A Glimpse at Linearity in the Haskell Compiler

Rodrigo Mesquita
Bernardo Toninho

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

September 10, 2023

Haskell is a functional language

$$\mathit{add1} :: \mathit{Int} \rightarrow \mathit{Int}$$
$$\mathit{add1}\ x = x + 1$$

Haskell is a functional language

```
add1 :: Int → Int
add1 x = x + 1
```

```
madd1 :: Bool → Int → Int
madd1 condition x =
  if condition
    then add1 x
    else x
```

With Lazy Evaluation

$$
\begin{array}{l}
\mathit{madd1} :: \mathit{Bool} \to \mathit{Int} \to \mathit{Int} \\
\mathit{madd1}\ \mathit{condition}\ x = \\
\quad \mathbf{let}\ y = \mathit{add1}\ x \\
\quad \mathbf{if}\ \mathit{condition} \\
\quad\quad \mathbf{then}\ y \\
\quad\quad \mathbf{else}\ x
\end{array}
$$

With Lazy Evaluation

$$
\begin{array}{l}
\mathit{madd1} :: \mathit{Bool} \to \mathit{Int} \to \mathit{Int} \\
\mathit{madd1}\ \mathit{condition}\ x = \\
\quad \textbf{let}\ y = \mathit{add1}\ x \\
\quad \textbf{if}\ \mathit{condition} \\
\qquad \textbf{then}\ y \\
\qquad \textbf{else}\ x
\end{array}
$$

- We don't compute *add1* *at all* if the condition is false

And Linear Types

A linear function ($\multimap$) consumes its argument *exactly once*

And Linear Types

A linear function ($\multimap$) consumes its argument *exactly once*

$$dup :: a \multimap (a, a)$$
$$dup\ x = (x, x)$$

And Linear Types

A linear function ($\multimap$) consumes its argument *exactly once*

$$dup :: a \multimap (a, a)$$
$$dup\ x = (x, x)$$

$$fst :: (Int, Int) \multimap Int$$
$$fst\ (x, y) = x$$

And Linear Types

$$\mathit{add1} :: \mathit{Int} \multimap \mathit{Int}$$
$$\mathit{add1}\ x = x + 1$$

And Linear Types

$$\mathit{add1} :: \mathit{Int} \multimap \mathit{Int}$$
$$\mathit{add1}\ x = x + 1$$

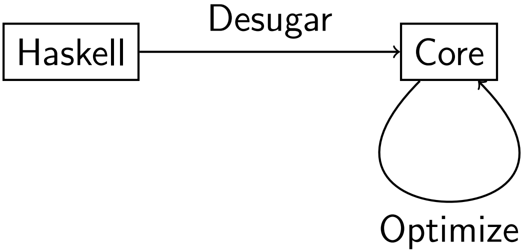
$$\mathit{madd1} :: \mathit{Bool} \to \mathit{Int} \multimap \mathit{Int}$$
$$\mathit{madd1}\ \mathit{condition}\ x =$$
$$\quad \textbf{if}\ \mathit{condition}$$
$$\qquad \textbf{then}\ \mathit{add1}\ x$$
$$\qquad \textbf{else}\ x$$

Which is aggressively optimized

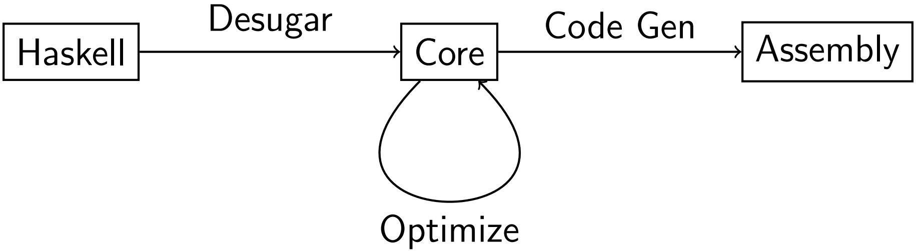
$\boxed{\text{Haskell}}$

Which is aggressively optimized

Which is aggressively optimized



Which is aggressively optimized



Hwvr, optimizations push linearity x laziness too far

- Optimizations move things around

Hwvr, optimizations push linearity x laziness too far

- Optimizations move things around
- In a lazy linear Core

Hwvr, optimizations push linearity x laziness too far

- Optimizations move things around
- In a lazy linear Core
- And programs stop *looking* linear

Hwvr, optimizations push linearity x laziness too far

- Optimizations move things around
- In a lazy linear Core
- And programs stop *looking* linear
- In spite of the laziness

Example program that is not *obviously linear*

$$
\begin{array}{l}
\mathit{madd1} :: \mathit{Bool} \to \mathit{Int} \multimap \mathit{Int} \\
\mathit{madd1}\ \mathit{condition}\ x = \\
\quad \mathbf{let}\ y = \mathit{add1}\ x \\
\quad \mathbf{if}\ \mathit{condition} \\
\qquad \mathbf{then}\ y \\
\qquad \mathbf{else}\ x
\end{array}
$$

Motivation: The short story

- Core's current linearity is violated after optimizations

Motivation: The short story

- Core's current linearity is violated after optimizations
- But the compiler doesn't duplicate/forget linear resources

Motivation: The short story

- Core's current linearity is violated after optimizations
- But the compiler doesn't duplicate/forget linear resources
- Core's *type system* does not understand linearity x laziness

Motivation: The short story

- Core's current linearity is violated after optimizations
- But the compiler doesn't duplicate/forget linear resources
- Core's *type system* does not understand linearity x laziness
- So it cannot use linearity for optimizations

Motivation: The short story

- Core's current linearity is violated after optimizations
- But the compiler doesn't duplicate/forget linear resources
- Core's *type system* does not understand linearity x laziness
- So it cannot use linearity for optimizations
- Neither validate linearity internally

Our contributions

- We developed a *type system* that understands linearity x laziness

Our contributions

- We developed a *type system* that understands linearity x laziness
- Wrote proofs about its safety

Our contributions

- We developed a *type system* that understands linearity x laziness
- Wrote proofs about its safety
- And implemented this system as a GHC plugin

Fim

System FC

- *System* F_C is a polymorphic lambda calculus with explicit type-equality coercions

System FC

- *System F_C* is a polymorphic lambda calculus with explicit type-equality coercions
- A coercion $\sigma_1 \sim \sigma_2$ can be used to safely *cast* an expression e of type σ_1 to type σ_2, written $e \blacktriangleright \sigma_1 \sim \sigma_2$.

System FC

Definition (Syntax)

$$
\begin{array}{rcll}
u & ::= & x \mid K & \text{Variables and data constructors} \\
e & ::= & u & \text{Term atoms} \\
 & \mid & \Lambda a{:}\kappa.\ e \ \mid\ e\ \varphi & \text{Type abstraction/application} \\
 & \mid & \lambda x{:}\sigma.\ e \ \mid\ e_1\ e_2 & \text{Term abstraction/application} \\
 & \mid & \mathsf{let}\ x{:}\sigma = e_1\ \mathsf{in}\ e_2 & \\
 & \mid & \mathsf{case}\ e_1\ \mathsf{of}\ \overline{p \rightarrow e_2} & \\
 & \mid & e \blacktriangleright \gamma & \text{Cast} \\
 & & & \\
p & ::= & K\ \overline{b{:}\kappa}\ \overline{x{:}\sigma} & \text{Pattern}
\end{array}
$$

Sample: Δ-bound variables

$$\frac{}{\Gamma, x{:_\Delta}\sigma; \Delta \vdash x : \sigma}\;(\mathit{Var}_\Delta)$$

Sample: Δ-bound variables

$$\frac{}{\Gamma, \mathit{x}{:}_{\Delta}\sigma; \Delta \vdash \mathit{x} : \sigma}\ (\mathit{Var}_{\Delta})$$

$$\textsc{(Let)}\ \frac{\Gamma; \Delta \vdash e : \sigma \qquad \Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e' : \varphi}{\Gamma; \Delta, \Delta' \vdash \textbf{let}\ x{:}_{\Delta}\sigma = e\ \textbf{in}\ e' : \varphi}$$