CS 421 Lecture 17: More Functional Programming

- Announcements
- Lecture outline
  - Using fold_right and fold_left
  - Expression evaluation
    - Substitution model
    - Scope of definitions
  - "Simple" examples
  - Combinator programming

Announcements

- 4-unit grad students:
  - Project proposal due today

Review: 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 ________________________ lis _____
```

Review: fold_left

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

- Use fold_left to compute the length of lis

  `fold_left _______________________   _____  lis`

- Use fold_left to compute map f lis

  `fold_left _______________________   _____  lis`

Review: 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)
```

- Define fold_left:

```
let rec fold_left f z lis =
```

Evaluation of expressions

- Problem: “free” variables in function definitions
- Two models: substitution and environment/closure
- Substitution:
  - Replace free variable with its value
- Closure:
  - Put free variables in an “environment” data structure
  - (expr, env) = closure

Evaluation of expressions

- Using substitution model – in function calls, substitute actual parameter for formal parameter in body of function.
  - No expressions with free variables evaluated
  - 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
  - Tomorrow: handling recursive functions

Evaluation of expressions

- Evaluate expression without free variables:
  - Constant n (int, bool, string, list, ..) ⇒ n
  - Abstraction fun x -> e ⇒
  - Application of built-in operator: e1 + e2 ⇒
  - if e1 then e2 else e3 ⇒

Evaluation of expressions

- Application of user-defined function: e1 e2 ⇒

```
(1) e1 ⇒ fun x -> e'
(2) e2 ⇒ v
(3) let e'' = substitute v for
                   free occurrence of x in e'
(4) eval e''
```

Example of evaluation

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

Example of evaluation

```
(fun x -> fun y -> x y) (fun y -> y 4) (fun z -> z+1)
```

Free variables

- In rule for applications, substitute v for *free occurrences* of x in e'. Need to define "free occurrence."
- Def. Free occurrences of x in e are those marked with an overbar after applying free to x and e:

```
free x e = match e with
    n ->
  | x ->
  | y ->
  | e1+e2 ->
  | (fun x -> e') ->

  | (fun y -> e') ->
```

Example of free occurrences

```
(fun x -> fun y -> x y) (fun y -> y 4) (fun z -> z+1)
```

Scope rules

- Programs introduce names via “declarations”, then refer to those names in “uses.” A given name can be introduced in more than one declaration, but every use corresponds to a particular declaration. The question is: which one?
- The *scope* of a declaration of a name x is the parts of the program in which a use of x refers to this declaration
- A use of a name is *in the scope of a declaration* if that use is in the scope of that declaration
- N.B. the scope of a declaration can have holes, where the declaration is covered up by another declaration of the same name.

Example: Scope rules in Java

```
class C {
  int y
  void f (x) { … x … f … y … g … }
  void g () { … }
}

class D extends C {
  int z
  void f (x) { … x … f … y … g … }
}
```

- Static vs. dynamic scope

Example: Scope rules in OCaml

- ```
  let x = 2
  in let f = fun x -> x+x
    in f x
  ```
- ```
  let x = 2
  in let y = x
    in let f z = let x=3 in y+z
        in f x
  ```
- ```
  let x = 2
  in let add = fun x -> fun y -> x+y
    in let addx = add x
        in let x = 3 in addx 1
  ```

- Only static scope
  ```

Scope rules in OCaml

- Scope rules are implied by expression evaluation rules.
- Declarations are just function definitions fun x ->e
- Scope of this declaration of x is exactly the free occurrences of x in e.
  - (Put differently, a use of a variable x is in the scope of the closest enclosing function definition for which x is the formal parameter.)

- **This is called *static scope*, or *lexical scope*, because the declaration corresponding to any use is known statically (before run time).**

The scope rule of LISP

- In Lisp, the declaration associated with a use of a variable x is determined as follows: at run-time, the most recent function application that has x as formal parameter (and which is still on the stack) gives the declaration of x.
- LISP vs. OCaml:

```
let h f = let x = 3 in f x
let f x = let g y = x + y in h g
f 5 => ?
```

“Simple” examples – currying

- Can define a two-argument function in two ways:
  - Uncurried:

```
let f x y = … x … y …
let f = fun x y -> … x … y …
let f = fun x -> fun y -> … x … y …
```

  - Curried:

```
let f (x,y) = … x … y …
let f = fun (x,y) -> … x … y …
let f = fun p -> … (fst p) … (snd p)
```

- Sometimes want to use the “same” function both ways.

"Simple" examples – currying

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

```
let curry f = fun x -> fun y -> f(x,y)
curry : (α->β->γ)->(α*β->γ)

let uncurry g = fun (x,y) -> g x y
uncurry : (α*β->γ)->(α->β->γ)


f ≡ uncurry (curry f)
```

"Simple" examples – reversing arguments

- Given `f: α->β->γ`, produce f_R`: β->α->γ`, s.t.:

  f_R `x y = f y x`

```
let reverse f =
```

```
reverse (-) 3 4 = ?
```

“Simple” examples – applying function twice

- Given `f: α->α->α`, produce `ff: α->α->α`, s.t.:

```
ff x = f (f x)


let double f =





(double incr) 5 = ?
```

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: ease of creating programs – programs are just expressions

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

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:

```
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']
```