Polymorphi

Christine Rizkallah
CSE, UNSW
Term 3 2020

Where we're at

- **Syntax Foundations** ✓
  Concrete/Abstract Syntax, Ambiguity, HOAS, Binding, Variables, Substitution
- **Semantic**
  Static Sem **Assignment 0**)
  Abstract M
- **Features**
  - Algebraic Data Types ✓
  - Polymorphism
  - Polymorphic Type Inference (**Assignment 2**
  - Overloading
  - Subtyping
  - Modules
  - Concurrency

A Swap Function

Consider the hum

In our MinHS with algebraic data types from last lecture, we can'

Monomorphic

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

$$\textbf{recfun}\ \mathit{swap_1} :: ((\mathtt{Int} \times \mathtt{Bool}) \rightarrow (\mathtt{Bool} \times \mathtt{Int}))$$

$$p = (\mathsf{snd}\ p, \mathsf{fst}\ p)$$

$$\textbf{recfun}\ \mathit{swap_3} :: ((\mathtt{Bool} \times \mathtt{Boo}$$

$$p = (\mathsf{snd}\ p, \mathsf{fst}\ p)$$

$$\cdots$$

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.

This is called *parametric polymorphism* (or just [illegible] programming circles). In Java and some other languages, this [illegible] *nerics* and polymorphism refers to something else. Don't be confused.

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). In MinHS, we will write using **type** expressions like so: (the literature uses Λ)

$$\textbf{recfun}\ \mathit{swap} :: (a \quad b) \quad (b \quad a)$$

2. *Type application* is the ability to *instantiate* cific types. In MinHS, we use @ signs.

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

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:

Example (Identity Function)

$$
\begin{array}{ll}
 & (\textbf{type}\ a.\ \textbf{recfun}\ f :: (a \ldots \\
\mapsto & (\textbf{recfun}\ f :: (\texttt{Int} \to \texttt{In} \ldots \\
\mapsto & 3
\end{array}
$$

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

Type Variables

What is the type of this?

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

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:

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

But these rules don't account for what type variables are availa

Type Wellformedness

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

$$\frac{}{\Delta \vdash \tau_1 \to \tau_2 \ \textbf{ok}}$$

(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.

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

(the other typing rules just pass Δ through)

Dynamic Semantics

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

$$\frac{\vdash}{\vdash}$$

Then we apply our β-reduction principle:

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

Curry-Howard

Previously we noted the correspondence between types and logic:

$\rightarrow$	$\Rightarrow$
$\mathbf{1}$	
$\mathbf{0}$	$\bot$
$\forall$	$\forall$

Curry-Howard

The type quantifier $\forall$ corresponds to a universal quantifier $\forall$, but it is *not* the same as the $\forall$ 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

These quantifier

second-order logic, not first-order:

$$\forall A.\ \forall B.\ A \wedge B$$
$$\forall A.\ \forall B.\ A \times B$$

The first-order quantifier has a type-theoretic analogue too (type indices), but this is not nearly as common as polymorphism.

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 *mo* [illegible] bles in τ can be instantiated to give the type ρ.

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 dyna

While we can easily

this is compiled to m

type in question.

There are two main approaches to solve this problem.

Template Instantiation

Key Idea

Automatically generate a monomorphic copy of each polymorphic functions based on the types applied to it.

For example, if we d

$$p =$$

Then a type application like $swap$@Int@Bool w compiler with the monomorphic version:

$$\begin{aligned} swap_{\texttt{IB}} = \textbf{recfun}\ swap :: (\texttt{Int} \times \texttt{Bool}) \rightarrow (\texttt{Bool} \times \texttt{Int}) \\ p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

A new copy is made for each unique type application.

Evaluating Template Instatiation

This approach has a number of advantages:

1. Little to no run-time cost
2. Simple men
3. Allows for cu
4. Easy to impl

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++, Cogent, some ML dialects

Polymorphic Recursion

Consider the following Haskell data type:

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

This describes a list

S Int

We can write a sum function like this:

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

How many different instantiations of the type variable a are there? We'd have to run the program to find out.

HM Types

Automatically generating a copy for each instantiation is great but can't handle all polymorphic programs.

In practice a static
programs can be w

1. Only allow ∀
   functions or type constructors).

2. Recursive functions *cannot* call themselves with differ

This restriction is sometimes called *Hindley-Miln* he subset for which *type inference* is both complete and tractable.

Boxing

An alternative to our copy-paste-heavy template instantiation approach is to make all types represente
function in the gen
Typically this is do s a
pointer to a data str
exactly 32 (or 64) bits of stack space.

The extra indirection has a run-time penalty, and it can make ga
necessary, but it results in smaller binaries and unrestricted po

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

Constraining Implementations

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

How about this typ

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

Polymorphic type signatures constrain implementation

Parametricity

Definition

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

More formally, su ... arameter. If run any arbitrary f ... he function $g@_\tau$ on the result ... st then f

Example

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

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 param

Example (Ans

For any f:

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

More Examples

$$(++) :: \quad a.\ [a] \quad [a] \quad [a]$$

What's the param

Example (Answer)

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

More Examples

$$concat :: \quad a.\ [[a]] \quad [a]$$

What's the param

Example (Answer)

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

Higher Order Functions

What's the param

Example (Ans

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

Parametricity Theorems

Follow a similar str *relational*
parametricity f r in
the famous paper,
Upshot: We can ask `lambdabot` on the Haskell IRC c

[2] `https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf`