Call Arity

Joachim Breitner
27 May 2014
15th Symposium on Trends in Functional Programming
Soesterberg, The Netherlands

More formally... the equations (I)

\[
\begin{align*}
A_n(\lambda x. e) &= C_n(e) \\
C_{n+1}(\lambda x. e) &= (\text{fv}(e))^2 \\
C_0(\lambda x. e) &= A_0(e) \sqcup A_n(e_1) \sqcup A_n(e_2) \\
A_n(e ? e_1 : e_2) &= C_0(e) \sqcup C_n(e_1) \sqcup C_n(e_2) \sqcup \text{fv}(e) \\
C_n(e ? e_1 : e_2) &= (\text{fv}(e))^2
\end{align*}
\]
How many lists do you see?

foldl (+) 0 [1..1000]
The bad and the good code

The bad code:

```ocaml
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in \z \rightarrow ds (z + x)
in go 1 0
```

The good code:

```ocaml
let go x z =
    let ds z' = if x == 1000 then z' else go (x + 1) z'
    in ds (z + x)
in go 1 0
```
The bad code:

```ml
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in ds (x + z)
in go 1 0
```

The good code:

```ml
let go x z =
  let ds z' = if x == 1000 then z' else go (x + 1) z'
  in ds (z + x)
in go 1 0
```
The bad and the good code

The bad code:

```plaintext
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in \z → ds (z + x)
in go 1 0
```

The good code:

```plaintext
let go x z =
    let ds z' = if x == 1000 then z' else go (x + 1) z'
    in ds (z + x)
in go 1 0
```

The goal: Eta-expand go and ds.
When is eta-expansion allowed?

The bad code:

```ml
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in \z → ds (z + x)
in go 1 0
```

We can eta-expand f with n arguments, if

- every call to f has (at least) n arguments on the stack
- if f is a thunk, i.e. not in head-normal form, if f is called at most once.
The analysis: What we want and what we need

The bad code:

```haskell
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in \z \rightarrow ds (z + x)
in go 1 0
```

What do we want to know, for a let-bound variable?
- A lower bound to the number of arguments it is called.
- If it may be called more than once.
The analysis: What we want and what we need

The bad code:

```haskell
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in \z \rightarrow ds (z + x)
in go 1 0
```

What do we want to know, for a let-bound variable?

- A lower bound to the number of arguments it is called.
- If it may be called more than once.

So for an expression, we need this information about all its free variables.
The analysis: What we want and what we need

The bad code:

```haskell
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in \z → ds (z + x)
in go 1 0
```

What do we want to know, for a let-bound variable?
- A lower bound to the number of arguments it is called.
- If it may be called more than once.

So for an expression, we need this information about all its free variables, under the assumption that the expression is called with a certain number of arguments.
We need more information:

```
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
    in go2 x
in go 1 0
```

`go2` is recursive, and calls `ds`. How do we know that `let go2 = ... in go2 x` calls `ds` at most once?

So the analysis finds out:

For every two variables `f` and `g`, can `e` call both `f` and `g`? (Includes as a special case: Can `e` call `f` twice?)
We need more information:

```plaintext
let go x =
  let ds = if x == 1000 then id else go (x + 1) in
  let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1) in
  go2 x
in go 1 0

go2 is recursive, and calls ds.
How do we know that let go2 = ... in go2 x calls ds at most once?
```
We need more information:

\[
\text{let } \text{go } x = \\
\quad \text{let } \text{ds } = \text{if } x == 1000 \text{ then } \text{id} \text{ else } \text{go } (x + 1) \\
\quad \text{in} \\
\quad \quad \text{let } \text{go2 } y = \text{if } \text{odd } y \text{ then } \lambda z \rightarrow \text{ds } (z + y) \text{ else } \text{go2 } (y + 1) \\
\quad \quad \text{in } \text{go2 } x \\
\text{in } \text{go } 1 \ 0
\]

go2 is recursive, and calls ds. How do we know that \text{let } \text{go2 } = \ldots \text{ in } \text{go2 } x \text{ calls ds at most once?}

So the analysis finds out:

For every two variables \( f \) and \( g \), can \( e \) call both \( f \) and \( g \)?

(Includes as a special case: Can \( e \) call \( f \) twice?)
Let’s see it happen

```ocaml
let go x =
  let ds = if x == 1000 then id else go (x + 1) in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1) in
  go2 x
in go 1 0
```

Incoming arity: 0
Free variables arity: ?
Co-call information: ?
Let’s see it happen

```plaintext
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 0
Free variables arity: ?
Co-call information: ?
Let’s see it happen

```ml
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 2
Free variables arity: \{ go \rightarrow 2 \}  
Co-call information: \{\}
Let's see it happen

```haskell
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
        let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
            in go2 x
    in go 1 0
```

Incoming arity: 0
Free variables arity: \{go ↦ 2\}
Co-call information: \{\}

Let’s see it happen

```ocaml
let go x =
  let ds = if x == 1000 then id else go (x + 1) in
  let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1) in
  go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: ?
Co-call information: ?
Let’s see it happen

```ocaml
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
    let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
    in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: ?
Co-call information: ?
Let’s see it happen

```ocaml
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: ?
Co-call information: ?
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z \mapsto ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0

Incoming arity: 2
Free variables arity: {go2 \mapsto 2}
Co-call information: { }
Let’s see it happen

```haskell
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
        let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
        in go2 x
    in go 1 0

Incoming arity: 1
Free variables arity: ?
Co-call information: ?
```
Let’s see it happen

```haskell
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
        let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
        in go2 x
    in go 1 0

Incoming arity: 0
Free variables arity: \{odd \mapsto 1\}
Co-call information: \{\}
```
Let’s see it happen

```haskell
let go x =
  let ds = if x == 1000 then id else go (x + 1) in
  let go2 y = if odd y then \( z \rightarrow ds (z + y) \) else go2 (y + 1) in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: ?
Co-call information: ?
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
    let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
    in go2 x
  in go 1 0

Incoming arity: 0
Free variables arity: ?
Co-call information: ?
Let’s see it happen

let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0

Incoming arity: 1
Free variables arity: \{ds ↦ 1\}
Co-call information: { }
Let’s see it happen

```plaintext
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 0
Free variables arity: \{ds \mapsto 1\}
Co-call information: {}
Let’s see it happen

```haskell
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z \rightarrow ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: \{ go2 \mapsto 2 \}
Co-call information: \{ \}
Let’s see it happen

```ocaml
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z -> ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: \{\text{odd} \mapsto 1, \text{ds} \mapsto 1, \text{go2} \mapsto 2\}
Co-call information: \{\text{odd}--\text{ds}, \text{odd}--\text{go2}\}
Let's see it happen

```plaintext
let go x =
  let ds = if x == 1000 then id else go (x + 1)
  in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
  in go2 x
in go 1 0

Incoming arity: 1
Free variables arity: \{odd → 1, ds → 1\}
Co-call information: \{odd—ds, odd—odd\}
```
Let's see it happen

```ml
let go x =
  let ds = if x == 1000 then id else go (x + 1) in
  let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1) in go2 x
in go 1 0
```

Incoming arity: 1
Free variables arity: \{id ↦ 1, go ↦ 2\}
Co-call information: {}
let go x =
    let ds = if x == 1000 then id else go (x + 1)
    in
    let go2 y = if odd y then \z → ds (z + y) else go2 (y + 1)
    in go2 x
in go 1 0

Incoming arity: 1
Free variables arity: {odd ↦ 1, id ↦ 1, go ↦ 2}
Co-call information: {odd—id, odd—go, odd—odd}
Results

- length \([1..2^{30}]\): 11.7s instead of 16.3s.
- Nofib, without changing foldl:
  
<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocations</td>
<td>-1.3%</td>
<td>-0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Runtime</td>
<td>-4.0%</td>
<td>-0.0%</td>
<td>+4.9%</td>
</tr>
</tbody>
</table>

- Nofib, with changing foldl:
  
<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocations</td>
<td>-79.0%</td>
<td>-5.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Runtime</td>
<td>-47.4%</td>
<td>-1.9%</td>
<td>+3.0%</td>
</tr>
</tbody>
</table>
Summary

- Self-contained, heuristics-free analysis
- Implemented and deployed in GHC
- Relevant for ubiquitous list fusion

Also in the paper:
- Precise description of the analysis (formulas! maths!)
- Notes on the implementation
- Limitations
- Comparison with related work and other approaches

Future work:
- Formal and machine-checked proof of correctness.
More formally... the components

\[ e \colon \text{Expr} \]
\[ e ::= x \mid e_1 \; e_2 \mid (\lambda x.\; e_1) \mid e \; ? 
\]  
\[ e_1 : e_2 \mid \text{let } x_i = e_i \text{ in } e \quad \text{expressions} \]

\[ A_n \colon \text{Expr} \to (\text{Var} \to \mathbb{N}) \quad \text{arity analysis} \]

\[ C_n \colon \text{Expr} \to \text{Graph(Var)} \quad \text{co-call analysis} \]

\[ \text{fv} \colon \text{Expr} \to \mathcal{P}(\text{Var}) \quad \text{free variables} \]

\[ \sqcap \colon (\text{Var} \to \mathbb{N}) \to (\text{Var} \to \mathbb{N}) \to (\text{Var} \to \mathbb{N}) \quad \text{point-wise minimum} \]

\[ \times : \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Graph(Var)} \quad \text{complete bi-partite graph} \]

\[ ^2 : \mathcal{P}(\text{Var}) \to \text{Graph(Var)} \quad \text{complete graph} \]
More formally... the equations (I)

\[
\begin{align*}
A_n(x) &= \{ x \mapsto n \} \\
C_n(x) &= \{} \\
A_n(e_1 e_2) &= A_{n+1}(e_1) \sqcup A_0(e_2) \\
C_n(e_1 e_2) &= C_{n+1}(e_1) \cup C_0(e_2) \cup \text{fv}(e_1) \times \text{fv}(e_2) \\
A_{n+1}(\lambda x.e) &= A_n(e) \\
A_0(\lambda x.e) &= A_0(e) \\
C_{n+1}(\lambda x.e) &= C_n(e) \\
C_0(\lambda x.e) &= (\text{fv}(e))^2 \\
A_n(e ? e_1 : e_2) &= A_0(e) \sqcup A_n(e_1) \sqcup A_n(e_2) \\
C_n(e ? e_1 : e_2) &= C_0(e) \cup C_n(e_1) \cup C_n(e_2) \cup \text{fv}(e) \times (\text{fv}(e_1) \cup \text{fv}(e_2))
\end{align*}
\]
More formally... the equations (II)

Non-recursive binding (let \( x = e_1 \) in \( e_2 \)):

\[
n_x = \begin{cases} 
0 & \text{if } x \!\vdash x \in C_n(e_2) \text{ and } e_1 \text{ not in HNF} \\
A_n(e_2)[x_i] & \text{otherwise}
\end{cases}
\]

\[
C_{\text{rhs}} = \begin{cases} 
C_{n_x}(e_1) & \text{if } x \!\vdash x \not\in C_n(e_2) \text{ or } n_x = 0 \\
fv(e_1)^2 & \text{otherwise}
\end{cases}
\]

\[
E = fv(e_1) \times \{ v \mid v \!\vdash x \in C_n(e_2) \}
\]

\[
A_n(\text{let } x = e_1 \text{ in } e_2) = A_{n_x}(e_1) \sqcup A_n(e_2)
\]

\[
C_n(\text{let } x = e_1 \text{ in } e_2) = C_{\text{rhs}} \cup A_n(e_2) \cup E
\]
More formally... the equations (III)

Mutually recursive bindings:
Let $A = A_n(\text{let } \overline{x_i = e_i} \text{ in } e)$ and $C = C_n(\text{let } \overline{x_i = e_i} \text{ in } e)$.

\[
A = A_n(e) \sqcup \bigsqcup_i A_{n_{x_i}}(e_i)
\]
\[
C = C_n(e) \sqcup \bigsqcup_i C^i \sqcup \bigsqcup_i E^i
\]

\[
n_{x_i} = \begin{cases} 
0 & \text{if } e_i \text{ not in HNF} \\
A[x_i] & \text{otherwise}
\end{cases}
\]

\[
C^i = \begin{cases} 
C_{n_{x_i}}(e_i) & \text{if } x_i \rightarrow x_i \notin C \text{ or } n_{x_i} = 0 \\
fv(e_i)^2 & \text{otherwise}
\end{cases}
\]

\[
E^i = \begin{cases} 
fv(e_i) \times \{v \mid v \rightarrow x_k \in C_n(e) \sqcup \bigsqcup_j C^j\} & \text{if } n_{x_i} \neq 0 \\
fv(e_i) \times \{v \mid v \rightarrow x_k \in C_n(e) \sqcup \bigsqcup_{j \neq i} C^j\} & \text{if } n_{x_i} = 0
\end{cases}
\]
Consider a data type for trees

```haskell
data Tree = Tip Int | Bin Tree Tree
```

and a function `toList :: Tree → [Int]`, set up for list fusion. Then `sum (toList t)` gets rewritten to

```haskell
let go t fn = case t of
    Tip x → (\a → fn (x + a))
    Bin l r → go l (go r fn)
in go t id 0
```

Call Arity does not eta-expand `go`, and even if it would, the code would still be bad.
## Detailed benchmark results: Allocations

<table>
<thead>
<tr>
<th>Arity Analysis</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Call Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>foldl via foldr</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Anna</th>
<th>Bernouilli</th>
<th>Calendar</th>
<th>FFT2</th>
<th>Gen_reregexps</th>
<th>Hidden</th>
<th>Integrate</th>
<th>Minimax</th>
<th>Rewrite</th>
<th>Simple</th>
<th>X2n1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Reduction</td>
<td>-1.3%</td>
<td>-1.4%</td>
<td>+0.0%</td>
<td>+0.0%</td>
<td>0.0%</td>
<td>-4.9%</td>
<td>+3.7%</td>
<td>+3.7%</td>
<td>-1.3%</td>
<td>-53.9%</td>
<td>+33.8%</td>
</tr>
<tr>
<td>Percentage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...and 89 more

<table>
<thead>
<tr>
<th>Summary</th>
<th>Min</th>
<th>Max</th>
<th>Geometric Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>-1.3%</td>
<td>-79.0%</td>
<td>-78.9%</td>
</tr>
<tr>
<td>Percentage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+0.0%</td>
<td>+0.0%</td>
<td>+84.0%</td>
</tr>
<tr>
<td>Percentage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.0%</td>
<td>-5.2%</td>
<td>-1.5%</td>
</tr>
</tbody>
</table>
## Detailed benchmark results: Runtime

<table>
<thead>
<tr>
<th>Arity Analysis</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Call Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>foldl via foldr</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Min</th>
<th>Max</th>
<th>Geometric Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>-6.0%</td>
<td>+4.7%</td>
<td>+0.9%</td>
</tr>
<tr>
<td>bernouilli</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>calendar</td>
<td>-6.0%</td>
<td>-8.9%</td>
<td>-48.7%</td>
</tr>
<tr>
<td>fft2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gen_regexp</td>
<td>-1.2%</td>
<td>-3.3%</td>
<td>-3.3%</td>
</tr>
<tr>
<td>hidden</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integrate</td>
<td>-6.0%</td>
<td>-48.7%</td>
<td>-48.7%</td>
</tr>
<tr>
<td>minimax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rewrite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2n1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>... and 89 more</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Max**: +224.8%  
**Min**: -48.7%  
**Geometric Mean**: +1.2%
foldl as foldr

A left fold implemented as a right fold:

\[ \text{foldl} \ k \ z \ \text{xs} = \text{foldr} (\lambda v \ fn \ z \rightarrow \ fn (k \ z \ v)) \ \text{id} \ \text{xs} \ \text{z} \]

The other code:

\([x..y] = \text{build} (\lambda c \ n \rightarrow \text{fromToFB} \ c \ n \ x \ y)\]
\(\text{build} \ g = g (:) []\]
\(\text{fromToFB} \ c \ n \ x0 \ y =\)

\[
\text{let} \ \text{go} \ x = x \ 'c' \ (\text{if} \ x == y \ \text{then} \ n \ \text{else} \ \text{go} \ (x+1))
\]
\(\text{in} \ \text{go} \ x0\]

The rewrite rule:

\{-# \text{RULES} \ \text{foldr} \ c \ n \ (\text{build} \ g) = g \ c \ n \ #-\}