# CS 320: Concepts of Programming Languages

## Lecture 7: Polymorphism

Ankush Das

Nathan Mull

Sep 24, 2024

# Abstraction of the Day



- Today, we will talk about polymorphism!
- Recall polymorphic functions:
  - Functions that are defined for any arbitrary type!
  - Just need to be defined once; can be called at arbitrary types
  - We will see more of those today!

# Reminder: Length of a List



```
'a list -> int
let rec length lst =
  match lst with
  | [] -> 0
  | h::t -> 1 + (length t)
```

- If the list is empty, i.e., `[]`, then return `0`
- Else, add `1` to the length of `t`
- Note the type: what is `'a list`?

# Polymorphic Length Function



- This length function can be applied to any argument
- ```
  length [1; 2; 3] = 3
  length [4.; 1.; 2.; 5.] = 4
  length [“cs320”, “cs599”] = 2
  ```
- In fact, they only need to be defined once and can be called with different types without defining them individually for each type
- We will do more such examples today!
  ```

# Simplest Polymorphic Function



- Suppose we need to define a function `f : 'a -> 'a`
- There's only one possibility:

```
'a -> 'a
let f x = x
```

- Why? Informally, you don't know anything about `x`, so you cannot modify it; you have no other expression of type `'a`

# Other Simple Polymorphic Functions



- Suppose we need to define a function `f : 'a -> 'a -> 'a`
- How many possibilities are there? Exactly two!
- ```
  let f1 x y = x
  let f2 x y = y
  ```
- You can try defining local variables, etc. but it won't help you!
- ```
  let f x y =
    let z = x in
    z
  ```

# Let's Take a More Difficult Type



- Suppose we need to define a function `f : 'a list -> 'a list`
- What can you do inside `f`? You can
  - Delete elements (delete the first, last element, etc.)
  - Duplicate elements
  - Change the order of elements (reverse, swap, etc.)
- You can't really do anything else. More importantly, you CANNOT MODIFY ANY element of the list, you don't know anything about it

# Some Examples



'a list -> 'a list

```
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

'a list -> 'a list

```
let rec delete_last l =
  match l with
  | [] -> []
  | [h] -> []
  | h::t -> h::(delete_last l)
```

'a list -> 'a list

```
let rec duplicate_last l =
  match l with
  | [] -> []
  | [h] -> [h; h]
  | h::t -> h::(duplicate_last t)
```

# What About Binary Trees?

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

- Suppose we define a function `f : ‘a tree -> ‘a tree`
- What are my choices?
  - Change the structure of the tree
  - Duplicate or delete elements
  - Can you insert elements?

# Some Example Functions



```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

'a tree -> 'a tree

```
let rec swap tr =
  match tr with
  | Leaf -> Leaf
  | Node(x, l, r) -> Node(x, swap r, swap l)
```

'a tree -> 'a tree

```
let rec delete_right tr =
  match tr with
  | Leaf -> Leaf
  | Node(x, l, r) -> Node(x, delete_right l, Leaf)
```

# Conclusion



- Think of ‘a type being like a black box; you can move it around but you CANNOT look inside
- Such types and functions are important for many applications, e.g., defining a polymorphic stack inside a module (we’ll see this later)
- Practice *typing derivations* for polymorphic list and tree functions