Functional Programming

CS 340: Programming Paradigms / Patterns

Questions:

- what is functional programming? (what is it not?)
- some f.p. history/background, please?
- how is f.p. different from imperative programming?
  - what are some advantages of f.p.?
    (per Hughes, what "glue" features does it provide?)
  - what are some disadvantages of f.p.?

Functional programming has roots in mathematical logic and computational theory.

what is computable?
- based on some universal model of computation

In 1936, three formal approaches to computability:

1. Turing machines
2. Kleene's recursive function theory
3. Church's lambda (λ) calculus

each specifies ① set of primitive operations, ② rules for structuring operations, ③ proof theory — "can I compute this?"

- all are equivalent! (what is computable in one model is computable in others) λ calculus:

- very simple model for functional programming
- basic syntax → universal machine code for f.p. languages

grammar:

$$expr \ := \ \overbrace{\lambda\, \underset{\substack{\uparrow\\ \text{identifier}\\ \text{(bound variable)}}}{var}.\ \underset{\substack{\uparrow\\ \text{fn body}}}{expr}}^{\text{fn. definition}} \ \Big| \ \overbrace{\big(\underset{\substack{\uparrow\\ \text{fn call}}}{expr}\ \ \underset{\substack{\uparrow\\ \text{arg}}}{expr}\big)}^{\text{fn. application}}$$

Evaluation :

- when evaluating fn application, the argument is substituted for all instances of the bound variable in the called function's body

e.g. $(\lambda x.x \quad 10) \Rightarrow \cancel{\lambda x.}\overset{10}{\cancel{x}} \Rightarrow 10$

$$(\lambda x.\underbrace{x^2+2x+5}_{\text{allowing arith. exps.}} \quad 10) \Rightarrow 10^2+2\cdot 10+5$$

$$\text{e.g., } id = \lambda x.x \quad \begin{cases} (id\ 5) \Rightarrow 5 \quad (id\ id) \Rightarrow id \\ (id\ y) \Rightarrow y \end{cases}$$

$$fst = \lambda x.\lambda y.x \quad \begin{cases} (fst\ 5) \Rightarrow \lambda y.5 \\ ((fst\ 5)\ 6) \Rightarrow 5 \end{cases}$$

$$app = \lambda f.\lambda x.(f\ x) \quad \begin{cases} ((app\ id)\ 5) \Rightarrow 5 \\ ((app\ ((app\ fst)\ 5))\ 10) \Rightarrow 5 \end{cases}$$

$$rec = \lambda f.(f\ f) \quad \begin{cases} (rec\ id) \Rightarrow id \\ (rec\ rec) \Rightarrow ? \end{cases}$$

Imperative vs. Functional

Imperative programs are made up of sequences of statements

```
a = 10
b = 20

c = a + b
```

Functional programs are based on nesting/composing function calls

```
f(w, g(x, y), h(z))
```

Imperative programs allow values associated w/ variables to change over time ("state mutations")

```
x = 5
y = 6
tmp = x
x = y
y = tmp
```

Functional programs do not allow variables to be re-bound. (no assignment statements)

equals means equals!

`a = 10`

~~a = 20~~

`swap x y = swap y x`

Imperative languages have built-in "control flow" constructs that alter the default ordering of statements

```
a = 10
b = 20
if a < b
    max = b
else
    max = a
```

```
sum = 0
x = 10
while x > 0
    sum += x
    x -= 1
```

Functional languages only have expressions and function calls. "Control flow" is meaningless!

```
max a b = if a < b then b
                    else a

sum 0 = 0
sum x = x + sum (x-1)
```

Imperative programs may define procedures w/ "side-effects."

```
def sum(x, y):
    global z
    z += x+y
    return x+y
```

```
z=0
sum(10,20)
sum(10,20)
```

result in different states!

Functional programs contain no side-effects. I.e., fns only compute their results — nothing else!

```
sum x y = x + y
```

"pure" function — result depends solely on input.

Imperative programs require "strict" evaluation (aka "eager"/"greedy" evaluation)

```
foo(sum(1,2), sum(2,3))
```

must be executed in order!

```
foo(print("hello"),
    print("bye"))
```

Functional programs permit "lazy" evaluation, due to referential transparency.

```
foo(bar(1,2,3,4),
    baz(2,3,4,5))
```

no need to evaluate immediately!

```
foo x y = 10
```

contain "thunks"

first-class fn = fn as a value

Imperative languages have varying support for first-class and higher-order functions.

HOF = fn that takes/returns fns

Functional languages universally support first-class and higher-order functions.

$(\lambda f.f\ \lambda g.g)$

```
def app(f, x):
    f(x)
```

PYTHON

```
app(lambda y: y+1, 10)
```

```
int app(int (*f)(int), int x) {
    return f(x);
}
```

```
int inc(int x) {
    return x+1;
}

app(inc, 10) => 11
```

Imperative programs
support direct input/output

```
def foo(x):
    y = input
    return x+y
```

foo is <u>not</u> a pure
function!

Functional languages
prohibit direct I/O in functions,
as they introduce side effects!

?

(how do we do input?)