Kinds Are Calling Conventions

Paul Downen, Zena M. Ariola, Simon Peyton Jones, Richard A. Eisenberg
Efficient Function Calls

Parameter Passing Techniques
Efficient Function Calls

- Representation — What & Where?
Efficient Function Calls

- Representation — What & Where?
- Arity — How many?
Efficient Function Calls

- Representation — What & Where?
- Arity — How many?
- Levity (aka Evaluation Strategy) — When to compute?
Efficient Function Calls

• Representation — What & Where?
• **Arity** — How many?
• Levity (aka Evaluation Strategy) — When to compute?
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]  
Type suggests arity 2
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{Type suggests arity 2} \]

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

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

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

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]  
Type suggests arity 2

\[ f_1 = \lambda x \to \lambda y \to \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity } 2 \quad f_2 = \lambda x \to f_1 x \]
\[ \quad \text{in } y + z \]
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \rightarrow f_1 x \]
\[ \quad \text{in } y + z \quad = \lambda x \rightarrow \lambda y \rightarrow f_1 x \ y \]
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \rightarrow f_1 x \quad \text{Arity 2} \]
\[ \text{in } y + z \quad = \lambda x \rightarrow \lambda y \rightarrow f_1 x y \]
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \to \lambda y \to \\
\quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \to f_1 x \quad \text{Arity 2} \\
\quad \text{in } y + z \quad = \lambda x \to \lambda y \to f_1 x \, y \]

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

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

Type suggests arity 2

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

f2 = \x -> f1 x

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

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

f4 = \x ->
    let z = expensive x
    in \y -> y + z

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

f1, f2, f3, f4 :: Int -> Int -> Int  
Type suggests arity 2

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 -> \y -> f1 x y

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

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \] 
Type suggests arity 2

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \text{let } z = \text{expensive } x \quad \text{Arity 2} \]
\[ \text{in } y + z \]
\[ f_2 = \lambda x \rightarrow f_1 x \quad \text{Arity 2} \]
\[ = \lambda x \rightarrow \lambda y \rightarrow f_1 x y \]

\[ f_3 = \lambda x \rightarrow \]
\[ \text{let } z = \text{expensive } x \quad \text{Arity 1} \]
\[ \text{in } \lambda y \rightarrow y + z \]
\[ f_4 = \lambda x \rightarrow f_3 x \]

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

\( f_1 = \lambda x \rightarrow \lambda y \rightarrow \) 
\hspace{2em} let \( z = \text{expensive} \ x \) \hspace{2em} \text{Arity 2} \hspace{2em} \text{f2 = } \lambda x \rightarrow f_1 \ x \hspace{2em} \text{Arity 2} \\
\hspace{2em} \text{in } y + z \hspace{2em} \hspace{2em} = \lambda x \rightarrow \lambda y \rightarrow f_1 \ x \ y \\
\hspace{2em} f_3 = \lambda x \rightarrow \) 
\hspace{2em} let \( z = \text{expensive} \ x \) \hspace{2em} \text{Arity 1} \hspace{2em} f_4 = \lambda x \rightarrow f_3 \ x \\
\hspace{2em} \text{in } \lambda y \rightarrow y + z \hspace{2em} \neq \lambda x \rightarrow \lambda y \rightarrow f_3 \ x \ y \\

f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \hspace{2em} \text{Type suggests arity 2} \\

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

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

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

\[
\quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \rightarrow f_1 x \quad \text{Arity 2}
\]

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

\[ f_3 = \lambda x \rightarrow \]

\[
\quad \text{let } z = \text{expensive } x \quad \text{Arity 1} \quad f_4 = \lambda x \rightarrow f_3 x \quad \text{Arity 1}
\]

\[
\quad \text{in } \lambda y \rightarrow y + z
\]

\[ \neq \lambda x \rightarrow \lambda y \rightarrow f_3 x y \]

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

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

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 ‘\(x y z \rightarrow f x y z\)’, then ‘f’ has arity 3

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 ‘$\lambda x \ y \ z \to f \ x \ y \ z$’, then ‘$f$’ has arity 3

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

**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?

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: An IL with *unrestricted* $\eta$ for functions, along with *restricted* $\beta$ for other types
Static Arity

In an Intermediate Language
Static Arity

• New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘$a \sim \rightarrow b$’)
  • 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
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
• Application may still be restricted for efficiency, like source functions
  • $(\lambda x. x + x) \ (\text{fact } 10^6)$ does not recompute \text{fact} $10^6$
Static Arity

In an Intermediate Language

- New $a \rightarrow b$ type of primitive functions (ASCII ‘$a \rightarrow b$’)
  - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are *fully extensional*, unlike source functions
  - $\lambda x. f x =_\eta f : a \rightarrow b$ unconditionally
- Application may still be *restricted* for efficiency, like source functions
  - $(\lambda x. x + x) \; (\text{fact} \; 10^6)$ does not recompute $\text{fact} \; 10^6$
- With full $\eta$, types express arity — just count the arrows
  - $f : \text{Int} \rightarrow \text{Bool} \rightarrow \text{String}$ has arity 2, no matter $f$’s definition
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

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

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

- Because of η, \( f_3 \) now has arity 2, not 1!
  - `map (f3 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 \( \eta \), \( f_3 \) now has arity 2, not 1!

- \( \text{map} (f_3 100) [1..10^6] \) 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

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

• 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 😞

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

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

\[ \text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\} \]
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!
  - \( \text{map} \ (f_3 \ 100) \ [1..10^6] \) 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
  - \( \text{map} \ (\text{App} \ (f_3' \ 100)) \ [1..10^6] \) computes ‘expensive 100’ only once 😊

\[ \text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\} \quad \text{App} :: \{\text{Int} \rightarrow \text{Int}\} \rightarrow \text{Int} \rightarrow \text{Int} \]
Functions are *Called*

Not *Evaluated*
Functions are *Called*

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

x = let f :: Int ~> Int = expensive 100 in ...f...f...

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

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

- **When is \text{expensive 100} evaluated?**
  - Call-by-value: first, before binding \( f \)
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

Not Evaluated
Functions are *Called*

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

- **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

Not *Evaluated*
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

\[ x' = \text{let } f :: \text{Int} \rightarrow \text{Int} = \\lambda y \rightarrow \text{expensive 100} y \text{ in } \ldots f \ldots f \ldots \]
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?
  • 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 \( \eta \), and \( x' \) always follows call-by-name order!
Functions are *Called*

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

- 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

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

- \( x = x' \) by \( \eta \), and \( x' \) always follows call-by-name order!
- Primitive functions are never just *evaluated*; they are always *called*
The Problem With Polymorphism

And Static Compilation
The Problem With Polymorphism

And Static Compilation

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

• What are the arities of f and g? Counting arrows...

```haskell
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```
The Problem With Polymorphism

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

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

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

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

And Static Compilation
The Problem With Polymorphism

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

• 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 Polymorphism

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

• 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 Polymorphism

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

• 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 Polymorphism

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

• 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 5’ a call? A partial application?
Arity Polymorphism

Kinds As Calling Conventions
Arity Polymorphism

- Generalize $a : \star$ to $a : \text{TYPE}$

Kinds As Calling Conventions
Arity Polymorphism

• Generalize $a : : \star$ to $a : : \text{TYPE} \quad r \quad c$
  • $r : : \text{Rep}$ is the \textit{runtime representation} of $a$
Arity Polymorphism

- Generalize $a::\star$ to $a::\text{TYPE}$ $r \ c$
  - $r::\text{Rep}$ is the runtime representation of $a$
  - $c::\text{Conv}$ is the calling convention of $a$
Arity Polymorphism

- Generalize $a::\star$ to $a::\text{TYPE} \ r \ c$
  - $r::\text{Rep}$ is the runtime representation of $a$
  - $c::\text{Conv}$ is the calling convention of $a$
- $a::\text{TYPE} \ Ptr \ \text{Call}[n]$ says $a$ values are pointers with arity $n$ (simplified)
Arity Polymorphism

- Generalize \( a::\star \) to \( a::\text{TYPE r c} \)
  - \( r::\text{Rep} \) is the \textit{runtime representation} of \( a \)
  - \( c::\text{Conv} \) is the \textit{calling convention} of \( a \)
  - \( a::\text{TYPE Ptr Call[n]} \) says \( a \) values are pointers with arity \( n \) (simplified)

```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)
```

Kinds As Calling Conventions
Arity Polymorphism

• Generalize \( a::\star \) to \( a::\text{TYPE } r \ c \)
  • \( r::\text{Rep} \) is the runtime representation of \( a \)
  • \( c::\text{Conv} \) is the calling convention of \( a \)
  • \( a::\text{TYPE } \text{Ptr } \text{Call}[n] \) says \( a \) values are pointers with arity \( n \) (simplified)

\[
\text{poly :: forall a::\text{TYPE } \text{Ptr } \text{Call}[2]. (Int \rightarrow Int \rightarrow a) \rightarrow (a,a)}
\]

\[
\text{poly } f = \text{let } g :: \text{Int } \rightarrow a = f 3 \text{ in } (g 4, g 5)
\]

• \( f :: \text{Int } \rightarrow \text{Int } \rightarrow a :: \text{TYPE } \text{Ptr } \text{Call}[4] \) has arity 4
Arity Polymorphism

- Generalize $a::\star$ to $a::\text{TYPE~r~c}$
  - $r::\text{Rep}$ is the runtime representation of $a$
  - $c::\text{Conv}$ is the calling convention of $a$
  - $a::\text{TYPE~Ptr~Call[n]}$ says $a$ values are pointers with arity $n$ (simplified)

$$\text{poly} :: \forall a::\text{TYPE~Ptr~Call[2]}. (\text{Int} \to \text{Int} \to a) \to (a,a)$$

$$\text{poly} f = \text{let } g :: \text{Int} \to a = f \ 3 \ \text{in } (g \ 4, g \ 5)$$

- $f :: \text{Int} \to \text{Int} \to a :: \text{TYPE~Ptr~Call[4]}$ has arity 4
- $g :: \text{Int} \to a :: \text{TYPE~PTR~Call[3]}$ has arity 3
Arity Polymorphism

- Generalize \( a::\star \) to \( a::\text{TYPE} \ r \ c \)
  - \( r::\text{Rep} \) is the *runtime representation* of \( a \)
  - \( c::\text{Conv} \) is the *calling convention* of \( a \)
  - \( a::\text{TYPE} \ \text{Ptr} \ \text{Call}[n] \) says \( a \) values are pointers with arity \( n \) (simplified)

\[
\text{poly :: forall } a::\text{TYPE} \ \text{Ptr} \ \text{Call}[2]. \ (\text{Int} \to \text{Int} \to a) \to (a,a)
\]

\[
\text{poly } f = \text{let } g :: \text{Int} \to a = f \ 3 \ \text{in } (g \ 4, g \ 5)
\]

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

\[
\text{revapp :: forall } (c::\text{Conv}) (r::\text{Rep})
\quad (a::\text{TYPE} \ \text{Ptr} \ c) (b::\text{TYPE} \ r \ \text{Call}[1]).
\quad a \to (a \to b) \to b
\]

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

- Generalize `a::★` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of `a`
  - `c::Conv` is the *calling convention* of `a`
  - `a::TYPE Ptr Call[n]` says `a` values are pointers with arity `n` (simplified)

```
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

```
revapp :: forall (c::Conv) (r::Rep)

          (a::TYPE Ptr c) (b::TYPE r Call[1]).

          a ~> (a ~> b) ~> b

revapp x f = f x
```

- `f :: a ~> b :: TYPE Ptr Call[2]` has arity 2
Arity Polymorphism

- **Generalize** \(a::\star\) to \(a::\text{TYPE \ r \ c}\)
  - \(r::\text{Rep}\) is the *runtime representation* of \(a\)
  - \(c::\text{Conv}\) is the *calling convention* of \(a\)
  - \(a::\text{TYPE \ Ptr \ Call}[n]\) says \(a\) values are pointers with arity \(n\) (simplified)

\[
\text{poly} :: \forall (a::\text{TYPE \ Ptr \ Call}[2]). (\text{Int} \Rightarrow \text{Int} \Rightarrow a) \Rightarrow (a,a)
\]

\[
\text{poly} \ f = \text{let} \ g :: \text{Int} \Rightarrow a = f \ 3 \ \text{in} \ (g \ 4, g \ 5)
\]

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

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

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

- \(f :: a \Rightarrow b :: \text{TYPE \ Ptr \ Call}[2]\) has arity 2
- \(x :: a :: \text{TYPE \ Ptr \ c}\) is represented as a pointer
Even More

- Levity Polymorphism
  - For when evaluation strategy doesn’t matter

- Compiling Source → Intermediate → Target
  - Via kind-directed $\eta$-expansion and register assignment

- Type system for ensuring static compilation
  - Of definitions with arity, levity, and representation polymorphism
Kinds capture the details of efficient calling conventions