Theory and Practice

Of Programming Languages
Theory and Practice

• **Goal**: Performance
Theory and Practice

• **Goal**: Performance

• **Subgoal**: Semantics
Theory and Practice

• **Goal**: Performance
• **Subgoal**: Semantics
• **Answer**: Logic
Compilation Funnel

Source → Intermediate → Target

Haskell
Compilation Funnel

Source $\rightarrow$ Intermediate $\rightarrow$ Target

Desugaring $\downarrow$

Haskell
Compilation Funnel

Source ➔ Intermediate ➔ Target

Desugaring ➔ Haskell ➔ Core
Compilation Funnel

Source $\rightarrow$ Intermediate $\rightarrow$ Target

Desugaring

Haskell

Core

Code Generation
Compilation Funnel

Desugaring → Haskell → Core → STG → Code Generation

Source → Intermediate → Target
Compilation Funnel

Source → Intermediate → Target

Desugaring

Code Generation

Haskell

Core

STG

x86

Machine Primitives
Compilation Funnel

Source $\rightarrow$ Intermediate $\rightarrow$ Target

Desugaring

Code Generation

Haskell

Core

STG

x86

Efficient Libraries

Machine Primitives
System F

Workhorse of Functional Compilers
System F

Workhorse of Functional Compilers

Core
System F

Workhorse of Functional Compilers

Core = System F
Core = System F (first-class functions, polymorphism)
System F

Core = System F (first-class functions, polymorphism)
+ Data Types

Workhorse of Functional Compilers
System F

Workhorse of Functional Compilers

Core = System F  (first-class functions, polymorphism)
+ Data Types   (Primitives, lists/trees, records)
System F

Workhorse of Functional Compilers

Core = System F (first-class functions, polymorphism) + Data Types (Primitives, lists/trees, records) + Type Equality
System F

Workhorse of Functional Compilers

Core = System F
+ Data Types
+ Type Equality

(first-class functions, polymorphism)
(Primitives, lists/trees, records)
(GADTs, type families, coercions)
System F

Workhorse of Functional Compilers

Core = System F  
+ Data Types  
+ Type Equality  
+ ...

(first-class functions, polymorphism)  
(Primitives, lists/trees, records)  
(GADTs, type families, coercions)
*GHC Core*

*In Greek*
GHC Core*  

Expr ∈ d, e, f ::= x | λx:τ. e | f e

λ-calculus: variables, functions, application

*In Greek
GHC Core*

Type $\exists \tau, \sigma ::= \ldots$

$Expr \ni d, e, f ::= x \mid \lambda x: \tau. e \mid f e$

$\lambda$-calculus: variables, functions, application
GHC Core*

Type \( \exists \tau, \sigma ::= \ldots \)

Expr \( \exists d, e, f ::= x \mid \lambda x: \tau. e \mid f e \mid \Lambda a: \kappa. e \mid e \tau \)

\( \lambda \)-calculus: variables, functions, application

System F: polymorphism & instantiation

*In Greek
GHC Core*

Type $\exists \tau, \sigma ::= \ldots$

Kind $\exists \kappa = \text{Type}$

Expr $\exists d, e, f ::= x | \lambda x : \tau . e | f e$

$| \Lambda a : \kappa . e | e \tau$

$\lambda$-calculus: variables, functions, application

System F: polymorphism & instantiation

*In Greek
GHC Core* 

Type $\exists \tau, \sigma ::= \ldots$

Kind $\exists \kappa = Type$

Expr $\exists d, e, f ::= x \mid \lambda x: \tau \cdot e \mid f e$

$\mid \Lambda a: \kappa \cdot e \mid e \tau$

$\mid l \mid let \ x: \tau = d \ in \ e$

$\lambda$-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

*In Greek
GHC Core*

Type \in \tau, \sigma ::= \ldots

Kind \in \kappa = Type

Expr \in d, e, f ::= x | \lambda x:\tau. e | f e
| \Lambda a:\kappa. e | e \tau
| l | let x:\tau = d in e
| case d of \{ \pi \to e; \ldots \}

\lambda\text{-calculus: variables, functions, application}

System F: polymorphism & instantiation

Literal primitives & let-bindings

Data constructor & literal matching
GHC Core*

\[ Type \ni \tau, \sigma ::= \ldots \quad Pattern \ni \pi ::= x \mid l \mid K \pi \ldots \]

\[ Kind \ni \kappa = Type \]

\[ Expr \ni d, e, f ::= x \mid \lambda \times \tau . e \mid f e \]

\[ \mid \Lambda a: \kappa . e \mid e \tau \]

\[ \mid l \mid \text{let } x: \tau = d \text{ in } e \]

\[ \mid \text{case } d \text{ of } \{ \pi \rightarrow e; \ldots \} \]

\[ \lambda\text{-calculus: variables, functions, application} \]
\[ \text{System F: polymorphism \& instantiation} \]
\[ \text{Literal primitives \& let-bindings} \]
\[ \text{Data constructor \& literal matching} \]

*In Greek
GHC Core*  

Type $\exists \tau, \sigma ::= \ldots$  \quad Pattern $\exists \pi ::= x | l | K x\ldots$

Kind $\exists \kappa = Type$

Expr $\exists d, e, f ::= x | \lambda x:\tau. e | f e$

$| \Lambda a:\kappa. e | e \tau$

$| l | let x:\tau = d in e$

$| \text{case } d \text{ of } \{ \pi \to e; \ldots \}$

$| \chi | e \triangleright \chi$

\[ \text{\(\lambda\)-calculus: variables, functions, application} \]

\[ \text{System F: polymorphism & instantiation} \]

\[ \text{Literal primitives & let-bindings} \]

\[ \text{Data constructor & literal matching} \]

\[ \text{Coercion evidence & casting} \]

*In Greek
GHC Core*

Type $\ni \tau, \sigma ::= \ldots$

Pattern $\ni \pi ::= x \mid l \mid K x \ldots$

Kind $\ni \kappa = \text{Type}$

Coercion $\ni \chi ::= \text{refl} \mid \chi^{-1} \mid \chi \circ \chi' \mid \ldots$

$\lambda$-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

Data constructor & literal matching

Coercion evidence & casting

*In Greek
\( \text{GHC Core*} \)

\[
\begin{align*}
\text{Type} \ & \ni \tau, \sigma ::= \ldots \\
\text{Kind} \ & \ni \kappa \quad = \text{Type} \\
\text{Pattern} \ & \ni \pi ::= x \mid l \mid K \mid x \ldots \\
\text{Expr} \ & \ni d, e, f ::= x \mid \lambda x: \tau. e \mid f e \\
& \quad \mid \Lambda a: \kappa. e \mid e \tau \\
& \quad \mid l \mid \text{let } x: \tau = d \text{ in } e \\
& \quad \mid \text{case } d \text{ of } \{ \pi \rightarrow e ; \ldots \} \\
& \quad \mid \chi \mid e \triangleright \chi \\
& \quad \mid \text{tick } tk \ e \\
\end{align*}
\]

\( \lambda \)-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

Data constructor & literal matching

Coercion evidence & casting

Profiling & instrumentation

*In Greek
GHC Core*

Type ⊇ τ, σ ::= ... Pattern ⊇ π ::= x | l | K x...
Kind ⊇ κ = Type Coercion ⊇ χ ::= refl | χ⁻¹ | χ ⪯ χ' | ...

Expr ⊇ d, e, f ::= x | λx:τ.e | f e
   | Λa:κ.e | e τ
   | l | let x:τ = d in e
   | case d of {π → e; ...}
   | χ | e ⪯ χ
   | tick tk e

λ-calculus: variables, functions, application
System F: polymorphism & instantiation
Literal primitives & let-bindings
Data constructor & literal matching
Coercion evidence & casting
Profiling & instrumentation

A real-world programming language in only 6 lines!
Compiling Polymorphism

Statically
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a
dup f x = f x x

Statically
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:

1. Accept parameters
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:

1. Accept parameters
   
   • f : a -> a -> a is a pointer; read from pointer register 1
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:

1. Accept parameters
   • f : a -> a -> a is a pointer; read from pointer register 1
   • Where is x : a?
Compiling Polymorphism

\[
\text{dup : } \forall a. (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow a
\]
\[
\text{dup } f \ x = f \ x \ x
\]

Compiled assembly code:

1. Accept parameters
   - \( f : a \rightarrow a \rightarrow a \) is a pointer; read from pointer register 1
   - Where is \( x : a \)?
   - **Assume \( x \) is a pointer; read from pointer register 2**
Compiling Polymorphism

\[
dup : \forall a. (a \to a \to a) \to a \to a
\]
\[
dup f x = f x x
\]

Compiled assembly code:

1. Accept parameters
   - \( f : a \to a \to a \) is a pointer; read from pointer register 1
   - Where is \( x : a \)?
   - \textbf{Assume } x \textbf{ is a pointer}; read from pointer register 2

2. Pass arguments
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a
dup f x = f x x

Compiled assembly code:

1. Accept parameters
   • f : a -> a -> a is a pointer; read from pointer register 1
   • Where is x : a?
   • Assume x is a pointer; read from pointer register 2

2. Pass arguments
   • Save f
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a
dup f x = f x x

Compiled assembly code:

1. Accept parameters
   • f : a -> a -> a is a pointer; read from pointer register 1
   • Where is x : a?
   • Assume x is a pointer; read from pointer register 2

2. Pass arguments
   • Save f
   • Copy x (pointer register 2) to the first argument (pointer register 1)
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:

1. Accept parameters
   • f : a -> a -> a is a pointer; read from pointer register 1
   • Where is x : a?
   • Assume x is a pointer; read from pointer register 2

2. Pass arguments
   • Save f
   • Copy x (pointer register 2) to the first argument (pointer register 1)

3. Call f
Compiling Polymorphism

\[ \text{dup} : \forall a. (a \to a \to a) \to a \to a \]
\[ \text{dup } f \ x = f \ x \ x \]

Compiled assembly code:

1. Accept parameters
   • \( f : a \to a \to a \) is a pointer; read from pointer register 1
   • Where is \( x : a \)?
   • Assume \( x \) is a pointer; read from pointer register 2

2. Pass arguments
   • Save \( f \)
   • Copy \( x \) (pointer register 2) to the first argument (pointer register 1)

3. Call \( f \)
   • How many arguments does \( f : a \to a \to a \) take? Is \( f \ x \ x : a \) a call? a closure?
Compiling Polymorphism

dup : forall a. (a -> a -> a) -> a -> a

dup f x = f x x

Compiled assembly code:

1. Accept parameters
   • f : a -> a -> a is a pointer; read from pointer register 1
   • Where is x : a?
   • Assume x is a pointer; read from pointer register 2

2. Pass arguments
   • Save f
   • Copy x (pointer register 2) to the first argument (pointer register 1)

3. Call f
   • How many arguments does f : a -> a -> a take? Is f x x : a a call? a closure?
   • Check the arity of f; read runtime closure info, and take appropriate action
Calling Conventions

In Systems Programming Languages
Calling Conventions

• Calls have statically known parameter #s
Calling Conventions

• Calls have statically known parameter #s
  • Just store arguments, push return pointer, and jump
Calling Conventions

- Calls have statically known parameter #s
  - Just store arguments, push return pointer, and jump
- Call-by-value versus call-by-reference
Calling Conventions

- Calls have statically known parameter #s
  - Just store arguments, push return pointer, and jump

- Call-by-value versus call-by-reference
  - Values may be passed directly, not just pointers
Calling Conventions

- Calls have statically known parameter #s
  - Just store arguments, push return pointer, and jump
- Call-by-value versus call-by-reference
  - Values may be passed directly, not just pointers
- Many shapes of values

In Systems Programming Languages
Calling Conventions

Call-by-value versus call-by-reference
• Values may be passed directly, not just pointers

Many shapes of values
• Different sizes of integers and words
Calling Conventions

In Systems Programming Languages

• Calls have statically known parameter #s
  • Just store arguments, push return pointer, and jump

• Call-by-value versus call-by-reference
  • Values may be passed directly, not just pointers

• Many shapes of values
  • Different sizes of integers and words
    • Built-in floating-point numbers & registers
Calling Conventions

In Systems Programming Languages

• Calls have statically known parameter #s
  • Just store arguments, push return pointer, and jump

• Call-by-value versus call-by-reference
  • Values may be passed directly, not just pointers

• Many shapes of values
  • Different sizes of integers and words
  • Built-in floating-point numbers & registers
  • Contiguous arrays and compound structures
Calling Conventions

In Systems Programming Languages

• Calls have statically known parameter #s
  • Just store arguments, push return pointer, and jump

• Call-by-value versus call-by-reference
  • Values may be passed directly, not just pointers

• Many shapes of values
  • Different sizes of integers and words
  • Built-in floating-point numbers & registers
  • Contiguous arrays and compound structures

• Checks for calling conventions \textit{statically} at compile time
Efficient Function Calls

Parameter Passing Techniques
Efficient Function Calls

• Representation — What & Where?
Efficient Function Calls

• Representation — What & Where?
  • Shape of data values
Efficient Function Calls

• Representation — What & Where?
  • Shape of data values

• Arity — How many arguments?
Efficient Function Calls

- Representation — What & Where?
  - Shape of data values
- Arity — How many arguments?
  - Shape of calling context
Efficient Function Calls

• Representation — What & Where?
  • Shape of data values

• Arity — How many arguments?
  • Shape of calling context

• Levity — When to compute?
Efficient Function Calls

• Representation — What & Where?
  • Shape of data values

• Arity — How many arguments?
  • Shape of calling context

• Levity — When to compute?
  • Aka Evaluation Strategy
Efficient Function Calls

- Representation — What & Where?
  - Shape of data values
- Arity — How many arguments?
  - Shape of calling context
- Levity — When to compute?
  - Aka Evaluation Strategy

**Goal:** A type safe high-level functional IL (System F) with fine-grained control over efficient calling conventions
The Long Road

To Intensional Static Polymorphism
The Long Road

To Intensional Static Polymorphism

  - Explicit monomorphic representations; implicit levities.
The Long Road

To Intensional Static Polymorphism

• S. Peyton Jones and J. Launchbury. 1991. Unboxed Values As First Class Citizens in a Non-Strict Functional Language.
  • Explicit monomorphic representations; implicit levities.

  • Explicit polymorphic representations; implicit levities.
The Long Road

To Intensional Static Polymorphism

  - Explicit monomorphic representations; implicit levities.

  - Explicit polymorphic representations; implicit levities.

  - Explicit monomorphic arities; implicit levities.
The Long Road

To Intensional Static Polymorphism

  - Explicit monomorphic representations; implicit levities.
  - Explicit polymorphic representations; implicit levities.
  - Explicit monomorphic arities; implicit levities.
  - Explicit polymorphic representations, arities, and levities.
Representation
Unboxed Types

And Their Representation
Unboxed Types

• Primitive types:
Unboxed Types

• Primitive types:
  • Int#, Float#, Char#, Word16#, Array#...
Unboxed Types

- Primitive types:
  - Int#, Float#, Char#, Word16#, Array#...

- Unboxed (Int#, Float#...) or Boxed (Array#)
Unboxed Types

• Primitive types:
  • Int#, Float#, Char#, Word16#, Array#...

• Unboxed (Int#, Float#...) or Boxed (Array#)

• **Pro**: Efficient memory
Unboxed Types

- Primitive types:
  - Int#, Float#, Char#, Word16#, Array#...
- Unboxed (Int#, Float#...) or Boxed (Array#)

**Pro:** Efficient memory

**Pro:** Efficient passing
Unboxed Types

- Primitive types:
  - Int#, Float#, Char#, Word16#, Array#...

- Unboxed (Int#, Float#...) or Boxed (Array#)

- **Pro**: Efficient memory
- **Pro**: Efficient passing

- **Con**: Different sizes
Unboxed Types

- **Primitive types:**
  - Int#, Float#, Char#, Word16#, Array#...

- **Unboxed** (Int#, Float#...) or **Boxed** (Array#)

- **Pro:** Efficient memory
- **Pro:** Efficient passing
- **Con:** Different sizes
- **Con:** Different locations
Unboxed Types

• Primitive types:
  • Int#, Float#, Char#, Word16#, Array#...

• Unboxed (Int#, Float#...) or Boxed (Array#)

• **Pro**: Efficient memory
• **Pro**: Efficient passing

• **Con**: Different sizes
• **Con**: Different locations
The Problem with Nonuniform Representation
And Compiling Static Polymorphism
The Problem with Nonuniform Representation
And Compiling Static Polymorphism

dup :: forall a. (a -> a -> a) -> a -> a
dup f x = f x x
The Problem with Nonuniform Representation
And Compiling Static Polymorphism

dup :: forall a. (a -> a -> a) -> a -> a
dup f x = f x x

(++) :: [a] -> [a] -> [a]

plusFloat# :: Float# -> Float# -> Float#
The Problem with Nonuniform Representation

And Compiling Static Polymorphism

dup :: forall a. (a -> a -> a) -> a -> a
dup f x = f x x

(++) :: [a]    -> [a]    -> [a]
plusFloat# :: Float# -> Float# -> Float#

dup (++) [0..3] — read/write pointer to [0..3]

versus

dup addFloat# 1.5 — read/write float 1.5
The Problem with Nonuniform Representation
And Compiling Static Polymorphism

dup :: forall a. (a -> a -> a) -> a -> a
dup f x = f x x

(++) :: [a] -> [a] -> [a]
plusFloat# :: Float# -> Float# -> Float#

dup (++) [0..3] — read/write pointer to [0..3]
versus
dup addFloat# 1.5 — read/write float 1.5

Assembly code of dup depends on type a!
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always represented as a pointer
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always represented as a pointer

• Restriction on quantifiers for all $\forall a::k. \ldots$
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always represented as a pointer

• Restriction on quantifiers *forall a::k. …*
  • Special kinds for unboxed types (#)
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is \textit{uniform}
  • Generic ‘a’ is always represented as a pointer

• Restriction on quantifiers \texttt{forall} a::k. ...
  • Special kinds for unboxed types (#)
  • k may be ★ or ★→★ but never #
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always represented as a pointer

• Restriction on quantifiers `forall a::k. ...`
  • Special kinds for unboxed types (#)
  • `k` may be ★ or ★->★ but never #

• Draconian restriction is unsatisfactory
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always represented as a pointer

• Restriction on quantifiers `forall a::k. ...`
  • Special kinds for unboxed types (#)
  • `k` may be ★ or ★->★ but never #

• Draconian restriction is unsatisfactory
  • *Too restrictive*: Identical definitions/code repeated for different types
    (like `error :: String -> a`)
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - Generic ‘a’ is always represented as a pointer

- Restriction on quantifiers `forall a::k. ...`
  - Special kinds for unboxed types (#)
  - `k` may be ★ or ★->★ but never #

- Draconian restriction is unsatisfactory
  - Too restrictive: Identical definitions/code repeated for different types (like `error :: String -> a`)
  - Incompatible with kind polymorphism: `forall k::Kind.forall a::k. ???`
Representation Polymorphism

Kinds As Representations
Representation Polymorphism

• Generalize $a :: \star$ to $a :: \text{TYPE}\ r$
Representation Polymorphism

• Generalize \( a :: \star \) to \( a :: \text{TYPE} \ r \)

• \( r :: \text{Rep} \) is the \textit{representation} of \( a \)
Representation Polymorphism

- Generalize $a :: ★$ to $a :: \text{TYPE} \ r$
  - $r :: \text{Rep}$ is the \textit{representation} of $a$
  - $★ = \text{TYPE} \ \text{Ptr}$
Representation Polymorphism

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
- $r :: \text{Rep}$ is the representation of $a$
- $\star = \text{TYPE } \text{Ptr}$
Representation Polymorphism

- Generalize $a :: \star$ to $a :: \text{TYPE} \ r$
  - $r :: \text{Rep}$ is the *representation* of $a$
  - $\star = \text{TYPE} \ \text{Ptr}$

$\text{error} :: \forall (a :: \star). \ \text{String} \to a$
• Generalize \( a :: \star \) to \( a :: \text{TYPE} \ r \)
  • \( r :: \text{Rep} \) is the representation of \( a \)
  • \( \star = \text{TYPE} \ \text{Ptr} \)

error :: forall (a :: \star). String -> a
errorInt# :: String -> Int#
Representation Polymorphism

• Generalize $a :: \star$ to $a :: \text{TYPE } r$
  • $r :: \text{Rep}$ is the representation of $a$
  • $\star = \text{TYPE } \text{Ptr}$

error :: forall ($a :: \star$). String $\rightarrow a$
errorInt# :: String $\rightarrow \text{Int}\#
errorFloat# :: String $\rightarrow \text{Float}\#$
Representation Polymorphism

- Generalize $a :: ★$ to $a :: \text{TYPE} \ r$
  - $r :: \text{Rep}$ is the representation of $a$
  - $★ = \text{TYPE} \ \text{Ptr}$

$error :: \forall (a :: ★). \text{String} \rightarrow a$
$error\text{Int#} :: \text{String} \rightarrow \text{Int#}$
$error\text{Float#} :: \text{String} \rightarrow \text{Float#}$

\ldots
Representation Polymorphism

- Generalize \( a :: \star \) to \( a :: \text{TYPE } r \)
  - \( r :: \text{Rep} \) is the \textit{representation} of \( a \)
  - \( \star = \text{TYPE } \text{Ptr} \)

\[
\text{error} :: \forall (a :: \star). \text{String} \rightarrow a \\
\text{errorInt#} :: \text{String} \rightarrow \text{Int#} \\
\text{errorFloat#} :: \text{String} \rightarrow \text{Float#} \\
\ldots
\]

\[
\text{error} :: \forall (r :: \text{Rep})(a :: \text{TYPE } r). \text{String} \rightarrow a
\]
Representation Polymorphism

In Function Definitions
Representation Polymorphism

In Function Definitions

```
revapp :: a -> (a -> b) -> b
revapp x f = f x
```
Representation Polymorphism

In Function Definitions

\[
\text{revapp} :: a \to (a \to b) \to b
\]
\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (r1, r2 :: \text{Rep}) (a :: \text{TYPE} \ r1) (b :: \text{TYPE} \ r2). a \to (a \to b) \to b
\]
Representation Polymorphism

In Function Definitions

\[
\text{revapp} :: a \rightarrow (a \rightarrow b) \rightarrow b
\]
\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (r_1, r_2 :: \text{Rep}) \ (a :: \text{TYPE } r_1) \ (b :: \text{TYPE } r_2). \ a \rightarrow (a \rightarrow b) \rightarrow b
\]
Representation Polymorphism

In Function Definitions

```
revapp :: a -> (a -> b) -> b
revapp x f = f x

revapp :: forall (r1, r2 :: Rep)
          (a :: TYPE r1) (b::TYPE r2).
          a -> (a -> b) -> b
```
Representation Polymorphism

In Function Definitions

\[
\text{revapp} :: \forall (r_1, r_2 :: \text{Rep}) \quad (a :: \text{TYPE } r_1) (b :: \text{TYPE } r_2).
\]
\[
a \to (a \to b) \to b
\]

\[
\text{revapp} :: a \to (a \to b) \to b
\]
\[
\text{revapp } x \ f = f \ x
\]
Representation Polymorphism

In Function Definitions

revapp :: \( a \to (a \to b) \to b \)

\[
\text{revapp } x \ f = f \ x
\]

revapp :: \( \forall \ (r1, \ r2 :: \text{Rep}) \)
\((a :: \text{TYPE} \ r1) \ (b :: \text{TYPE} \ r2) \).
\(a \to (a \to b) \to b \)

revapp :: \( \forall \ (r :: \text{Rep}) \)
\((a :: \text{TYPE} \ \text{Ptr}) \ (b :: \text{TYPE} \ r) \).
\(a \to (a \to b) \to b \)
In Function Definitions

revapp :: a -> (a -> b) -> b
revapp x f = f x

revapp :: forall (r1, r2 :: Rep) (a :: TYPE r1) (b :: TYPE r2).
        a -> (a -> b) -> b

revapp :: forall (r :: Rep) (a :: TYPE Ptr) (b :: TYPE r).
        a -> (a -> b) -> b
Representation Polymorphism

In Function Definitions

\[
\text{revapp} :: a \to (a \to b) \to b \\
\text{revapp } x \ f = f \ x
\]

\[
\text{revapp} :: \forall (r_1, r_2 :: \text{Rep})(a :: \text{TYPE } r_1)(b :: \text{TYPE } r_2). a \to (a \to b) \to b
\]

\[
\text{revapp} :: \forall (r :: \text{Rep})(a :: \text{TYPE } \text{Ptr})(b :: \text{TYPE } r). a \to (a \to b) \to b
\]
Representation Polymorphism

In Function Definitions

revapp :: a -> (a -> b) -> b
revapp x f = f x

revapp :: forall (r1, r2 :: Rep)
          (a :: TYPE r1) (b::TYPE r2).
          a -> (a -> b) -> b

revapp :: forall (r :: Rep)
          (a :: TYPE Ptr) (b :: TYPE r).
          a -> (a -> b) -> b  Assume tail-call elimination
Representation Polymorphism

In Function Definitions

revapp :: a -> (a -> b) -> b
revapp x f = f x

revapp :: forall (r1, r2 :: Rep)
          (a :: TYPE r1) (b::TYPE r2).
          a -> (a -> b) -> b

revapp :: forall (r :: Rep)
          (a :: TYPE Ptr) (b :: TYPE r).
          a -> (a -> b) -> b

Assume tail-call elimination
Restricting Representation Polymorphism
To Ensure Static Compilability

Never move or store representation-polymorphic values
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

- Moving, storing, reading, writing depends on representation
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

• Moving, storing, reading, writing depends on representation

• When this happens in assembly depends on the compiler
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

- Moving, storing, reading, writing depends on representation
- When this happens in assembly depends on the compiler
- Examples:
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

- Moving, storing, reading, writing depends on representation
- When this happens in assembly depends on the compiler
- Examples:
  - (\x. ... x ...) reads x
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

• Moving, storing, reading, writing depends on representation
• When this happens in assembly depends on the compiler
• Examples:
  • (\x. ... x ...) reads x
  • (let x = ... in ...) stores and writes x
Restricting Representation Polymorphism

To Ensure Static Compilability

Never move or store representation-polymorphic values

• Moving, storing, reading, writing depends on representation
• When this happens in assembly depends on the compiler
• Examples:
  • (\x. ... x ...) reads x
  • (let x = ... in ...) stores and writes x
  • (f x) moves (reads and writes) x
Efficient Code Abstraction

For Numeric Operations
class Num (a :: TYPE r) where
  (+) :: a -> a -> a
...
class Num (a :: TYPE r) where
  (+) :: a -> a -> a

...
Efficient Code Abstraction

For Numeric Operations

class Num (a :: TYPE r) where
  (+) :: a -> a -> a

instance Num Float# where
  x + y = addFloat# x y
Efficient Code Abstraction

For Numeric Operations

class Num (a :: TYPE r) where

  (+) :: a -> a -> a

  ...

instance Num Float# where

  x + y = addFloat# x y

  ...


class Num (a :: TYPE r) where

(+) :: a -> a -> a

instance Num Float# where

x + y = addFloat# x y

...
Efficient Code Abstraction

For Numeric Operations

class Num (a :: TYPE r) where
  (+) :: a -> a -> a
...

data NumDict (a :: TYPE r) = NumD (a -> a -> a) ...

instance Num Float# where
  x + y = addFloat# x y
...

NumFloat# = NumD addFloat# ...
class Num (a :: TYPE r) where
  (+) :: a -> a -> a
...

instance Num Float# where
  x + y = addFloat# x y
...

data NumDict (a :: TYPE r) = NumD (a -> a -> a) ...

NumFloat# = NumD addFloat# ...

(+): forall (r :: Rep) (a :: TYPE r).
  NumDict a -> (a -> a -> a)
  (+) (NumD plus ...) = plus
Arity
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]

\[ \quad \text{let } z = \text{expensive } x \]

\[ \quad \text{in } y + z \]
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \]

\[ f_1 = \lambda x \to \lambda y \to \]

\[ \text{let } z = \text{expensive } x \]

\[ \text{in } y + z \]

Arity 2
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \text{Arity 2} \quad f_2 = \lambda x \rightarrow f_1 x \]

\[ \text{let } z = \text{expensive } x \]

\[ \text{in } y + z \]
Determining Function Arity

f1, f2, f3, f4 :: Int -> Int -> Int

f1 = \x -> \y ->
    let z = expensive x
    in y + z

Arity 2

f2 = \x -> f1 x

= \x -> \y -> f1 x y

Type suggests arity 2
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \text{let } z = \text{expensive } x \]
\[ \text{in } y + z \]

Type suggests arity 2

\[ f_2 = \lambda x \rightarrow f_1 x \]

Arity 2

\[ = \lambda x \rightarrow \lambda y \rightarrow f_1 x y \]
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \)

\[ f_1 = \lambda x \to \lambda y \to \]
\[ \quad \text{let } z = \text{expensive } x \]
\[ \quad \text{in } y + z \]

\( \text{Arity 2} \quad f_2 = \lambda x \to f_1 x \quad \text{Arity 2} \]
\[ = \lambda x \to \lambda y \to f_1 x y \]

\[ f_3 = \lambda x \to \]
\[ \quad \text{let } z = \text{expensive } x \]
\[ \quad \text{in } \lambda y \to y + z \]
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \]

\[ f_1 = \lambda x \to \lambda y \to \]
\[ \text{let } z = \text{expensive } x \]
\[ \text{in } y + z \]
\[ \text{Arity 2} \]

\[ f_2 = \lambda x \to f_1 x \]
\[ = \lambda x \to \lambda y \to f_1 x y \]
\[ \text{Arity 2} \]

\[ f_3 = \lambda x \to \]
\[ \text{let } z = \text{expensive } x \]
\[ \text{in } \lambda y \to y + z \]

Hint: ‘expensive x’ may be costly, or even cause side effects
Determining Function Arity

Type suggests arity 2

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \text{let } z = \text{expensive } x \]
\[ \text{in } y + z \]
\[ \text{Arity 2} \]

\[ f_2 = \lambda x \rightarrow f_1 \ x \]
\[ \text{Arity 2} \]

\[ f_3 = \lambda x \rightarrow \]
\[ \text{let } z = \text{expensive } x \]
\[ \text{in } \lambda y \rightarrow y + z \]
\[ \text{Arity 1} \]

\[ f_4 = \lambda x \rightarrow \lambda y \rightarrow f_1 \ x \ y \]

Hint: ‘expensive \ x’ may be costly, or even cause side effects
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)

\( f_1 = \lambda x \rightarrow \lambda y \rightarrow \) \hspace{1cm} \text{Arity 2} \hspace{1cm} \text{Type suggests arity 2}

\hspace{1cm} \text{let } z = \text{expensive } x \hspace{1cm} = \lambda x \rightarrow \lambda y \rightarrow f_1 \ x \ y \hspace{1cm} \text{Arity 2}

\hspace{1cm} \text{in } y + z \hspace{1cm}

\( f_2 = \lambda x \rightarrow f_1 \ x \)

\( f_3 = \lambda x \rightarrow \) \hspace{1cm} \text{Arity 1} \hspace{1cm} \text{Arity 1}

\hspace{1cm} \text{let } z = \text{expensive } x \hspace{1cm} \text{f4 = } \lambda x \rightarrow f_3 \ x \hspace{1cm}

\hspace{1cm} \text{in } \lambda y \rightarrow y + z \hspace{1cm}

\( f_4 = \lambda x \rightarrow f_3 \ x \)

\text{Hint: ‘expensive } x\text{’ may be costly, or even cause side effects}
Determining Function Arity

f1, f2, f3, f4 :: Int -> Int -> Int

f1 = \x -> \y ->
    let z = expensive x
    in y + z

f2 = \x -> f1 x

f3 = \x ->
    let z = expensive x
    in \y -> y + z

f4 = \x -> f3 x

f1, f2, f3, f4 :: Int -> Int -> Int

Type suggests arity 2

f2 = \x -> f1 x

f3 = \x ->
    let z = expensive x
    in \y -> y + z

f4 = \x -> f3 x

Hint: ‘expensive x’ may be costly, or even cause side effects
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \)

\[
\begin{align*}
f_1 &= \lambda x \to \lambda y \to \\
&\quad \text{let } z = \text{expensive } x \\
&\quad \text{in } y + z
\end{align*}
\]
\( \text{Arity 2} \)

\[
\begin{align*}
f_2 &= \lambda x \to f_1 x
\end{align*}
\]
\( \text{Arity 2} \)

\[
\begin{align*}
f_3 &= \lambda x \to \\
&\quad \text{let } z = \text{expensive } x \\
&\quad \text{in } \lambda y \to y + z
\end{align*}
\]
\( \text{Arity 1} \)

\[
\begin{align*}
f_4 &= \lambda x \to f_3 x
\end{align*}
\]
\( \neq \lambda x \to \lambda y \to f_3 x y \)
\( \text{Arity 1} \)

Hint: ‘\text{expensive } x’ may be costly, or even cause side effects
What Is Arity?

For Curried Functions
What Is Arity?

**Definition 1.** The number of arguments a function needs before doing “serious work.”
What Is Arity?

For Curried Functions

**Definition 1.** The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3
What Is Arity?

Definition 1. The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

Definition 2. The number of times a function may be soundly $\eta$-expanded.
What Is Arity?

Definition 1. The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

Definition 2. The number of times a function may be soundly η-expanded.

- If ‘f’ is equivalent to ‘\x y z -> f x y z’, then ‘f’ has arity 3
What Is Arity?

**Definition 1.** The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be soundly $\eta$-expanded.

- If ‘f’ is equivalent to ‘\x y z -> f x y z’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.

For Curried Functions
What Is Arity?

**Definition 1.** The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be soundly \( \eta \)-expanded.

- If ‘f’ is equivalent to ‘\( \lambda x \ y \ z \rightarrow f \ x \ y \ z \)’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.

- If ‘f’ has arity 3, then ‘f 1 2 3’ can be implemented as a single call

For Curried Functions
What Is Arity?

**Definition 1.** The number of arguments a function needs before doing “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be soundly η-expanded.

- If ‘f’ is equivalent to ‘\(x \ y \ z \rightarrow f \ x \ y \ z\)’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.

- If ‘f’ has arity 3, then ‘f 1 2 3’ can be implemented as a single call
Goal: A core language with unrestricted $\eta$ for functions
Static Arity

In an Intermediate Language
Static Arity

In an Intermediate Language

• New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘a ~> b’)
  • To distinguish from the source-level $a \rightarrow b$ with different semantics
Static Arity

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘a ~> b’)
  - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are fully extensional, unlike source functions
  - $\lambda x. f x =_\eta f : a \rightsquigarrow b$ unconditionally
  - error “not a function” /= \x -> (error “not a function”) x in Haskell
Static Arity

In an Intermediate Language

• New \( a \leadsto b \) type of primitive functions (ASCII ‘\( a \leadsto b \)’)
  • To distinguish from the source-level \( a \rightarrow b \) with different semantics

• Primitive functions are **fully extensional**, unlike source functions
  • \( \lambda x . f x =^\eta f : a \leadsto b \) unconditionally
  • error “not a function” \( /= \ \lambda x \rightarrow (\text{error “not a function”}) \) \( x \) in Haskell

• With full \( \eta \), types express arity — just count the arrows
  • \( f : \text{Int} \leadsto \text{Bool} \leadsto \text{String} \) has arity 2, no matter \( f \)’s definition
Static Arity


In an Intermediate Language

• New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘$a \rightsquigarrow b$’)
  • To distinguish from the source-level $a \rightarrow b$ with different semantics

• Primitive functions are fully extensional, unlike source functions
  • $\lambda x. f x =_{\eta} f : a \rightsquigarrow b$ unconditionally
  • error “not a function” /= $\lambda x \rightarrow$ (error “not a function”) $x$ in Haskell

• With full $\eta$, types express arity — just count the arrows
  • $f : Int \rightsquigarrow Bool \rightsquigarrow String$ has arity 2, no matter $f$’s definition
The Problem With Nonuniform Arity
And Compiling Static Polymorphism
The Problem With Nonuniform Arity
And Compiling Static Polymorphism

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
The Problem With Nonuniform Arity

And Compiling Static Polymorphism

def poly :: (Int ~> Int ~> a) ~> (a, a)
  poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

- What are the arities of f and g? Counting arrows...
The Problem With Nonuniform Arity

And Compiling Static Polymorphism

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
The Problem With Nonuniform Arity

The Problem With Nonuniform Arity And Compiling Static Polymorphism

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1
The Problem With Nonuniform Arity
And Compiling Static Polymorphism

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
The Problem With Nonuniform Arity

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
  • f :: Int ~> Int ~> Bool ~> Bool has arity 3...
The Problem With Nonuniform Arity

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
  • f :: Int ~> Int ~> Bool ~> Bool has arity 3...
  • g :: Int ~> Bool ~> Bool has arity 2... oops...
The Problem With Nonuniform Arity
And Compiling Static Polymorphism

poly :: (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
  • f :: Int ~> Int ~> Bool ~> Bool has arity 3...
  • g :: Int ~> Bool ~> Bool has arity 2... oops...

• How to statically compile? Is ‘g 4’ a call? A partial application?
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always has arity 0
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always has arity 0

• Restriction on quantifiers for all \( a : : k \). ...
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always has arity 0

• Restriction on quantifiers `forall a::k. ...`
  • Special kinds for non-0 arity types (~)
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is uniform
  • Generic ‘a’ is always has arity 0

• Restriction on quantifiers \( \forall a : k. \) ...
  • Special kinds for non-0 arity types (~)
    • \( k \) may be ★ or ★→★ but never ~
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always has arity 0

• Restriction on quantifiers for all $a::k$. ...
  • Special kinds for non-0 arity types (~)
  • $k$ may be ★ or ★->★ but never ~

• Draconian restriction is unsatisfactory
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  - Generic ‘\(a\)’ is always has arity 0

• Restriction on quantifiers \(\forall a::k. \ldots\)
  - Special kinds for non-0 arity types (~)
  - \(k\) may be ★ or ★->★ but never ~

• Draconian restriction is unsatisfactory
  - **Too restrictive**: Identical definitions/code repeated for different types
    (like \(\text{repeat} :: a -> [a]\) and \([\,] :: ★ -> ★\))
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is *uniform*
  • Generic ‘a’ is always has arity 0

• Restriction on quantifiers forall a::k. ...
  • Special kinds for non-0 arity types (~)
  • k may be ★ or ★->★ but never ~

• Draconian restriction is unsatisfactory
  • Too restrictive: Identical definitions/code repeated for different types (like repeat :: a -> [a] and [] :: ★ -> ★)
  • Incompatible with kind polymorphism: forall k::Kind. forall a::k. ???
Another Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - Generic ‘a’ is always has arity 0

- Restriction on quantifiers forall a::k. ...
  - Special kinds for non-0 arity types (~)
  - k may be ★ or ★->★ but never ~

- Draconian restriction is unsatisfactory
  - **Too restrictive:** Identical definitions/code repeated for different types (like repeat :: a -> [a] and [] :: ★ -> ★)
  - **Incompatible with kind polymorphism:** forall k::Kind. forall a::k. ???

- Wait… this sounds awfully familiar…
Arity Polymorphism

Kinds As Calling Conventions
Arity Polymorphism

- Generalize $a :: TYPE \ r$ to $a :: TYPE \ r \ v$

Kinds As Calling Conventions
Arity Polymorphism

• Generalize $a : \text{TYPE} \ r$ to $a : \text{TYPE} \ r \ v$
  
  • $v : \text{Conv}$ is the calling convention of $a$
Arity Polymorphism

- Generalize $a::\text{TYPE } r$ to $a::\text{TYPE } r \ v$
- $v::\text{Conv}$ is the calling convention of $a$
- $a::\text{TYPE } r \ \text{Call}[n]$ says $a$ has arity $n$ (simplified)
Arity Polymorphism

- Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ v$
- $v::\text{Conv}$ is the *calling convention* of $a$
- $a::\text{TYPE} \ r \ \text{Call}[n]$ says $a$ has arity $n$ (simplified)
Arity Polymorphism

• Generalize $\texttt{a::TYPE r}$ to $\texttt{a::TYPE r v}$
  • $\texttt{v::Conv}$ is the calling convention of $\texttt{a}$
  • $\texttt{a::TYPE r \ Call[n]}$ says $\texttt{a}$ has arity $\texttt{n}$ (simplified)

```haskell
revapp \ x \ f = f \ x
```
Arity Polymorphism

- Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ v$
- $v::\text{Conv}$ is the *calling convention* of $a$
- $a::\text{TYPE} \ r \ \text{Call}[n]$ says $a$ has arity $n$ (simplified)

```
revapp x f = f x
```

```
revapp :: \forall (\forall v1, v2 :: \text{Conv}) (r :: \text{Rep})
        (a :: \text{TYPE} \ Ptr \ v1) (c :: \text{Type} \ r \ v2).
        a \Rightarrow (a \Rightarrow b) \Rightarrow b
```
Arity Polymorphism

- Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ v$
- $v::\text{Conv}$ is the calling convention of $a$
- $a::\text{TYPE} \ r \ \text{Call}[n]$ says $a$ has arity $n$ (simplified)

```
revapp \ x \ f = f \ x
```

```
revapp :: \forall (v1, v2 :: \text{Conv}) (r :: \text{Rep})
\ (a :: \text{TYPE} \ \text{Ptr} \ v1) (c :: \text{Type} \ r \ v2).
\ a \ \rightarrow \ (a \ \rightarrow \ b) \ \rightarrow \ b
```
Arity Polymorphism

- Generalize $a :: TYPE \ r$ to $a :: TYPE \ r \ v$
- $v :: Conv$ is the calling convention of $a$
- $a :: TYPE \ r$ Call[$n$] says $a$ has arity $n$ (simplified)

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (v1, v2 :: Conv) (r :: Rep) (a :: TYPE \ Ptr \ v1) (c :: Type \ r \ v2). a \leadsto (a \leadsto b) \leadsto b
\]
Arity Polymorphism

- Generalize \( \texttt{a::TYPE \ r} \) to \( \texttt{a::TYPE \ r \ v} \)
- \( \texttt{v::Conv} \) is the calling convention of \( \texttt{a} \)
- \( \texttt{a::TYPE \ r \ \text{Call}[n]} \) says \( \texttt{a} \) has arity \( \text{n} \) (simplified)

\[
\text{revapp \ x \ f = f \ x}
\]

\[
\text{revapp :: forall (v1, v2 :: Conv) (r :: Rep) (a :: TYPE \ Ptr \ v1) (c :: Type \ r \ v2).}
\]

\[
\text{a ~> (a ~> b) ~> b}
\]
Arity Polymorphism

- Generalize \( a :: \text{TYPE} \ r \) to \( a :: \text{TYPE} \ r \ v \)
- \( v :: \text{Conv} \) is the \textit{calling convention} of \( a \)
- \( a :: \text{TYPE} \ r \ \text{Call}[n] \) says \( a \) has arity \( n \) (simplified)

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (v1, v2 :: \text{Conv}) (r :: \text{Rep}) (a :: \text{TYPE} \ \text{Ptr} \ v1) (c :: \text{Type} \ r \ v2). a \Rightarrow (a \Rightarrow b) \Rightarrow b
\]

\[
\text{revapp} :: \forall (v :: \text{Conv}) (r :: \text{Rep}) (a :: \text{TYPE} \ \text{Ptr} \ c) (c :: \text{Type} \ r \ \text{Call}[1]). a \Rightarrow (a \Rightarrow b) \Rightarrow b
\]
Arity Polymorphism

- Generalize \( a::\text{TYPE} \rightarrow r \) to \( a::\text{TYPE} \rightarrow r \rightarrow v \)
- \( v::\text{Conv} \) is the calling convention of \( a \)
- \( a::\text{TYPE} \rightarrow \text{Call}[n] \) says \( a \) has arity \( n \) (simplified)

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (v_1, v_2 :: \text{Conv}) (r :: \text{Rep}) (a :: \text{TYPE} \rightarrow \text{Ptr} v_1) (c :: \text{Type} r \rightarrow v_2).
\]

\[
a \rightarrow (a \rightarrow b) \rightarrow b
\]

\[
\text{revapp} :: \forall (v :: \text{Conv}) (r :: \text{Rep}) (a :: \text{TYPE} \rightarrow \text{Ptr} c) (c :: \text{Type} r \rightarrow \text{Call}[1]).
\]

\[
a \rightarrow (a \rightarrow b) \rightarrow b
\]
Arity Polymorphism

• Generalize $a :: \text{TYPE} \ r$ to $a :: \text{TYPE} \ r \ v$
• $v :: \text{Conv}$ is the calling convention of $a$
• $a :: \text{TYPE} \ r \ \text{Call}[n]$ says $a$ has arity $n$ (simplified)

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (v1, v2 :: \text{Conv}) \ (r :: \text{Rep}) \ (a :: \text{TYPE} \ \text{Ptr} \ v1) \ (c :: \text{Type} \ r \ v2). \ a \Rightarrow (a \Rightarrow b) \Rightarrow b
\]

\[
\text{revapp} :: \forall (v :: \text{Conv}) \ (r :: \text{Rep}) \ (a :: \text{TYPE} \ \text{Ptr} \ c) \ (c :: \text{Type} \ r \ \text{Call}[1]). \ a \Rightarrow (a \Rightarrow b) \Rightarrow b
\]
Arity Polymorphism

- Generalize `a :: TYPE r` to `a :: TYPE r v`
- `v :: Conv` is the *calling convention* of `a`
- `a :: TYPE r Call[n]` says `a` has arity `n` (simplified)

```haskell
revapp x f = f x

revapp :: forall (v1, v2 :: Conv) (r :: Rep) (a :: TYPE Ptr v1) (c :: Type r v2).
  a ~> (a ~> b) ~> b
```

```haskell
revapp :: forall (v :: Conv) (r :: Rep) (a :: TYPE Ptr c) (c :: Type r Call[1]).
  a ~> (a ~> b) ~> b
```
Arity Polymorphism

And Higher-Order Functions
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
    (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]). (Int ~> Int ~> a) ~> (a,a)

poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
  (Int ~> Int ~> a) ~> (a,a)

poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3
Arity Polymorphism

poly :: forall (a :: TYPE_PTR Call[2]).
      (Int ~> Int ~> a) ~> (a, a)

poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

- f :: Int ~> Int ~> a :: TYPE_PTR Call[4] has arity 4
- g :: Int ~> a :: TYPE_PTR Call[3] has arity 3

And Higher-Order Functions
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
    (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

\[ \text{poly} :: \forall (v :: \text{Conv}) \ (a :: \text{TYPE Ptr } v).\]
\[ (\text{Int} \rightarrow \text{Int} \rightarrow a) \rightarrow (a,a) \]
\[ \text{poly } f = \text{let } g :: \text{Int} \rightarrow a = f 3 \text{ in } (g 4, g 5) \]
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]). (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

poly :: forall (a :: TYPE Ptr Call[2]). (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

And Higher-Order Functions

f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v). (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
      (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v).
      (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]). (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

- f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
- g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v). (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
      (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v).
       (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

And Higher-Order Functions
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
       (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v).
       (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
      (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v).
      (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[2+?] has an unknown arity ≥ 2
Arity Polymorphism

\[
poly :: \forall (a :: \text{TYPE\,Ptr\,Call}[2]).
\quad \text{(Int \to Int \to a) \to (a,a)}
\]
\[
poly f = \text{let } g :: \text{Int \to a} = f 3 \text{ in (g 4, g 5)}
\]

- \( f :: \text{Int \to Int \to a :: TYPE\,Ptr\,Call}[4] \) has arity 4
- \( g :: \text{Int \to a :: TYPE\,Ptr\,Call}[3] \) has arity 3

\[
poly :: \forall (v :: \text{Conv}) (a :: \text{TYPE\,Ptr\,v}).
\quad \text{(Int \to Int \to a) \to (a,a)}
\]
\[
poly f = \text{let } g :: \text{Int \to a} = f 3 \text{ in (g 4, g 5)}
\]

- \( f :: \text{Int \to Int \to a :: TYPE\,Ptr\,Call}[2+?] \) has an unknown arity \( \geq 2 \)
- \( g :: \text{Int \to Int \to a :: TYPE\,Ptr\,Call}[1+?] \) has an unknown arity \( \geq 1 \)
Arity Polymorphism

poly :: forall (a :: TYPE Ptr Call[2]).
        (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[4] has arity 4
• g :: Int ~> a :: TYPE Ptr Call[3] has arity 3

poly :: forall (v :: Conv) (a :: TYPE Ptr v).
        (Int ~> Int ~> a) ~> (a,a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)

• f :: Int ~> Int ~> a :: TYPE Ptr Call[2+] has an unknown arity ≥ 2
• g :: Int ~> Int ~> a :: TYPE Ptr Call[1+] has an unknown arity ≥ 1

And Higher-Order Functions
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code

- Calling and defining function code depends on arity
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code

• Calling and defining function code depends on arity

• When this happens in assembly depends on the compiler
Restricting Arity Polymorphism

Never invoke or define arity-polymorphic code

• Calling and defining function code depends on arity
• When this happens in assembly depends on the compiler
• Examples:
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code

• Calling and defining function code depends on arity
• When this happens in assembly depends on the compiler
• Examples:
  • (let f = \x y z -> ... in ...) defines code for f
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code

- Calling and defining function code depends on arity
- When this happens in assembly depends on the compiler
- Examples:
  - (let \( f = \lambda x \ y \ z \rightarrow \ldots \) in \ldots) defines code for \( f \)
  - (\( \lambda x \ y \rightarrow f \ y \ x \)) calls code at \( f \)
Restricting Arity Polymorphism

To Ensure Static Compilability

Never invoke or define arity-polymorphic code

- Calling and defining function code depends on arity
- When this happens in assembly depends on the compiler
- Examples:
  - (let f = \x y z -> … in …) defines code for f
  - (\x y -> f y x) calls code at f
  - (f (\x -> ...)) creates code for function pointer passed to f
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a) = Nil | Cons a (List a)
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a) = Nil | Cons a (List a)

Nil ::
    List a
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a) = Nil | Cons a (List a)

Nil :: List a

Cons :: a ~> List a ~> List a
data List (a :: TYPE Ptr v)
  = Nil | Cons a (List a)

Nil ::
  List a

Cons ::
  a ~> List a ~> List a
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a :: TYPE Ptr v)
  = Nil ∣ Cons a (List a)

Nil :: forall (v :: Conv) (a :: TYPE Ptr v).
  List a

Cons ::
  a ~> List a ~> List a
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a :: TYPE Ptr v)
    = Nil | Cons a (List a)

Nil :: forall (v :: Conv) (a :: TYPE Ptr v). List a

Cons :: forall (v :: Conv) (a :: TYPE Ptr v). a ~> List a ~> List a
data List (a :: TYPE Ptr v)  
  = Nil | Cons a (List a)

Nil :: forall (v :: Conv) (a :: TYPE Ptr v).  
  List a

Cons :: forall (v :: Conv) (a :: TYPE Ptr v).  
  a ~> List a ~> List a

repeat x = Cons x (repeat x)
Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

data List (a :: TYPE Ptr v)
    = Nil | Cons a (List a)

Nil :: forall (v :: Conv) (a :: TYPE Ptr v).
    List a

Cons :: forall (v :: Conv) (a :: TYPE Ptr v).
    a ~> List a ~> List a

repeat x = Cons x (repeat x)

repeat :: forall (v :: Conv) (a :: TYPE Ptr v).
    a ~> List a
Efficient and Correct Abstractions
For Higher-Order Type Classes
Efficient and Correct Abstractions

For Higher-Order Type Classes

class Functor (f)
  fmap :: (a -> b) -> f a -> f b

) where
class Functor (f :: TYPE r v -> TYPE r’ v’) where 
  fmap :: (a -> b) -> f a -> f b
class Functor (f :: TYPE r v -> TYPE r’ v’) where
  fmap :: (a -> b) -> f a -> f b

newtype Reader (e :: TYPE r v) (a :: TYPE r’ v’)
  = Read (e ~> a)
Efficient and Correct Abstractions

For Higher-Order Type Classes

class Functor (f :: TYPE r v -> TYPE r’ v’) where
  fmap :: (a -> b) -> f a -> f b

newtype Reader (e :: TYPE r v) (a :: TYPE r’ v’)
  = Read (e ~> a)

instance Functor (Reader e) where
class Functor (f :: TYPE r v -> TYPE r’ v’) where
    fmap :: (a -> b) -> f a -> f b

newtype Reader (e :: TYPE r v) (a :: TYPE r’ v’)
    = Read (e ~> a)

instance Functor (Reader e) where
    fmap f (Read g) = Read (\x ~> f (g x))
Efficient and Correct Abstractions

For Higher-Order Type Classes

class Functor (f :: TYPE r v -> TYPE r’ v’) where
    fmap :: (a -> b) -> f a -> f b

newtype Reader (e :: TYPE r v) (a :: TYPE r’ v’)
    = Read (e ~> a)

instance Functor (Reader e) where
    fmap f (Read g) = Read (\x ~> f (g x))

• But now fmap id (Read g) = Read g! (hint: requires η)
Efficient and Correct Abstractions

For Higher-Order Type Classes

class Functor (f :: TYPE r v -> TYPE r' v') where
  fmap :: (a -> b) -> f a -> f b

newtype Reader (e :: TYPE r v) (a :: TYPE r' v')
  = Read (e ~> a)

instance Functor (Reader e) where
  fmap f (Read g) = Read (\x ~> f (g x))

• But now fmap id (Read g) = Read g! (hint: requires η)

• Better for performance and correctness
Levity
Unrestricted $\eta$ Is \textbf{Inconsistent} With Restricted $\beta$

In the $\lambda$-calculus

$$\lambda x. M \, x =_\eta M$$
Unrestricted $\eta$ is **Inconsistent** With Restricted $\beta$

In the $\lambda$-calculus

\[
\lambda x. M\ x =_\eta M
\]

\[
\lambda x. \bot\ x =_\eta \bot
\]
Unrestricted $\eta$ Is Inconsistent With Restricted $\beta$

In the $\lambda$-calculus

\[ \lambda x. M \ x =_{\eta} M \]

\[ \lambda x. \bot \ x =_{\eta} \bot \]

\[ (\lambda z. 5) (\lambda x. \bot \ x) =_{\eta} (\lambda z. 5) \bot \]
Unrestricted $\eta$ Is Inconsistent With Restricted $\beta$

In the $\lambda$-calculus

$$\lambda x. M \ x =_{\eta} M$$

$$\lambda x. \bot \ x =_{\eta} \bot$$

$$(\lambda z. \ 5) \ (\lambda x. \bot \ x) =_{\eta} (\lambda z. \ 5) \bot$$

$\beta_v$
Unrestricted $\eta$ Is Inconsistent With Restricted $\beta$

In the $\lambda$-calculus

\[ \lambda x. M \ x =_\eta M \]

\[ \lambda x. \perp \ x =_\eta \perp \]

\[ (\lambda z. \ 5 \) (\lambda x. \perp \ x) =_\eta (\lambda z. \ 5) \perp \]

\[ \beta_v \]

5
Unrestricted $\eta$ Is Inconsistent With Restricted $\beta$

In the $\lambda$-calculus

$$\lambda x. M \ x =_\eta M$$

$$\lambda x. \bot \ x =_\eta \bot$$

$$(\lambda z. 5) \ (\lambda x. \bot \ x) =_\eta (\lambda z. 5) \bot$$

$\beta_v$ $\beta_v$

5
Unrestricted $\eta$ Is Inconsistent With Restricted $\beta$

In the $\lambda$-calculus

$$\lambda x. M \ x =_\eta M$$
$$\lambda x. \bot x =_\eta \bot$$

$$(\lambda z. \, 5) \ (\lambda x. \bot x) =_\eta (\lambda z. \, 5) \bot$$
Unrestricted \( \eta \) Is Inconsistent With Restricted \( \beta \)

In the \( \lambda \)-calculus

\[
\begin{align*}
\lambda x . M \ x &= \eta M \\
\lambda x . \bot \ x &= \eta \ \bot \\
(\lambda z . 5) \ (\lambda x . \bot \ x) &= \eta (\lambda z . 5) \ \bot
\end{align*}
\]

\[
\frac{\beta_v}{5} \neq \frac{\bot}{\bot}
\]
Goal: A core language with *unrestricted* $\eta$ for functions and *restricted* $\beta$ for other types
Unboxed Data Is *Eager*

Not *Lazy*
Unboxed Data Is *Eager*

\[ \text{addFloat} : \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \]

*Not Lazy*
Unboxed Data Is *Eager*

`addFloat# :: Float# ~> Float# ~> Float#`

- Compiles to machine primop for float addition in specialized registers

Not *Lazy*
Unboxed Data Is **Eager**

addFloat# :: Float# ~> Float# ~> Float#

- Compiles to machine primop for float addition in specialized registers

let x :: Float# = addFloat# 1.5 3.5 in …
Unboxed Data Is **Eager**

```
addFloat# :: Float# ~> Float# ~> Float#

• Compiles to machine primop for float addition in specialized registers

let x :: Float# = addFloat# 1.5 3.5 in …

• Compiles to code that stores \((1.5 + 3.5)\) in float register \(x\)
```
Unboxed Data Is **Eager**

addFloat# :: Float# ~> Float# ~> Float#

- Compiles to machine primop for float addition in specialized registers

let x :: Float# = addFloat# 1.5 3.5 in …

- Compiles to code that stores \((1.5 + 3.5)\) in float register \(x\)

- Can \(x\) be lazy?
Unboxed Data Is *Eager*

addFloat# :: Float# ~> Float# ~> Float#

• Compiles to machine primop for float addition in specialized registers

let x :: Float# = addFloat# 1.5 3.5 in ...

• Compiles to code that stores \((1.5 + 3.5)\) in float register \(x\)

• Can \(x\) be lazy?

• No!
Unboxed Data Is **Eager**

\[
\text{addFloat} \quad :: \quad \text{Float} \quad \rightarrow \quad \text{Float} \quad \rightarrow \quad \text{Float}
\]

- Compiles to machine primop for float addition in specialized registers

\[
\text{let x} \quad :: \quad \text{Float} \quad = \quad \text{addFloat} \ 1.5 \ 3.5 \ \text{in} \ldots
\]

- Compiles to code that stores \((1.5 + 3.5)\) in float register \(x\)

- **Can x be lazy?**
  - **No!**
  - \(x\) stores a floating-point number
Unboxed Data Is *Eager*

addFloat# :: Float# ~> Float# ~> Float#

- Compiles to machine primop for float addition in specialized registers

let x :: Float# = addFloat# 1.5 3.5 in …

- Compiles to code that stores \((1.5 + 3.5)\) in float register \(x\)

- **Can \(x\) be lazy?**
  - No!
  - \(x\) stores a floating-point number
  - Lazy thunks must be represented as pointers
Primitive Functions are *Called*

Not *Evaluated*
Primitive Functions are *Called*

\[ x = \text{let } f :: \text{Int } \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f \ldots \]
Primitive Functions are *Called*

```haskell
x = let f :: Int -> Int = expensive 100 in ...f...f...
```

- *When is expensive 100 evaluated?*
Primitive Functions are *Called*

\[
x = \text{let } f :: \text{Int} \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f\ldots
\]

- **When is** `expensive 100` **evaluated**?
  - Call-by-value: first, before binding `f`
Primitive Functions are *Called*

\[
x = \text{let } f :: \text{Int} \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots \!f\!\ldots\!f\!
\]

- **When is \text{expensive 100} evaluated?**
  - **Call-by-value:** first, before binding \( f \)
  - **Call-by-need:** later, but only once, when \( f \) is first demanded
Primitive Functions are *Called*

\[
x = \text{let } f :: \text{Int} \to \text{Int} = \text{expensive } 100 \text{ in } ...f...f...
\]

- **When is \text{expensive } 100 evaluated?**
  - Call-by-value: first, before binding \( f \)
  - Call-by-need: later, but only once, when \( f \) is first demanded
  - Call-by-name: later, re-evaluated every time \( f \) is demanded
Primitive Functions are \textit{Called}

\[ x = \text{let } f :: \text{Int} \to \text{Int} = \text{expensive 100} \text{ in } ...f...f... \]

• When is \text{expensive 100} evaluated?
  • Call-by-value: first, before binding \( f \)
  • Call-by-need: later, but only once, when \( f \) is first demanded
  • Call-by-name: later, re-evaluated every time \( f \) is demanded

\[ x' = \text{let } f :: \text{Int} \to \text{Int} = \lambda y \to \text{expensive 100 } y \text{ in } ...f...f... \]
Primitive Functions are \textit{Called}

\[ x = \text{let } f :: \text{Int } \rightarrow \text{Int} = \text{expensive 100 } \text{in } \ldots f \ldots f \ldots \]

- When is \text{expensive 100} evaluated?
  - \text{Call-by-value:} first, before binding \( f \)
  - \text{Call-by-need:} later, but only once, when \( f \) is first demanded
  - \text{Call-by-name:} later, re-evaluated every time \( f \) is demanded

\[ x' = \text{let } f :: \text{Int } \rightarrow \text{Int} = \lambda y \rightarrow \text{expensive 100 } y \text{ in } \ldots f \ldots f \ldots \]

- \( x = x' \) by \( \eta \), so they must be the same
Primitive Functions are **Called**

\[ x = \text{let } f :: \text{Int} \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f \ldots \]

- **When is \text{expensive 100} evaluated?**
  - Call-by-value: first, before binding \( f \)
  - Call-by-need: later, but only once, when \( f \) is first demanded
  - Call-by-name: later, re-evaluated every time \( f \) is demanded

\[ x' = \text{let } f :: \text{Int} \rightarrow \text{Int} = \lambda y \rightarrow \text{expensive 100 } y \text{ in } \ldots f \ldots f \ldots \]

- \( x = x' \) by \( \eta \), so they must be the same
- \( x' \) always follows call-by-name order! So \( x \) does, too
Primitive Functions are Called

\[ x = \text{let } f :: \text{Int } \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f \ldots \]

• When is expensive 100 evaluated?
  • Call-by-value: first, before binding f
  • Call-by-need: later, but only once, when f is first demanded
  • Call-by-name: later, re-evaluated every time f is demanded

\[ x' = \text{let } f :: \text{Int } \rightarrow \text{Int} = \backslash y \rightarrow \text{expensive 100 } y \text{ in } \ldots f \ldots f \ldots \]

• \( x = x' \) by \( \eta \), so they must be the same
• \( x' \) always follows call-by-name order! So \( x \) does, too
• Primitive functions are never just evaluated; they are always called
Currying

When Partial Application Matters
Currying

\[ f_3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z \]
Currying

When Partial Application Matters

\[ f_3 :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ f_3 = \lambda x \to \text{let } z = \text{expensive } x \text{ in } \lambda y \to y + z \]

- Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
Currying

\[ f_3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z \]

- Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
  - \( \text{map } (f_3 \ 100) \ [1..10^6] \) recomputes ‘expensive 100’ a million times 😞
Currying

\[ f_3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z \]

• Because of η, \( f_3 \) now has arity 2, not 1!
  • map \( (f_3 \ 100) \ [1..10^6] \) recomputes ‘expensive 100’ a million times 😞

\[ f_3' :: \text{Int} \rightarrow \{ \text{Int} \rightarrow \text{Int} \} \]
\[ f_3' = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos}(\lambda y \rightarrow y + z) \]

\text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\}
Currying

When Partial Application Matters

\( f_3 :: \text{Int} \to \text{Int} \to \text{Int} \)
\( f_3 = x \to \text{let } z = \text{expensive } x \text{ in } y \to y + z \)

• Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
  • map \((f_3 100) [1..10^6]\) recomputes ‘expensive 100’ a million times 😞

\( f_3' :: \text{Int} \to \{ \text{Int} \to \text{Int} \} \)
\( f_3' = x \to \text{let } z = \text{expensive } x \text{ in } \text{Clos} (y \to y + z) \)

• \( f_3' \) is an arity 1 function; returns a closure \{Int~>Int\} of an arity 1 function

\text{Clos} :: (\text{Int} \to \text{Int}) \to \{\text{Int} \to \text{Int}\}
Currying

When Partial Application Matters

\[ f_3 \, :: \, \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z \]

- Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
  - map \( f_3 \, 100 \) \([1..10^6]\) recomputes ‘expensive 100’ a million times ☹

\[ f_3' \, :: \, \text{Int} \rightarrow \{ \text{Int} \rightarrow \text{Int} \} \]
\[ f_3' = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos } (\lambda y \rightarrow y + z) \]

- \( f_3' \) is an arity 1 function; returns a closure \( \{\text{Int} \rightarrow \text{Int}\} \) of an arity 1 function
  - map \( \text{App } (f_3' \, 100) \) \([1..10^6]\) computes ‘expensive 100’ only once ☺

Clos :: (Int ~> Int) ~> {Int ~> Int}  
App :: {Int ~> Int} ~> Int ~> Int
Levity and Evaluation Strategy

Denotationally and Logically
Levity and Evaluation Strategy

• $A_\perp$ is the *lifted* version of $A$
Levity and Evaluation Strategy

• $A_\bot$ is the lifted version of $A$
  
  • $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
Levity and Evaluation Strategy

• $A\perp$ is the *lifted* version of $A$

• $A\perp$ adds a special, unique value $\perp$ to $A$ denoting divergent computation

• E.g., $\mathbb{N}_\perp = \{\perp, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \perp$, and $(A \rightarrow B)_\perp = \{\perp\} \cup \{\lambda x.f(x) | f \in A \rightarrow B\}$
Levity and Evaluation Strategy

- $A_\bot$ is the *lifted* version of $A$
  - $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  - E.g., $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \bot$, and $(A \rightarrow B)_\bot = \{\bot\} \cup \{\lambda x . f(x) \mid f \in A \rightarrow B\}$
- Unboxed types and primitive functions are *unlifted*
Levity and Evaluation Strategy

- \( A_\perp \) is the *lifted* version of \( A \)
  - \( A_\perp \) adds a special, unique value \( \perp \) to \( A \) denoting divergent computation
  - E.g., \( \mathbb{N}_\perp = \{ \perp, 0, 1, 2, 3, \ldots \} \) so that \( 1/0 = \perp \), and \( (A \rightarrow B)_\perp = \{ \perp \} \cup \{ \lambda x . f(x) \mid f \in A \rightarrow B \} \)
- Unboxed types and primitive functions are *unlifted*
  - \( \text{Int#} = \{ 0, 1, -1, 2, -2, \ldots \} \) and \( A \rightsquigarrow B = \{ \lambda x . f(x) \mid f \in A \rightarrow B \} \) denotes only real functions
Levity and Evaluation Strategy

• $A_\bot$ is the *lifted* version of $A$

  • $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  
  • E.g., $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots \}$ so that $1/0 = \bot$, and $(A \to B)_\bot = \{\bot\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$

• Unboxed types and primitive functions are *unlifted*

  • $\text{Int}# = \{0, 1, -1, 2, -2, \ldots \}$ and $A \mapsto B = \{\lambda x . f(x) \mid f \in A \to B\}$ denotes only real functions

  • Lifting implies *worse performance* (for data, functions)
Levity and Evaluation Strategy

• $A_\perp$ is the \textit{lifted} version of $A$
  • $A_\perp$ adds a special, unique value $\perp$ to $A$ denoting divergent computation
  • E.g., $\mathbb{N}_\perp = \{\perp, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \perp$, and $(A \to B)_\perp = \{ \perp \} \cup \{ \lambda x . f(x) \mid f \in A \to B \}$

• Unboxed types and primitive functions are \textit{unlifted}
  • $\text{Int}\# = \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightarrow B = \{ \lambda x . f(x) \mid f \in A \to B \}$ denotes only real functions
  • Lifting implies \textit{worse performance} (for data, functions)
    • Indirection, dynamic checks, multiple function calls/jumps

\textit{Denotationally and Logically}
Levity and Evaluation Strategy

• $A_\bot$ is the *lifted* version of $A$
  • $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  • E.g., $\mathbb{N}_\bot = \{ \bot, 0, 1, 2, 3, \ldots \}$ so that $1/0 = \bot$, and $(A \rightarrow B)_\bot = \{ \bot \} \cup \{ \lambda x . f(x) \mid f \in A \rightarrow B \}$

• Unboxed types and primitive functions are *unlifted*
  • $\mathbb{I} \rightarrow \mathbb{I} \rightarrow \mathbb{I}$
    • $\text{Int}# = \{ 0, 1, -1, 2, -2, \ldots \}$ and $A \rightsquigarrow B = \{ \lambda x . f(x) \mid f \in A \rightarrow B \}$ denotes only real functions
    • Lifting implies *worse performance* (for data, functions)
    • Indirection, dynamic checks, multiple function calls/jumps

• Denotation of computations of type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ is:
Levity and Evaluation Strategy

- $A_\bot$ is the *lifted* version of $A$
  - $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  - E.g., $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \bot$, and $(A \to B)_\bot = \{\bot\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - $\mathbb{N}# = \{0, 1, -1, 2, -2, \ldots\}$ and $A \leftrightarrow B = \{\lambda x . f(x) \mid f \in A \to B\}$ denotes only real functions
  - Lifting implies *worse performance* (for data, functions)
  - Indirection, dynamic checks, multiple function calls/jumps
- Denotation of computations of type $\text{Int} \to \text{Int} \to \text{Int}$ is:
  - Call-by-name: $\text{Int}_\bot \to \text{Int}_\bot \to \text{Int}_\bot$
Levity and Evaluation Strategy

- $A_\perp$ is the *lifted* version of $A$
  - $A_\perp$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  - E.g., $\mathbb{N}_\perp = \{ \bot, 0, 1, 2, 3, \ldots \}$ so that $1/0 = \bot$, and $(A \rightarrow B)_\perp = \{ \bot \} \cup \{ \lambda x. f(x) \mid f \in A \rightarrow B \}$

- Unboxed types and primitive functions are *unlifted*
  - $\text{Int}_\# = \{ 0, 1, -1, 2, -2, \ldots \}$ and $A \leftrightarrow B = \{ \lambda x. f(x) \mid f \in A \rightarrow B \}$ denotes only real functions
  - Lifting implies *worse performance* (for data, functions)
  - Indirection, dynamic checks, multiple function calls/jumps

- Denotation of computations of type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ is:
  - Call-by-name: $\text{Int}_\perp \rightarrow \text{Int}_\perp \rightarrow \text{Int}_\perp$
  - Call-by-value: $(\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}_{\perp}))_{\perp}$
Levity and Evaluation Strategy

• $A_{\bot}$ is the *lifted* version of $A$
  • $A_{\bot}$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  • E.g., $\mathbb{N}_{\bot} = \{\bot, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \bot$, and $(A \to B)_{\bot} = \{\bot\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$

• Unboxed types and primitive functions are *unlifted*
  • $\text{Int#} = \{0, 1, -1, 2, -2, \ldots\}$ and $A \leadsto B = \{\lambda x . f(x) \mid f \in A \to B\}$ denotes only real functions
  • Lifting implies *worse performance* (for data, functions)
  • Indirection, dynamic checks, multiple function calls/jumps

• Denotation of computations of type $\text{Int} \to \text{Int} \to \text{Int}$ is:
  • Call-by-name: $\text{Int}_{\bot} \to \text{Int}_{\bot} \to \text{Int}_{\bot}$
  • Call-by-value: $(\text{Int} \to (\text{Int} \to \text{Int}_{\bot})_{\bot})_{\bot}$
  • Call-by-push-value: $\text{Int} \to \text{Int} \to \text{Int}_{\bot}$
Levity and Evaluation Strategy

• $A_\bot$ is the lifted version of $A$
  • $A_\bot$ adds a special, unique value $\bot$ to $A$ denoting divergent computation
  • E.g., $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots\}$ so that $1/0 = \bot$, and $(A \to B)_\bot = \{\bot\} \cup \{\lambda x.f(x) \mid f \in A \to B\}$
• Unboxed types and primitive functions are unlifted
  • Int$\# = \{0, 1, -1, 2, -2, \ldots\}$ and $A \hookrightarrow B = \{\lambda x. f(x) \mid f \in A \to B\}$ denotes only real functions
  • Lifting implies worse performance (for data, functions)
  • Indirection, dynamic checks, multiple function calls/jumps
• Denotation of computations of type $Int \to Int \to Int$ is:
  • Call-by-name: $Int_\bot \to Int_\bot \to Int_\bot$
  • Call-by-value: $(Int \to (Int \to Int_\bot)_\bot)_\bot$
  • Call-by-push-value: $Int \to Int \to Int_\bot$
• Logical polarity reveals the semantics for best performance
Levity Polymorphism

Call vs Eval, Revisited
Levity Polymorphism

• Code that isn’t called is evaluated
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
Levity Polymorphism

- Code that isn’t **called** is **evaluated**
  
  -Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  -Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  -Eval g :: Conv — polymorphic evaluation, with levity variable g
Levity Polymorphism


Call vs Eval, Revisited

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with levity variable g
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval \( U \) :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval \( L \) :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval \( g \) :: Conv — polymorphic evaluation, with levity variable \( g \)

\( \text{Int } g :: \text{TYPE } \text{Ptr } (\text{Eval } g) -- \text{boxed, levity-}g \text{ ints} \)
Levity Polymorphism


Call vs Eval, Revisited

- Code that isn’t called is evaluated
  - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  - Eval g :: Conv — polymorphic evaluation, with levity variable g

Int g :: TYPE Ptr (Eval g) -- boxed, levity-g ints

sum :: forall (g1 g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

• Code that isn’t **called** is **evaluated**
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with levity variable g

Int g :: TYPE Ptr (Eval g) -- boxed, levity-g ints

sum :: forall (g1 g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with levity variable g

\[ \text{Int } g :: \text{TYPE } \text{Ptr } (\text{Eval } g) \text{ -- boxed, levity-}g \text{ ints} \]

\[ \text{sum} :: \forall (g1 \ g2 :: \text{Levity}). \ [\text{Int } g1] \rightarrow \text{Int } g2 \]

\[ \text{sum } [] = 0 \]

\[ \text{sum } (x : \text{x}s) = x + \text{sum } \text{x}s \]
Levity Polymorphism


Call vs Eval, Revisited

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with levity variable g

Int g :: TYPE Ptr (Eval g) -- boxed, levity-g ints

sum :: forall (g1 g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval $U$ :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval $L$ :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval $g$ :: Conv — polymorphic evaluation, with levy variable $g$

$Int\ g :: TYPE\ Ptr\ (Eval\ g)$ -- boxed, levy-$g$ ints

$sum :: forall\ (g1\ g2 :: Levity).\ [Int\ g1] \rightarrow\ Int\ g2$

$sum\ [] = 0$

$sum\ (x : xs) = x + sum\ xs$
Levity Polymorphism

- Code that isn’t **called** is **evaluated**
  - Eval $U : : \text{Conv} -$ eager (call-by-value) evaluation, Unlifted values
  - Eval $L : : \text{Conv} -$ lazy (call-by-need) evaluation, Lifted values
  - Eval $g : : \text{Conv} -$ polymorphic evaluation, with levity variable $g$

Int $g : : \text{TYPE Ptr (Eval } g) -$ boxed, levity-$g$ ints

$\text{sum} : : \forall (g1 \ g2 : : \text{Levity}). [\text{Int } g1] \rightarrow \text{Int } g2$

$\text{sum } [] = 0$

$\text{sum } (x : : xs) = x + \text{sum } xs$
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  - Eval g :: Conv — polymorphic evaluation, with levity variable g

Int g :: TYPE Ptr (Eval g) -- boxed, levity-g ints

sum :: forall (g1 g2 :: Levity). [Int g1] ~> Int g2
sum []       = 0
sum (x : xs) = x + sum xs

sum (I# z : xs) = case sum xs of I# y -> I# (z +# y)
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass

levity-polymorphic computations
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass levity-polymorphic computations

• Evaluation order of serious arguments and let's depends on levity
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass levity-polymorphic computations

• Evaluation order of serious arguments and lets depends on levity
• What counts as “serious computation” depends on the compiler
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass levity-polymorphic computations

• Evaluation order of serious arguments and let's depends on levity
• What counts as “serious computation” depends on the compiler
• Examples:
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass levity-polymorphic computations

- Evaluation order of serious arguments and `let`s depends on levity
- What counts as “serious computation” depends on the compiler
- Examples:
  - `(let x = expensive 100 in ...) binds x to expensive 100`
Restricting Levity Polymorphism

To Ensure Static Compilability

Never bind or pass
levity-polymorphic computations

• Evaluation order of serious arguments and let's depends on levity
• What counts as “serious computation” depends on the compiler
• Examples:
  • (let x = expensive 100 in ...) binds x to expensive 100
  • (f (expensive 100)) passes expensive 100 to f
Code Reuse

Between Eager and Lazy Programs
Code Reuse

Between Eager and Lazy Programs

data List (g :: Levity) (a :: TYPE Ptr v) :: TYPE Ptr (Eval g)
  = Nil | Cons a (List g
data List (g :: Levity) (a :: TYPE Ptr v) ::
   = Nil | Cons a (List g a)
Code Reuse

Between Eager and Lazy Programs

data List (g :: Levity) (a :: TYPE Ptr ν) :: TYPE Ptr (Eval g) = Nil | Cons a (List g a)
data List (g :: Levity) (a :: TYPE Ptr v) :: TYPE Ptr (Eval g)
  = Nil | Cons a (List g a)

foldl :: (b ~> a ~> b) ~> b ~> List ? a ~> b
foldl f z Nil = z
foldl f z (Cons x xs) = foldl f (f z x) xs
Code Reuse

data List (g :: Levity) (a :: TYPE Ptr v) :: TYPE Ptr (Eval g)
  = Nil | Cons a (List g a)

foldl :: (b ~> a ~> b) ~> b ~> List ? a ~> b
foldl f z Nil = z
foldl f z (Cons x xs) = foldl f (f z x) xs

foldl :: forall (v :: Conv) (g :: Levity) (a :: TYPE Ptr v) (b :: *).
  (b ~> a ~> b) ~> b ~> List g a ~> b
Code Reuse

data List (g :: Levity) (a :: TYPE Ptr v) :: TYPE Ptr (Eval g)
  = Nil | Cons a (List g a)

foldl :: (b ~> a ~> b) ~> b ~> List ? a ~> b
foldl f z Nil = z
foldl f z (Cons x xs) = foldl f (f z x) xs

foldl :: forall (v :: Conv) (g :: Levity) (a :: TYPE Ptr v) (b :: ★).
  (b ~> a ~> b) ~> b ~> List g a ~> b

foldl' f z Nil = z
foldl' f z (Cons x xs) = let !z' = f z x in foldl' f z' xs
Code Reuse

data List (g :: Levity) (a :: TYPE Ptr v) :: TYPE Ptr (Eval g)
  = Nil | Cons a (List g a)

foldl :: (b ~> a ~> b) ~> b ~> List ? a ~> b
foldl f z Nil = z
foldl f z (Cons x xs) = foldl f (f z x) xs

foldl :: forall (v :: Conv) (g :: Levity)
  (a :: TYPE Ptr v) (b :: ★).
  (b ~> a ~> b) ~> b ~> List g a ~> b

foldl' f z Nil = z
foldl' f z (Cons x xs) = let !z' = f z x in foldl' f z' xs

foldl’ :: forall (v :: Conv) (g, g’ :: Levity)
  (a :: TYPE Ptr v) (b :: TYPE Ptr (Eval g’)).
  (b ~> a ~> b) ~> b ~> List g a ~> b

Between Eager and Lazy Programs
Compilation
If it type checks, it can be compiled.
Static Compilation

To the Machine
Static Compilation

• Only basic types (pointer, integer, float); no polymorphism
Static Compilation

• Only basic types (pointer, integer, float); no polymorphism
• Only fully saturated functions and calls
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```haskell
poly :: forall a::TYPE Ptr Call[2]. (Int~>Int~>a) ~> (a,a)
poly f = let g :: Int ~> a = f 3
          in (g 4, g 5)
```
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```haskell
poly :: forall a :: TYPE Ptr Call[2]. (Int -> Int -> a) -> (a, a)
poly f = let g :: Int -> a = f 3
        in (g 4, g 5)

To the Machine
```
Static Compilation

• Only basic types (pointer, integer, float); no polymorphism
• Only fully saturated functions and calls

poly :: forall a::TYPE Ptr Call[2]. (Int~>Int~>a) ~> (a,a)

poly f = let g :: Int ~> a = f 3
          in (g 4, g 5)

poly = \(f::Ptr) ~>
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

\[
poly :: \forall a::\text{TYPE} \ 	ext{Ptr} \ 	ext{Call}[2]. \ (\text{Int} \to \text{Int} \to a) \to (a,a)
\]
\[
poly f = \text{let } g :: \text{Int} \to a = f 3
\]
\[
\text{in } (g 4, g 5)
\]
\[
poly = \lambda (f::\text{Ptr}) \to
\]
\[
\text{let } g::\text{Ptr} = \lambda (x::\text{Ptr}, y::?, z::?) \to f(3, x, y, z)
\]
Static Compilation

With Polymorphic $\eta$-Expansion
Static Compilation

With Polymorphic η-Expansion

poly :: forall a::TYPE Ptr Call[Ptr,Flt].
   (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3
         in (g 4, g 5)
Static Compilation

poly :: forall a::TYPE Ptr Call[Ptr,Flt].
     (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3
           in (g 4, g 5)

With Polymorphic η-Expansion
Static Compilation

With Polymorphic η-Expansion

definitions:

poly :: forall a::TYPE Ptr Call[Ptr,Flt].
       (Int ~> Int ~> a) ~> (a, a)

poly f = let g :: Int ~> a = f 3
          in (g 4, g 5)

poly = \(f::Ptr) ~>
Static Compilation

poly :: forall a::TYPE Ptr Call[Ptr,Flt].

        (Int ~> Int ~> a) ~> (a, a)

poly f = let g :: Int ~> a = f 3

         in (g 4, g 5)

poly = \(f::Ptr) ~> 

        let g::Ptr = \(x::Ptr, y::Ptr, z::Flt) ~> f(3,x,y,z)

With Polymorphic η-Expansion
poly :: forall a::TYPE Ptr Call[Ptr,Flt].
       (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3
       in (g 4, g 5)

poly = \(f::Ptr) ~>
     let g::Ptr = \(x::Ptr, y::Ptr, z::Flt) ~> f(3,x,y,z)
     in (\(y::Ptr, z::Flt) ~> g(4, y, z),
         \(y::Ptr, z::Flt) ~> g(5, y, z))
Lessons Learned

• Efficient performance requires good semantics
• Good semantics comes from logic
• Kinds capture efficient calling conventions
New Goal: a foundation for functional systems programming?