Higher-Order Functions

Equivalent Notations

\[
\begin{align*}
(\text{define } (f \ x \ y) \ (\ldots \text{body} \ldots) ) \\
= \\
(\text{define } f \\
\quad (\lambda (x \ y) \\
\qquad (\ldots \text{body} \ldots) \\
\quad ) \\
\quad )
\end{align*}
\]

Function Values

\[
(\text{define } \text{tag} \\
\quad (\lambda (t \ l) \ (\text{cons } t \ l)) \\
\quad )
\]

\[
(\text{tag 'int '(1)}) \rightarrow \text{(int 1)}
\]

What characterizes a function?
- Expression in the body of the definition.
- The sequence of formal parameters.

Anonymous Functions

\[
\begin{align*}
( (\lambda (t \ l) \ (\text{cons } t \ l)) \\
\quad '\text{int '}(1) \\
\quad )
\end{align*}
\]

- Name of the function is “irrelevant”.
- Simplification:

\[
(\text{define } \text{tag } \text{cons})
\]

- Assigning function values is similar to assigning primitive values.

(\text{first-class values}).
Higher-order Functions

- In Scheme, *function values* can be
  - a) passed to other functions.
  - b) returned from functions.
  - c) stored in data structures.

- FORTRAN, Ada, etc prohibit a), b), c).
- Pascal allows only a).
- LISP/C++/Java support approximations in a round about way.
  - LISP: Meta-programming, C++: Function pointers.
  - Java: via Objects (Closures coming in JDK 7)
- Scala, Python and C# (delegates) are multi-paradigm languages with support for higher-order functions.
- ML/Scheme are functional languages that treat function values as first-class citizens.

*Implementing subprograms using a stack breaks down here. Heap and garbage collection required.*

Applications

- Abstracting commonly occurring patterns of control. (*factoring*)
- Defining generic functions.
- Instantiating generics. (*ORTHOGONALITY*)

- Eventually contribute to readability, reliability, and reuse.
- *Library*: collection of higher-order functions.

Factoring Commonality

```
(1 2 ... n)  ->  (1 4 ... n^2)
```

```
(define (sql L)
  (if (null? L) ()
      (cons (* (car L) (car L))
        (sql (cdr L)))))
```

```
(a b c)  ->  ((a) (b) (c))
```

```
(define (brac L)
  (if (null? L) ()
      (cons (list (car L))
        (brac (cdr L)))))
```

The map function

```
(define (map fn lis)
  (if (null? lis) ()
      (cons (fn (car lis))
        (map fn (cdr lis))))
)
```

```
(map (lambda(x) (* x x)) '(1 2 3))
(map (lambda(x) (list x)) '(a b c))
```

Built-in map:

```
(map + '(1 2 3) '(4 5 6)) = (5 7 9)
```
**foldr (reduce) and foldl (accumulate)**

(Racket : Intermediate Level)

- \( \text{foldr} \ (\text{reduce}) \) and \( \text{foldl} \ (\text{accumulate}) \)

\[
(\text{foldr } + 100 \ (\text{list} \ 2 \ 4 \ 8)) = 2 + 4 + 8 + 100 = 114
\]

- \( \text{foldl} \ (\text{cons}) \) \( \text{'(c)'(a b)} \)

\[
(\text{foldl } \text{cons} \ (\text{'}c\text{'}) \ (\text{a b})) = (b \ a \ c)
\]

---

**Templates/Generics**

- Generalization

\[
(\text{define } (\text{imax} \ x \ y))
(\text{if } (< x \ y) \ y \ x)
\]

- Specialization

\[
(\text{define } (\text{cmax} \ x \ y))
(\text{if } (\text{char-<} x \ y) \ y \ x)
\]

- Parameterize wrt comparison

- Passing function values

---

**Instantiation**

\[
(\text{define } (\text{maxg} \text{ lt})
(\text{lambda } (x \ y)
(\text{max} \text{ lt} \ x \ y))
\)
\]

\[
(\text{maxg} >) \ 4 \ 5
\]

(* customization at run-time *)

- Function generating function.
  - Closure construction.
- Higher-order functions in a library may be tailored this way for different use.

---

**Recursion vs Higher-Order Functions**

\[
(\text{define-struct} \ \text{cost} \ ($) \)
(\text{generate code for make-cost & cost-\$})
\]

\[
(\text{define} \ \text{costList}
(\text{list} \ (\text{make-cost} 5) \ (\text{make-cost} 10)
(\text{make-cost} 25)))
\]

\[
(\text{check-expect} \ (\text{totalCost} \ \text{costList}) \ 40)
(\text{check-expect} \ (\text{totalCostHF} \ \text{costList})
(\text{totalCostHF} \ \text{costList}))
\]
Simple Recursion vs Map-Reduce version

(define (totalCost cl)
  (if (null? cl) 0
    (+ (cost-\$ (car cl))
      (totalCost (cdr cl)))))

(define (totalCostHF cl)
  (foldr + 0 (map cost-\$ cl)))

• Both foldl and foldr admissible.
• Requires DrRacket : > HtDP Intermediate Level

Scoping

• Rules governing the association of use of a variable with the corresponding declaration.
  proc p (x: int) ;
  proc q;
  var y: int;
  begin \[x]: = y end;
  begin q end;

• Imperative Language: Local / non-local variables.
• Functional Language: Bound / free variables.

Scoping rules enable determination of declaration corresponding to a non-local / free variable.

Alternatives
  - Fixed at function definition time
    • Static / lexical scoping
      - Scheme, Ada, C++, Java, C#, etc
    - Fixed at function call time
      • Dynamic scoping
        - Franz LISP, APL, etc.

Pascal Example

proc p;
var z:int;
proc q;
begin z := 5 end;
proc r;
var z:int;
begin q end;
proc s;
var z:int;
begin q end;
begin ... end;
**Scoping:**

**Static**
- \( z := 5; \)
- \( s: \)
  - \( z := 5; \)

Calls to \( q \) in \( r \) and \( s \) update the variable \( z \) declared in \( p \).

**Dynamic**
- \( z := 5; \)
- \( s: \)
  - \( z := 5; \)

Calls to \( q \) in \( r \) and \( s \) update the variables \( z \) and \( z \) declared in \( r \) and \( s \) respectively.

---

**Scoping: functional style**

```
(define y 5)
(lambda (x) (+ x y))
```

- Point of definition = Point of call.

```
(define (f x) (+ x y))
(define (g y) (f y))
```

- Naming context of definition =/= Naming context of call.

---

**Scoping Rules**

- **Static Scoping**
  - \( (g y) \) \( y <= 16 \)
  - \( (f y) \) \( y <= 16 \)
  - \( (+ x y) \) \( x <= 16 \)
  - \( y <= 5 \)

- **Dynamic Scoping**
  - \( (g y) \) \( y <= 16 \)
  - \( (f y) \) \( y <= 16 \)
  - \( (+ x y) \) \( x <= 16 \)
  - \( y <= 16 \)
  - \( y <= 5 \)

---

**Closure**

```
(define (addn n)
  (lambda (x) (+ x n))
)
```

```
(addn 5) =
```

```
(lambda (x) (+ x n))
```

```
\( n <= 5 \)
```

**Closure** = Function Definition + Creation-time environment

( to enforce lexical scoping for free variables )
Application

- Instantiating generic functions.
- **Object-oriented Programming**
  Using (1) `let`-construct (to introduce local variables) and (2) assignments, objects and classes can be simulated.
- **Streams**
  Creation of “infinite” data structures.

Lambda Expressions

- `(define (id x) x)`
- `(define id (lambda (x) x) )`
- `(lambda (x) x)`
- `( (lambda (x) x) 5 )`
- `(lambda () 5)`

Lambda Expressions

- `(lambda () 1)`
- `( (lambda () 1) )`
- `( (lambda (x) x)
  (lambda (y) y) )`
- `( (lambda (x) (x x))
  (lambda (y) y) )`