Kinds Are Calling Conventions:
Intensional Static Polymorphism

Paul Downen

POPV, November 14, 2020

Theory and Practice

Of Programming Languages

Theory and Practice

Of Programming Languages

- **Goal:** Performance

Theory and Practice

Of Programming Languages

- **Goal:** Performance
- **Subgoal:** Semantics

Theory and Practice

Of Programming Languages

- **Goal:** Performance
- **Subgoal:** Semantics
- **Answer:** Logic

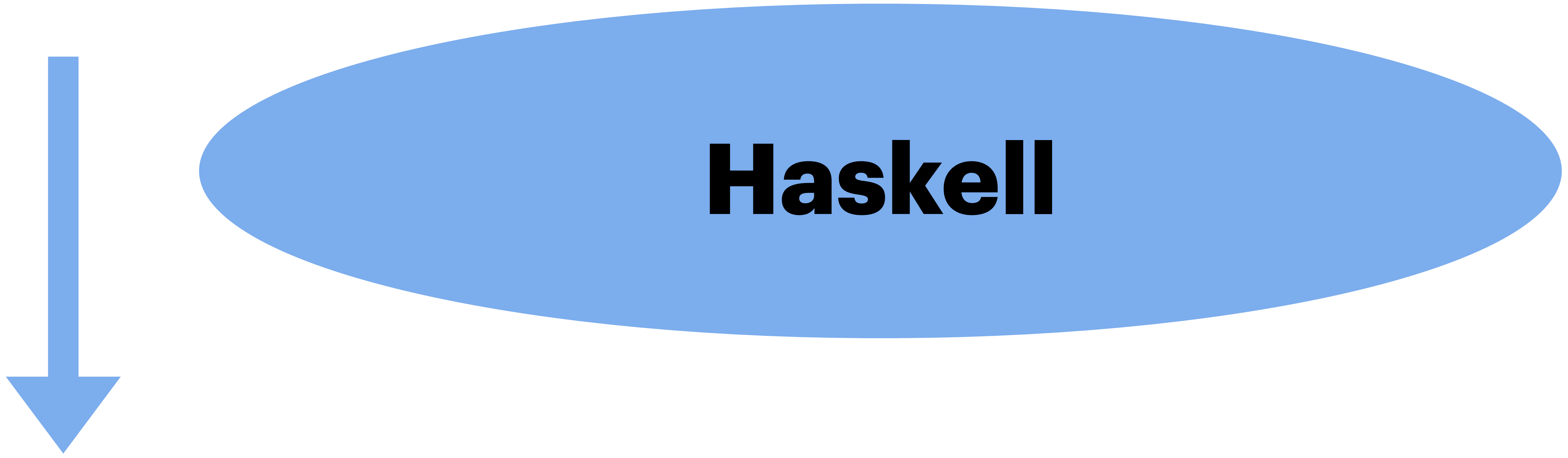
Compilation Funnel

Source → Intermediate → Target

Compilation Funnel

Source → Intermediate → Target

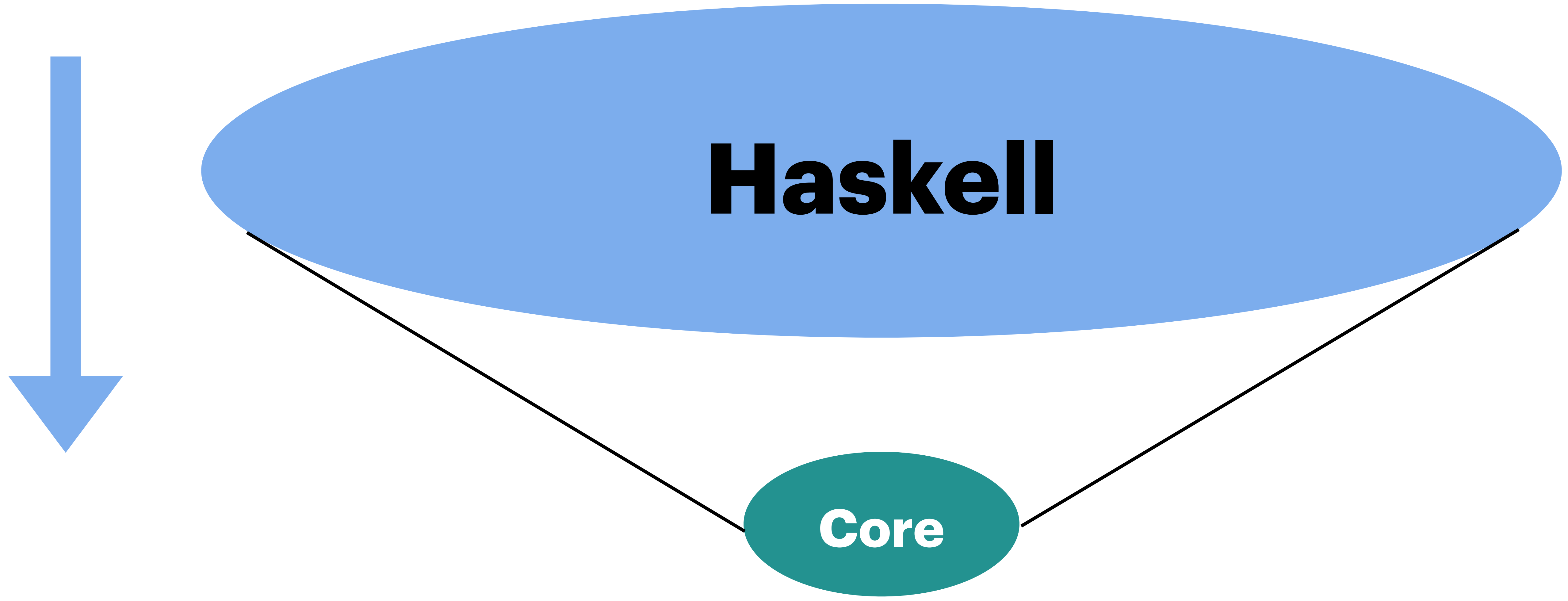
Desugaring



Compilation Funnel

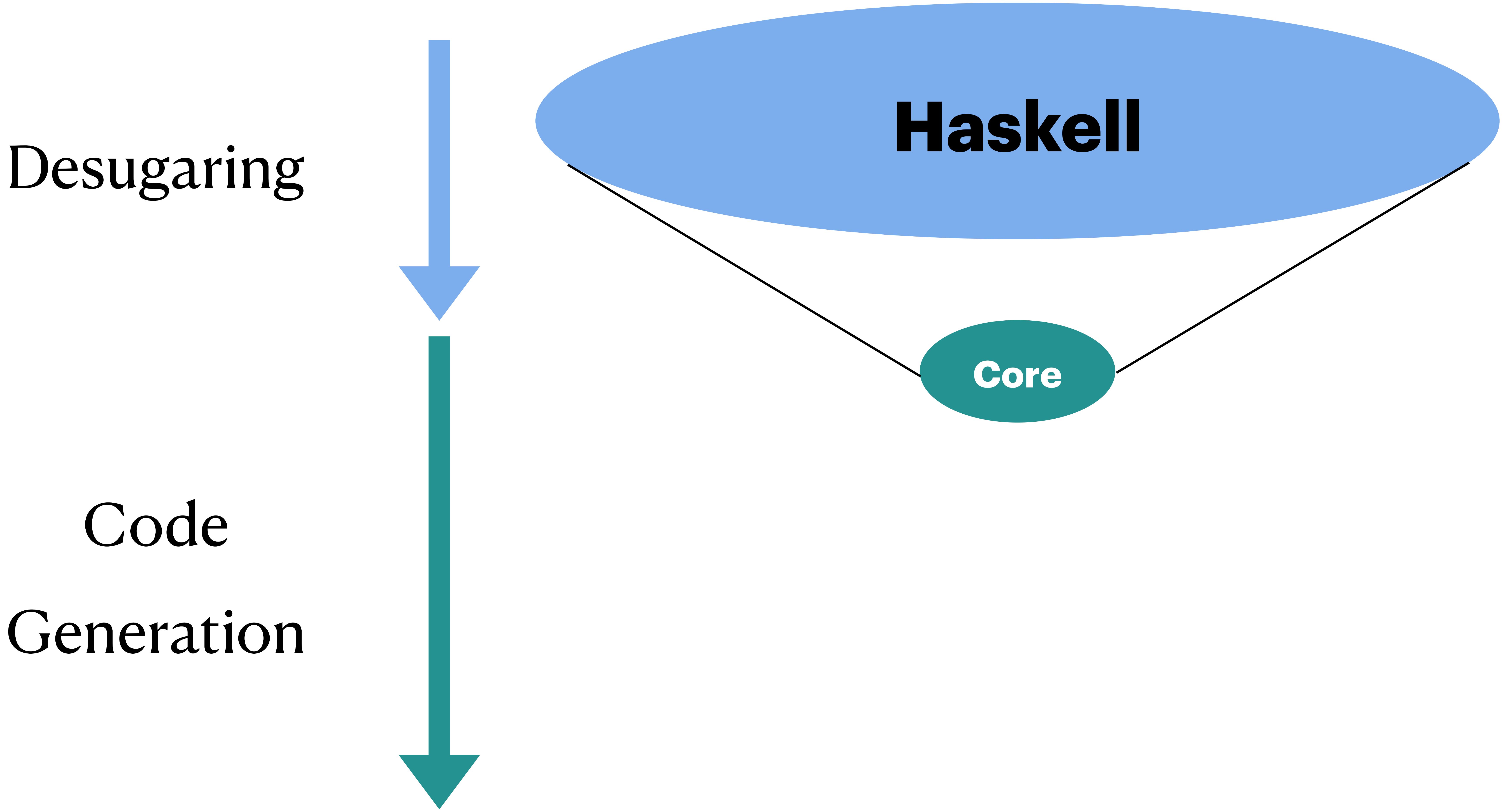
Source → Intermediate → Target

Desugaring



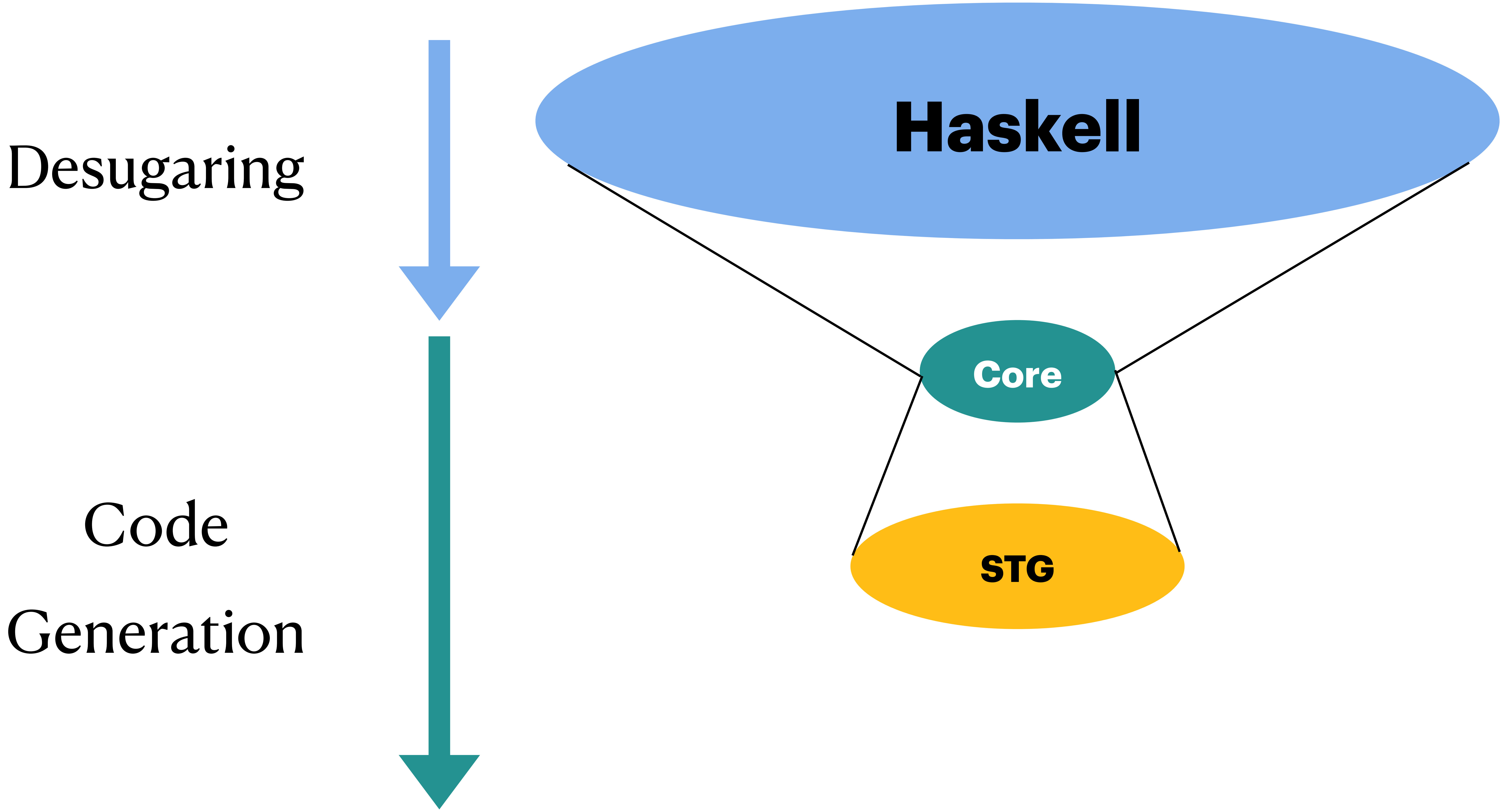
Compilation Funnel

Source → Intermediate → Target

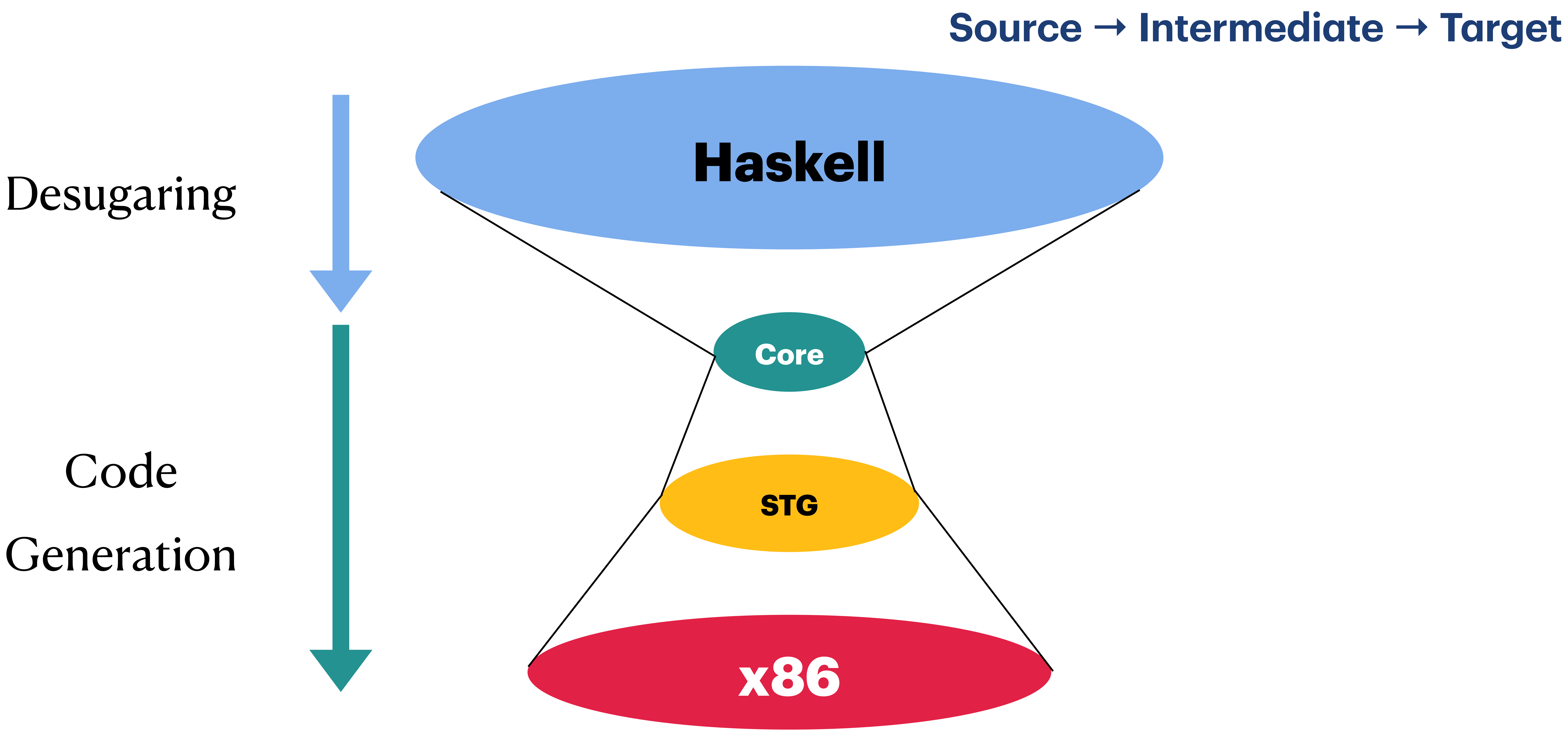


Compilation Funnel

Source → Intermediate → Target

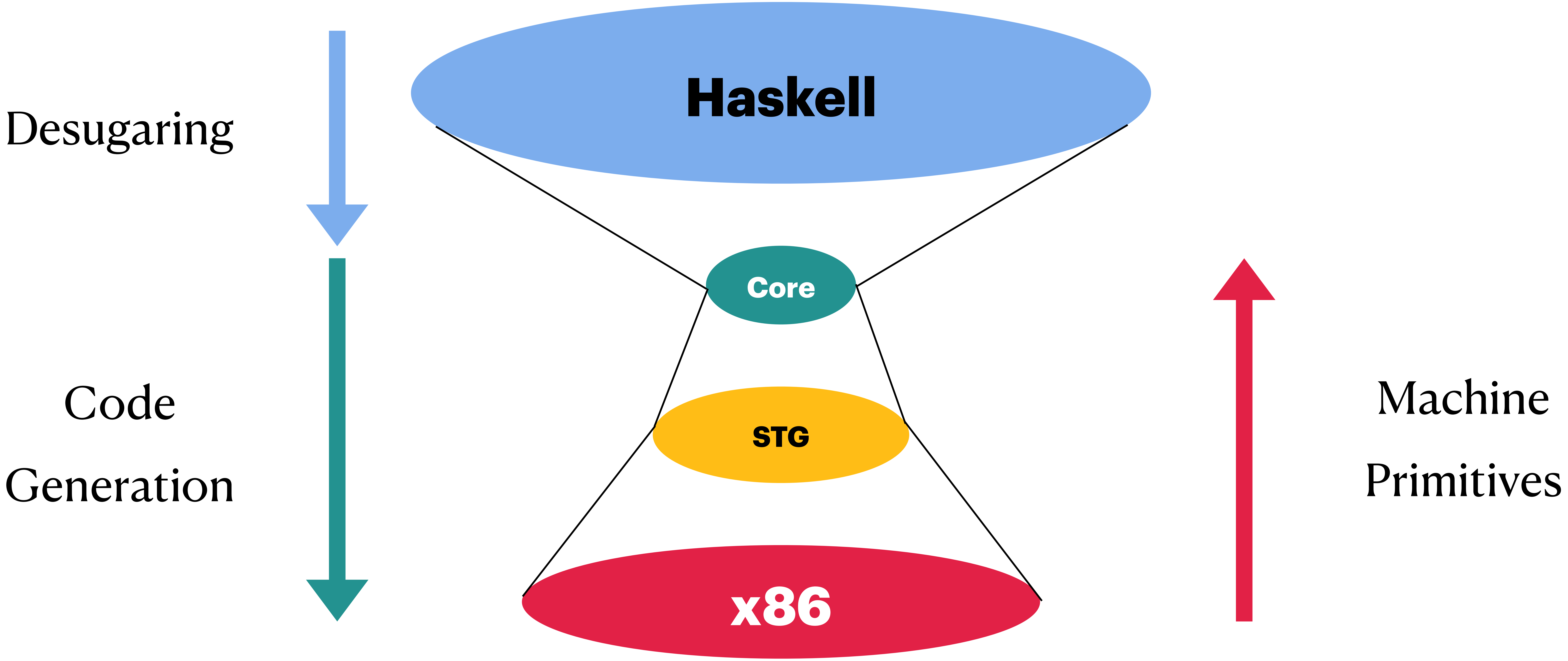


Compilation Funnel



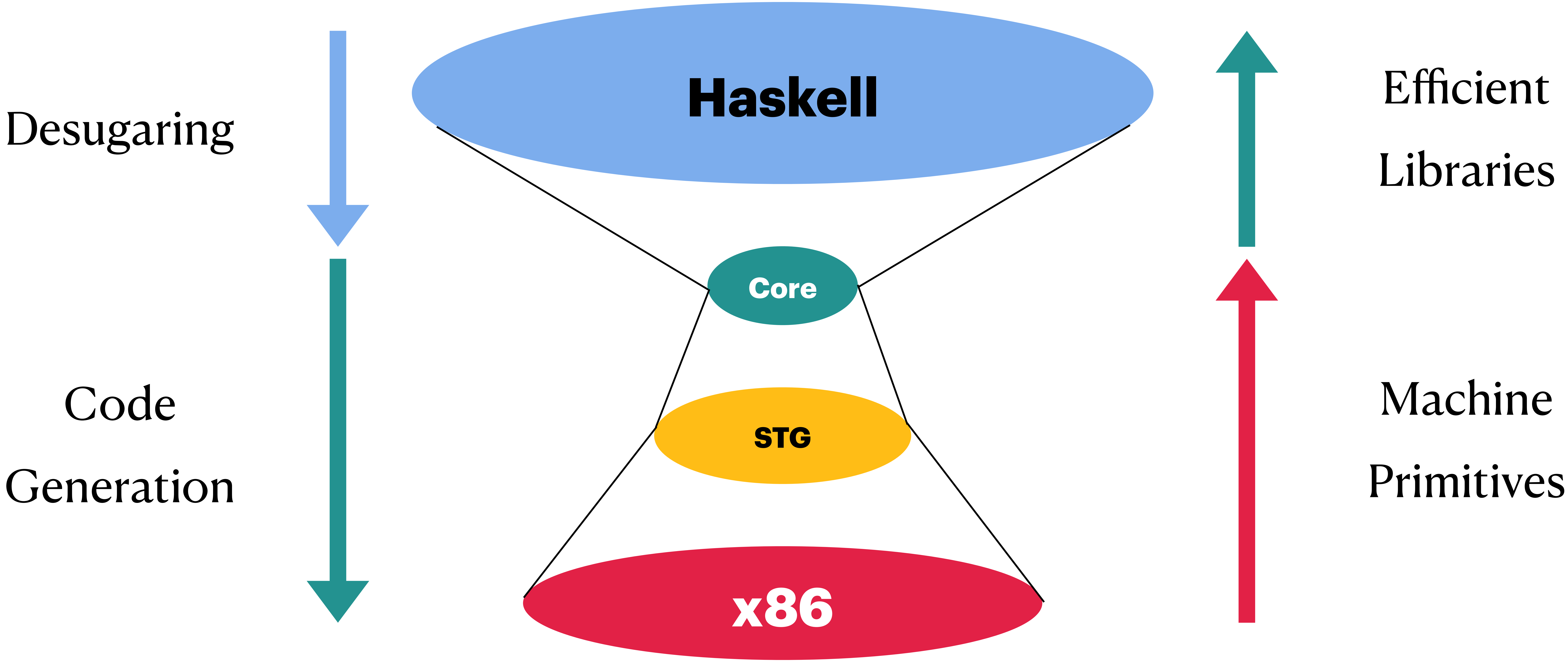
Compilation Funnel

Source → Intermediate → Target



Compilation Funnel

Source → Intermediate → Target



System F

Workhorse of Functional Compilers

System F

Workhorse of Functional Compilers

Core

System F

Workhorse of Functional Compilers

Core = System F

System F

Workhorse of Functional Compilers

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

System F

Workhorse of Functional Compilers

$$\begin{aligned} \text{Core} &= \text{System F} \\ &+ \text{Data Types} \end{aligned}$$

(first-class functions, polymorphism)

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	(first-class functions, polymorphism)
	+	Data Types	(Primitives, lists/trees, records)
	+	Type Equality	(GADTs, type families, coercions)

System F

Workhorse of Functional Compilers

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

GHC Core*

*In Greek

GHC Core*

*In Greek

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

λ-calculus: variables, functions, application

GHC Core*

*In Greek

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

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

λ-calculus: variables, functions, application

GHC Core*

*In Greek

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

$$\begin{aligned} Expr \ni d, e, f ::= {} & x \mid \lambda x{:}\tau\,.\,e \mid f\ e \\ & \mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau \end{aligned}$$

λ-calculus: variables, functions, application

System F: polymorphism & instantiation

GHC Core*

*In Greek

$$
\begin{aligned}
\mathit{Type} \ni \tau, \sigma &::= \ldots \\
\mathit{Kind} \ni \kappa \quad &= \mathit{Type} \\
\mathit{Expr} \ni d, e, f &::= x \mid \lambda x{:}\tau \,.\, e \mid f\ e \\
&\mid \Lambda a{:}\kappa \,.\, e \mid e\ \tau
\end{aligned}
$$

λ-calculus: variables, functions, application

System F: polymorphism & instantiation

GHC Core*

*In Greek

$$
\begin{aligned}
\mathit{Type} \ni \tau, \sigma &::= \ldots \\
\mathit{Kind} \ni \kappa \quad &= \mathit{Type} \\
\mathit{Expr} \ni d, e, f &::= x \mid \lambda x{:}\tau\,.\,e \mid f\ e \\
&\mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau \\
&\mid l \mid \mathtt{let}\ x{:}\tau = d\ \mathit{in}\ e
\end{aligned}
$$

λ-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

GHC Core*

*In Greek

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

$$Kind \ni \kappa \quad = Type$$

$Expr \ni d, e, f ::= x \mid \lambda x{:}\tau\,.\,e \mid f\ e$	λ-calculus: variables, functions, application
$\mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau$	System F: polymorphism & instantiation
$\mid l \mid \texttt{let}\ x{:}\tau = d\ in\ e$	Literal primitives & let-bindings
$\mid \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\}$	Data contructor & literal matching

GHC Core*

*In Greek

$$\mathit{Type} \ni \tau, \sigma ::= \ldots \qquad \mathit{Pattern} \ni \pi ::= x \mid l \mid K\ x \ldots$$

$$\mathit{Kind} \ni \kappa \qquad = \mathit{Type}$$

$$\mathit{Expr} \ni d, e, f ::= x \mid \lambda x{:}\tau\,.\,e \mid f\ e$$ λ-calculus: variables, functions, application

$$\mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau$$ System F: polymorphism & instantiation

$$\mid l \mid \texttt{let}\ x{:}\tau = d\ \mathit{in}\ e$$ Literal primitives & let-bindings

$$\mid \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\}$$ Data contructor & literal matching

GHC Core*

*In Greek

$$Type \ni \tau, \sigma ::= \ldots \qquad Pattern \ni \pi ::= x \mid l \mid K\ x\ldots$$

$$Kind \ni \kappa \quad = Type$$

$$\begin{aligned} Expr \ni d, e, f ::= {} & x \mid \lambda x{:}\tau\,.\,e \mid f\ e \\ \mid {} & \Lambda a{:}\kappa\,.\,e \mid e\ \tau \\ \mid {} & l \mid \texttt{let}\ x{:}\tau = d\ in\ e \\ \mid {} & \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\} \\ \mid {} & \chi \mid e \rhd \chi \end{aligned}$$

λ-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

Data contructor & literal matching

Coercion evidence & casting

GHC Core*

*In Greek

$$Type \ni \tau, \sigma ::= \ldots \qquad Pattern \ni \pi ::= x \mid l \mid K\ x\ldots$$

$$Kind \ni \kappa \quad = Type \qquad Coercion \ni \chi ::= \texttt{refl} \mid \chi^{-1} \mid \chi \circ \chi' \mid \ldots$$

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

λ-calculus: variables, functions, application

$$\mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau$$

System F: polymorphism & instantiation

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

Literal primitives & let-bindings

$$\mid \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\}$$

Data contructor & literal matching

$$\mid \chi \mid e \rhd \chi$$

Coercion evidence & casting

GHC Core*

*In Greek

$$Type \ni \tau, \sigma ::= \ldots \qquad Pattern \ni \pi ::= x \mid l \mid K\ x\ldots$$

$$Kind \ni \kappa \quad = Type \qquad Coercion \ni \chi ::= \texttt{refl} \mid \chi^{-1} \mid \chi \circ \chi' \mid \ldots$$

$Expr \ni d, e, f ::= x \mid \lambda x{:}\tau\,.\,e \mid f\ e$	λ-calculus: variables, functions, application
$\mid \Lambda a{:}\kappa\,.\,e \mid e\ \tau$	System F: polymorphism & instantiation
$\mid l \mid \texttt{let}\ x{:}\tau = d\ in\ e$	Literal primitives & let-bindings
$\mid \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\}$	Data contructor & literal matching
$\mid \chi \mid e \rhd \chi$	Coercion evidence & casting
$\mid \texttt{tick}\ tk\ e$	Profiling & instrumentation

GHC Core*

*In Greek

$$Type \ni \tau, \sigma ::= \ldots \qquad Pattern \ni \pi ::= x \mid l \mid K\ x \ldots$$

$$Kind \ni \kappa \quad = Type \qquad Coercion \ni \chi ::= \texttt{refl} \mid \chi^{-1} \mid \chi \circ \chi' \mid \ldots$$

$$\begin{aligned} Expr \ni d, e, f ::= {} & x \mid \lambda x{:}\tau\,.\,e \mid f\ e \\ \mid {} & \Lambda a{:}\kappa\,.\,e \mid e\ \tau \\ \mid {} & l \mid \texttt{let}\ x{:}\tau = d\ in\ e \\ \mid {} & \texttt{case}\ d\ \texttt{of}\ \{\pi \to e; \ldots\} \\ \mid {} & \chi \mid e \triangleright \chi \\ \mid {} & \texttt{tick}\ tk\ e \end{aligned}$$

λ-calculus: variables, functions, application

System F: polymorphism & instantiation

Literal primitives & let-bindings

Data contructor & literal matching

Coercion evidence & casting

Profiling & instrumentation

A real-world programming language in only 6 lines!

Compiling Polymorphism

Statically

Compiling Polymorphism

Statically

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

Compiling Polymorphism

Statically

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

Compiled assembly code:

Compiling Polymorphism

Statically

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

Compiled assembly code:

1. Accept parameters

Compiling Polymorphism

Statically

```
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

Statically

```
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

Statically

```
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

Compiling Polymorphism

Statically

```
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

Compiling Polymorphism

Statically

```
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

Statically

```
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

Statically

```
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

Statically

```
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?

Compiling Polymorphism

Statically

```
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

In Systems Programming Languages

- Calls have statically known parameter #s

Calling Conventions

In Systems Programming Languages

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

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

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

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

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

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 statically at compile time

Efficient Function Calls

Parameter Passing Techniques

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?

Efficient Function Calls

Parameter Passing Techniques

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

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
  - Shape of data values
- Arity — How many arguments?

Efficient Function Calls

Parameter Passing Techniques

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

Efficient Function Calls

Parameter Passing Techniques

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

Efficient Function Calls

Parameter Passing Techniques

- 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

Parameter Passing Techniques

- 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

- S. Peyton Jones and J. Launchbury. 1991. Unboxed Values As First Class Citizens in a Non-Strict Functional Language.
  - 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.
- R.A. Eisenberg and S. Peyton Jones. 2017. Levity polymorphism.
  - Explicit polymorphic 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.
- R.A. Eisenberg and S. Peyton Jones. 2017. Levity polymorphism.
  - Explicit polymorphic representations; implicit levities.
- P. Downen, Z. Sullivan, Z.M. Ariola, and S. Peyton Jones. 2019. Making a Faster Curry with Extensional Types.
  - Explicit monomorphic arities; 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.
- R.A. Eisenberg and S. Peyton Jones. 2017. Levity polymorphism.
  - Explicit polymorphic representations; implicit levities.
- P. Downen, Z. Sullivan, Z.M. Ariola, and S. Peyton Jones. 2019. Making a Faster Curry with Extensional Types.
  - Explicit monomorphic arities; implicit levities.
- P. Downen, Z.M. Ariola, S. Peyton Jones, and R.A. Eisenberg. 2020. Kinds Are Calling Conventions.
  - Explicit polymorphic representations, arities, and levities.

Representation

Unboxed Types

And Their Representation

Unboxed Types

And Their Representation

- Primitive types:

Unboxed Types

And Their Representation

- Primitive types:
  - `Int#`, `Float#`, `Char#`, `Word16#`, `Array#`…

Unboxed Types

And Their Representation

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

Unboxed Types

And Their Representation

- Primitive types:
  - `Int#`, `Float#`, `Char#`, `Word16#`, `Array#`...
- Unboxed (`Int#`, `Float#`...) or Boxed (`Array#`)
- **Pro:** Efficient memory

Unboxed Types

And Their Representation

- Primitive types:
  - `Int#`, `Float#`, `Char#`, `Word16#`, `Array#`…
- Unboxed (`Int#`, `Float#`…) or Boxed (`Array#`)
- **Pro**: Efficient memory
- **Pro**: Efficient passing

Unboxed Types

And Their Representation

- Primitive types:
  - `Int#`, `Float#`, `Char#`, `Word16#`, `Array#`…
- Unboxed (`Int#`, `Float#`…) or Boxed (`Array#`)
- **Pro:** Efficient memory
- **Pro:** Efficient passing
- **Con:** Different sizes

Unboxed Types

And Their Representation

- 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

S.L. Peyton Jones and J. Launchbury. 1991.

And Their Representation

- 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 `forall a::k. …`

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 *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 #

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 :: ⋆` to `a :: TYPE r`

Representation Polymorphism

Kinds As Representations

- Generalize a `::` ⋆ to a `:: TYPE r`
  - `r :: Rep` is the *representation* of a

Representation Polymorphism

Kinds As Representations

- Generalize `a :: ★` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `★ = TYPE Ptr`

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ⋆` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `⋆ = TYPE Ptr`

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ★` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `★ = TYPE Ptr`

```
error :: forall (a :: ★). String -> a
```

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ⋆` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `⋆ = TYPE Ptr`

```
error :: forall (a :: ⋆). String -> a
errorInt# :: String -> Int#
```

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ⋆` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `⋆ = TYPE Ptr`

```
error :: forall (a :: ⋆). String -> a
errorInt# :: String -> Int#
errorFloat# :: String -> Float#
```

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ★` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `★ = TYPE Ptr`

```
error :: forall (a :: ★). String -> a
errorInt# :: String -> Int#
errorFloat# :: String -> Float#
...
```

Representation Polymorphism

R.A. Eisenberg and S. Peyton Jones. 2017.

Kinds As Representations

- Generalize `a :: ⋆` to `a :: TYPE r`
  - `r :: Rep` is the *representation* of a
  - `⋆ = TYPE Ptr`

```
error :: forall (a :: ⋆). String -> a
errorInt# :: String -> Int#
errorFloat# :: String -> Float#
...
error :: forall (r::Rep) (a :: TYPE r). String -> 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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

Efficient Code Abstraction

For Numeric Operations

```
class Num (a              ) where
  (+) :: a -> a -> a
  …
```

Efficient Code Abstraction

For Numeric Operations

```
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
  …
```

Efficient Code Abstraction

For Numeric Operations

```
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) …
```

Efficient Code Abstraction

For Numeric Operations

```
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# …
```

Efficient Code Abstraction

For Numeric Operations

```
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

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

Determining Function Arity

Type suggests arity 2

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

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

Determining Function Arity

Type suggests arity 2

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

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

Arity 2

Determining Function Arity

Type suggests arity 2

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

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

Arity 2

```
f2 = \x -> f1 x
```

Determining Function Arity

Type suggests arity 2

```
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
```

Determining Function Arity

Type suggests arity 2

```
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
```

Arity 2

Determining Function Arity

Type suggests arity 2

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

Arity 2

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

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

Arity 2

```
f2 = \x -> f1 x

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

Determining Function Arity

Type suggests arity 2

```
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
```

Arity 2

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

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

Type suggests arity 2

```
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
```

Arity 2

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

Arity 1

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

Type suggests arity 2

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

Arity 2

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

Arity 1

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

Arity 2

```
f2 = \x -> f1 x
   = \x -> \y -> f1 x y
```

```
f4 = \x -> f3 x
```

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

Type suggests arity 2

```
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
```

Arity 2

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

Arity 1

```
f4 = \x -> f3 x
   ≠ \x -> \y -> f3 x y
```

Hint: ‘`expensive x`’ may be costly, or even cause side effects

Determining Function Arity

Type suggests arity 2

```
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
```

Arity 2

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

Arity 1

```
f4 = \x -> f3 x
   ≠ \x -> \y -> f3 x y
```

Arity 1

Hint: '`expensive x`' may be costly, or even cause side effects

What Is Arity?

For Curried Functions

What Is Arity?

For Curried Functions

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?

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

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

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

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?

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

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

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

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

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

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

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

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

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*** η 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 $a \to b$ with different semantics

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘`a ~> b`’)
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like 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 $a \rightsquigarrow b$ type of primitive functions (ASCII '`a ~> b`')
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like source functions
  - $\lambda x\,.f\,x =_\eta f : a \rightsquigarrow b$ *unconditionally*
  - `error "not a function" /= \x -> (error "not a function") x` in Haskell
- With full η, types express arity — just count the arrows
  - $f : \mathit{Int} \rightsquigarrow \mathit{Bool} \rightsquigarrow \mathit{String}$ has arity 2, no matter f's definition

Static Arity

P. Downen, Z. Sullivan, Z.M. Ariola, and S. Peyton Jones. 2019.

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII '`a ~> b`')
  - To distinguish from the source-level $a \to b$ with different semantics
- Primitive functions are *fully extensional*, **un**like source functions
  - $\lambda x\,.f\;x =_\eta f : a \rightsquigarrow b$ *unconditionally*
  - `error "not a function" /= \x -> (error "not a function") x` in Haskell
- With full η, types express arity — just count the arrows
  - $f : \mathit{Int} \rightsquigarrow \mathit{Bool} \rightsquigarrow \mathit{String}$ 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

```
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

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

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...

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...

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 `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 (~)

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 `forall 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. …`
  - 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 `[] :: ⋆ -> ⋆`)

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`

Arity Polymorphism

- Generalize `a::TYPE r` to `a::TYPE r v`
  - `v::Conv` is the *calling convention* of `a`

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)

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

$\otimes$

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

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

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

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

Arity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

Kinds As Calling Conventions

- 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)

```
revapp x f = f x

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

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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

And Higher-Order Functions

```
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

Arity Polymorphism

And Higher-Order Functions

```
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

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

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:

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 = \x y z -> … in …)` defines code for f
  - `(\x y -> 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
```

Primitive Functions are First-Class Values

Arity-Polymorphic Data Types

```
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
```

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)
```

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                                            ) where
  fmap :: (a -> b) -> f a -> f b
```

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
```

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)
```

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
```

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))
```

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 η Is Inconsistent With Restricted β

In the λ-calculus

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

Unrestricted η Is Inconsistent With Restricted β

In the λ-calculus

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

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

Unrestricted η Is Inconsistent With Restricted β

In the λ-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 η Is Inconsistent With Restricted β

In the λ-calculus

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

$$\lambda x\,.\perp x =_\eta \perp$$

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

$$\beta_v$$

Unrestricted η Is Inconsistent With Restricted β

In the λ-calculus

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

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

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

$$\downarrow \beta_v$$

$$5$$

Unrestricted η Is Inconsistent With Restricted β

In the λ-calculus

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

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

$$\begin{tikzcd}
(\lambda z\,.\,5)\;(\lambda x\,.\,\bot\;x) \arrow[d, "\beta_v"] & =_\eta & (\lambda z\,.\,5)\;\bot \arrow[d, "\beta_v"] \\
5 & & {}
\end{tikzcd}$$

Unrestricted η Is Inconsistent With Restricted β

In the λ-calculus

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

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

$$\begin{tikzcd}
(\lambda z\,.\,5)\;(\lambda x\,.\,\bot\;x) \arrow[d, "\beta_v"] \arrow[r, phantom, "=_\eta"] & (\lambda z\,.\,5)\;\bot \arrow[d, "\beta_v"] \\
5 & \bot
\end{tikzcd}$$

Unrestricted η Is Inconsistent With Restricted β

In the λ-calculus

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

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

$$\begin{tikzcd} (\lambda z\,.\,5)\ (\lambda x\,.\,\bot\ x) \arrow[d, "\beta_v"] & =_\eta & (\lambda z\,.\,5)\ \bot \arrow[d, "\beta_v"] \\ 5 & \neq & \bot \end{tikzcd}$$

Goal: A core language with ***un**restricted η* for functions and *restricted β* for other types

Unboxed Data Is *Eager*

Not *Lazy*

Unboxed Data Is *Eager*

Not *Lazy*

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

Unboxed Data Is *Eager*

Not *Lazy*

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

- Compiles to machine primop for float addition in specialized registers

Unboxed Data Is *Eager*

Not *Lazy*

```
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*

Not *Lazy*

```
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*

Not *Lazy*

`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*

Not *Lazy*

```
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*

Not *Lazy*

```
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

Unboxed Data Is *Eager*

Not *Lazy*

`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*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

Primitive Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?

Primitive Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`

Primitive Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- When is `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*

Not *Evaluated*

`x = let f :: Int ~> Int = expensive 100 in …f…f…`

- 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

Primitive Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- 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' = let f :: Int ~> Int = \y ~> expensive 100 y in …f…f…
```

Primitive Functions are *Called*

Not *Evaluated*

`x = let f :: Int ~> Int = expensive 100 in …f…f…`

- 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’ = let f :: Int ~> Int = \y ~> expensive 100 y in …f…f…`

- `x = x’` by η, so they must be the same

Primitive Functions are *Called*

Not *Evaluated*

```
x = let f :: Int ~> Int = expensive 100 in …f…f…
```

- 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’ = let f :: Int ~> Int = \y ~> expensive 100 y in …f…f…
```

- `x = x’` by η, so they must be the same
- `x’` always follows call-by-name order! So `x` does, too

Primitive Functions are *Called*

Not *Evaluated*

`x = let f :: Int ~> Int = expensive 100 in …f…f…`

- 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’ = let f :: Int ~> Int = \y ~> expensive 100 y in …f…f…`

- `x = x’` by η, 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

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

- Because of η, f3 now has arity 2, not 1!

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes ‘expensive 100’ a million times ☹

```
f3’ :: Int ~> { Int ~> Int }
f3’ = \x ~> let z = expensive x in Clos (\y ~> y + z)
```

```
Clos :: (Int ~> Int) ~> {Int ~> Int}
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

```
f3' :: Int ~> { Int ~> Int }
f3' = \x ~> let z = expensive x in Clos (\y ~> y + z)
```

- `f3'` is an arity 1 function; returns a closure `{Int~>Int}` of an arity 1 function

```
Clos :: (Int ~> Int) ~> {Int ~> Int}
```

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
f3 = \x ~> let z = expensive x in \y ~> y + z
```

- Because of η, f3 now has arity 2, not 1!
  - map (f3 100) [1..10^6] recomputes 'expensive 100' a million times ☹

```
f3' :: Int ~> { Int ~> Int }
f3' = \x ~> let z = expensive x in Clos (\y ~> y + z)
```

- f3' is an arity 1 function; returns a closure {Int~>Int} of an arity 1 function
  - map (App (f3' 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

Denotationally and Logically

- $A_\perp$ is the *lifted* version of A

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ is the *lifted* version of A
  - $A_\perp$ adds a special, unique value $\perp$ to A denoting divergent computation

Levity and Evaluation Strategy

Denotationally and Logically

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

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\ \perp\ \} \cup \{\lambda x\,.\,f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ is the *lifted* version of A
  - $A_\perp$ adds a special, unique value $\perp$ to A denoting divergent computation
  - E.g., $\mathbb{N}_\perp = \{\perp, 0, 1, 2, 3, \dots\}$ 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 *unlifted*
  - $\texttt{Int\#} = \{0, 1, -1, 2, -2, \dots\}$ and $A \rightsquigarrow B = \{\lambda x . f(x) \mid f \in A \to B\}$ denotes only real functions

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\perp\} \cup \{\lambda x\,.\,f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - $\texttt{Int\#} = \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x\,.\,f(x) \mid f \in A \to B\}$ denotes only real functions
  - Lifting implies worse performance (for data, functions)

Levity and Evaluation Strategy

Denotationally and Logically

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

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{ \perp \} \cup \{\lambda x \,.\, f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - `Int#` $= \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x \,.\, f(x) \mid f \in A \to B\}$ 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:

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\perp\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - `Int#` $= \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x . f(x) \mid f \in A \to B\}$ 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: $Int_\perp \to Int_\perp \to Int_\perp$

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\ \perp\ \} \cup \{\lambda x\,.\,f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - `Int#` $= \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x\,.\,f(x) \mid f \in A \to B\}$ 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: $Int_\perp \to Int_\perp \to Int_\perp$
  - Call-by-value: $(Int \to (Int \to Int_\perp)_\perp)_\perp$

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\perp\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - `Int#` $= \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x . f(x) \mid f \in A \to B\}$ 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: $Int_\perp \to Int_\perp \to Int_\perp$
  - Call-by-value: $(Int \to (Int \to Int_\perp)_\perp)_\perp$
  - Call-by-push-value: $Int \to Int \to Int_\perp$

Levity and Evaluation Strategy

Denotationally and Logically

- $A_\perp$ 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 $(A \to B)_\perp = \{\perp\} \cup \{\lambda x . f(x) \mid f \in A \to B\}$
- Unboxed types and primitive functions are *unlifted*
  - $\texttt{Int\#} = \{0, 1, -1, 2, -2, \ldots\}$ and $A \rightsquigarrow B = \{\lambda x . f(x) \mid f \in A \to B\}$ 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: $Int_\perp \to Int_\perp \to Int_\perp$
  - Call-by-value: $(Int \to (Int \to Int_\perp)_\perp)_\perp$
  - Call-by-push-value: $Int \to Int \to Int_\perp$
- Logical polarity reveals the semantics for best performance

Levity Polymorphism

Call vs Eval, Revisited

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn’t called is evaluated

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't called is evaluated
  - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values

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

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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
```

Levity Polymorphism

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. 2020.

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

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 `lets` 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 (                  :: TYPE Ptr v) ::
  = Nil | Cons a (List g
```

Code Reuse

Between Eager and Lazy Programs

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

Code Reuse

Between Eager and Lazy Programs

```
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

Between Eager and Lazy Programs

```
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

Between Eager and Lazy Programs

```
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

Between Eager and Lazy Programs

```
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
```

Compilation

If it type checks, it can be compiled.

P. Downen, Z.M. Ariola, S. Peyton Jones, R.A. Eisenberg. ICFP 2020.

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

To the Machine

- 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)
```

Static Compilation

To the Machine

- 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)
```

Static Compilation

To the Machine

- 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

To the Machine

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

Static Compilation

With Polymorphic η-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

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

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) ~>
```

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)
```

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

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)
```

```
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?