Karlsruhe Institute of Technology

Deutsche Telekom Stiftung

Call Arity

Joachim Breitner

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

PROGRAMMING PARADIGMS GROUP

$$A_0(\lambda x.e) = \ldots$$

$$C_{n+1}(\lambda x.e) = C_n(e)$$

$$C_0(\lambda x.e) = (\mathrm{fv}(e))^2$$

$$A_n(e\ ?\ e_1 : e_2) = A_0(e) \sqcup A_n(e_1) \sqcup A_n(e_2)$$

$$C_n(e\ ?\ e_1 : e_2) = C_0(e) \cup C_n(e_1) \cup C_n(e_2) \cup \mathrm{fv}(e \ldots$$

How many lists do you see?

foldl (+) 0 [1..1000]

The bad and the good code

Karlsruhe Institute of Technology

The bad code:

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

The bad and the good code

The bad code:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

The analysis: What we want and what we need

The bad code:

```
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.

The analysis: What we want and what we need

The bad code:

```
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
```

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?

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?)

Let's see it happen

Karlsruhe Institute of Technology

```
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

Karlsruhe Institute of Technology

```
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

```
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: 2
Free variables arity: $\{go \mapsto 2\}$
Co-call information: $\{\}$

Let's see it happen

Karlsruhe Institute of Technology

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: $\{go \mapsto 2\}$
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: ?
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: ?
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 $\rightarrow$ 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

```
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: 2
Free variables arity: $\{go2 \mapsto 2\}$
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: ?
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: 0
Free variables arity: $\{\text{odd} \mapsto 1\}$
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: ?
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: 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: $\{ds \mapsto 1\}$
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: 0
Free variables arity: $\{ds \mapsto 1\}$
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: $\{go2 \mapsto 2\}$
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: $\{\text{odd} \mapsto 1, \text{ds} \mapsto 1, \text{go2} \mapsto 2\}$
Co-call information: $\{\text{odd—ds}, \text{odd—go2}\}$

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 $\rightarrow$ ds (z + y) **else** go2 (y + 1)
    in go2 x
in go 1 0

Incoming arity: 1
Free variables arity: $\{\text{odd} \mapsto 1, \text{ds} \mapsto 1\}$
Co-call information: $\{\text{odd—ds}, \text{odd—odd}\}$

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: $\{\text{id} \mapsto 1, \text{go} \mapsto 2\}$

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: $\{\text{odd} \mapsto 1, \text{id} \mapsto 1, \text{go} \mapsto 2\}$

Co-call information: $\{\text{odd—id}, \text{odd—go}, \text{odd—odd}\}$

Results

- length [1..2^30]: 11.7s instead of 16.3s.
- Nofib, without changing foldl:

	min	mean	max
Allocations	-1.3%	-0.0%	0.0%
Runtime	-4.0%	-0.0%	+4.9%

- Nofib, with changing foldl:

	min	mean	max
Allocations	-79.0%	-5.2%	0.0%
Runtime	-47.4%	-1.9%	+3.0%

Summary

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

Also in the paper:

- Precise description of the analyis (formulas! maths!)
- Notes on the implementation
- Limitations
- Comparision with related work and other approaches

Future work:

- Formal and machine-checked proof of correctness.

More formally... the components

$e\colon \text{Expr}$	expressions
$e ::= x \mid e_1\ e_2 \mid (\lambda x. e_1) \mid e\ ?\ e_1 : e_2 \mid \text{let}\ \overline{x_i = e_i}\ \text{in}\ e$	
$A_n\colon \text{Expr} \to (\text{Var} \rightharpoonup \mathbb{N})$	arity analysis
$C_n\colon \text{Expr} \to \text{Graph}(\text{Var})$	co-call analysis
$\text{fv}\colon \text{Expr} \to \mathcal{P}(\text{Var})$	free variables
$\sqcup\colon (\text{Var} \rightharpoonup \mathbb{N}) \to (\text{Var} \rightharpoonup \mathbb{N}) \to (\text{Var} \rightharpoonup \mathbb{N})$	point-wise minimum
$\times\colon \mathcal{P}(\text{Var}) \to \mathcal{P}(\text{Var}) \to \text{Graph}(\text{Var})$	complete bi-partite graph
${}^2\colon \mathcal{P}(\text{Var}) \to \text{Graph}(\text{Var})$	complete graph

More formally. . . the equations (I)

$$
\begin{aligned}
A_n(x) &= \{x \mapsto n\} \\
C_n(x) &= \{\} \\
A_n(e_1\ e_2) &= A_{n+1}(e_1) \sqcup A_0(e_2) \\
C_n(e_1\ e_2) &= C_{n+1}(e_1) \cup C_0(e_2) \cup \mathrm{fv}(e_1) \times \mathrm{fv}(e_2) \\
A_{n+1}(\lambda x.e) &= A_n(e) \\
A_0(\lambda x.e) &= A_0(e) \\
C_{n+1}(\lambda x.e) &= C_n(e) \\
C_0(\lambda x.e) &= (\mathrm{fv}(e))^2 \\
A_n(e\ ?\ e_1 : e_2) &= A_0(e) \sqcup A_n(e_1) \sqcup A_n(e_2) \\
C_n(e\ ?\ e_1 : e_2) &= C_0(e) \cup C_n(e_1) \cup C_n(e_2) \cup \mathrm{fv}(e) \times (\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2))
\end{aligned}
$$

More formally... the equations (II)

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

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

$$C_{\text{rhs}} = \begin{cases} C_{n_x}(e_1) & \text{if } x\text{—}x \notin C_n(e_2) \text{ or } n_x = 0 \\ \text{fv}(e_1)^2 & \text{otherwise} \end{cases}$$

$$E = \text{fv}(e_1) \times \{v \mid v\text{—}x \in C_n(e_2)\}$$

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

$$C_n(\text{let } x = e_1 \text{ in } e_2) = C_{\text{rhs}} \cup A_n(e_2) \cup E$$

More formally. . . the equations (III)

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

$$A = A_n(e) \sqcup \bigsqcup_i A_{n_{x_i}}(e_i)$$

$$C = C_n(e) \cup \bigcup_i C^i \cup \bigcup_i E^i$$

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

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

$$E^i = \begin{cases} fv(e_i) \times \{v \mid v \text{—} x_k \in C_n(e) \cup \bigcup_j C^j\} & \text{if } n_{x_i} \neq 0 \\ fv(e_i) \times \{v \mid v \text{—} x_k \in C_n(e) \cup \bigcup_{j \neq i} C^j\} & \text{if } n_{x_i} = 0 \end{cases}$$

Limitations

Consider a data type for trees

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

```
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

Arity Analysis	✓	✓		✓
Co-Call Analysis	✓	✓		
foldl via foldr		✓	✓	✓
anna	-1.3%	-1.4%	+0.0%	+0.0%
bernouilli	-0.0%	-4.9%	+3.7%	+3.7%
calendar	-0.1%	-0.2%	-0.1%	-0.1%
fft2	-0.0%	-79.0%	-78.9%	-78.9%
gen_regexps	0.0%	-53.9%	+33.8%	+33.8%
hidden	-0.3%	-6.3%	+1.2%	+1.2%
integrate	-0.0%	-61.7%	-61.7%	-61.7%
minimax	0.0%	-15.6%	+4.0%	+4.0%
rewrite	-0.0%	-0.0%	-0.0%	-0.0%
simple	0.0%	-9.4%	+8.1%	+8.1%
x2n1	-0.0%	-77.4%	+84.0%	+84.0%
. . . and 89 more				
Min	-1.3%	-79.0%	-78.9%	-78.9%
Max	+0.0%	+0.0%	+84.0%	+84.0%
Geometric Mean	-0.0%	-5.2%	-1.5%	-1.5%

Detailed benchmark results: Runtime

Karlsruhe Institute of Technology

Arity Analysis	✓	✓		✓
Co-Call Analysis	✓	✓		
foldl via foldr		✓	✓	✓
`anna`				
`bernouilli`				
`calendar`	+4.7%	+0.8%	+0.8%	+2.3%
`fft2`				
`gen_regexps`	-1.2%	-8.9%	+223.6%	+224.8%
`hidden`	-3.3%	-3.3%	0.0%	0.0%
`integrate`	-6.0%	-48.7%	-48.7%	-48.7%
`minimax`				
`rewrite`	+0.9%	+6.1%	+3.5%	+0.9%
`simple`				
`x2n1`				
. . . and 89 more				
Min	-6.0%	-48.7%	-48.7%	-48.7%
Max	+4.7%	+6.1%	+223.6%	+224.8%
Geometric Mean	-0.2%	-1.4%	+1.0%	+1.2%

foldl as foldr

Karlsruhe Institute of Technology

A left fold implemented as a right fold:

```
foldl k z xs = foldr (\v fn z → fn (k z v)) id xs z
```

The other code:

```
[x..y] = build (\c n → fromToFB c n x y)
build g = g (:) [ ]
fromToFB c n x0 y =
  let go x = x ‘c‘ (if x == y then n else go (x+1))
  in  go x0
```

The rewirte rule:

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