Karlsruhe Institute of Technology

Deutsche Telekom Stiftung

Formally Proving a Compiler Transformation Safe

Joachim Breitner
Haskell Symposium 2015
3 August 2015, Vancouver

PROGRAMMING PARADIGMS GROUP

Short summary

I formally proved that

Call Arity is safe.

Short summary

I formally proved that

Call Arity is safe.

W
H
A
B

Short summary

I formally proved that

Call Arity is safe.

“What exactly have you shown?”

H

A

B

Short summary

I formally proved that

Call Arity is safe.

“What exactly have you shown?”

“How did you prove that?”

A

B

Short summary

I formally proved that

Call Arity is safe.

“What exactly have you shown?”

“How did you prove that?”

“Are you sure about this?”

B

Short summary

I formally proved that

Call Arity is safe.

“What exactly have you shown?”

“How did you prove that?”

“Are you sure about this?”

“But, . . . !”

What exactly is... Call Arity?

Karlsruhe Institute of Technology

Call Arity is an arity analysis:

```
let fac 10 = id                                let fac 10 y = y
    fac x   = λy. fac (x+1) (y∗x)      ⟹          fac x   y = fac (x+1) (y∗x)
in fac 0 1                                     in fac 0 1
```

What exactly is... Call Arity?

Call Arity is an arity analysis:

$$\begin{array}{ll} \textbf{let}\ \text{fac}\ 10 & = \text{id} \\ \quad \text{fac}\ x & = \lambda y.\ \text{fac}\ (x{+}1)\ (y*x) \\ \textbf{in}\ \text{fac}\ 0\ 1 & \end{array} \quad \Longrightarrow \quad \begin{array}{ll} \textbf{let}\ \text{fac}\ 10\ y & = y \\ \quad \text{fac}\ x\quad y & = \text{fac}\ (x{+}1)\ (y*x) \\ \textbf{in}\ \text{fac}\ 0\ 1 & \end{array}$$

So far: Naive forward arity analysis, see Gill's PhD thesis from 96

What exactly is... the problem?

Eta-expanding a thunk is tricky:

let thunk = f x **in** ...	$\Longrightarrow$	**let** thunk y = f x y **in** ...

What exactly is... the problem?

Eta-expanding a thunk is tricky:

let thunk = f x **in** ...	$\Longrightarrow$	**let** thunk y = f x y **in** ...

Sharing can be lost!

What exactly is... the problem?

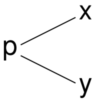
Eta-expanding a thunk is tricky:

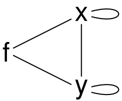
let thunk = f x **in** . . .	$\Longrightarrow$	**let** thunk y = f x y **in** . . .

Sharing can be lost!

(unless “thunk” is used at most once in “. . . ”)

What exactly is... co-call cardinality analysis?

$\mathcal{G}_0(\textbf{if } \mathsf{p} \textbf{ then } \mathsf{x} \textbf{ else } \mathsf{y}) =$ 

$\mathcal{G}_0(\mathsf{f}\ \mathsf{x}\ \mathsf{y}) =$ 

What exactly is... Call Arity?

Call Arity
=
Arity analysis with co-call cardinality analysis

What exactly is... Call Arity?

Call Arity
=
Arity analysis with co-call cardinality analysis

Now foldl can be a good consumer in list-fusion!

What exactly is. . . "safe"?

Safety: It is safe for the compiler to apply the transformation, i.e. the performance will not degrade.

What exactly is… “safe”?

Safety: It is safe for the compiler to apply the transformation, i.e. the performance will not degrade.

Yes, it is synonymous to “improvement”.

What exactly ~~is~~. . . could possibly go wrong?

A bug in Call Arity

$\Downarrow$

What exactly ~~is~~. . . could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

What exactly ~~is~~... could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

Loss of sharing

$\Downarrow$

What exactly ~~is~~. . . could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

Loss of sharing

$\Downarrow$

Work is duplicated

$\Downarrow$

What exactly ~~is~~… could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

Loss of sharing

$\Downarrow$

Work is duplicated

$\Downarrow$

Allocation is increasing

What exactly ~~is~~. . . could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

Loss of sharing

$\Downarrow$

Work is duplicated

$\Downarrow$

Allocation is increasing

Theorem: Call Arity does not increase the number of allocations

What exactly ~~is~~… could possibly go wrong?

A bug in Call Arity

$\Downarrow$

Too much eta-expansion

$\Downarrow$

Loss of sharing

$\Downarrow$

Work is duplicated

$\Downarrow$

Allocation is increasing

No (such) bug

$\Uparrow$

Theorem: Call Arity does not increase the number of allocations

How did you prove that?

1st ingredient Sufficiently detailed semantics:

Launchbury's natural semantics for lazy evaluation.

$$\Gamma : e \Downarrow \Delta : v$$

heap before
current expression
heap afterwards
final value

How did you prove that?

1st ingredient Sufficiently detailed semantics:

Sestoft's mark-1 virtual machine

$$(\Gamma, e, S) \Rightarrow (\Gamma', e', S')$$

current heap
current expression
current stack
next heap
next expression
next stack

How did you prove that?

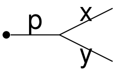
2nd ingredient Abstract view on what calls what:

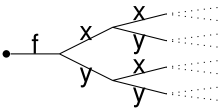
Trace trees!

How did you prove that?

2nd ingredient Abstract view on what calls what:

Trace trees!

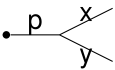
$\mathcal{T}_0(\textbf{if}\ \mathrm{p}\ \textbf{then}\ \mathrm{x}\ \textbf{else}\ \mathrm{y}) =$ 

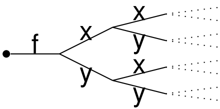
$\mathcal{T}_0(\mathrm{f\ x\ y}) =$ 

How did you prove that?

2nd ingredient Abstract view on what calls what:

Trace trees!

$\mathcal{T}_0(\textbf{if } \mathsf{p} \textbf{ then } \mathsf{x} \textbf{ else } \mathsf{y}) =$ 

$\mathcal{T}_0(\mathsf{f}\ \mathsf{x}\ \mathsf{y}) =$ 

Co-call graphs approximates trace trees

It even is a Galois immersion.

How did you prove that?

3nd ingredient A way to handle a large proof:

Refinement proofs

How did you prove that?

3nd ingredient A way to handle a large proof:

Refinement proofs

Arity analysis + any cardinality analysis	$\xleftarrow{\text{impl.}}$	Arity analysis + a trace tree analysis	$\xleftarrow{\text{approx.}}$	Arity analysis + a co-call graph analysis	$\xleftarrow{\text{impl.}}$	Call Arity

Are you sure?

- Syntax (using Nominal logic)
- Semantics (Launchbury, Sestoft, denotational)
- Data types (Co-call graphs, trace trees)
- ... and of course the proofs

```
lemma end2end_closed:
  assumes closed: "fv e = ({} :: var set)"
  assumes "([], e, []) ⇒* (Γ,v,[])"
  assumes "isVal v"
  obtains Γ' and v'
  where "([], transform 0 e, []) ⇒* (Γ',v',[])"
    and "card (domA Γ') ≤ card (domA Γ)"
    and "isVal v'"
proof-
```

But...

The formalization gap!

But. . .

The formalization gap!

But. . .

The formalization gap!

But. . .

The formalization gap!

Bug #10176

let foo x = error "..."
in ... **case** foo a b **of** ...

⇓ Strictness analyzer

let foo x = error "..."   -- *Strictness:* `<L,U>b`
in ... **case** foo a b **of** ...

⇓ Call Arity

let foo x y = error "..." y   -- *Strictness:* `<L,U>b`
in ... **case** foo a b **of** ...

⇓ Simplifier

let foo x y = error "..." y   -- *Strictness:* `<L,U>b`
in ... **case** foo a **of** {}

Conclusion

Karlsruhe Institute of Technology

Yes, we can. . .

formally prove a compiler transformation to be safe.

Conclusion

Karlsruhe Institute of Technology

Yes, we can. . .

formally prove a compiler transformation to be safe.

- Increased the quality
  Uncovered a bug missed by tests.
- Refactorable
  when the code changes
- Provides high assurance

Conclusion

Yes, we can. . .

formally prove a compiler transformation to be safe.

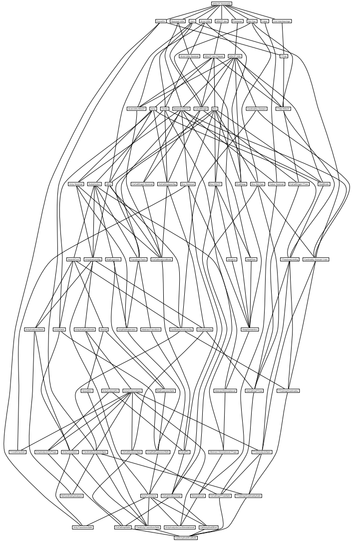
- Increased the quality
  Uncovered a bug missed by tests.
- Refactorable
  when the code changes
- Provides high assurance

- Very tedious
  Still only worth it in certain domains?
- Formalization gap
  Is GHC the wrong target?

Thank you for your attention.

Backup slide: How tedious, really?

- 9 man-months
- 12,000 loc
- 1,200 lemmas
- 79 theories



Backup slide: That bug that was found

Karlsruhe Institute of Technology

Call Arity initially would eta-expand thunks in a recursive group, as long as the recursion is linear.

```
foo a =
   let go | a == "m"
          = λ x. if x == 0
                 then 1
                 else x ∗ go (x-1)
          | a == "p"
          = λ x. if x == 0
                 then 0
                 else x + go (x-1)
   in  go 100
```