CS 421 Lecture 17 – Higher-order functions

- Using fold_right
- Expression evaluation via substitution
- Short examples
- Combinator-style programming

fold_right

```
fold_right f [x1;x2;...xn] z
    = f x1 (f x2 (...(f xn z)...))
fold_right : (α->β->β)->(α list)->β->β
```

Use `fold_right` to remove all negative elements from a list:

```
fold_right (fun x y -> if (x<0)
                       then y
                       else x::y ) lis []
```

Defining higher-order functions

```
let rec fold_right f lis z =
  if lis = [] then z
  else f (hd lis)
    (fold_right f (tl lis) z)
```

Evaluation of expressions

Use substitution model: *in function calls, substitute actual parameter for formal parameter in body of function.*

- Details on following slides:
  - Expressions: constants, function definitions (fun x -> e), application of built-in functions, if, application of user-defined functions
  - let expressions syntactic sugar for function application; top-level definitions implicitly in let let x=e1 in e2 ≡ (fun x→e2)e1
  - Will handle recursive functions after break; also will discuss closure model after break
  - Key feature of substitution model: never evaluate expressions that have "free" variables; e.g. when evaluating e1+e2, e1 and e2 will consist solely of constants; when evaluating fun x -> e, the only "free" variable in e is x.

Evaluation of expressions

Evaluate expression:

- Constant n (int, bool, string, list, ..) $\Rightarrow$ n
- Abstraction fun x -> e $\Rightarrow$ fun $x \to e$
- Application of built-in operator: e1 + e2
  evaluate e1 $\Rightarrow$ v1
  evaluate e2 $\Rightarrow$ v2
  $\Rightarrow$ v1+v2
- if e1 then e2 else e3
  evaluate e1 $\Rightarrow$ v1
  if v1 is true, $\Rightarrow$ evaluate e2
  o.w. $\to$ evaluate e3

Evaluation of expressions (cont.)

- Application of user-defined function: e1 e2
  - Evaluate e1 $\Rightarrow$ fun $x \to e$ (for some x, e)
  - Evaluate e2 $\Rightarrow$ v
  - Substitute v for x in e, yielding $\bar{e}$
  - $\Rightarrow$ Evaluate $\bar{e}$

let add = fun x → fun y → x+y

(add 3) 4

let add3 = add 3

add3 4

Example of evaluation

(fun x -> fun y -> x+y) 1 2

- eval, yielding "fun z -> e"
  - eval => fun x -> fun y -> x+y
  - eval => 1
  - Subst 1 for x in => fun y -> 1+y
  - Eval fun y -> 1+y => fun y -> 1+y

  z e
- Eval 2 => 2
- Subst 2 for z in e, yielding $\bar{e}$ 1+2
- => Evaluate $\bar{e}$ - 1+2
  - Eval 1 => 1
  - Eval 2 => 2
  - => 1+2 = 3

Example of evaluation

$$\Big(\underbrace{(\text{fun } x \text{ -> fun } y \text{ -> } x\ y)}_{e3}\ \underbrace{(\text{fun } w \text{ -> } w\ 4)}_{e4}\Big)\ (\text{fun } z \text{ -> } z+1) \Rightarrow 5$$

e1 e2

(1) Eval. e1

(1) Eval e3. ⇒ e3

(2) Eval e4 ⇒ e4

(3) Subst. e4 for x in fun y → x y

⇒ fun y → (fun w → w 4) y

(4) Eval , ⇒ fun y → (fun w → w 4) y

(2) Eval e2 ⇒ fun z → z+1

(3) Subst e2 for y in (fun w → w 4) y

⇒ (fun w → w 4)(fun z → z+1)

$$(4)\ \text{Eval}\ \underbrace{(\text{fun } w \to w\ 4)}_{e5}\underbrace{(\text{fun } z \to z+1)}_{e6} \Rightarrow 5$$

(1) Eval e5 ⇒ fun w → w 4

(2) Eval e6 ⇒ fun z → z+1

(3) Subst. e6 for w in (w 4)

⇒ ((fun z → z+1) 4)

$$(4)\ \text{Eval}\ \underbrace{(\text{fun } z \to z+1)}_{e7}\ \underbrace{4}_{e8} \Rightarrow 5$$

(1) Eval e7 ⇒ fun z → z+1]

(2) Eval e8 ⇒ 4

(3) Subst 4 for z in z+1 ⇒ 4+1

(4) Eval 4+1

⇒ 5

Short examples - Currying

- Can define two-argument functions in two ways:
  - Curried: let f x y = … x … y …
    (or, let f = fun x y -> … x … y …
    or, let f = fun x -> fun y -> … x … y …)
  - Uncurried: let f (x,y) = … x … y …
    (or, let f = fun (x,y) -> … x … y …
    or, let f = fun p -> … (fst p) … (snd p) …)

Sometimes want to use the "same" function both ways…

Short examples - Currying

- Can use higher-order function to turn curried function to ~~uncurried form, and vice versa~~

let curry = fun f -> fun x -> fun y -> f(x, y)

If $f : \alpha * \beta \to \gamma$

then $(\text{curry } f) : \alpha \to \beta \to \gamma$

and for any e1, e2, $f(e1, e2) = ((\text{curry } f)\ e_1)\ e_2$

Suppose mult (x, y) = x * y

((curry mult) 4) 5 = 20 = mult (4, 5)

((fun x -> fun y -> mult (x, y)) 4) 5 ⟶ (fun y -> mult (4, y)) 5 ⟶ mult (4, 5)

$$\text{let uncurry} = \text{fun } f \to \underbrace{\text{fun } (a,b) \to f\ a\ b}$$

Then if $f : \alpha \to \beta \to \gamma$

then $\text{uncurry } f : \alpha * \beta \to \gamma$

and $\forall e_1, e_2 : (\text{uncurry } f)(e_1, e_2) = (f\ e_1)\ e_2$

$$(\text{fun } (a,b) \to f\ a\ b)(e_1, e_2)$$

$$\equiv f\ e_1\ e_2 \rightsquigarrow$$

Short examples – reversing arguments

Given f: $\alpha \longrightarrow \beta \longrightarrow \gamma$, produce f_R: $\beta \longrightarrow \alpha \longrightarrow \gamma$, s.t.

$f_R \; x \; y = f \; y \; x$

let reverseargs = fun g →
(fun a → fun b → g b a)

((reverseargs (-)) 4) 3
= (fun a → fun b → (-) b a) 4 3
= (-) 3 4 = -1

let decr = (reverseargs (-)) 1 ;;

Short examples – applying function twice

Given f: $\alpha \to \alpha \to \alpha$, want ff: $\alpha \to \alpha \to \alpha$ such that
ff x = f (f x)

Combinator-style programming

Can write complex programs by defining a library of higher-order functions and applying them to one another (and to first-order or built-in functions).

Advantage: easy of creating programs – programs are just expressions

Example: build a parser by writing “parser combinators”.

type α option = Some α | None

Parser combinators

Define a parser to be a function from token list -> (token list) option.

Idea is to define functions that build parsers, rather than building parsers "by hand."

E.g. Parser to recognize a single token:

```
let token s = fun cl -> if cl=[] then None
                        else if s=hd cl then Some (tl cl)
                        else None;;
let parsex = token 'x';;
parsex ['x'];;
parsex['a'];;
```

Parser combinators

"Combinators" to combine parsers into larger parsers:

= fun p -> fun q ->

let (++) ~~p q =~~ fun cl -> match p cl with None -> None
| Some cl' -> q cl';;

let parsexy = token 'x' ++ token 'y'

parsexy ['x', 'y']

parsexy ['x', 'z']

let (||) p q = fun cl -> match p cl with None -> q cl
| Some cl' -> Some cl';;

let parsexyorz = parsexy || token 'z'

parsexyorz['x', 'y']

parsexyorz ['z']

Parser combinators

Put this together to define parser for grammar:

A -> aB | b
B -> cB | A

```
let rec parseA cl = ((token 'a' ++ parseB) || token 'b') cl
  and parseB cl = ((token 'c' ++ parseB) || parseA) cl;;
```

```
parseA ['a';'c';'c';'a';'b']
```