CS 320 : Review of functional Programming in Ocaml

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- The second theory homework is is due today.
- We will post the third programming assignment later this week.

Plan for today

- Quick review of I/O on files.
- Review of partial application
- Review of polymorphism
- Program equivalence
- Review of Functional Programming part
- Intro to Part 2

Input/Output on files in OCaml

Input and Output Channels

The normal way of opening a file in OCaml returns a **channel**. There are two kinds of channels:

- channels that write to a file: type `out_channel`
- channels that read from a file: type `in_channel`

Four operations that will be useful are:

- Open input file: `open_in: string -> in_channel`
- Open out file: `open_out: string -> out_channel`
- Close input file: `close_in: in_channel -> unit`
- Close input file: `close_out: out_channel -> unit`

If you want to use a channel, you can use `let`, as usual.

Reading a line


```
let read_line (ine_in_channel) : string option =
match input_line inc with
| l -> Some l
| exception End_of_file -> None
```


Writing a line

```
Printf.fprintf ouc "%s\n" str
```

The types need to match

An example – copying one line:


```
let ic=open_in "tmp_input.txt" in
let oc=open_out "tmp_output.txt" in
let l=read_line ic in
let _ = match l with
  None -> ()
  | Some s -> Printf.fprintf oc "%s\n"  s
in
let _= close_in ic in
let _= close_out oc in
()
```

Functions

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

Example:

```
add_one : int -> int

3 + 4 : int

add_one (3 + 4) : int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:


```
add : int -> int -> int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:


```
add : int -> int -> int
```

Same as:

```
add : int -> (int -> int)
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : ???
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) :
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

(add (3 + 4)) 7 : int
```

Rule for type checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

add (3 + 4) 7 : int
```

Rule for type checking functions

Example:


```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string of int x
  else
    "hello"
;;

let y = 17;;
```



```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : ??

munge true (g munge) : ??
  g : ??
```

One key thing to remember

- If you have a function f with a type like this:

A -> B -> C -> D -> E -> F

- Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

f a1 : B -> C -> D -> E -> F (if a1 : A)

f a1 a2 : C -> D -> E -> F (if a2 : B)

f a1 a2 a3 : D -> E -> F (if a3 : C)

f a1 a2 a3 a4 a5 : F (if a4 : D and a5 : E)

Some examples


```
let f x = [x ^ "str"]
```



```
let foo g n =
  if n = "Hello"
  then ["1"]
  else (g n)
```



```
let partial = foo f


partial "str"
```

Some examples


```
let h = fun x -> fun y-> [x + y]
```



```
let rec foo1 n g =
  if n = 1
  then [[1]]
  else (g n) :: foo1 (n-1) g
```



```
let partial = fun x -> foo1 x h

partial 10
```

Polymorphism

Type of the undecorated map?


```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```



```
map : ('a -> 'b) -> 'a list -> 'b list
```

How about reduce?


```
let rec reduce (f:'a -> 'b -> 'b) (u:'b) (xs: 'a list) : 'b =
   match xs with
   | [] -> u
   | hd::tl -> f hd (reduce f u tl)
```


What's the most general type of reduce?


```
('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

Are these expressions well typed? What is their type?


```
let foo x y g = Some [g x (y+1)]
```



```
let foo f x y = f x y
```



```
let foo1 f x y = f x y in foo1 (fun u w -> u+w+1)
```



```
let foo2 f x y = f x y in foo2 (fun u w -> (u,w))
```



```
let foo3 f x y = f (x y)
```

```
let foo4 f x y = f (x y) in fun f -> foo4 f (fun x -> x)
```

```
let foo5 f = f f
```

Program equivalence

Are these two programs equivalent?


```
let rec foo1 l =
    match l with
    | [] -> []
    | hd::tl -> hd :: (foo1 tl)
```



```
let rec foo2 l =
    match l with
    | [] -> []
    | hd::tl -> hd :: (foo2 tl)
```

Are these two programs equivalent?


```
let rec foo1 l a =
    match l with
    | [] -> []
    | hd::tl -> hd :: (foo1 tl)
```


```
let rec foo2 l =
    match l with
    | [] -> []
    | hd::tl -> hd :: (foo2 tl)
```

Are these two programs equivalent?


```
let rec foo1 l =
    match l with
    | [] -> [1]
    | hd::tl -> hd :: (foo1 tl)
```


```
let rec foo2 l =
    match l with
    | [] -> []
    | hd::tl -> hd :: (foo2 tl)
```

When are two programs equivalent?

Two programs are (operationally) equivalent if and only if:

- they have the same type
- for every possible input they return the same output

Here we are assuming that the only way to interact with a program is through its inputs. This is true for pure functional languages, it requires more care for languages with side-effects.

TopHat Q1-Q4

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Reasoning about definitions

Reasoning About Definitions

We can factor this program


```
let square_all ys =
  match ys with
    | [] -> []
    | hd::tl -> (square hd)::(square_all tl)
```


into this program:


```
let square_all = map square
```

assuming we already have a definition of map

Reasoning About Definitions


```
let square_all ys =
  match ys with
    | [] -> []
    | hd::tl -> (square hd)::(square_all tl)
```



```
let square_all = map square
```


Goal: Rewrite definitions so my program is simpler, easier to understand, more concise, …

Question: What are the reasoning principles for rewriting programs without breaking them? For reasoning about the behavior of programs? About the equivalence of two programs?

I want some *rules* that never fail.

Simple Equational Reasoning

Rewrite 1 (Function de-sugaring):

```
let f x = body        ==   let f = (fun x -> body)
```


Rewrite 2 (Substitution):

if **arg** is a value or, when executed, **will always terminate without effect** and produce a value

```
(fun x -> ... x ...) arg   ==   ... arg ...
```

roughly: all occurrences of x replaced by arg (though getting this *exactly* right is shockingly difficult)

Rewrite 3 (Eta-expansion):

```
let f = def        ==   let f x = (def) x
```

if f has a function type

chose name x wisely so it does not shadow other names used in def

Eta-expansion is an example of Leibniz's law

Gottfried Wilhelm von Leibniz
German Philosopher
1646 - 1716

Leibniz's law:

If every predicate possessed by x is also possessed by y and vice versa, then entities x and y are identical. Frequently invoked in modern logic and philosophy.

Rewrite 3 (Eta-expansion):

```
let f = def
```

==

```
let f = fun x -> (def)x
```

if f has a function type

chose name x wisely so it does not shadow other names used in def

Eliminating the Sugar in Map


```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Eliminating the Sugar in Map


```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```



```
let rec map =
  (fun f ->
    (fun xs ->
       match xs with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)))
```

Consider square_all


```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

let square_all =
    map square
```

Substitute map definition into square_all


```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

let square_all =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl)
        )
    ) square
```

Substitute map definition into square_all


```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

let square_all =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl)
        )
    ) square
```

Substitute map definition into square_all


```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

let square_all =
    (fun f ->
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (f hd)::(map f tl)
        )
    ) square
```

Substitute Square


```
let rec map =
   (fun f ->
     (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (f hd)::(map f tl)))

let square_all =
    (
        (fun xs ->
            match xs with
            | [] -> []
            | hd::tl -> (square hd)::(map square tl)
        )
```

argument square substituted for parameter f

Expanding map square


```
let rec map =
  (fun f ->
    (fun xs ->
       match xs with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)))

let square_all ys =
      (fun xs ->
         match xs with
         | [] -> []
         | hd::tl -> (square hd)::(map square tl)
      ) ys
```

add argument
via eta-expansion

Expanding map square


```
let rec map =
  (fun f ->
    (fun xs ->
       match xs with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)))
```

```
let square_all ys =
```

substitute again (argument ys for parameter xs)

```
          match ys with
          | [] -> []
          | hd::tl -> (square hd)::(map square tl)
```

So Far


```
let rec map f xs =
      match xs with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)

let square_all xs = map square xs
```



```
let square_all ys =
  match ys with
     | [] -> []
     | hd::tl -> (square hd)::(map square tl)
```


proof by simple rewriting unrolls definition once

Next Step


```
let rec map f xs =
      match xs with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)

let square_all xs = map square xs
```

```
let square_all ys =
  match ys with
     | [] -> []
     | hd::tl -> (square hd)::(map square tl)
```

```
let square_all ys =
  match ys with
     | [] -> []
     | hd::tl -> (square hd)::(square_all tl)
```

proof by simple rewriting unrolls definition once

proof *by induction* eliminates recursive function map

Summary

We saw this:


```
let rec map f xs =
        match xs with
        | [] -> []
        | hd::tl -> (f hd)::(map f tl);;

let square_all ys = map square
```


Is equivalent to this:


```
let square_all ys =
  match ys with
    | [] -> []
    | hd::tl -> (square hd)::(map square tl)
```


Morals of the story:

(1) OCaml's *hot* (higher-order, typed) functions capture recursion patterns

(2) we can figure out what is going on by *equational reasoning*.

(3) ... but we typically need to do *proofs by induction* to reason about recursive (inductive) functions

TopHat Q5-Q8

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Why study functional programming?

1. Learning a different style of programming teach you that programming transcends programming in a language/style
2. Functional languages predict the future (several concepts adopted in other languages)
3. Functional languages are more and more used in industry
4. Functional languages help writing correct code (types, function encapsulation, equational reasoning, etc.)

Thinking Functionally

In imperative languages like Java or C, you get most work done by changing something


```
temp   = pair.x;
pair.x = pair.y;
pair.y = temp;
```


Commands modify or change the state – in this case an existing data structure (pair)

In functional languages you get most work done by producing something new

```
let
  (x,y)  = pair
in
  (y,x)
```

Commands analyze an existing data (pair) and produce a new data (y,x)

Writing Functions Over Typed Data

- Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. Deconstruct input data structures
  5. Build new output values
  6. Clean up by identifying repeated patterns

- For option types:

when the input has type t option, deconstruct with:

```
match … with
  | None -> …
  | Some s -> …
```

when the output has type t option, construct with:

```
Some (…)
```

```
None
```

Factoring Code in OCaml

A higher-order function captures the recursion pattern:


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


We can use an *anonymous* function instead.

Originally, Church wrote this function using λ instead of **fun**: (λx. x+1) or (λx. x*x)

Uses of the function:


```
let inc_all xs = map (fun x -> x + 1) xs


let square_all xs = map (fun y -> y * y) xs
```

More on Anonymous Functions

Function declarations:


```
let square x = x*x

let add x y = x+y
```


are *syntactic sugar* for:


```
let square = (fun x -> x*x)
```



```
let add = (fun x y -> x+y)
```

In other words, *functions are values* we can bind to a variable, just like 3 or “moo” or true.

Functions are 2nd class no more!

Curried Functions

Currying: *verb. gerund or present participle*

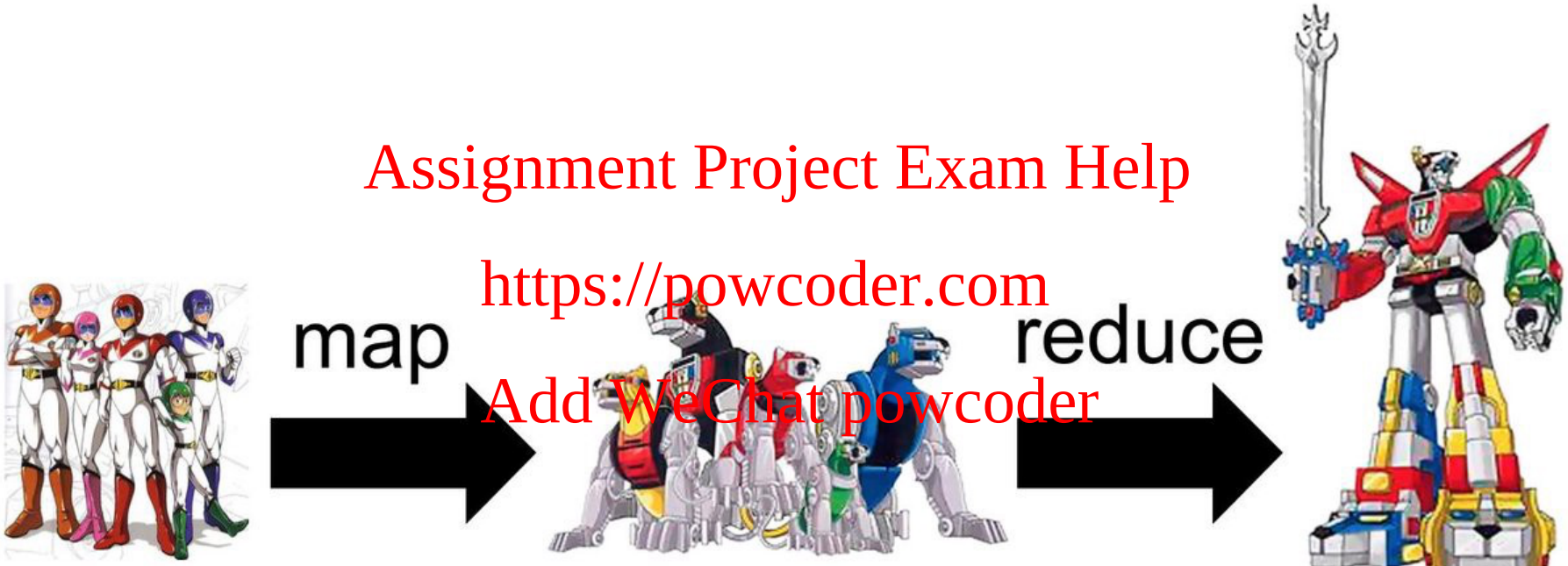
(1) to prepare or flavor with hot-tasting spices

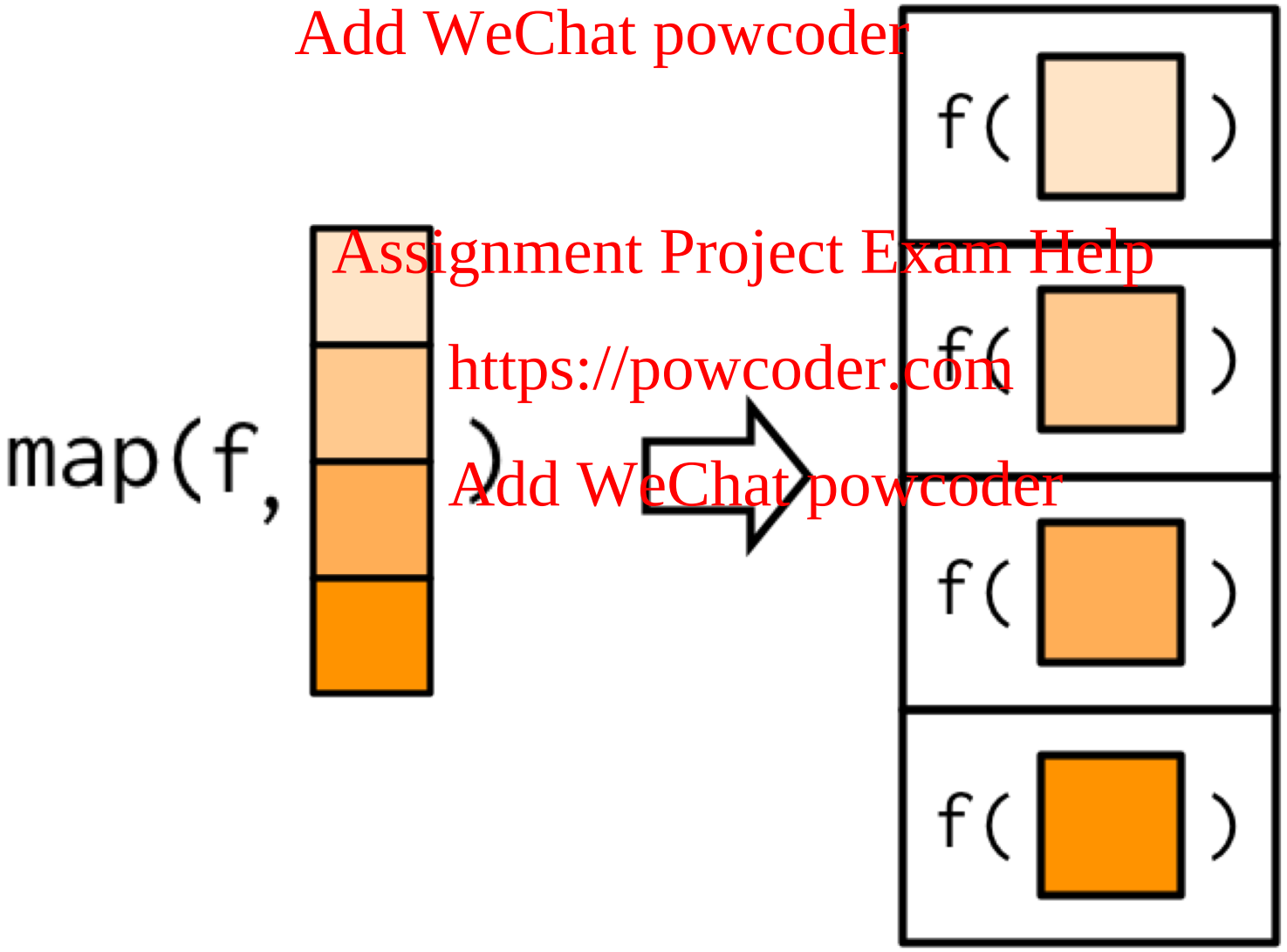
(2) to encode a multi-argument function using nested, higher-order functions.

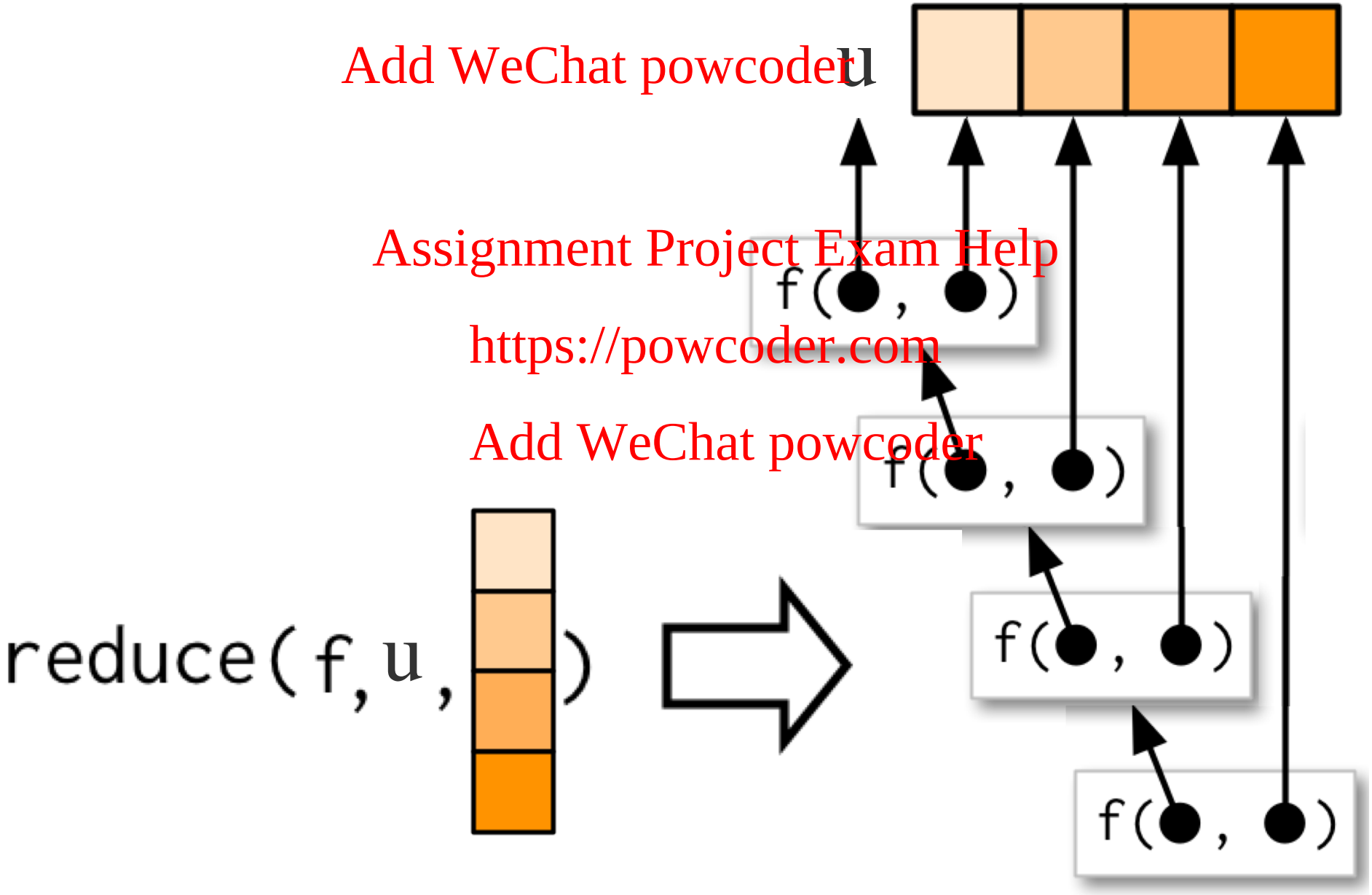
(1)

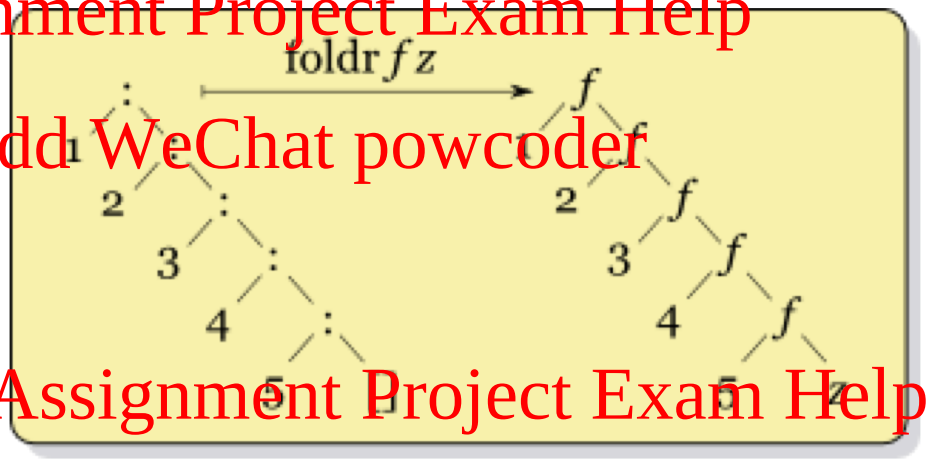
(2)

```
fun x -> (fun y -> x+y) (* curried *)
fun x y -> x + y        (* curried *)
fun (x,y) -> x+y        (* uncurried *)
```





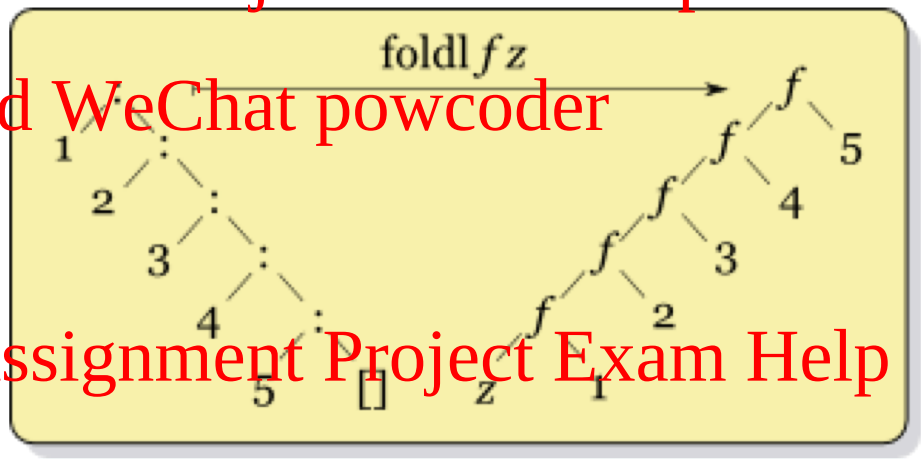





```
let rec fold_right (f: ‘a -> ‘b -> ‘b) (xs: ‘a list) (z: ‘b) : ‘b =
match  xs with
   [] -> z
   | hd::tl -> f hd (fold_right f tl z)



#  fold_right (+) [1;2;3;4;5] 0
```




```
let rec fold_left l (f: ‘b -> ‘a -> ‘b) (z: ‘b) (xs: ‘a list) : ‘b =
match  xs with
   [] -> z
 | hd::tl -> fold_left f  (f z hd) tl



#  fold_left (+) 0 [1;2;3;4;5]
```

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```



```
type color = Blue | Yellow | Green | Red
```

- Creating values:

```
let b1 : my_bool = Tru
let b2 : my_bool = Fal
let c1 : color = Yellow
let c2 : color = Red
```

use constructors to create values

Data types


```
type color = Blue | Yellow | Green | Red
```



```
let c1 : color = Yellow
let c2 : color = Red
```


- Using data type values:


```
let print_color (c:color) : unit =
  match c with
   | Blue ->
   | Yellow ->
   | Green ->
   | Red ->
```


use pattern matching to determine which color you have; act accordingly

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree


```
type key = string
type value = int
type tree =
  Leaf
| Node of key * value * tree * tree
```

Type Soundness

"*well typed programs do not go wrong*"

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc.

(but not immune to all bugs!)

Safe languages: ML, Java, Python, …

Unsafe languages: C, C++, Pascal

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Type of the undecorated map?


```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b lis
```


We often use greek letters like α or β to represent type variables.

Read as:

- for any types 'a and 'b,
- if you give map a function from 'a to 'b,
- it will return a function
  - which when given a list of 'a values
  - returns a list of 'b values.

Simple Equational Reasoning

Rewrite 1 (Function de-sugaring):

```
let f x = body        ==   let f = (fun x -> body)
```


Rewrite 2 (Substitution):

if **arg** is a value or, when executed, **will always terminate without effect** and produce a value

```
(fun x -> ... x ...) arg   ==   ... arg ...
```

roughly: all occurrences of x replaced by arg (though getting this *exactly* right is shockingly difficult)

Rewrite 3 (Eta-expansion):

```
let f = def        ==   let f x = (def) x
```

if f has a function type

chose name x wisely so it does not shadow other names used in def

Summary

We saw this:


```
let rec map f xs =
        match xs with
        | [] -> []
        | hd::tl -> (f hd)::(map f tl);;

let square_all ys = map square
```


Is equivalent to this:


```
let square_all ys =
  match ys with
    | [] -> []
    | hd::tl -> (square hd)::(map square tl)
```


Morals of the story:

(1) OCaml's *hot* (higher-order, typed) functions capture recursion patterns

(2) we can figure out what is going on by *equational reasoning*.

(3) ... but we typically need to do *proofs by induction* to reason about recursive (inductive) functions

TopHat Q9-Q21