
```

When to lift?

When to lift?

When *not* to lift?

When to lift?

```
let f = \a b c -> a*x + b*y + z
in g 5 x f
```

When *not* to lift?

- Argument occurrences

When to lift?

When *not* to lift?

- Argument occurrences

```
let f = \a b c -> a*x + b*y + z
in g 5 x f
```


```
f' a b c = a*x + b*y + z;
g 5 x (f' x y)
```

When to lift?

When *not* to lift?

- Argument occurrences

```
let f = \a b c -> a*x + b*y + z
in g 5 x f
```


```
f' a b c = a*x + b*y + z;
let f = f' x y
in g 5 x f
```

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth

```
let f = \a b -> a*x + b*y
    g = \d -> f d d + x
in g 5
```

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth

```
let f = \a b -> a*x + b*y
    g = \d -> f d d + x
in g 5
```


```
f' x y a b = a*x + b*y;
let g = \d -> f x y d d + x
in g 5
```

When to lift?

```
let f = \a b c d -> a*b*c*d*x*y*z
in f 1 2 3 4
```

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention

```
let f = \a b c d -> a*b*c*d*x*y*z
in f 1 2 3 4
```


```
f' a b c d = a*b*c*d*x*y*z;
in f' x y z 1 2 3 4
```

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention
- Known calls to FVs

```
let f = \x -> 2*x
    mapF = \xs -> case xs of
      []     -> []
      x:xs' -> f x : mapF xs'
in mapF [1..n]
```

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention
- Known calls to FVs

```
let f = \x -> 2*x
    mapF = \xs -> case xs of
      []     -> []
      x:xs'  -> f x : mapF xs'
in mapF [1..n]
```

⇓ LL *mapF*

```
mapF []       = [];
mapF (x:xs') = f' x : mapF xs';
let f = \x -> 2*x
in mapF' f [1..n]
```

When to lift?

```
let p = (,) x y
in fst p + snd p
```

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention
- Known calls to FVs
- Sharing

When to lift?

When *not* to lift?

- Argument occurrences
- Closure growth
- Calling convention
- Known calls to FVs
- Sharing

```
let p = (,) x y
in fst p + snd p
```


```
p x y = (,) x y
fst (p x y) + snd (p x y)
```

Estimating Closure Growth

Estimating Closure Growth

```
let f = \a b -> a*x + b*y
    g = \d -> f d d + x
in g 5
```


```
f' x y a b = a*x + b*y;
let g = \d -> f x y d d + x
in g 5
```

Estimating Closure Growth

- Closure alloc minus syntactic call sites?

```
let f = \a b -> a*x + b*y
    g = \d -> f d d + x
in g 5
```

$\Downarrow$ LL f

```
f' x y a b = a*x + b*y;
let g = \d -> f x y d d + x
in g 5
```

Estimating Closure Growth

- Closure alloc minus syntactic call sites?

```
let f = \a b -> a*x + b*y
    g = \d ->
      let h = \e -> f e e
      in h x
in g 1 + g 2 + g 3
```


```
f' x y a b = a*x + b*y;
let g = \d ->
      let h = \e -> f' x y e e
      in h x
in g 1 + g 2 + g 3
```

Estimating Closure Growth

- Closure alloc minus syntactic call sites? ✗

```
let f = \a b -> a*x + b*y
    g = \d ->
      let h = \e -> f e e
      in h x
in g 1 + g 2 + g 3
```


```
f' x y a b = a*x + b*y;
let g = \d ->
      let h = \e -> f' x y e e
      in h x
in g 1 + g 2 + g 3
```

Estimating Closure Growth

- Closure alloc minus syntactic call sites? ✗
- Don't lift multi-shot occurrences?
-

```
let f = \a b -> a*x + b*y
    g = \d ->
      let h = \e -> f e e
      in h x
in g 1 + g 2 + g 3
```

⇓ LL *f*

```
f' x y a b = a*x + b*y;
let g = \d ->
      let h = \e -> f' x y e e
      in h x
in g 1 + g 2 + g 3
```

Estimating Closure Growth

- Closure alloc minus syntactic call sites? ✗
- Don't lift multi-shot occurrences? ✗
- ✗

```
let f = \a b -> a*x + b*y
    g = \d ->
      let h = \e -> f e e
      in h x
in g 1 + g 2 + g 3
```

⇓ LL *f*

```
f' x y a b = a*x + b*y;
let g = \d ->
      let h = \e -> f' x y e e
      in h x
in g 1 + g 2 + g 3
```

Example that slows down by 10%