Lecture 2

Categories

Category Theory

- You can use Categories to model Logic.
- You can use Categories to model Lambda calculus.
- Logic = Categories = Lambda calculus.
- Here is a category

$$\begin{tikzcd}
X \arrow[loop left, "id_A"] \arrow[r, "f"] \arrow[rr, bend right, "f\,;\,g"'] & Y \arrow[r, "g"] & Z
\end{tikzcd}$$

Category Theory

Object

Arrow

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[loop left, "id_A"] \arrow[rr, bend right, "f ; g"'] & Y \arrow[r, "g"] & Z \end{tikzcd}$$

$id_A ; f = f = f ; id_B$ (Identity function)

$(f ; g) ; h = f ; (g ; h)$ (Composition)

Products

- The two most important structures in Programming Languages (Or the most two important logical connectives)

$$\begin{tikzcd}
& Z \arrow[dl, "f"'] \arrow[d, dashed, "{<f,g>}"] \arrow[dr, "g"] & \\
X & X*Y \arrow[l, "fst"] \arrow[r, "snd"'] & Y
\end{tikzcd}$$

- For every object X, and every object Y, there is an object X times Y ($X * Y$)

Products

$$\begin{tikzcd}
& Z \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[d, dashed, "{<f,g>}"] & \\
X & X*Y \arrow[l, "fst"] \arrow[r, "snd"'] & Y
\end{tikzcd}$$

$$<f,g> ; fst = f$$

$$<f,g> ; snd = g$$

- **In Computing:** X could be a data structure (Integer), Y is another type (String), and Z is a structure with two fields. **(X and Y)**

Products

If we have three different items of type X for example a string [“a”, “b”, “c”], and two different items of type Y [0,1]. How many different records can be created?

3x2

(“a”,0)	**(“a”,1)**
(“b”,0)	**(“b”,1)**
(“c”,0)	**(“c”,1)**

Let’s see Products in Java

```
public class Product<X,Y>{
      private X first;
      private Y second;
      public Product (X first,Y second){
            this.first=first; this.second=second;
      }
      public X getFst(){ return this.first;}
      public Y getSnd(){ return this.second;}
}

public class Test{
      public Product<Integer,String> pair= new
      Product(1,”two”);
      public Integer one=pair.getFst();
      public String two=pair.getSnd();
}
```

Sums

$$\begin{tikzcd}
X \arrow[r, "\text{left}"] \arrow[dr, "f"'] & X+Y \arrow[d, dashed, "{[f,g]}"] & Y \arrow[l, "\text{right}"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}$$

left ; [f,g] = f

right ; [f,g] = g

- If I know X or Y, I can conclude Z.
- In computing: for either a value of type X or a value of type Y, you can get to Z. **(X OR Y)**

Sums

If we have three different items of type X [a,b,c], and two different items of type Y [0,1]. How many different sums can be created?

3+2

left “a”	**right 0**
left “b”	**right 1**
left “c”	

Let’s see Sums in Java

```
public interface Sum<X,Y>{
 private <Z> Z Selection(Function<X,Z> f, Function<Y,Z>
g);
}
public class Left<X,Y> implements Sum<X,Y>{
 private X x;
 public Left(X x) {this.x=x;}
 public <Z> Z Selection(Function<X,Z> f,
       Function<Y,Z> g){
  return f.apply(x);
 }}
public class Right<X,Y> implements Sum<X,Y>{
 private Y y;
 public Right(Y y) {this.y=y;}
 public <Z> Z Selection(Function<X,Z> f, Function<Y,Z> g){
  return g.apply(y);
 }}
```

Let’s see Sums in Java

```
public class ErrInt extends Sum<Integer,String>{
 public ErrInt err= new Left(“Error”);
 public ErrInt one= new Right(1);
 public ErrInt add(ErrInt that){
   return this.Selection(
       e-> new Left(e),
       m-> that.Selection(
             e-> new Left(e),
             n-> new Right(m+n)
            )
       );
 public ErrInt test=one.add(err);
}
```

You cannot add if an error exists

Duals

$$\begin{tikzcd}
& Z \arrow[dl, "f"'] \arrow[d, dashed, "{<f,g>}"] \arrow[dr, "g"] & \\
X & X*Y \arrow[l, "\text{fst}"] \arrow[r, "\text{snd}"'] & Y
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "\text{left}"] \arrow[dr, "f"'] & X+Y \arrow[d, dashed, "{[f,g]}"] & Y \arrow[l, "\text{right}"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}$$

- If you flip the arrows twice, you’ll get back where you started. Sounds Familiar?

De Morgan's laws

- The expression of conjunctions and disjunctions can be expressed in terms of each other via negation.

$$\overline{\boldsymbol{A} \vee \boldsymbol{B}} = \overline{\boldsymbol{A}} \wedge \overline{\boldsymbol{B}}$$

$$\overline{\boldsymbol{A} \wedge \boldsymbol{B}} = \overline{\boldsymbol{A}} \vee \overline{\boldsymbol{B}}$$

Exponentials

How many different functions can be created from a type Y with two items [0,1] to a type X with three items [a,b,c]?

$2 \Longrightarrow 3 = 3^2$

0→”a” 1→”a”	0→”b” 1→ "a”	0→”c” 1→”a”
0→”a” 1→”b”	0→”b” 1→”b”	0→”c” 1→”b”
0→”a” 1→”c”	0→”b” 1→”c”	0→”c” 1→”c”

Isomorphisms

- Functions are Exponentials!

$$\mathcal{C}(Z, X * Y) \cong \mathcal{C}(Z, X) \; * \; \mathcal{C}(Z, Y)$$
$$\mathcal{C}(X + Y, Z) \cong \mathcal{C}(X, Z) \; * \; \mathcal{C}(Y, Z)$$

$$\Downarrow$$

$$(X * Y)^Z = X^Z \; * Y^Z$$
$$Z^{(X+Y)} = Z^X \; * Z^Y$$

Or-structure Example in Rust

std::result

- Result<T, E> is an enum with two variants:
  - Ok(T) → success and containing a value.
  - Err(E) → error and containing an error value.

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

Attempt 1 = 💣

```
use std::io::{stdin, stdout, Write}; //flush uses Write

fn main(){
    let mut my_string = String::new();
    print!(“Enter a number: “); // prompt user
    stdout().flush().unwrap();
    stdin().read_line(&mut my_string)
        .expect(“Did not enter a correct string”);
    let my_number: f64 = my_string.trim().parse();   💣
    println!(“Yay! You entered a number. It was {:?}”,
        my_num);
}
```

because you need to handle a Result that has been returned by a function

```
error[E0308]: mismatched types
  --> main.rs:20:26
   |
20 |     let my_number: f64 = my_string.trim().parse();
//.unwrap();
   |                          ^^^^^^^^^^^^^^^^^^^^^^^^
expected f64, found enum `std::result::Result`
   |
   = note: expected type `f64`
              found type `std::result::Result<_, _>`
error: aborting due to previous error
```

```
pub fn parse<F>(&self) -> Result<F, <F as … >::Err>
```

- So, how do we set the correct type?

Fixing the Error with Expect()

- The expect() function looks at the Result type:
  - OK → returns the converted value.
  - Err → lets the program crash.

```
let my_number: f64 = my_string.trim().parse().expect("Parse
failed");
```

- If the Result is Err, expect() will let the program crash and display the string that was passed to it.

Fixing 1 Problem

```
use std::io::{stdin, stdout, Write}; //flush uses Write
fn main(){
    let mut my_string = String::new();
    print!(“Enter a number: “);
    stdout().flush().unwrap(); //like expect(), ignore it
    let my_num = loop {
        my_string.clear();  //clearing any errors.
        stdin().read_line(&mut my_string)
            .expect(“Did not enter a correct string”);
        match my_string.trim().parse::<f64>() {
            Ok(okay) => break okay,
            Err(_) => println!(“Try again. Enter a number.”)
        }
    };
    println!(“You entered {:?}”, my_num);
}
```

Using Result in Your Own Functions

```
fn is_it_fifty(num: u32) -> Result<u32, &’static str> {
        let error = “It didn’t work”;
        if num == 50 {
            Ok(num)
        } else {
            Err(error)
        }
}

fn main(){
    let my_num = 50;  //50 for example
    match is_it_fifty(my_num) {
        Ok(_) => println!(“Good! my_num is 50”),
        Err(err) => println!("Error: {:?}", err)
    }
}
```

Panic, panic – controlled chaos

(C Style in Rust)

Can ....we try again?

```
fn open_config() -> Result<std::fs::File, std::io::Error> {
use std::fs::File;
match File::open("config.toml") {
Ok(f) => Ok(f),
                    // If not found, search in second location:
Err(e) => match File::open("data/config.toml") {
Ok(f) => Ok(f),
                    // Otherwise, bubble first error up to caller
_ => Err(e),
}}}
```

In main ...

```
fn main() {
let mut result = open_config();
if result.is_err() {                        // If failed the first time
                                            // Try again in about 5 seconds ...
std::thread::sleep(std::time::Duration::from_secs(5));
                                                      // Reattempt
result = open_config();
} match result {
                                                      // Proceed as usual ...
Ok(cfg) => println!("Opened the config file"),
                                  // Print the cause to stderr, and DONT PANIC
Err(e) => {eprintln!("File could not be opened: {:?}", e.kind());
std::process::exit(1);                                // Exit with code 1 (fail)
}
}
```

Does it exist?

Choosing between existing OR non-existing things

Example 3 -- Optional

An output can have either Some value or no value (None).

```
enum Option<T> { // T is a generic and it can
                        contain any type of value.
   Some(T),
   None,
}
```

Basic usages of Option

- if an argument of the function is optional,
- If the function is non-void and if the output it returns can be empty,
- If the value, of a property of the data type can be empty,
- We have to use their data type as an Option type

```
fn get_an_optional_value() -> Option<&str>
{       //if the optional value is not empty

        return Some("Some value");
        //else
                None
}
```

In a data structure

```
struct Name

{ first_name: String,


middle_name: Option<String>, // middle_name
                                  can be empty

last_name: String, }
```

Pattern Matching

```
use std::env;


fn main()
{ let home_path = env::home_dir();
match home_path {
Some(p) => println!("{:?}", p), // prints "/root",
None => println!("Can not find the home directory!"),
} }
```

Options and Results

```
fn main() {

let o: Result<i8, &str> = Ok(8);
let e: Result<i8, &str> = Err("message");

 assert_eq!(o.ok(), Some(8)); // Ok(v) ok = Some(v)
assert_eq!(e.ok(), None); // Err(v) ok = None
assert_eq!(o.err(), None); // Ok(v) err = None
assert_eq!(e.err(), Some("message")); // Err(v) err = Some(v)
}
```