Programming Languages and Techniques (CIS120)

Lecture 8

Feb 4, 2019

Generics & First-class functions
Chapters 8 and 9
Announcements

• Homework 2
  – Due tomorrow night at 11:59pm

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

• Reading: Chapters 8, 9, and 10 of the lecture notes
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:

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

```ocaml
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.
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

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

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

The return type is the same as the inputs.

Pattern matching works over generic types!

In the body of the branch:
  h has type 'a
  tl has type 'a list

The return type is the same as the inputs.
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 function

zip [1;2;3] ["a";"b";"c"]
→ [(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
• Recall our integer tree type:

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

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

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

Note that the recursive uses of `tree` also mention `'a`.

Parameter `'a` used here
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?

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?

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

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

```haskell
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:

```ocaml
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
```

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

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

*Function type:* argument of type int and result of type int

*Argument is an expression that produces a function*

*The function add_one is passed as an argument to twice!*
Functions as Data

- You can store functions in data structures

```haskell
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

```haskell
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

$\rightarrow$ add_one (add_one 3)  
  substitute add_one for f, 3 for x

$\rightarrow$ add_one (3 + 1)  
  substitute 3 for z in add_one

$\rightarrow$ add_one 4  
  3+1⇒4

$\rightarrow$ 4 + 1  
  substitute 4 for z in add_one

$\rightarrow$ 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

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

```
make_incr 3
```

(substitute 3 for n)

\[
\rightarrow \text{let helper (x:int) = 3 + x in helper}
\]

\[
\rightarrow \text{fun (x:int) -> 3 + x}
\]

Anonymous function value

- Keyword "fun"
- "->" after arguments
  no return type annotation
Named function values

A standard function definition...

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

really has two parts:

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

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

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

- Examples:

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

• Functions have types that look like this:

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

• Examples:

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

Parentheses matter!

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

\[ \text{int input} \]

\[ \text{function input} \]
Function Types

Hang on... did we just say that

\[ \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

and

\[ \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]

mean the same thing??

Yes!
Multiple Arguments

We can decompose a standard function definition

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

into parts

```plaintext
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

```plaintext
let sum : int -> int -> int
```
Partial Application

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

```r
cis120
sum 3
⇒ (fun (x:int) -> fun (y:int) -> x + y) 3  definition
⇒ fun (y:int) -> 3 + y substitute 3 for x
```
What is the value of this expression?

1. A
2. B
3. C
4. D
What is the value of this expression?

```
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?
What is the value of this expression?

```ocaml
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?

```haskell
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 expression?

\[
\begin{array}{l}
(fun (x:int) \rightarrow x + 1); \\
(fun (x:int) \rightarrow x - 1)
\end{array}
\]

1. int
2. int \rightarrow int
3. (int \rightarrow int) list
4. int list \rightarrow int list
5. ill typed

Answer: 3