A Glimpse at Linearity in the Haskell Compiler

Rodrigo Mesquita
Bernardo Toninho

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

September 11, 2023

Haskell is a functional language

$$
\begin{aligned}
&\mathit{add1} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{add1}\ x = x + 1
\end{aligned}
$$

Haskell is a functional language

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

$$\mathit{madd1} :: \mathit{Bool} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$$

$$
\begin{array}{l}
\mathit{madd1}\ \mathit{condition}\ x = \\
\quad \textbf{if}\ \mathit{condition} \\
\qquad \textbf{then}\ \mathit{add1}\ x \\
\qquad \textbf{else}\ x
\end{array}
$$

With Lazy Evaluation

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

With Lazy Evaluation

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

- 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*

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

And Linear Types

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

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

$$\mathit{dup} :: \mathit{Int} \multimap (\mathit{Int}, \mathit{Int})$$
$$\mathit{dup}\ x = (x, x)$$

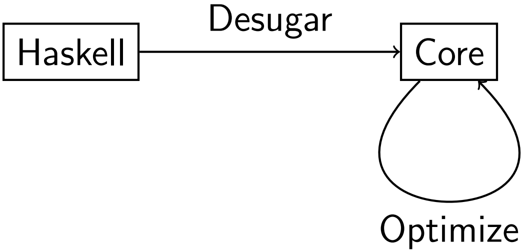
Which is aggressively optimized

$\boxed{\text{Haskell}}$

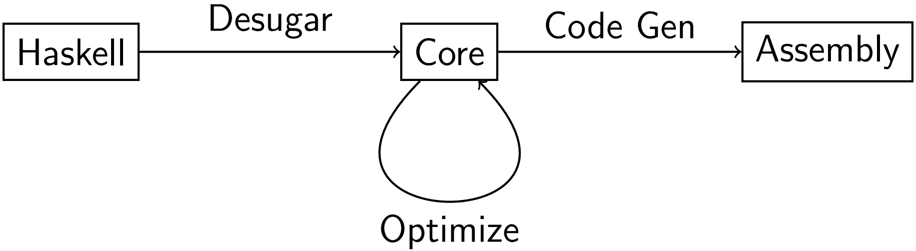
Which is aggressively optimized

$$\boxed{\text{Haskell}} \xrightarrow{\text{Desugar}} \boxed{\text{Core}}$$

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
- And programs stop *looking* linear

Example program that is not *obviously linear*

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

Motivation: The short story

- Core's *type system* does not understand linearity x laziness

Motivation: The short story

- Core's *type system* does not understand linearity x laziness
- So it cannot use linearity for optimizations

Motivation: The short story

- 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
- Validating that optimisations preserve linearity

Our contributions

- We developed a *type system* that understands linearity x laziness
- Validating that optimisations preserve linearity
- And implemented it 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)

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

Sample: Δ-bound variables

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

Sample: Δ-bound variables

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

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