Programming Languages and Techniques (CIS1200)

Lecture 8

Generics & First-class functions

Chapters 8 and 9

Announcements

- Homework 2 due tomorrow night at 11:59pm
- Complete the intro survey (link on Ed) when available
- Homework 3 available Wednesday
  - Practice with BSTs, generic functions, first-class functions and abstract types
  - *Start early!*
- Read: Chapters 8, 9, and 10 of the lecture notes
- Midterm 1: Friday, September 27th
  - Covers chapters 1-10 in the lecture notes
  - Details posted on Ed later this week

Generic Functions and Data

Wow, implementing BSTs took quite a bit of typing... Do we have to do it all again if we want to use BSTs containing strings, and again for characters, and again for floats, and…?

or

How not to repeat yourself, Part I.

Structurally Identical Functions

- Observe: Many functions on lists don't depend on the contents of the list, only on the list structure
- Compare:

```
let rec length (l: int list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length tl
  end
```

```
let rec length (l: string list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length tl
  end
```

These functions are *identical* aside from the type annotations.

Notation for Generic Types

- In OCaml, functions can have *generic* types

```
let rec length (l:'a list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length tl)
  end
```

- Notation: `'a` is a *type variable*, indicating that the function `length` can be used on a `t list` for *any* type `t`
- Examples:
  - `length [1;2;3]` use length on an int list
  - `length ["a";"b";"c"]` use length on a string list
- Idea: OCaml chooses an appropriate type for `'a` whenever length is used

Generic List Append

Note that the two input lists must have the *same* type of elements, namely `'a`.

The return type is also the *same* as the inputs.

```
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
  | [] -> l2
  | h::tl -> h::(append tl l2)
  end
```

Pattern matching works over generic types!

In the body of the branch:

`h` has type `'a`

`tl` has type `'a list`

Zip function

- Combine two lists into one list
  - ignore elements from longer list if they are not the same length

```
let rec zip (l1:int list) (l2:string list)
          : (int*string) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

```
zip [1;2;3] ["a";"b";"c"]
  ⟼ [(1,"a"); (2,"b"); (3,"c")]
```

- Does it matter what type of elements are in these lists?

Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a * 'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- *Distinct* type variables *can* be instantiated differently:

```
zip [1;2;3] ["a";"b";"c"]
```

- Here, `'a` is instantiated to `int`, `'b` to `string`
- Result is

```
[(1,"a");(2,"b");(3,"c")]
```

of type `(int * string) list`

Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a * 'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- Distinct type variables *do not need to be* instantiated differently:

```
zip [1;2;3] [4;5;6]
```

- Here, `'a` is instantiated to `int`, `'b` to `int`
- Result is

```
[(1,4);(2,5);(3,6)]
```

of type `(int * int) list`

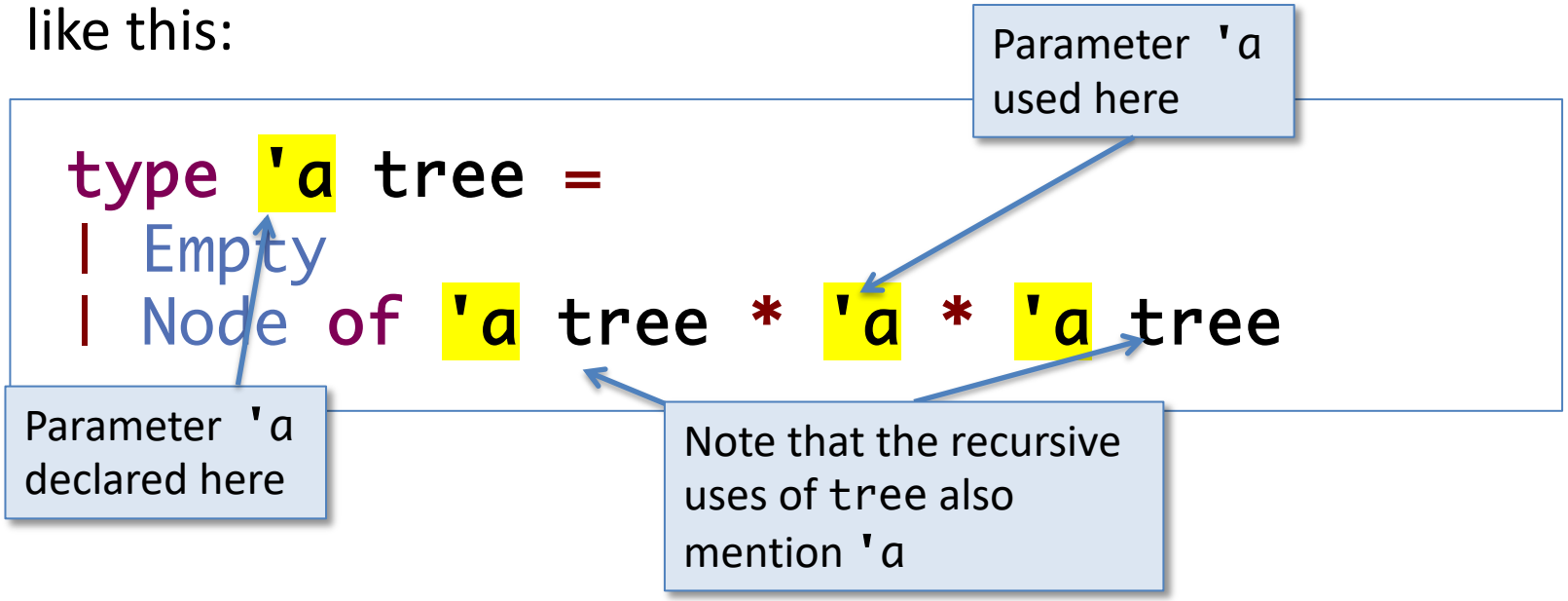
Intuition: OCaml tracks instantiations of type variables (`'a` and `'b`) and makes sure they are used consistently.

User-Defined Generic Datatypes

- Recall our integer tree type:

```
type tree =
| Empty
| Node of tree * int * tree
```

- We can define a generic version by adding a type parameter, like this:



User-Defined Generic Datatypes

BST operations can be generic too; the only change is to the type annotations

```
(* Insert n into the BST t *)

let rec insert (t:'a tree) (n:'a) : 'a tree =
  begin match t with
  | Empty -> Node(Empty,n,Empty)
  | Node(lt,x,rt) ->
      if x = n then t
       else if n < x then Node(insert lt n, x, rt)
       else Node(lt, x, insert rt n)
  end
```

Equality and comparison are *generic* — they work for *any* type of data, even strings, lists, and tuples!

8: Does the following function typecheck?

0

yes

0%

no

0%

Does the following function typecheck?

```
let f (l : 'a list) : 'b list =
begin match l with
| [] -> true::l
| _::rest -> 1::l
end
```

1. yes
2. no

Answer: no: even though the return type is generic, the two branches must agree (so that ‘b can be consistently instantiated).

8: Does the following code typecheck?

0

yes

0%

no

0%

Does the following code typecheck?

```
let f (x : 'a) : 'a =
  x + 1

;; print_endline (f “hello”)
```

1. yes
2. no

Answer: no, the type annotations and uses of f aren’t consistent.

However, it is a bit subtle: without the use (`f "hello"`) the code *would* be correct – so long as all uses of `f` provide only '`int`' the code is consistent! Despite the "generic" type annotation, `f` really has type `int -> int`.

First-class Functions

Higher-order Programs

or

How not to repeat yourself, Part II.

First-class Functions

- You can pass a function as an *argument* to another function

```
let twice (f:int->int) (x:int) : int =
  f (f x)

let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

function type: *argument* of type `int` and *result* of type `int`

The function `add_one` is passed as an argument to twice!

- You can *return* a function as the result of another function

```
let make_incr (n:int) : int->int =
  let helper (x:int) : int =
        n + x
  in
  helper
let y = twice (make_incr 1) 3
```

Result type is an arrow (function) type

Argument is an expression that produces a function

Functions as Data

You can store functions in data structures!

```
let add_one   (x:int) : int = x+1
let add_two   (x:int) : int = x+2
let add_three (x:int) : int = x+3

let func_list : (int -> int) list =
   [ add_one; add_two; add_three ]
```

a list of functions

```
let func_list1 : (int -> int) list =
   [ make_incr 1; make_incr 2; make_incr 3 ]
```

a list of expressions that produce functions

Simplifying First-Class Functions

```
let twice (f:int->int) (x:int) : int =
  f (f x)

let add_one (z:int) : int = z + 1
```

twice add_one 3	
⟼ add_one (add_one 3)	*substitute add_one for f, 3 for x*
⟼ add_one (3 + 1)	*substitute 3 for z in add_one*
⟼ add_one 4	*3+1⇒4*
⟼ 4 + 1	*substitute 4 for z in add_one*
⟼ 5	*4+1⇒5*

Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =
  let helper (x:int) : int = n + x in
  helper
```

```
make_incr 3
```

substitute 3 for n

```
⟼ let helper (x:int) = 3 + x in helper
⟼ ???
```

Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =
  let helper (x:int) : int = n + x in
  helper
```

```
make_incr 3
```

substitute 3 for n

```
⟼ let helper (x:int) = 3 + x in helper
⟼ fun (x:int) -> 3 + x
```

anonymous function value

keyword “fun”

“->” after arguments,
no return type annotation

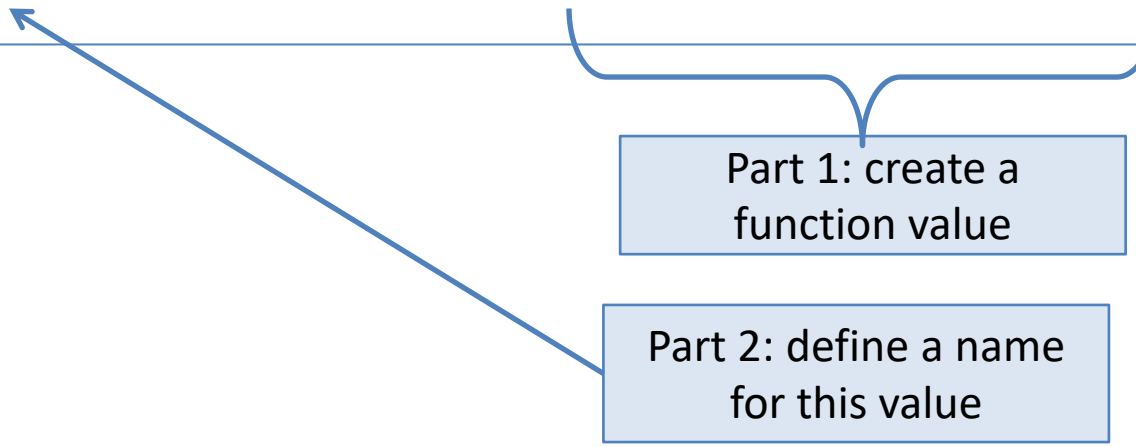
Function values

A standard function definition…

```
let add_one (x:int) : int = x+1
```

…is really an abbreviation for this:

```
let add_one : int->int = fun (x:int) -> x+1
```



The two definitions of add_one have exactly the same type and behave exactly the same. (The first is just an abbreviation* for the second.)

*computer scientists like to use the term "*syntactic sugar*" for such abbreviations. Such abbreviations make it "sweeter "to write simpler, tastier code, which "desugars" into more complex stuff

Anonymous functions

```
let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

```
let y = twice (fun (z:int) -> z+1) 3
let w = twice (fun (z:int) -> z+2) 3
```

a function value, passed as an argument to twice

Function Types

- Functions have types that look like this:

$$t_{in} \texttt{ -> } t_{out}$$

- Examples:

```
int -> int

int -> bool * int

int -> int -> int

(int -> int) -> int
```

Parentheses matter!

```
int -> int -> int
  = int -> (int -> int)
  ≠ (int -> int) -> int
```

int input

function input

Function Types

Hang on… did we just say that

```
int -> int -> int
```

and

```
int -> (int -> int)
```

mean the same thing??

Yes!

2 = 1 + 1

A function that takes *two* arguments…

```
int -> int -> int
```

has the same type as a function that takes *one* argument and returns a function that takes *one* argument

```
int -> (int -> int)
```

This is actually useful!

Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```

into parts:

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

define a variable with that value

create a function value

that returns a function value

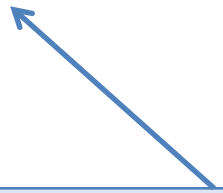
The two definitions of sum have the same type and behave the same!

```
let sum : int -> int -> int
```

Partial Application

```
let sum (x : int) (y:int) : int = x + y
```

```
sum 3
⟼ (fun (x:int) -> fun (y:int) -> x + y) 3      definition
⟼ fun (y:int) -> 3 + y                         substitute 3 for x
```



the result of a “partially applied function” is itself a function (that can later be applied)

What good is partial application?

Consider this *filter* function:

```
let rec filter (p:'a -> bool) (l:'a list) : 'a list =
  begin match l with
  | [] -> []
  | x::xs -> if p x then x :: (filter p xs) else (filter p xs)
  end
```

`filter` selects elements of a list based on a predicate p.

```
let larger  = filter (fun x -> x > 10)
let smaller = filter (fun x -> x <= 10)
```

We can create specialized "instances" by partial application…

```
larger  [1;17;120;4;10] ⇒ [17;120]
smaller [1;17;120;4;10] ⇒ [1;4;10]
```

and use them as ordinary list-processing functions.

Upshot: higher-order functions like filter are a very useful way to structure library interfaces…

8: What is the value of this expression?

0

A. 1

0%

B. True

0%

C. fun (y:int) -> if true then 1 else y

0%

D. fun (x:bool) -> if x then 1 else y

0%

What is the value of this expresssion?

```
let f (x:bool) (y:int) : int =
    if x then 1 else y in

f true
```

1. `1`
2. `true`
3. `fun (y:int) -> if true then 1 else y`
4. `fun (x:bool) -> if x then 1 else y`

Answer: 3

8: What is the value of this expression?

0

1 0%

2 0%

3 0%

4 0%

5 0%

6 0%

What is the value of this expression?

```
let f (g : int->int) (y: int) : int =
    g 1 + y in

f (fun (x:int) -> x + 1) 3
```

1. 1
2. 2
3. 3
4. 4
5. 5

Answer: 5

8: What is the type of this expression?

0

1. int
0%

2. int -> int
0%

3. int -> int -> int
0%

4. (int -> int) -> int -> int
0%

5. Ill-typed
0%

What is the type of this expression?

```
let f (g : int->int) (y: int) : int =
   g 1 + y in

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

1. `int`
2. `int -> int`
3. `int -> int -> int`
4. `(int -> int) -> int -> int`
5. `ill-typed`

Answer: 2