Polymorphism

Concepts of Programming Languages
Lecture 4

CAS CS 320

Practice Problem

```
let k = fun x -> fun y -> x in
let x = 3 + k k 2 3 4 in
k x (k x)
```

$$\texttt{let x = } \underbrace{\texttt{3 + } \underbrace{\underbrace{\texttt{k k 2}}_{K}\texttt{ 3 4}}_{3}}_{6}\texttt{ in}$$

$$\underbrace{\texttt{k x (k x)}}_{x=6}$$

Would UTop throw a type error on the above expressions? If not, what does it evaluate to?

Answer

```
let k = fun x -> fun y -> x in
let x = 3 + k k 2 3 4 in
k x (k x)
```

Practice Problem

Implement the function

```
val bitonic : int -> int -> int -> int list
```

so that **bitonic i j k** is a list of consecutive integers from **i** to **j** to **k**, e.g.,

```
bitonic 1 5 3 = [1;2;3;4;5;4;3]
```

Outline

- Discuss **polymorphism** in general
- Demo **examples** of polymorphic functions

Polymorphism

Recall: Polymorphism and Lists

```
let rec length l =
  match l with
  | [] -> 0
  | x :: xs -> 1 + length xs
```

What is the type of the length function?

Does this function depend on the values in the list?

Recall: The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	`string list`	`int list list`

Recall: The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

Recall: The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

Recall: The List Type

[1;2;3]	["1";"2";"3"]	[[1;1];[2;2];[3;3]]
int list	**string list**	**int list list**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

Recall: The List Type

`[1;2;3]`	`["1";"2";"3"]`	`[[1;1];[2;2];[3;3]]`
`int list`	**`string list`**	**`int list list`**

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

`'a, 'b, 'c,...`

Not all functions are polymorphic

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Not all functions are polymorphic

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Not all functions are polymorphic

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int list**s

Not all functions are polymorphic

```ocaml
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int list**s

OCaml's type inference is good at "guessing" when functions are polymorphic

Example

Implement the function

reverse : 'a list -> 'a list

such that ***reverse l*** *is the same as* ***l*** *but in reverse order*

High Level

```ocaml
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

Polymorphism allows for better code reuse. The *same* function can be applied in multiple contexts

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Important: We can evaluate this if we *don't* type check

Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Important: We can evaluate this if we *don't* type check

But if we type-check, what should be the type of ***id****?*

Polymorphism

Polymorphism

There are two common kinds of polymorphism

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

our focus

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

Then we can define code against *interfaces* (this is common in object oriented programming)

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

For example, we can write a single identity function and use it in multiple contexts

Type Parameters

```
let id : 'a -> 'a = fun x -> x
```

Type Parameters

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *parameters*

Type Parameters

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *parameters*

Type parameters are instantiated at particular types according to the context

Looking Ahead: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

Looking Ahead: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Looking Ahead: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

Looking Ahead: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

We read this "**id** has type **t -> t** for any type **t**"

Polymorphism and Type Inference

(int → 'a) → int → 'a

```
fun f -> fun x -> f (x + 1)
```

In OCaml, the type of an expression depends on how its subexpressions are *used*

OCaml can figure out what is the "most" polymorphic type we can give to an expression

This is called **type inference**

demo

(remove)

There are many subtleties to this...

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

There are PLs *without* polymorphism *or* type annotations

Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Parametric polymorphism is *not* just removing type annotations

There are PLs *without* polymorphism *or* type annotations

There are PLs *with* polymorphism that *require* type annotations

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

In OCaml, polymorphism is deeply connected with it's type inference system, but they are distinct (we can choose to annotated all our OCaml code)

Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let id : 'a -> 'a = fun x -> x
```

Polymorphism is *not* the same has having type inference

In OCaml, polymorphism is deeply connected with it's type inference system, but they are distinct (we can choose to annotated all our OCaml code)

We will take up this topic at the end of the course

Subtlety 3: Dynamic Dispatch

```
let to_string (x : 'a) : string = ...
(* This is not possible in OCaml *)
```

Subtlety 3: Dynamic Dispatch

```
let to_string (x : 'a) : string = ...
(* This is not possible in OCaml *)
```

Parametric polymorphism cannot be used for *dispatch*

Subtlety 3: Dynamic Dispatch

```
let to_string (x : 'a) : string = ...
(* This is not possible in OCaml *)
```

Parametric polymorphism cannot be used for *dispatch*

We can't write a polymorphic function that "checks the type" to see what to do

Practice Problem

```
let rec f x = f (f (x + 1)) in f
```

What is the type of the above OCaml expressions?

Answer

```
let rec f x = f (f (x + 1)) in f
```

Aside: Parametricity

Aside: Parametricity

Can you write function of type ***'a -> 'a****?*

Aside: Parametricity

Can you write function of type ***'a -> 'a****?*

Can you write a another?

Aside: Parametricity

Can you write function of type ***'a -> 'a****?*

Can you write a another?

Parametricity refers the (very cool) observation that polymorphic types often restrict the kinds of functions you can write in a *mathematically rigorous sense*

Summary

Polymorphism allows functions to be agnostic to (parts of) the types of its parameters

Parametric polymorphism does not allow for dynamic dispatch, it must given the *same* implementation at any particular type