Formally Proving a Compiler Transformation Safe

Joachim Breitner
Haskell Symposium 2015
3 August 2015, Vancouver
I formally proved that Call Arity is safe.
I formally proved that

Call Arity is safe.
I formally proved that Call Arity is safe.

“What exactly have you shown?”
I formally proved that Call Arity is safe.

“What exactly have you shown?”

“How did you prove that?”
I formally proved that

Call Arity is safe.

"What exactly have you shown?"

"How did you prove that?"

"Are you sure about this?"
I formally proved that

**Call Arity is safe.**

“**W**hat exactly have you shown?”

“**H**ow did you prove that?”

“**A**re you sure about this?”

“**B**ut, ...!”
What exactly is... Call Arity?

Call Arity is an arity analysis:

\[
\text{let } \text{fac } 10 = \text{id} \\
\quad \text{fac } x = \lambda y. \text{fac } (x+1) (y \times x) \\
\text{in } \text{fac } 0 1
\]  \[\Rightarrow\]  \[
\text{let } \text{fac } 10 \ y = y \\
\quad \text{fac } x \ y = \text{fac } (x+1) (y \times x) \\
\text{in } \text{fac } 0 1
\]
What exactly is... Call Arity?

Call Arity is an arity analysis:

```latex
let fac 10 = id
    fac x = \lambda y. fac (x+1) (y*x)
in fac 0 1
\Rightarrow
let fac 10 y = y
    fac x y = fac (x+1) (y*x)
in fac 0 1
```

So far: Naive forward arity analysis, see Gill’s PhD thesis from 96
What exactly is... the problem?

Eta-expanding a thunk is tricky:

\[
\begin{align*}
\text{let} & \quad \text{thunk} = f \ x \\
\text{in} & \quad \ldots \\
\end{align*}
\]

\[
\begin{align*}
\implies & \quad \text{let} \quad \text{thunk} \ y = f \ x \ y \\
\text{in} & \quad \ldots
\end{align*}
\]
What exactly is... the problem?

Eta-expanding a thunk is tricky:

```
let thunk = f x
in...
```

\[ \Rightarrow \]

```
let thunk y = f x y
in...
```

Sharing can be lost!
What exactly is... the problem?

Eta-expanding a thunk is tricky:

\[
\begin{align*}
\text{let } \text{thunk} &= f \, x \\
\text{in} \ldots
\end{align*}
\quad \implies \quad
\begin{align*}
\text{let } \text{thunk} \, y &= f \, x \, y \\
\text{in} \ldots
\end{align*}
\]

Sharing can be lost!

(unless “thunk” is used at most once in “...”)
What exactly is... co-call cardinality analysis?

\[ G_0(\text{if } p \text{ then } x \text{ else } y) = p \]

\[ G_0(f \ x \ y) = f \]

\[ G_0(f \ x \ y) = f \]
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

↓

Theorem: Call Arity does not increase the number of allocations
What exactly is... could possibly go wrong?

A bug in Call Arity

⇓

Too much eta-expansion

⇓
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

⇓

Too much eta-expansion

⇓

Loss of sharing

⇓

Work is duplicated

⇓

Allocation is increasing
What exactly is... could possibly go wrong?

A bug in Call Arity

⇒

Too much eta-expansion

⇒

Loss of sharing

⇒

Work is duplicated

⇒

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

final value

heap afterwards
How did you prove that?

1st ingredient  Sufficiently detailed semantics:

Sestoft’s mark-1 virtual machine

\[(\Gamma, e, S) \Rightarrow (\Gamma', e', S')\]
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!

$$\mathcal{T}_0(\text{if } p \text{ then } x \text{ else } y) = p \quad \begin{array}{c} \text{if } p \text{ then } x \text{ else } y \end{array}$$

$$\mathcal{T}_0(f \times y) = f \quad \begin{array}{c} f \times y \end{array}$$
How did you prove that?

2nd ingredient Abstract view on what calls what:

Trace trees!

\[ T_0(\text{if } p \text{ then } x \text{ else } y) = \text{p} \]

\[ T_0(f \times y) = \text{f} \]

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

\[
\begin{align*}
\text{Arity analysis} & \quad + \quad \text{impl.} \quad \text{Arity analysis} & \quad + \quad \text{approx.} \quad \text{Arity analysis} & \quad + \quad \text{impl.} \quad \text{Call Arity} \\
+ \quad \text{any cardinality analysis} & \quad \leftrightarrow & \quad + \quad \text{a trace tree analysis} & \quad \leftrightarrow & \quad + \quad \text{a co-call graph analysis} & \quad \leftrightarrow
\end{align*}
\]
Are you sure?

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

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

The formalization gap!
But... The formalization gap!
But... The formalization gap!
But... The formalization gap!
Bug #10176

```haskell
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 {}
```
Yes, we can... formally prove a compiler transformation to be safe.
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
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
Call Arity initially would eta-expand thunks in a recursive group, as long as the recursion is linear.

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