Logic and Computation Notes

William Traub

Contents

1 1/7, Defining Functions

mk-requests:(ListID)
$\forall S, \forall r, \forall a, \exists R,\ \text{mk-requests}(S, r, a) = R$
$\forall S, \forall r, \forall a, \forall R, \text{mk-requests}(S, r, a) = R \implies \Sigma_{(\text{fr, to, amt})} amt = a$
$\forall S, \forall r, \forall a, a < 0 \implies \nexists R,\ \text{mk-requests}(S, r, a) = R$
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 (ternerary 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.

$$\forall i, j, k \in \mathbb{Z}$$

$$\begin{gathered} max3(i,j,k) \geq i \wedge \\ max3(i,j,k) \geq j \wedge \\ max3(i,j,k) \geq k \wedge \end{gathered}$$

$$max3(i,j,k) \in \{i,j,k\}$$

4 1/14 Pattern Matching and Data

4.1 Functions

```
fun (x : int -> string) -> (string -> string) (z : int)
```

Functions are everything. You can think of let bindings as function application.

```
let x = 19 in              (fun (x) ->
let y = x < 10               (fun (y) -> y
in y ;;                      ) (x < 10)
                           ) 19
```

What is pattern matching?

```
let show (x : int) =
  if x = 1 then "one" else
  if x = 2 then "two" else
  if x = 3 then "three" else
  "> three";;

let show (x : int) =
  match x with
  | 1 -> "one"
  | 2 -> "two"
  | 3 -> "three"
  | _ -> "> three";;
```

Because we have typing, we can more aggresively optimize. `https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/`
In OCaml, we have pairs or tuples. Always use pattern matching to unfold or destruct.

```
let student_of_2800 (name_stu : string * bool) : bool =
match (name, is_student) with
| (_, false) -> false
| (_, true) -> true ;;
```

You can have a nested match but you must wrap with parenthesis.

```
let shift_x (pt : int * int) (delta : int) : int * int =
  match pt with
    | (x, y) -> (x + delta, y)
    ;;

let shift_x ((x, y) : int * int) (delta : int) : int * int =
  (x + delta, y)
  ;;
```

Definition 1. *Sometimes we have basic types but sometimes we want to provide a light form of documentation. We can write a* ***type alias*** *to define our own.*

```
type point2d = int * int;;
let shift_x (point2d : int * int) (dx, dy : int * int) :
   point2d =
  match (xy, dxdy) with | (x, y), (dx, dy) -> (x + dx, y + dy)
     ;;
```

4.2 ADT's

Definition 2. *And **Algebraic Data Type (ADT)** is a composite data type. This is similar to a union or Enum. An example of a sumtype.*

```
type weekday =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday ;;

type two_days = (weekday * weekday);;

let next_weekday (d : weekday) : weekday =
  match d with
  | Monday  -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday -> Monday ;;

type my_bool =  | myTrue | myFalse ;;

type distance =
  | Inches of float
  | Feet of float
  | Yards of float ;;

let d1 = Inches 10. ;;
let d2 = Yards 12. ;;

let distance_to_inches (d : distance) : float =
match d with
  | Inches x -> x
  | Feet x -> x *. 12.
  | Yards x -> x *. 36. ;;

distance_to_inches d2 ;; (* returns )
```

```
type quantity =
  | Fraction of int * int
  | Float of float
  | Arbitrary of string ;;

let string_of_quantity (q : quantity) : string =
  match q with
  | Fraction (n, d) -> string_of_int n ^ "/" ^ string_of_int d
  | Float f -> string_of_float f
  | Arbitrary s -> s ;;
```

Record type in OCaml

```
type offset
  = { location : point2d ; delta distance * distance}

let small_offset = {location = (3, 3)
                   ; delta = (Float 3., Fraction (1, 2))}
let _ =
  match small_offset.delta with
    | (_, Fraction (_, _)) -> print_endline "fractional"
    | (_, Float _) -> print_endline "float"
    | (_, Arbitrary _) -> print_endline "arbitrary" ;;
```

5 1/15 Recap and Structural Recursion

```
type int_result =
  | IntOk of int
  | IntError of string ;;

type person = { name : string ; age : int; registered : bool} ;;
let alice = { name = "alice" ; age = 24 ; registered = true } ;;

let string_of_person (p : person) : string =
  p.name ^ " is " string_of_int p.age ;;

let other_string_of_person { name ; age = their_age ; _ } :
   string =
  name ^ " is " string_of_int their_age ;;
```

The million dollar mistake: nullity. Can be solved by options.

```
type int_option =
  | IntOpt of int
  | IntOptEmpty
```

5.1 Recursive types

Lists in OCaml

```
  [1 ; 2 ; 3]
```

Statically typed, can only have single type.

```
  type int_list =
    | Empty
    | Cons of int * int_list

  let intlist1 = Cons (2, (Cons (1, Empty)))

  let rec int_with_list_sum (xs : int_list) : int =
    match xs with
    | Empty -> 0
    | Cons (x, xs') -> x + int_list_sum xs'

  let rec int_list_length (xs : int_list) : int =
    | Empty -> 0
    | Cons (_x, xs) -> 1 + int_list_length xs

  let rec int_list_add1 (xs : int_list) : int_list =
    | Empty -> Empty
    | Cons (x, xs) -> Cons (x * 2, int_list_add1 xs)

  let rec int_list_mul2 (xs : int list) : int list =
    | [] -> []
    | x::xs -> (x * 2) :: (int_list_mul2 xs)
```

We can create a high-order function map for this

```
let rec map (f : int -> int) (xs : int list) : int list =
  match xs with
  | [] -> []
  | x::xs -> (f x) :: (map f xs)

let add1 = map (fun (x : int) -> x + 1)
let mul2 = map (fun (x : int) -> x * 2)
```

We can reduce or fold to find the sum of a list. We are taking a data structure and an accumulator and apply a binary function to the acc and each element of the structure.

```
let rec fold ( f : int -> int -> int) (acc : int) (xs : int
   list) : int_list =
  match xs with
  | [] -> acc
  | (x::xs') ->
    let acc' = f acc x in
    fold f acc' xs'

let sum = fold (fun (acc : int) (x : int) -> acc + x) 0
let product = fold (fun (acc : int) (x : int) -> acc * x) 1

let rec append (xs : int list) (ys : int list) : int list =
  match xs with
  | [] -> ys
  | (x :: xs') -> x :: append xs' ys

let rec reverse (xs : int list): int list =
  match xs with
  | [] -> []
  | (x :: xs') -> append (reverse xs') [x]
```

Functions sum and reverse are commutative. $\forall x \in$ int list, sum(reverse(xs)) = sum(xs)
How to test a list function.
Generating a random list: start with deciding the length.

```
let rec mk_list_of_length (len : int) : int list =
  if len <= 0 then [] else
    let hd = Random_int 500 in
    let tl = mk_list_of_length (len - 1) in
    hd :: tl

let list_gen () : int list =
  let len = Random.int 300 in
  mk_list_of_length len
```

We need to serialize a counterexample to create a tester

```
let prop_sum_rev (sum L int list -> int) (rev : int list -> int
    list) : int list option =
let xs = list_gen () in
if sum (reverse xs) = sum xs then None else
  Some xs
```

```
let rec do_tests (sum : int list -> int) (rev : int list -> int
    list) (n : int) =
  if  n <= 0 then
    print_endline "no issues"
  else
    match (prop_sum_rev sum rev) with
    | None -> do_tests sum rev (n - 1)
    | Some s -> print_endline("found a bug: " ^ stringify s)
```

6 1/21 - Specification cont.

We might use a comment to annotate a function;

```
(* return the larger of the two numbers *)
let max (i : int) (j : int) = ...
```

This can be written as $S = \{P | P$ implements "return the larger of two numbers" $\}$
This is somewhat ambiguous, the set is infinite.

```
(* sorts the input list *)
let rec sort (xs : int list) = ...

type row = {
  name : string;
  gid : string; (* global id *)
  lid : string (* local id *)
}

let rec sort_rows (xs : row list) : row list = ...
```

It's unclear how this needs to be sorted.
A data type is its own specification. $S = \{P | P$ has int list $\rightarrow$ int list$\}$. This set is also infinite.
We want something more powerfull then the second while still expressing the meaning of the first.

1. $\forall xs, ys,$ sort xs $= ys \implies \forall i, j, 0 \leq i < j <$ length $ys \implies ys[i] \leq ys[j]$
2. $\forall x,\ x \in ys \wedge x \in xs$
   or $\forall xs, ys,$ sort xs = ys $\implies \forall i, |xs\{i\}| = |ys\{i\}|$

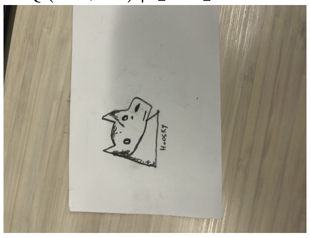
We can restate this as $S = \{P | \forall i, R(i, P(i))\}$.

7 1/22 - Formalization, Coding and Bug Hunting

Plan

- Formalize our sort specification
- Discuss computability
- Implement a the specification and find a bug

CODE3
We now have our implementation of our specification and can write a full test function.
CODE4
We take in our sort function as an input as we are validating if a certain function works.
Because we are talking about our computable test function, we should also specify that.
$S = \{P | \forall xs,\ \text{prop-sort-correct}(xs, P)\}$ where prop-sort-correct is int list $\rightarrow$ bool.
This is an undecidable problem so we must modify it. $S = \{(xs, P) |\ \text{prop-sort-correct}(xs, P)\}$



We were told to draw a pretty picture so here is mine.
PBT recipie:

1. Write down our logical spec as a program
2. Use randomness to perform (directed) sampling of the input values
3. If no counterexample: increase confidence and reexamine the generator and proposition
4. If the is a counterexample, the prop is invalid

CODE5 List init is like a racket buildlist
We are now making our testing harness to make forall use random sampling
CODE6
The final step in this bug hunt is to write a bad implementation of sort.

8 Type Inference, Modules

We have a function and we want to identify it's type signature.

```
fun a -> (snd a, fst a);;
  snd: X * Y -> Y
  fst: X * Y -> X
```

First we must look at the top-level expression. We can express this as
$X? * Y? \to C? * D?$
Where $a : X? * Y?$
$snd(a) : Y?$
$fst(a) : X?$ We can substitute the functions in and fill in the polymorphic types
$f :' a *' b \to' b *' a$.
There is a website to practice polymorphic type inference but I didn't get the url.
We have the sample code

```
  type 'a queue = 'a list
  let size : 'a queue -> int = failwith "..."

  type 'a q_ops = {
    push : 'a queue -> 'a -> 'a queue;
    peek : 'a queue -> 'a option;
  }
```

This is a queue data structure which allows adding an element to the end and looking at the front. This is an example of a module?
In ocaml, files are modules. These are similar to a Java class with static methods.

```
  let x : int Queue.t = create () ;;
  Queue.add 1234 x;;
  Queue.take_opt x;; (* int option = Some 1234 *)
```

Now to create a set, an unordered list of elements.

```
  type 'a set = 'a list
  let empty : 'a set = []

  let rec mem (x : 'a) (s : 'a set) : bool
    = match s with
    | [] -> false
    | x' :: s' -> if x' = x then true else mem x s'

  let rec remove (x : 'a) (s : 'a set) : 'a set
    = match s with
    | [] -> []
    | x' :: s' -> if x' = x the s' else x' :: remove x s'

  let add (el : 'a) (s : 'a set) : 'a set = el :: s
```

We care about module and interface design in order to ensure usability. When storing a set as a list, we allow the user to call List. commands on it.
In order to prevent this we can define an interface file.

9 2/4

$f :' a.'a \rightarrow' a$ is a universal quantifier $\forall$.
A module is equivalent to the existensial quantifier $\exists$. We are stating that there exists an instance of a type that follows a proposition, in this case the proposition is our interface.

When creating a module interface, you can comment out a function to make a private method.
We can create module types in files:

```
module type MyIntSetSig = sig
  type t
  val empty : t
  val mem : int -> t -> bool
  val add : int -> t -> t
  val remove : int -> t -> t
```

We use t for our type in order to unify the output type for all of our methods.

```
module type MyIntSet : MyIntSetSig = struct
  type t = int list
  let empty : t = []
  let mem : int -> t -> bool = ...
  val add : int -> t -> t = ...
  val remove : int -> t -> t = ...
```

This struct signature defines a structed module.
We can also open up the scope of the module using a "scoped open". We can pass in a statement (such as a function into the instance of the module).

```
let ex1 = MyIntSet.(add 1 (add 2 ex0))
```

We can design a recursive random generator for random sets.

```
let rec build_set_acc (n : int) (acc : MyIntSet.t) : MyIntSet.t
  = if n <= 0
    then acc
    else (build_set_acc (n - 1) (MyIntSet.add (Random.int 10)
       acc))

let build_set (n : int) = build_set_acc n MyIntSet.empty
```

Even though this module holds an int list, we cannot call int functions on it because it is abstract.
Some say that interface design is the most important part of programming so we need to make meaningful choices in their creation.
If we supply a large collection of helper functions in an interface the type somewhat loses semantic meaning. Specifiying very few functions makes the interface simpler and easier to read, but inflexible.

```
module type StringSetSig = sig
  type t
  val empty : t
  val mem : string -> t -> bool
  val add : string -> t -> t
```

```
    val remove : string -> t -> t
  end

  module HashSet : StringSetSig = struct
    type t = string list
    let empty = []
    let add x xs = Digest.string x :: xs
    let mem x xs = List.mem (Digest.string x) xs
    let remove x xs = List.filter ((<>) (Digest.string x)) xs
  end
```

We can use these abstract types to reason about invariance. Our interface can make specifications such as a bijection between two types.

```
  type bo = unit option
```

This specifies an isomorphism between the booleans and unit option.

```
  module type BSig = sig
    type t

  ende
```

10 Invariants and Generalizing Modules

We can create the type Even with the emergent inavariant property $\forall (n : even), n \mod 2 = 0$

We can equip our struct with a well-formedness property that ensures the invariance holds. This pushes some responsibility to the implementer and requires some hacking for testing.

Instead we can define a CounterSig module and corresponding Counter that can be a fresh variable and ”counts” how many times it gets bumped.

This counter lacks the ability to actually count as we do not have mutability. We can instead define CounterWithId with t as a record type that uses unique ids.

We currently have our count˙evens function that can only work on Counter types but we might want to ma

For example:

```
let read_network_string () : string = ...
let read_local () : string = ...

let process () : int =
  let s = read_network () in
  if s = "hello" the 0 else 1
```

This is an example of an application where we might want to make our process either read network or local. The desired solution to this problem is to parameterize our read function i.e.

```
let process (read: unit -> string) : int =
  let s = read () in
  if s = "hello" the 0 else 1
```

To extend this to modules we need a wrapper module. These are called functors (which are different from all other things called functors)

We can now write properties such as ∀ *(c : CounterSig), get (bump c) = get c + 1*

There are some strange properties with polymorphism in this context. In the past we worked with MySetSig. We are reliant on the definition of equality where $\{(3, 2)(2, 3)\} = \{(2, 3)\}$

Our solution is to design a new definition of equality, a functor called EqType that has the equals property. Now we can parameterize our set struct with EqType, using a second type elem = T.t

We can now create UnorderedPair with an equals function. Even though we cannot annotate the type, the compiler infers this based on structure.