Programming Languages and Techniques (CIS120)

Lecture 8

Generics & First-class functions

Chapters 8 and 9

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, trees, and other datatypes don’t depend on the contents, only on the 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
```

The functions are *identical*, except for the type annotation.

Notation for Generic Types

- OCaml allows defining functions with *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 fills in `'a` whenever length is used

Generic List Append

Note that the two input lists must have the *same* type of elements.

The return type is 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 lists these are?

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`

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

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:

Parameter 'a used here

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

Note that the recursive uses of tree also mention 'a

User-Defined Generic Datatypes

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

```
(* 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.

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).

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
```

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)
⟼  add_one (3 + 1)
⟼  add_one 4
⟼  4 + 1
⟼  5
```

substitute add_one for f, 3 for x

substitute 3 for z in add_one

3+1⇒4

substitute 4 for z in add_one

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

Named function values

A standard function definition…

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

really has two parts:

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

define a name for the value

create a function value

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

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
```

an expression that is a function value

Function Types

- Functions have types that look like this:

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

- Examples:

```
int -> int

int -> bool * int

int -> int -> int            int input

(int -> int) -> int          function input
```

Function Types

- Functions have types that look like this:

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

- Examples:

```
int -> int

int -> (bool * int)

int -> (int -> int)          int input

(int -> int) -> int          function input
```

Parentheses matter!

int -> int -> int is equivalent to int -> (int -> int) but not to (int -> int) -> int

Function Types

Hang on… did we just say that

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

and

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

mean the same thing??

Yes!

2 = 1 + 1

The type of a function that takes *two* arguments

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

is the same as the type of a function that takes *one* argument and returns a function that takes *one* argument.

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

We can exploit this in OCaml!

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 have the same type and behave exactly 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
↦ fun (y:int) -> 3 + y
```

definition

substitute 3 for x

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

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

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

What is the type of this expresssion?

```
[ (fun (x:int) -> x + 1);
  (fun (x:int) -> x - 1) ]
```

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

Answer: 3