Polymorphism

Rob Sison
UNSW
Term 3 2024

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:

$$\mathit{swap} :: (t_1, t_2) \to (t_2, t_1)$$
$$\mathit{swap}\ (a, b) = (b, a)$$

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:

$$\textbf{recfun}\ \mathit{swap}_1 :: ((\texttt{Int} \times \texttt{Bool}) \to (\texttt{Bool} \times \texttt{Int}))$$
$$p = (\mathsf{snd}\ p, \mathsf{fst}\ p)$$

$$\textbf{recfun}\ \mathit{swap}_2 :: ((\texttt{Bool} \times \texttt{Int}) \to (\texttt{Int} \times \texttt{Bool}))$$
$$p = (\mathsf{snd}\ p, \mathsf{fst}\ p)$$

$$\textbf{recfun}\ \mathit{swap}_3 :: ((\texttt{Bool} \times \texttt{Bool}) \to (\texttt{Bool} \times \texttt{Bool}))$$
$$p = (\mathsf{snd}\ p, \mathsf{fst}\ p)$$

...

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.

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

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)

$$\begin{array}{l} \textit{swap} = \textbf{type}\ a.\ \textbf{type}\ b. \\ \qquad \textbf{recfun}\ \textit{swap} :: (a \times b) \rightarrow (b \times a) \\ \qquad\qquad p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{array}$$

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)

$$\begin{aligned} \mathit{swap} = \mathbf{type}\ a.\ \mathbf{type}\ b.\ & \\ & \mathbf{recfun}\ \mathit{swap} :: (a \times b) \to (b \times a) \\ & \qquad p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

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)

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

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

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

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:

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

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:

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

Example (Identity Function)

$$(\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \to a)\ x = x)@\texttt{Int}\ 3$$

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:

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

Example (Identity Function)

$$\begin{array}{ll} & (\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \to a)\ x = x)@\texttt{Int}\ 3 \\ \mapsto & (\textbf{recfun}\ f :: (\texttt{Int} \to \texttt{Int})\ x = x)\ 3 \end{array}$$

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:

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

Example (Identity Function)

$$\begin{array}{ll} & (\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \to a)\ x = x)@\texttt{Int}\ 3 \\ \mapsto & (\textbf{recfun}\ f :: (\texttt{Int} \to \texttt{Int})\ x = x)\ 3 \\ \mapsto & 3 \end{array}$$

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

Type Variables

What is the type of this?

$$(\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \rightarrow a)\ x = x)$$

[1] Technically, they already could with recursive types.

Type Variables

What is the type of this?

$$(\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \rightarrow a)\ x = x)$$

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

Types can mention *type variables* now[1].

[1] Technically, they already could with recursive types.

Type Variables

What is the type of this?

$$(\textbf{type } a.\ \textbf{recfun } f :: (a \to a)\ x = x)$$

$$\forall a.\ a \to a$$

Types can mention *type variables* now[1].

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

[1] Technically, they already could with recursive types.

Type Variables

What is the type of this?

$$(\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \to a)\ x = x)$$

$$\forall a.\ a \to a$$

Types can mention *type variables* now[1].

If $id : \forall a.a \to a$, what is the type of $id@\texttt{Int}$?

$$(a \to a)[a := \texttt{Int}] = (\texttt{Int} \to \texttt{Int})$$

[1] Technically, they already could with recursive types.

Typing Rules Sketch

We would like rules that look something like this:

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

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

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.

$$\frac{t\ \textbf{bound} \in \Delta}{\Delta \vdash t\ \textbf{ok}} \qquad \frac{}{\Delta \vdash \texttt{Int}\ \textbf{ok}} \qquad \frac{}{\Delta \vdash \texttt{Bool}\ \textbf{ok}}$$

$$\frac{\Delta \vdash \tau_1\ \textbf{ok} \quad \Delta \vdash \tau_2\ \textbf{ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2\ \textbf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \textbf{ok} \quad \Delta \vdash \tau_2\ \textbf{ok}}{\Delta \vdash \tau_1 \times \tau_2\ \textbf{ok}}$$

(etc.)

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

Typing Rules, Properly

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

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

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

(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:

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

Dynamic Semantics

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

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

Then we apply our β-reduction principle:

$$\frac{}{(\textbf{type}\ a.\ e)@\tau \quad \mapsto_M \quad e[a := \tau]}$$

Curry-Howard

Previously we noted the correspondence between types and logic:

$\times$	$\wedge$
$+$	$\vee$
$\rightarrow$	$\Rightarrow$
$\mathbf{1}$	$\top$
$\mathbf{0}$	$\bot$
$\forall$	?

Curry-Howard

Previously we noted the correspondence between types and logic:

$\times$	$\wedge$
$+$	$\vee$
$\rightarrow$	$\Rightarrow$
$\mathbf{1}$	$\top$
$\mathbf{0}$	$\bot$
$\forall$	$\forall$

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*:

$$\forall A.\ \forall B.\ A \land B \Rightarrow B \land A$$
$$\forall A.\ \forall B.\ A \times B \to B \times A$$

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

Generality

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

Generality

If we need a function of type `Int` $\rightarrow$ `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 $\texttt{Int} \rightarrow \texttt{Int}$, a polymorphic function of type $\forall a.\ a \rightarrow a$ will do just fine, we can just instantiate the type variable to $\texttt{Int}$. 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)

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

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:

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

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:

$$\begin{array}{l} \mathit{swap} = \textbf{type}\ a.\ \textbf{type}\ b. \\ \qquad \textbf{recfun}\ \mathit{swap} :: (a \times b) \to (b \times a) \\ \qquad\qquad p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{array}$$

Then a type application like $\mathit{swap}@\mathtt{Int}@\mathtt{Bool}$ would be replaced statically by the compiler with the monomorphic version:

$$\begin{array}{l} \mathit{swap}_{\mathtt{IB}} = \textbf{recfun}\ \mathit{swap} :: (\mathtt{Int} \times \mathtt{Bool}) \to (\mathtt{Bool} \times \mathtt{Int}) \\ \qquad p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{array}$$

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:

$$\textbf{data}\ \mathit{Dims}\ a = \mathsf{Step}\ a\ (\mathit{Dims}\ [a])\ |\ \mathsf{Epsilon}$$

Example: Polymorphic Recursion

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Dims}\ a = \mathrm{Step}\ a\ (\mathit{Dims}\ [a])\ |\ \mathrm{Epsilon}$$

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

$$\mathrm{Step}\ 1\ (\mathrm{Step}\ [1,2]\ (\mathrm{Step}\ [[1,2],[3,4]]\ \mathrm{Epsilon})) :: \mathit{Dims}\ \mathtt{Int}$$

Example: Polymorphic Recursion

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Dims}\ a = \mathsf{Step}\ a\ (\mathit{Dims}\ [a])\ |\ \mathsf{Epsilon}$$

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

$$\mathsf{Step}\ 1\ (\mathsf{Step}\ [1,2]\ (\mathsf{Step}\ [[1,2],[3,4]]\ \mathsf{Epsilon})) :: \mathit{Dims}\ \mathtt{Int}$$

We can write a sum function like this:

$$\begin{array}{l} \mathit{sumDims} :: \forall a.\ (a \to \mathtt{Int}) \to \mathit{Dims}\ a \to \mathtt{Int} \\ \mathit{sumDims}\ f\ \mathsf{Epsilon} = 0 \\ \mathit{sumDims}\ f\ (\mathsf{Step}\ a\ t) = (f\ a) + \mathit{sumDims}\ (\mathit{sum} \circ (\mathit{map}\ f))\ t \end{array}$$

Example: Polymorphic Recursion

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Dims}\ a = \mathrm{Step}\ a\ (\mathit{Dims}\ [a]) \mid \mathrm{Epsilon}$$

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

$$\mathrm{Step}\ 1\ (\mathrm{Step}\ [1,2]\ (\mathrm{Step}\ [[1,2],[3,4]]\ \mathrm{Epsilon})) :: \mathit{Dims}\ \mathtt{Int}$$

We can write a sum function like this:

$$\begin{aligned}
&\mathit{sumDims} :: \forall a.\ (a \to \mathtt{Int}) \to \mathit{Dims}\ a \to \mathtt{Int}\\
&\mathit{sumDims}\ f\ \mathrm{Epsilon} = 0\\
&\mathit{sumDims}\ f\ (\mathrm{Step}\ a\ t) = (f\ a) + \mathit{sumDims}\ (\mathit{sum} \circ (\mathit{map}\ f))\ t
\end{aligned}$$

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

Example: Polymorphic Recursion

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Dims}\ a = \mathrm{Step}\ a\ (\mathit{Dims}\ [a])\ |\ \mathrm{Epsilon}$$

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

$$\mathrm{Step}\ 1\ (\mathrm{Step}\ [1,2]\ (\mathrm{Step}\ [[1,2],[3,4]]\ \mathrm{Epsilon})) :: \mathit{Dims}\ \mathtt{Int}$$

We can write a sum function like this:

$$\begin{array}{l} \mathit{sumDims} :: \forall a.\ (a \to \mathtt{Int}) \to \mathit{Dims}\ a \to \mathtt{Int} \\ \mathit{sumDims}\ f\ \mathrm{Epsilon} = 0 \\ \mathit{sumDims}\ f\ (\mathrm{Step}\ a\ t) = (f\ a) + \mathit{sumDims}\ (\mathit{sum} \circ (\mathit{map}\ f))\ t \end{array}$$

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?

$$\texttt{Int} \rightarrow \texttt{Int}$$

Constraining Implementations

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

$$\texttt{Int} \rightarrow \texttt{Int}$$

How about this type?

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

Constraining Implementations

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

$$\texttt{Int} \rightarrow \texttt{Int}$$

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 $f : \tau \to \tau$ 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

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

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

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

What's the parametricity theorem for *head*?

Example (Answer)

For any f:

$$f\ (head\ \ell) = head\ (map\ f\ \ell)$$

More Examples

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

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)

$$map\ f\ (a +\!\!+ b) = map\ f\ a +\!\!+ map\ f\ b$$

More Examples

$$concat :: \forall a.\ [[a]] \to [a]$$

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

More Examples

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

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

Example (Answer)

$$map\ f\ (concat\ ls) = concat\ (map\ (map\ f)\ ls)$$

Higher Order Functions

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

What's the parametricity theorem for *filter*?

Higher Order Functions

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

What's the parametricity theorem for *filter*?

Example (Answer)

$$filter\ \ p\ (map\ \ f\ \ ls) = map\ \ f\ \ (filter\ (p \circ f)\ \ ls)$$

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