Type-checking Linearity in Core: Semantic Linearity for a Lazy Optimising Compiler

Rodrigo Mesquita
Advisor: Bernardo Toninho

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Linear Haskell

Haskell has Linear Types!

Linear Haskell

Haskell has Linear Types!

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

Linear Haskell

Haskell has Linear Types!

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

$$
\begin{aligned}
&\mathit{bad} :: \mathit{Ptr} \multimap \mathit{IO}\ () \\
&\mathit{bad}\ x = \mathbf{do} \\
&\quad \mathit{free}\ x \\
&\quad \mathit{free}\ x
\end{aligned}
$$

Linear Haskell

Haskell has Linear Types!
A linear function $\multimap$ consumes its argument *exactly once*

$$
\begin{aligned}
&\mathit{bad} :: \mathit{Ptr} \multimap \mathit{IO}\ ()\\
&\mathit{bad}\ x = \mathbf{do}\\
&\quad \mathit{free}\ x\\
&\quad \mathit{free}\ x
\end{aligned}
$$

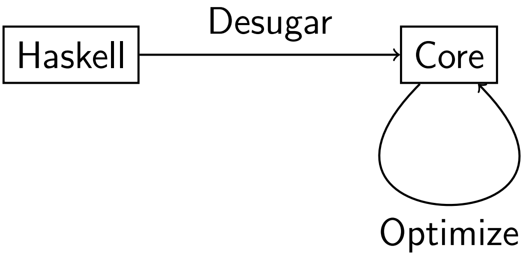
$$
\begin{aligned}
&\mathit{ok} :: \mathit{Ptr} \multimap \mathit{IO}\ ()\\
&\mathit{ok}\ x = \mathit{free}\ x
\end{aligned}
$$

Linearity in the Glasgow Haskell Compiler (GHC)

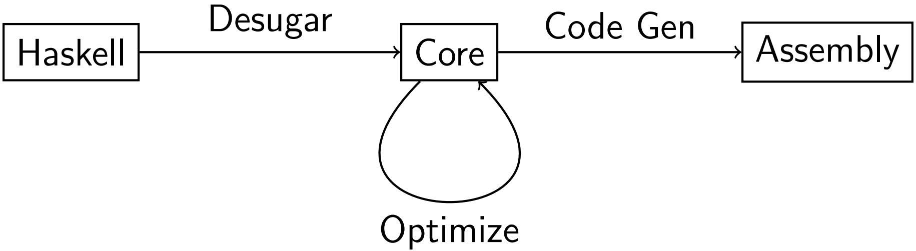
$\boxed{\text{Haskell}}$

Linearity in the Glasgow Haskell Compiler (GHC)

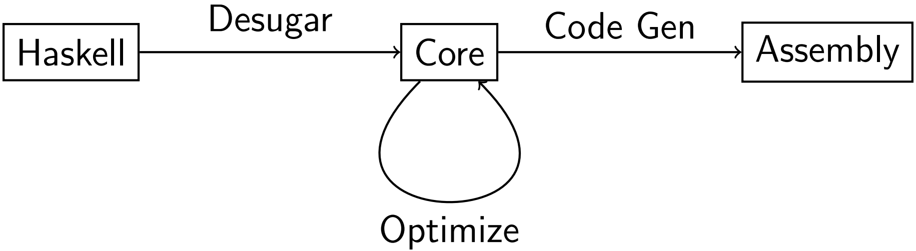
Linearity in the Glasgow Haskell Compiler (GHC)



Linearity in the Glasgow Haskell Compiler (GHC)

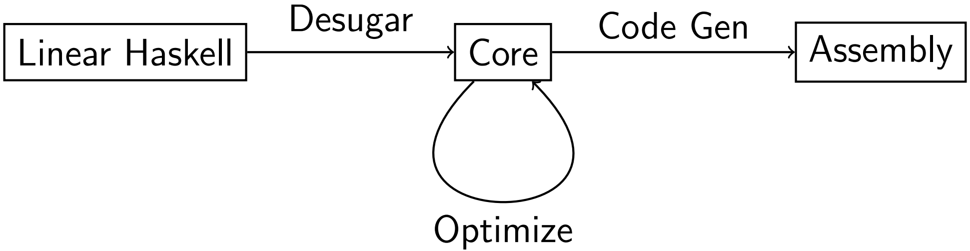


Linearity in the Glasgow Haskell Compiler (GHC)



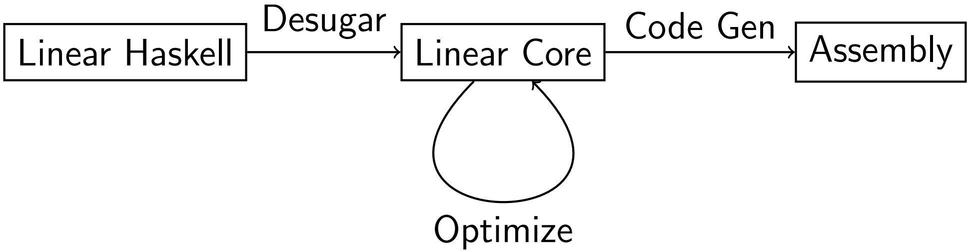
Core is both lazy and typed

Linearity in the Glasgow Haskell Compiler (GHC)



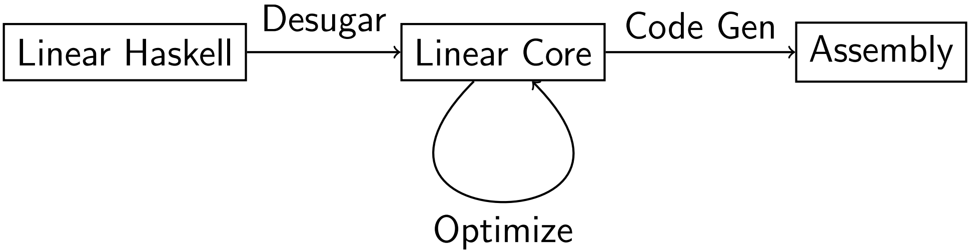
Core is both lazy and typed

Linearity in the Glasgow Haskell Compiler (GHC)



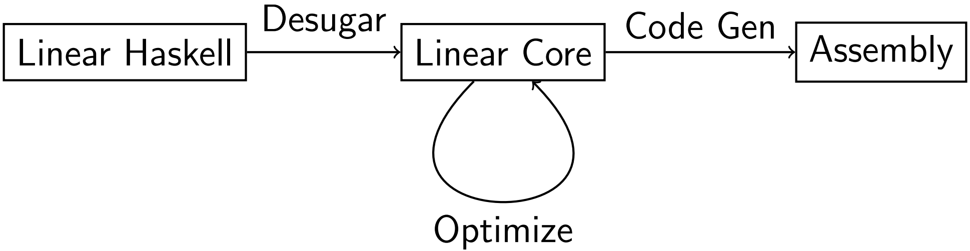
Core is both lazy and typed

Linearity in the Glasgow Haskell Compiler (GHC)



Linear Core is both lazy and *linearly* typed

Linearity in the Glasgow Haskell Compiler (GHC)



Linear Core ~~is~~ *should be* both lazy and *linearly* typed

So, why isn't Core linear?

Optimised programs stop *looking* linear, but are linear *semantically*

So, why isn't Core linear?

Optimised programs stop *looking* linear, but are linear *semantically*

$$\begin{array}{l} \textbf{let } y = \mathit{free}\ x \textbf{ in } y \\ \Longrightarrow \\ \textbf{let } y = \mathit{free}\ x \textbf{ in } \mathit{free}\ x \end{array}$$

So, why isn't Core linear?

Optimised programs stop *looking* linear, but are linear *semantically*

$$\textbf{let}\ y = \textit{free}\ x\ \textbf{in}\ y$$
$$\Longrightarrow$$
$$\textbf{let}\ y = \textit{free}\ x\ \textbf{in}\ \textit{free}\ x$$

Linearity is ignored in Core, or most programs would be rejected

Semantic vs Syntactic Linearity

- Programs are still linear *semantically* because of laziness

Semantic vs Syntactic Linearity

- Programs are still linear *semantically* because of laziness
- **Key insight:** Under lazy evaluation,

$$\textit{syntactic}\ \text{occurrence} \not\Rightarrow \textit{consuming}\ \text{a resource}$$
$$\textit{syntactic}\ \text{linearity} \neq \textit{semantic}\ \text{linearity}$$

Semantic vs Syntactic Linearity

- Programs are still linear *semantically* because of laziness
- **Key insight:** Under lazy evaluation,

$$\textit{syntactic}\text{ occurrence} \not\Rightarrow \textit{consuming}\text{ a resource}$$
$$\textit{syntactic}\text{ linearity} \neq \textit{semantic}\text{ linearity}$$

- We type *syntactic* linearity in Core, but that is not enough

Semantic vs Syntactic Linearity

- Programs are still linear *semantically* because of laziness
- **Key insight:** Under lazy evaluation,

$$\textit{syntactic}\ \text{occurrence} \not\Rightarrow \textit{consuming}\ \text{a resource}$$
$$\textit{syntactic}\ \text{linearity} \neq \textit{semantic}\ \text{linearity}$$

- We type *syntactic* linearity in Core, but that is not enough
- Optimisations push laziness x linearity to the limit

Our Contributions

- Linear Core: a type system that understands semantic linearity in the presence of laziness

Our Contributions

- Linear Core: a type system that understands semantic linearity in the presence of laziness
- We proved Linear Core and multiple optimising transformations to be sound

Our Contributions

- Linear Core: a type system that understands semantic linearity in the presence of laziness
- We proved Linear Core and multiple optimising transformations to be sound
- We implemented Linear Core as a GHC plugin

Semantic Linearity, by example

Semantic Linearity: Lets

let $y = \mathit{free\ ptr}$
in if *condition*
  then y
  else *return ptr*

Semantic Linearity: Lets

let $y = \mathit{free\ ptr}$
in if *condition*
  then y
  else *return ptr*

Resources in lets are only consumed if the binder is evaluated

Semantic Linearity: Case

$$\begin{array}{l} \textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \mathit{something}\ a\ b \end{array}$$

Semantic Linearity: Case

$$\begin{array}{l}\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \mathit{something}\ a\ b\end{array}$$

$$\begin{array}{l}\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \mathit{something}\ x\ y\end{array}$$

Semantic Linearity: Case

$$\begin{array}{l}\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \textit{something } a\ b\end{array}$$

$$\begin{array}{l}\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \textit{something } x\ y\end{array}$$

$$\begin{array}{l}\textbf{case } \textit{free } x \textbf{ of} \\ \quad \textit{Result } v \rightarrow \textit{free } x\end{array}$$

Semantic Linearity: Case

$$\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \textit{something } a\ b$$

$$\textbf{case } (x, y) \textbf{ of} \\ \quad (a, b) \rightarrow \textit{something } x\ y$$

$$\textbf{case } \textit{free } x \textbf{ of} \\ \quad \textit{Result } v \rightarrow \textit{free } x$$

$$\textbf{case } \textit{use } x \textbf{ of} \\ \quad \textit{Result } v \rightarrow ()$$

Semantic Linearity: Case

$$\textbf{case}\ (x, y)\ \textbf{of}\\ \quad (a, b) \to \mathit{something}\ a\ b$$

$$\textbf{case}\ (x, y)\ \textbf{of}\\ \quad (a, b) \to \mathit{something}\ x\ y$$

$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to \mathit{free}\ x$$

$$\textbf{case}\ \mathit{use}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to ()$$

Resources are *kind of* consumed if the expression is evaluated

Linear Core

Linear Core: *Let*-vars

$$\textbf{let } y_{\{ptr\}} = \mathit{free}\ \mathit{ptr} \textbf{ in } y_{\{ptr\}}$$

Linear Core: *Let*-vars

$$\textbf{let}\ y_{\{ptr\}} = \mathit{free\ ptr}\ \textbf{in}\ y_{\{ptr\}}$$

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

Linear Core: *Let*-vars

$$\textbf{let } y_{\{ptr\}} = \textit{free ptr } \textbf{in } y_{\{ptr\}}$$

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

Let-binder bodies don't consume resources

Linear Core: *Let*-vars

$$\textbf{let } y_{\{ptr\}} = \textit{free ptr } \textbf{in } y_{\{ptr\}}$$

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

Let-binder bodies don't consume resources

- Annotate Let-vars with linear resources Δ used in its body

Linear Core: *Let*-vars

$$\textbf{let } y_{\{ptr\}} = \mathit{free\ ptr} \textbf{ in } y_{\{ptr\}}$$

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

Let-binder bodies don't consume resources

- Annotate Let-vars with linear resources Δ used in its body
- Using a Let-var entails using all of its Δ

Linear Core: Lets

let $y_{\{ptr\}} = free\ ptr$
in if *condition*
  then $y_{\{ptr\}}$
  else *return ptr*

Linear Core: Lets

$\textbf{let } y_{\{ptr\}} = \textit{free ptr}$
$\textbf{in if } \textit{condition}$
$\quad \textbf{then } y_{\{ptr\}}$
$\quad \textbf{else } \textit{return ptr}$

(Let)

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

Linear Core: Lets

$$
\begin{array}{l}
\textbf{let } y_{\{ptr\}} = \mathit{free\ ptr} \\
\textbf{in if } \mathit{condition} \\
\quad \textbf{then } y_{\{ptr\}} \\
\quad \textbf{else } \mathit{return\ ptr}
\end{array}
$$

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

Resources used in the binder are still available in the body:

- Can consume them using the let-var
- Or directly, if the let-var is unused

Linear Core: Case

Case scrut evaluate to WHNF, unless they are already in WHNF

Linear Core: Case

Case scrut evaluate to WHNF, unless they are already in WHNF

$$
\begin{array}{l}
\textbf{case } (x, y) \textbf{ of} \\
\quad (a, b) \rightarrow \mathit{something}\ x\ y
\end{array}
$$

Linear Core: Case

Case scrut evaluate to WHNF, unless they are already in WHNF

$$\textbf{case}\ (x, y)\ \textbf{of}\\ \quad (a, b) \to \mathit{something}\ x\ y$$

$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to \mathit{free}\ x$$

Linear Core: Case

Case scrut evaluate to WHNF, unless they are already in WHNF

$$\textbf{case}\ (x, y)\ \textbf{of}\\ \quad (a, b) \to \textit{something}\ x\ y$$

$$\textbf{case}\ \textit{free}\ x\ \textbf{of}\\ \quad \textit{Result}\ v \to \textit{free}\ x$$

Key idea: We need to branch on *WHNF-ness*

Linear Core: Case WHNF

$$
\begin{array}{l}
\textbf{case } (x, y) \textbf{ of} \\
\quad (a_{\{x\}}, b_{\{y\}}) \rightarrow \mathit{use}\ x\ y
\end{array}
$$

Linear Core: Case WHNF

$$\begin{array}{l}\textbf{case}\ (x, y)\ \textbf{of}\\ \quad (a_{\{x\}}, b_{\{y\}}) \to \mathit{use}\ x\ y\end{array}$$

$$\frac{}{\cdot; x, y \vdash \textbf{case}\ (x, y)\ \textbf{of}\ (a, b) \to \ldots}$$

Linear Core: Case WHNF

$$
\begin{array}{l}
\textbf{case}\ (x, y)\ \textbf{of} \\
\quad (a_{\{x\}}, b_{\{y\}}) \to \mathit{use}\ x\ y
\end{array}
$$

$$
\frac{\cdot; x, y \vdash (x, y)}{\cdot; x, y \vdash \textbf{case}\ (x, y)\ \textbf{of}\ (a, b) \to \dots}
$$

Linear Core: Case WHNF

$$\begin{aligned}&\textbf{case}\ (x, y)\ \textbf{of}\\ &\quad (a_{\{x\}}, b_{\{y\}}) \rightarrow \mathit{use}\ x\ y\end{aligned}$$

$$\frac{\cdot; x, y \vdash (x, y)}{\cdot; x, y \vdash \textbf{case}\ (x, y)\ \textbf{of}\ (a, b) \rightarrow \ldots}$$

Scrut resources are available in the body, pattern vars are Δ-vars

Linear Core: Case WHNF

$$\textbf{case } (x, y) \textbf{ of} \\ \quad (a_{\{x\}}, b_{\{y\}}) \to \mathit{use}\ x\ y$$

$$\frac{\cdot; x, y \vdash (x, y) \qquad a{:}_{\{x\}}, b{:}_{\{y\}}; x, y \vdash \mathit{use}\ x\ y}{\cdot; x, y \vdash \textbf{case } (x, y) \textbf{ of } (a, b) \to \dots}$$

Scrut resources are available in the body, pattern vars are Δ-vars

Linear Core: Case Not-WHNF

case *free* x **of**
  Result $v \rightarrow$ *free* x

Linear Core: Case Not-WHNF

$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}$$
$$\quad \mathit{Result}\ v \rightarrow \mathit{free}\ x$$

$$\frac{}{\cdot; x \vdash \textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \ \ldots}$$

Linear Core: Case Not-WHNF

$$\begin{array}{l}\textbf{case}\ \mathit{free}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to \mathit{free}\ x\end{array}$$

$$\frac{\cdot; x \vdash \mathit{free}\ x}{\cdot; x \vdash \textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \ldots}$$

Linear Core: Case Not-WHNF

$$\begin{array}{l}\textbf{case}\ \mathit{free}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to \mathit{free}\ x\end{array}$$

$$\frac{\cdot; x \vdash \mathit{free}\ x}{\cdot; x \vdash \textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \ldots}$$

Scrut resources are *irrelevant* in the body

- They cannot be instantiated with *Var*
- But must still be used exactly once

Linear Core: Case Not-WHNF

$$\begin{array}{l}\textbf{case}\ \mathit{free}\ x\ \textbf{of}\\ \quad \mathit{Result}\ v \to \mathit{free}\ x\end{array}$$

$$\frac{\cdot; x \vdash \mathit{free}\ x \qquad v{:}_{\{[x]\}}; [x] \vdash \mathit{free}\ x}{\cdot; x \vdash \textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \dots}$$

Scrut resources are *irrelevant* in the body

- They cannot be instantiated with *Var*
- But must still be used exactly once

Metatheory: Linear Core

- Not obvious whether these rules make sense together
- We proved the system is type safe via preservation + progress

Metatheory: Linear Core

- Not obvious whether these rules make sense together
- We proved the system is type safe via preservation + progress
  - *Irrelevance* lemma
  - Linear-var substitution lemma
    - + substitution on case alternatives
  - Δ-var substitution lemma
    - + substitution on case alternatives
  - Unr-var substitution lemma
    - + substitution on case alternatives

Metatheory: Optimising Transformations

- Inlining
- β-reduction
- β-reduction with sharing
- β-reduction for multiplicity abstractions
- Case-of-known-constructor
- Full laziness
- Local transformations (three of them)
- η-expansion
- η-reduction
- Binder swap
- Reverse binder swap (contentious!)
- Case-of-case

GHC Plugin: Linear Core Implementation

We implemented Linear Core as a GHC plugin

Library	Total Accepted	Total Rejected	Unique Rejected	Linear modulo Call-by-name	Linear Rejected	¬ Linear Rejected	Unknown Rejected
linear-smc	19438	4	1	1	0	0	0
priority-sesh	6781	19	1	0	0	0	1
linear-base	112311	538	87	10	8	2	67

Figure: Linear Core Plugin on Linear Libraries

Conclusion

- Linear Core is a suitable type system for Core, as it understands the interaction between linearity and laziness that the optimiser pushes to the limit

Conclusion

- Linear Core is a suitable type system for Core, as it understands the interaction between linearity and laziness that the optimiser pushes to the limit
- Not every single program is accepted by Linear Core
  - Future work: *multiplicity coercions*
  - Discuss linearity modulo call-by-name
  - Iron out quirks (rewrite rules, ...)

Conclusion

- Linear Core is a suitable type system for Core, as it understands the interaction between linearity and laziness that the optimiser pushes to the limit
- Not every single program is accepted by Linear Core
  - Future work: *multiplicity coercions*
  - Discuss linearity modulo call-by-name
  - Iron out quirks (rewrite rules, ...)
- Builds on the shoulders of Linear Haskell and Linear Mini-Core

Conclusion

- Linear Core is a suitable type system for Core, as it understands the interaction between linearity and laziness that the optimiser pushes to the limit
- Not every single program is accepted by Linear Core
  - Future work: *multiplicity coercions*
  - Discuss linearity modulo call-by-name
  - Iron out quirks (rewrite rules, ...)
- Builds on the shoulders of Linear Haskell and Linear Mini-Core
- There's much more in the thesis!

Fim

Semantic Linearity: Case of Var

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)$$

Semantic Linearity: Case of Var

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)$$

$$\Longrightarrow_{\text{call by name}}$$

Semantic Linearity: Case of Var

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)$$
$$\Longrightarrow_{\text{call by name}}$$
$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \_ \to \mathit{free}\ x$$

Semantic Linearity: Case of Var

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)$$

$$\Longrightarrow_{\text{call by name}}$$

$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \_ \to \mathit{free}\ x$$

$$\Longrightarrow_{\text{call by need}}$$

Semantic Linearity: Case of Var

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)$$

$$\Longrightarrow_{\text{call by name}}$$

$$\textbf{case}\ \mathit{free}\ x\ \textbf{of}\ \_ \to \mathit{free}\ x$$

$$\Longrightarrow_{\text{call by need}}$$

$$\textbf{let}\ y = \mathit{free}\ x\ \textbf{in}\ \textbf{case}\ y\ \textbf{of}\ \_ \to y$$

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)

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