

Logic and Computation Notes

William Traub

Contents

1	1/7, Defining Functions	1
1.1	Course Plan	1
2	1/8 Functional Programming	2
2.1	OCaml notes	2
3	1/12	3
3.1	Functions	3
3.2	Specification	3

1 1/7, Defining Functions

mk-requests:(ListID)

$$\forall S, \forall r, \forall a, \exists R, \text{mk-requests}(S, r, a) = R$$

$$\forall S, \forall r, \forall a, \forall R, \text{mk-requests}(S, r, a) = R \implies \sum_{(\text{fr}, \text{to}, \text{amt})} \text{amt} = a$$

$$\forall S, \forall r, \forall a, a < 0 \implies \nexists R, \text{mk-requests}(S, r, a) = R$$

Advantages of proofs:

- Computable - we can turn these statements into tests and proofs
- Independent - decoupled from the code itself. We don't need to worry about runtime in order to test.

Coding a property?

Because we are universally quantifying, we would generally need to enumerate over the entire set of each \forall

Write a testing harness to substitute for the enumeration. Use randomness to generate test data.

1.1 Course Plan

Modules:

1. Programming and PBT
2. Engineering with Abstractions
3. Analyzing Programming Languages

2 1/8 Functional Programming

What is functional programming?

When we write software and it gets complicated, it helps to break it down into smaller pieces. With functional programming, we get 'glue' where you are composing functions with each other.

Goal: get us to write a fizzbuzz function that takes in an int and gives a fizzbuzz string output

```
function fizzbuzz:  
    input n number  
    output string  
  
    if (n % 3) = (n % 5) = 0 return "fizzbuzz"  
    else if n % 3 = 0 return "fizz"  
    else if n % 5 = 0 return "buzz"  
    else return n as a string
```

2.1 OCaml notes

- in the REPL (utop) terminate expressions with ;;
- standard arithmetic, carrot is string concat
- must use +. to add floats
- use float of int to cast float to int
- double quotes for string
- not binds tighter than other
- if false then 2 else 3 (ternary must have same type)
- comparison operators, single =
- let x = 4 in x / 2
- overwriting variables instead of mutating them
- ---

```
let <var> = <expr> in <body> ;;  
< store, let <var> = <expr>> -> <store[<var> ;;
```

3 1/12

```
let x = 19 in let x = x < 10 in x;;
(* goes to *)  
  
let x = 3 in x + x;; (* -> 6 *)
let x = 4 in let y = 5 in x * y;; (* 20 *)
let x = 19 in (let x = x < 10 in x);;
```

3.1 Functions

Functions are boring and everywhere.

```
(* anonymous function *)
let f = fun (x : int) -> x + 1;;
f : int -> int -> int
```

OCaml is right-associative so

```
let f = fun (x : int) -> fun (y : int) -> x * y;;
let f (x : int) (y : int) = x * y;;
```

These functions are isomorphic and will result in about the same thing.

```
let f (g : int -> int) (y : int) = g y;;
let adder = f (fun (y : int) )
```

Lexical Scope

```
let y = 1 in
let f x = x + y in
let y = 2 in
f 3;;
```

There is closure used where the function substitutes y into its body and just becomes $f(x) = x + 1$.

3.2 Specification

We have high-order functions that can take in other functions.

```
let max3 (i : int) (j : int) (k : int) =
  if i >= j
  then if i >= k
    then i
    else k
  else if l >= k
    then j
    else k
```

We want to specify the function in formal logic.

$$\begin{aligned} \forall i, j, k \in \mathbb{Z} \\ max3(i, j, k) \geq i \wedge \\ max3(i, j, k) \geq j \wedge \\ max3(i, j, k) \geq k \wedge \\ max3(i, j, k) \in \{i, j, k\} \end{aligned}$$