Programming Languages and Techniques (CIS120)

Lecture 8
September 17, 2018

Generics & First-class functions
Chapters 8 and 9
• Homework 2
  – Due tomorrow night at 11:59pm

• Homework 3 available soon
  – Practice with BSTs, generic functions, first-class functions and abstract types

• Reading: Chapters 8, 9, and 10 of the lecture notes
Wow, implementing BSTs took quite a bit of typing... Do we have to repeat it all again if we want to use BSTs containing strings, or characters, or floats?

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.
They are "generic" with respect to the input type.
Notation for Generic Types

• OCaml provides syntax for functions with *generic* types

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

• Notation: `'a` is a *type variable*; 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
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

Note that the two input lists must have the same type of elements.
The return type is the same as the inputs.
Zip function

```ocaml
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
```

• Call with two lists
  
  \[
  \text{zip } [1;2;3] \ ["a";"b";"c"] \\
  \mapsto [(1,"a"); (2,"b"); (3,"c")] \\
  \]

• Does it matter what type of lists these are?
Distinct type variables can be instantiated differently:

\[
\text{zip } [1;2;3] \ ["a";"b";"c"]
\]

Here, 'a is instantiated to int, 'b to string

Result is

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

of type (int * string) list
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:

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

Note that the recursive uses also mention 'a.

Parameter 'a used here
User-Defined Generic Datatypes

• BST operations can be generic too; 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 too.
Does the following function typecheck?

When poll is active, respond at PollEv.com/120fall18

Text 120FALL18 to 22333 once to join

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

```ocaml
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 function typecheck?

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

;; print_endline (f "hello")
```
Does the following code typecheck?

```ocaml
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:

```plaintext
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
```

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

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

• You can *return* a function as the result of another function.

```plaintext
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

CIS120
Functions have types that look like this:

\[ t_{in} \rightarrow t_{out} \]

Examples:

- \( \text{int} \rightarrow \text{int} \)
- \( \text{int} \rightarrow \text{bool} \ast \text{int} \)
- \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \quad \text{int input} \)
- \( (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \quad \text{function input} \)
Function Types

• Functions have types that look like this:

\[ t_{\text{in}} \rightarrow t_{\text{out}} \]

• Examples:

\[
\begin{align*}
\text{int} & \rightarrow \text{int} \\
\text{int} & \rightarrow (\text{bool} \times \text{int}) \\
\text{int} & \rightarrow (\text{int} \rightarrow \text{int}) \\
(\text{int} \rightarrow \text{int}) & \rightarrow \text{int}
\end{align*}
\]

Parentheses matter!

\[ \text{int} \rightarrow \text{int} \rightarrow \text{int} \] is equivalent to
\[ \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \] but not to
\[ (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \]
First-class Functions

• You can store functions in data structures

```ocaml
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

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

Not just names, use any expression that produces a function
let rec compose (fs:(int->int) list) (x:int) : int =
    begin match fs with
    | []  -> x
    | f::rest -> f (compose rest x)
    end

let ans : int = compose func_list1 0
Simplifying First-Class Functions

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

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

let 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

```ocaml
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
→ ???
```
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...

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

really has two parts:

```plaintext
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.
Anonymous functions

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

an expression that is a function value
Multiple Arguments

We can decompose a standard function definition into parts:

\[
\text{let } \text{sum} (x : \text{int}) (y : \text{int}) : \text{int} = x + y
\]

\[
\text{let } \text{sum} = \text{fun } (x : \text{int}) \to \text{fun } (y : \text{int}) \to x + y
\]

This allows us to 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:

\[
\text{let } \text{sum} : \text{int} \to \text{int} \to \text{int}
\]
Partial Application

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

sum 3
\rightarrow (\text{fun } (x:\text{int}) \rightarrow \text{fun } (y:\text{int}) \rightarrow x + y) \ 3 \quad \text{definition}
\rightarrow \text{fun } (y:\text{int}) \rightarrow 3 + y \quad \text{substitute 3 for x}