COL 100 - Lec 11

Formulating Abstractions With higher order functions

- function or procedure → abstraction
- higher order functions → functions taking functions as inputs

Currying

- A technique of translating the evaluation of a function with multiple arguments into evaluating a sequence of functions each with a single argument

Eg:

```
add c y x = y+x
```

fn: int → (int → int)

as opposed to

```
add c (y, x) = y + x
```

fn: (int * int) → int

if

f: α → β → γ → δ

then

f a: β → γ → δ

f a b: γ → δ

f a b c : δ

Consider

```
Cube x = x * x * x
```

] An abstraction. It computes a cube for any value of x

Now

1. Sum_int a, b = if a > b then 0 Else a + Sum_int (a+1) b

$\sum_{a}^{b} i$

2. Sum-cubes a b = if a > b then 0
else a*a*a + Sum-cubes (a+1) b

$$\sum_{a}^{b} i^3$$

3. Sum of a sequence $\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \cdots$

that slowly converges to $\pi/8$

```
Sum-pi a b = if a > b then 0
             else 1.0 div (a * (a+2)) +
                  Sum-pi (a+4), b
```

Examples (1) to (3) share a common pattern!

which is

```
fun <name> a b
    = if a > b then 0
      else (<term> a) +
           (<name> (<next> a) b)
```

useful abstraction for

$$\sum_{n=a}^{b} f(n)$$

Therefore

```
Sum term a next b = if a > b then 0
                    else (term a) + Sum term
                                    (next a) next b
```

How I define (1)

(1) sum-int a b = sum identity a inc b

(2) sum-cubes a b = sum cube a inc b

(3) sum-pi a b = sum pi-term a pi-next b

pi-term: $\frac{1}{x * (x+2)}$

pi-next: $x+4$

There is more!

Once you have 'sum' as the building block,
you can use it in formulating further concepts

Eg: definite integral f betⁿ limits a, b

$\int_a^b f$ can be approximated numerically using the formula:

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + \frac{3dx}{2}\right) + f\left(a + \frac{5dx}{2}\right) + \cdots \right] dx$$

for small values of dx

One could replace this with "Simpson's" rule which is more accurate

```
fun integral f a b dx
    = let add-dx x = x + dx in
      [Sum f (a + dx/2.0) add-dx b] * dx
```

Exercise: Iterative version of Sum

```
fun Sum term a next b =
    let fun iter a result =
            if <??> then <??>
            else iter <??> <??>
    in
      iter <??><??>
```

1 : a > b

2 : result

3 : next a

4 : term a + result

5 : a 6 : 0

Inv:

$$result = \sum_{i=c_0}^{c-1} \langle term\ i \rangle$$

Polymorphic functions

- $\sum_{i=a}^{b} f(i)$
- $\prod_{i=a}^{b} f(i)$
- $\bigotimes_{i=a}^{b} f(i)$

→ Operators are changing

→ and correspondingly the identity elements are Changing

Accumulator

As long as $\otimes$ is associative and has an identity Elements

i.e. $a \otimes (b \otimes c) = (a \otimes b) \otimes c$

$a \otimes e = e = e \otimes a$

Then if

$f, next : \alpha \longrightarrow \alpha$

id-val,

$Acc(op, \lambda f, l, next, U)$

$$= \begin{cases} e \quad [\text{identity value}] & \text{if } l > U \\ f(l) \; op_{\text{id-val},} \; Acc(op, \lambda f, next(l), next, U) & \text{Else} \end{cases}$$

Eg: Accumulators

→ Concatenation on strings

→ $(+, 0)$, $(*, 1)$ on vectors and matrices

⋮

One can obtain even more general versions of accumulator → filters on the terms to be combined

→ i.e. Combine only those terms in the range that satisfy a certain condition.

Eg: Sum of squares of the prime numbers in [a, b]

— Utility fnes

```
fun gcd a b = if (b=0) then a
              else gcd b (a mod b)

fun sq x = x * x

fun divisible? b a = (b mod a) = 0

fun smallest_divisor n = (find_divisor n 2)

fun find_divisor n test = ??
```

```
fun prime? x = (smallest-divisor x = x)

fun filtered-accumulate op filter idValue term a
    next b
  = let rec a
        = if a > b then idValue
          else if filter? a then
                 op (term a) (rec next a)
               else rec (next a)
    in
      rec a
```

```
. fun sum-of-sq-of-primes_between a b
    = filtered-accumulate
        + prime? o sq a inc b
```