

Where we're at

- **Syntax Foundations** ✓

Concrete/Abstract Syntax, Ambiguity, HOAS, Binding, Variables, Substitution

- **Semantics Foundations** ✓

Static Semantics, Dynamic Semantics (Small-Step/Big-Step),
(**Assignment 0**) Abstract Machines, Environments
(**Assignment 1**)

- **Features**

- Algebraic Data Types ✓
- Polymorphism
- Polymorphic Type Inference (**Assignment 2**)
- Overloading
- Subtyping
- Modules
- Concurrency

A Swap Function

Consider the humble `swap` function in Haskell:

$$\begin{aligned} \text{swap} &:: (t_1, t_2) \rightarrow (t_2, t_1) \\ \text{swap } (a, b) &= (b, a) \end{aligned}$$

In our MinHS with algebraic data types from last lecture, we can't define this function.

Monomorphic

In MinHS, we're stuck copy-pasting our function over and over for every different type we want to use it with:

```
recfun swap1 :: ((Int × Bool) → (Bool × Int))
                p = (snd p, fst p)
```

```
recfun swap2 :: ((Bool × Int) → (Int × Bool))
                p = (snd p, fst p)
```

```
recfun swap3 :: ((Bool × Bool) → (Bool × Bool))
                p = (snd p, fst p)
```

...

This is an acceptable state of affairs for some domain-specific languages, but not for general purpose programming.

Solutions

We want some way to specify that we **don't care** what the types of the tuple elements are.

$$\text{swap} :: (\forall a b. (a \times b) \rightarrow (b \times a))$$

This is called *parametric polymorphism* (or just *polymorphism* in functional programming circles). In Java and some other languages, this is called *generics* and polymorphism refers to something else.

How it works

There are two main components to parametric polymorphism:

- 1 *Type abstraction* is the ability to define functions regardless of specific types (like the `swap` example before).

How it works

There are two main components to parametric polymorphism:

- 1 *Type abstraction* is the ability to define functions regardless of specific types (like the `swap` example before). We will write **type** expressions like so: (the literature uses Λ)

swap = **type** *a*. **type** *b*.

recfun *swap* :: (*a* × *b*) → (*b* × *a*)

p = (`snd p`, `fst p`)

How it works

There are two main components to parametric polymorphism:

- 1 *Type abstraction* is the ability to define functions regardless of specific types (like the `swap` example before). We will write **type** expressions like so: (the literature uses Λ)

swap = **type** *a*. **type** *b*.

recfun *swap* :: (*a* × *b*) → (*b* × *a*)

p = (snd *p*, fst *p*)

- 2 *Type application* is the ability to *instantiate* polymorphic functions to specific types.

How it works

There are two main components to parametric polymorphism:

- 1 *Type abstraction* is the ability to define functions regardless of specific types (like the `swap` example before). We will write **type** expressions like so: (the literature uses Λ)

$$\begin{aligned} \text{swap} &= \mathbf{type} \ a. \mathbf{type} \ b. \\ &\quad \mathbf{recfun} \ \text{swap} :: (a \times b) \rightarrow (b \times a) \\ &\quad \quad p = (\text{snd } p, \text{fst } p) \end{aligned}$$

- 2 *Type application* is the ability to *instantiate* polymorphic functions to specific types. We will often write like so:

$$\text{swap@Int@Bool} \ (3, \text{True})$$

NB: *differs* from MinHS language in Assignment 2!

Analogies

The reason they're called type abstraction and application is that they behave analogously to λ -calculus.

We have a β -reduction principle, but for types:

$$(\mathbf{type} \ a. \ e)@_{\tau} \ \mapsto_{\beta} \ (e[a := \tau])$$

Analogies

The reason they're called type abstraction and application is that they behave analogously to λ -calculus.

We have a β -reduction principle, but for types:

$$(\mathbf{type} \ a. \ e)@_{\tau} \mapsto_{\beta} (e[a := \tau])$$

Example (Identity Function)

```
(type a. refun f :: (a → a) x = x)@Int 3
```

Analogies

The reason they're called type abstraction and application is that they behave analogously to λ -calculus.

We have a β -reduction principle, but for types:

$$(\mathbf{type} \ a. \ e)@_{\tau} \mapsto_{\beta} (e[a := \tau])$$

Example (Identity Function)

$$\begin{aligned} & (\mathbf{type} \ a. \ \mathbf{refun} \ f :: (a \rightarrow a) \ x = x)@_{\text{Int}} \ 3 \\ \mapsto & (\mathbf{refun} \ f :: (\text{Int} \rightarrow \text{Int}) \ x = x) \ 3 \end{aligned}$$

Analogies

The reason they're called type abstraction and application is that they behave analogously to λ -calculus.

We have a β -reduction principle, but for types:

$$(\mathbf{type} \ a. \ e)@_{\tau} \ \mapsto_{\beta} \ (e[a := \tau])$$

Example (Identity Function)

```
(type a. recfun f :: (a → a) x = x)@Int 3
↳ (recfun f :: (Int → Int) x = x) 3
↳ 3
```

This means that **type** expressions can be thought of as **functions** from types to values.

Type Variables

What is the type of this?

(type a. recfun f :: (a → a) x = x)

¹Technically, they already could with recursive types.

Type Variables

What is the type of this?

(type a. recfun f :: (a → a) x = x)

$\forall a. a \rightarrow a$

Types can mention *type variables* now¹.

¹Technically, they already could with recursive types.

Type Variables

What is the type of this?

(type a. recfun f :: (a → a) x = x)

$\forall a. a \rightarrow a$

Types can mention *type variables* now¹.

If $id : \forall a. a \rightarrow a$, what is the type of $id@Int$?

¹Technically, they already could with recursive types.

Type Variables

What is the type of this?

(type a. recfun f :: (a → a) x = x)

$\forall a. a \rightarrow a$

Types can mention *type variables* now¹.

If $id : \forall a. a \rightarrow a$, what is the type of $id@Int$?

$(a \rightarrow a)[a := Int] = (Int \rightarrow Int)$

¹Technically, they already could with recursive types.

Typing Rules Sketch

We would like rules that look something like this:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{type} \ a. e : \forall a. \tau}$$

$$\frac{\Gamma \vdash e : \forall a. \tau}{\Gamma \vdash e@{\rho} : \tau[a := \rho]}$$

But these rules don't account for what **type variables** are available or in scope.

Type Wellformedness

With variables in the picture, we need to check our types to make sure that they only refer to well-scoped variables.

$$\begin{array}{c}
 \frac{t \text{ bound} \in \Delta}{\Delta \vdash t \text{ ok}} \quad \frac{}{\Delta \vdash \text{Int ok}} \quad \frac{}{\Delta \vdash \text{Bool ok}} \\
 \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok}} \quad \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \times \tau_2 \text{ ok}} \\
 \text{(etc.)}
 \end{array}$$

$$\frac{\Delta, a \text{ bound} \vdash \tau \text{ ok}}{\Delta \vdash \forall a. \tau \text{ ok}}$$

Typing Rules, Properly

We add a **second context** of type variables that are bound.

$$\frac{a \text{ bound}, \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \text{type } a. e : \forall a. \tau}$$

$$\frac{\Delta; \Gamma \vdash e : \forall a. \tau \quad \Delta \vdash \rho \text{ ok}}{\Delta; \Gamma \vdash e@_a \rho : \tau[a := \rho]}$$

(the other typing rules just pass Δ through)

NB: **differs** from MinHS language in Assignment 2!

Dynamic Semantics

First we evaluate the LHS of a type application as much as possible:

$$\frac{e \mapsto_M e'}{e@T \mapsto_M e'@T}$$

Dynamic Semantics

First we evaluate the LHS of a type application as much as possible:

$$\frac{e \mapsto_M e'}{e@_{\tau} \mapsto_M e'@_{\tau}}$$

Then we apply our β -reduction principle:

$$\overline{(\mathbf{type} \ a. \ e)@_{\tau} \mapsto_M e[a := \tau]}$$

Curry-Howard

Previously we noted the correspondence between types and logic:

\times	\wedge
$+$	\vee
\rightarrow	\Rightarrow
1	\top
0	\perp
\forall	?

Curry-Howard

Previously we noted the correspondence between types and logic:

\times	\wedge
$+$	\vee
\rightarrow	\Rightarrow
1	\top
0	\perp
\forall	\exists

Curry-Howard

The type quantifier \forall corresponds to a universal quantifier \forall , but **not** from first-order logic. What's the difference?

Curry-Howard

The type quantifier \forall corresponds to a universal quantifier \forall , but **not** from first-order logic. What's the difference?

First-order logic quantifiers range over a set of *individuals* or values, for example the natural numbers:

$$\forall x. x + 1 > x$$

The type quantifier ranges over types (not values!), or invoking Curry-Howard: **propositions**. Analogous to *second-order logic*:

$$\begin{aligned} \forall A. \forall B. A \wedge B &\Rightarrow B \wedge A \\ \forall A. \forall B. A \times B &\rightarrow B \times A \end{aligned}$$

(First-order quantifier's type-theoretic analogue: dependent types!)

Generality

If we need a function of type $\text{Int} \rightarrow \text{Int}$, a polymorphic function of type $\forall a. a \rightarrow a$ will do just fine, we can just instantiate the type variable to Int . But the reverse is not true. This gives rise to an ordering.

Generality

If we need a function of type $\text{Int} \rightarrow \text{Int}$, a polymorphic function of type $\forall a. a \rightarrow a$ will do just fine, we can just instantiate the type variable to Int . But the reverse is not true. This gives rise to an ordering.

Generality

A type τ is *more general* than a type ρ , often written $\rho \sqsubseteq \tau$, if type variables in τ can be instantiated to give the type ρ .

NB: $\rho \sqsubseteq \tau$ is often written $\tau \prec \rho$ in the literature.

Generality

If we need a function of type $\text{Int} \rightarrow \text{Int}$, a polymorphic function of type $\forall a. a \rightarrow a$ will do just fine, we can just instantiate the type variable to Int . But the reverse is not true. This gives rise to an ordering.

Generality

A type τ is *more general* than a type ρ , often written $\rho \sqsubseteq \tau$, if type variables in τ can be instantiated to give the type ρ .

NB: $\rho \sqsubseteq \tau$ is often written $\tau \prec \rho$ in the literature.

Example (Functions)

$$\text{Int} \rightarrow \text{Int} \quad \sqsubseteq \quad \forall z. z \rightarrow z \quad \sqsubseteq \quad \forall x y. x \rightarrow y \quad \sqsubseteq \quad \forall a. a$$

Implementation Strategies

Our simple dynamic semantics belies an implementation headache.

We can easily **define** functions that operate uniformly on multiple types. But when they are **compiled** to machine code, the results may differ depending on the **size** of the type in question.

There are two main approaches to solve this problem.

Approach 1: Template Instantiation

Key Idea

Automatically generate monomorphic copies of each polymorphic function, based on the types applied to it.

Approach 1: Template Instantiation

Key Idea

Automatically generate monomorphic copies of each polymorphic function, based on the types applied to it.

For example, if we defined our polymorphic swap function:

```
swap = type a. type b.  
      recfun swap :: (a × b) → (b × a)  
              p = (snd p, fst p)
```

Approach 1: Template Instantiation

Key Idea

Automatically generate monomorphic copies of each polymorphic function, based on the types applied to it.

For example, if we defined our polymorphic swap function:

```
swap = type a. type b.  
      recfun swap :: (a × b) → (b × a)  
      p = (snd p, fst p)
```

Then a type application like `swap@Int@Bool` would be replaced **statically** by the compiler with the monomorphic version:

```
swapIB = recfun swap :: (Int × Bool) → (Bool × Int)  
      p = (snd p, fst p)
```

A new copy is made for each unique type application.

Template Instantiation Pros and Cons

Template Instantiation Pros and Cons

This approach has a number of advantages:

- ① Little to no run-time cost
- ② Simple mental model
- ③ Allows for custom specialisations (e.g. list of booleans into bit-vectors)
- ④ Easy to implement

Template Instantiation Pros and Cons

This approach has a number of advantages:

- ① Little to no run-time cost
- ② Simple mental model
- ③ Allows for custom specialisations (e.g. list of booleans into bit-vectors)
- ④ Easy to implement

However the downsides are just as numerous:

- ① Large binary size if many instantiations are used
- ② This can lead to long compilation times
- ③ Restricts the type system to **statically** instantiated type variables.

Languages that use Template Instantiation: Rust, C++, some ML dialects

Approach 2: Boxing

An alternative to our copy-paste-heavy template instantiation approach is to make **all types** represented the same way. Thus, a polymorphic function only requires one function in the generated code.

Approach 2: Boxing

An alternative to our copy-paste-heavy template instantiation approach is to make **all types** represented the same way. Thus, a polymorphic function only requires one function in the generated code.

Typically this is done by *boxing* each type. That is, all data types are represented as a **pointer** to a data structure on the heap. If everything is a pointer, then all values use exactly 32 (or 64) bits of stack space.

Approach 2: Boxing

An alternative to our copy-paste-heavy template instantiation approach is to make **all types** represented the same way. Thus, a polymorphic function only requires one function in the generated code.

Typically this is done by **boxing** each type. That is, all data types are represented as a **pointer** to a data structure on the heap. If everything is a pointer, then all values use exactly 32 (or 64) bits of stack space.

The extra indirection has a run-time penalty, but it results in smaller binaries and **unrestricted polymorphism**.

Languages that use boxing: Haskell, Java, C#, OCaml

Example: Polymorphic Recursion

Consider the following Haskell data type:

```
data Dims a = Step a (Dims [a]) | Epsilon
```

Example: Polymorphic Recursion

Consider the following Haskell data type:

```
data Dims a = Step a (Dims [a]) | Epsilon
```

This describes a list of matrices of increasing dimensionality, e.g:

```
Step 1 (Step [1, 2] (Step [[1, 2], [3, 4]] Epsilon)) :: Dims Int
```

Example: Polymorphic Recursion

Consider the following Haskell data type:

```
data Dims a = Step a (Dims [a]) | Epsilon
```

This describes a list of matrices of increasing dimensionality, e.g:

```
Step 1 (Step [1, 2] (Step [[1, 2], [3, 4]] Epsilon)) :: Dims Int
```

We can write a sum function like this:

```
sumDims :: ∀a. (a → Int) → Dims a → Int
```

```
sumDims f Epsilon = 0
```

```
sumDims f (Step a t) = (f a) + sumDims (sum ∘ (map f)) t
```

Example: Polymorphic Recursion

Consider the following Haskell data type:

```
data Dims a = Step a (Dims [a]) | Epsilon
```

This describes a list of matrices of increasing dimensionality, e.g:

```
Step 1 (Step [1, 2] (Step [[1, 2], [3, 4]] Epsilon)) :: Dims Int
```

We can write a sum function like this:

```
sumDims :: ∀a. (a → Int) → Dims a → Int
```

```
sumDims f Epsilon = 0
```

```
sumDims f (Step a t) = (f a) + sumDims (sum ∘ (map f)) t
```

How many different instantiations of the type variable *a* are there?

Example: Polymorphic Recursion

Consider the following Haskell data type:

```
data Dims a = Step a (Dims [a]) | Epsilon
```

This describes a list of matrices of increasing dimensionality, e.g:

```
Step 1 (Step [1, 2] (Step [[1, 2], [3, 4]] Epsilon)) :: Dims Int
```

We can write a sum function like this:

```
sumDims :: ∀a. (a → Int) → Dims a → Int
```

```
sumDims f Epsilon = 0
```

```
sumDims f (Step a t) = (f a) + sumDims (sum ∘ (map f)) t
```

How many different instantiations of the type variable a are there?

We'd have to run the program to find out.

HM Types

Template instantiation can't handle all polymorphic programs.

In practice a **statically determined subset** can be carved out by **restricting** what sort of programs can be written:

- 1 Only allow \forall quantifiers on the **outermost** part of a type declaration (not inside functions or type constructors).
- 2 Recursive functions **cannot** call themselves with different type parameters.

This restriction is sometimes called *Hindley-Milner* polymorphism. This is also the subset for which *type inference* is both complete and tractable.

Constraining Implementations

How many possible implementations are there of a function of the following type?

$$\text{Int} \rightarrow \text{Int}$$

Constraining Implementations

How many possible implementations are there of a function of the following type?

$$\text{Int} \rightarrow \text{Int}$$

How about this type?

$$\forall a. a \rightarrow a$$

Constraining Implementations

How many possible implementations are there of a function of the following type?

$$\text{Int} \rightarrow \text{Int}$$

How about this type?

$$\forall a. a \rightarrow a$$

Polymorphic type signatures constrain implementations.

Parametricity

Definition

The principle of **parametricity** states that the result of polymorphic functions cannot depend on **values** of an abstracted type.

More formally, suppose I have a polymorphic function g that takes a type parameter. If run any arbitrary function $f : \tau \rightarrow \tau$ on some values of type τ , then run the function $g@_{\tau}$ on the result, that will give the same results as running $g@_{\tau}$ first, then f .

Example

$$foo :: \forall a. [a] \rightarrow [a]$$

Parametricity

Definition

The principle of **parametricity** states that the result of polymorphic functions cannot depend on **values** of an abstracted type.

More formally, suppose I have a polymorphic function g that takes a type parameter. If run any arbitrary function $f : \tau \rightarrow \tau$ on some values of type τ , then run the function $g@_{\tau}$ on the result, that will give the same results as running $g@_{\tau}$ first, then f .

Example

$$foo :: \forall a. [a] \rightarrow [a]$$

We know that **every** element of the output occurs in the input. The parametricity theorem we get is, for all f :

$$foo \circ (map f) = (map f) \circ foo$$

More Examples

$$head :: \forall a. [a] \rightarrow a$$

What's the parametricity theorem for *head*?

More Examples

$$\text{head} :: \forall a. [a] \rightarrow a$$

What's the parametricity theorem for *head*?

Example (Answer)

For any f :

$$f (\text{head } \ell) = \text{head } (\text{map } f \ell)$$

More Examples

$$(++) :: \forall a. [a] \rightarrow [a] \rightarrow [a]$$

What's the parametricity theorem for list append $(++)$?

More Examples

$$(++) :: \forall a. [a] \rightarrow [a] \rightarrow [a]$$

What's the parametricity theorem for list append $(++)$?

Example (Answer)

$$\text{map } f (a ++ b) = \text{map } f a ++ \text{map } f b$$

More Examples

$$\text{concat} :: \forall a. [[a]] \rightarrow [a]$$

What's the parametricity theorem for list concatenation *concat*?

More Examples

$$\text{concat} :: \forall a. [[a]] \rightarrow [a]$$

What's the parametricity theorem for list concatenation *concat*?

Example (Answer)

$$\text{map } f (\text{concat } ls) = \text{concat } (\text{map } (\text{map } f) ls)$$

Higher Order Functions

$$filter :: \forall a. (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$$

What's the parametricity theorem for *filter*?

Higher Order Functions

$$\text{filter} :: \forall a. (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

What's the parametricity theorem for *filter*?

Example (Answer)

$$\text{filter } p (\text{map } f \text{ } l\text{s}) = \text{map } f (\text{filter } (p \circ f) \text{ } l\text{s})$$

Parametricity Theorems

Follow a similar structure. In fact it can be mechanically derived, using the *relational parametricity* framework invented by John C. Reynolds, and popularised by Wadler in the famous paper, “Theorems for Free!”².

²<https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf>