Higher order functions

Paliath Narendran

March 3, 2008
Declarative Programming

Functional Programming

- LISP
  - Scheme
  - Common Lisp
  - ML
  - Haskell
  based on the λ-calculus

Logic Programming

- PROLOG
  based on the first-order predicate calculus
Functional Programming

- based on the λ-calculus
Functional Programming

- based on the λ-calculus
- functions as “first-class” values
Functional Programming

- based on the \(\lambda\)-calculus
- functions as “first-class” values
- recursion is the main paradigm for repeated execution
Functional Programming

- based on the \( \lambda \)-calculus
- functions as “first-class” values
- recursion is the main paradigm for repeated execution
- \textit{List} is the basic data structure
Functional Programming

- based on the \(\lambda\)-calculus
- functions as “first-class” values
- recursion is the main paradigm for repeated execution
- \textit{List} is the basic data structure
- garbage collection
Mathematical Functions

Function: a mapping from a set (the *domain*) to another set (the *co-domain*)

\[ f : \mathcal{D} \rightarrow \mathcal{C} \]

Composition: \( h = f \circ g \)

\[ h(x) = f(g(x)) \]

used to define functions in terms of other functions
\( f(x) = x^2 \) (mathematical notation)

\( f = \lambda x . x^2 \) (\( \lambda \)-calculus notation)

useful in creating unnamed functions

\( \lambda x . E \)

formal parameter

body
Higher order functions
Paliath Narendran

Functional Programming
The Lambda Calculus
Scope
Nested definitions
Higher-order functions

λ-calculus

"pure" λ-calculus:

<expression> ::= <identifier>
  ::= λ <identifier> . <expression>
  ::= (<expression> <expression>)

λx.x stands for the identity function
\(\lambda\)-expressions in Scheme

built-in functions +, *, ...

\(\lambda\text{<list of formal parameters> <expression>}

Examples:

\(\lambda\text{(x) (* x x)}\)

\(\lambda\text{(x) (+ x 2)}\)

\(\lambda\text{(x y) x}\)
(\lambda (x\ y) (+\ x\ y))

is different from

(\lambda (x) (\lambda (y) (+\ x\ y)))
Creating functions

\((\text{lambda} \ (x_1 \ x_2 \ \ldots \ x_k) \ E)\)

When this functional object is invoked,

- the parameters are call-by-value
Creating functions

\[(\text{lambda } (x_1 \; x_2 \; \ldots \; x_k) \; E)\]

When this functional object is invoked,

- the parameters are call-by-value
- \(E\) is evaluated and
Creating functions

(lambda \((x_1 \ x_2 \ldots \ x_k)\) \(E\))

When this functional object is invoked,

- the parameters are call-by-value
- \(E\) is evaluated and
- the resulting value is returned
Lexical scoping

(\texttt{(define f \texttt{\( (\lambda (x_1 \ x_2 \ldots \ x_k) \ E) \)})})

The value of a \( \lambda \)-expression is a procedure (\textit{closure}) that consists of

- the list of parameters
- the body \( E \)
- the environment in which the \textit{free} variables in the body are bound at the time the \( \lambda \)-expression is evaluated

\textbf{lexical scoping:} values of free variables are looked up in the environment in which the procedure was \textit{defined}
Suppose the value of x is 2

1 ]=> (define foo (lambda (y) (+ x y)))

;Value: foo

1 ]=> (foo 1000)

;Value: 1002

1 ]=> (let ((x 1)) (foo 1000))

;Value: 1002
(define factorial
  (letrec
   ((f (lambda (n)
       (if (= n 0) 1
          (* n (f (- n 1))))
       )
     )
   )
  )
)

(define remove
  (letrec
    ((rhelp
       (lambda (x ls)
         (cond ((null? ls) '())
               ((equal? x (car ls))
                (rhelp x (cdr ls)))
               (else (cons (car ls)
                            (rhelp x (cdr ls)))))
         )
    ))
)
)
(define (<fun_name> <fp_1> ... <fp_n>) E)

is equivalent to

(define <fun_name>  
  (lambda (<fp_1> ... <fp_n>) E)
)
Nested definitions

\textit{internal definitions}

\begin{verbatim}
(define (remove x ls)
  (define (loop L M)
    (cond ((null? L) M)
          ((equal? x (car L)) (loop (cdr L) M))
          (else
           (loop (cdr L) (cons (car L) M))
          ))
  )
  (reverse (loop ls '()))
)

loop is not visible outside remove
\end{verbatim}
Higher-order functions

- functions as arguments
- functions as (part of) the return value

\[
\begin{align*}
\text{(define (compose f g) }
  \text{ (lambda (x) (f (g x)))))}
\end{align*}
\]

\[
\begin{align*}
((\text{compose 1+ 1+}) 100) & \rightarrow 102
\end{align*}
\]
map

\[(\text{map } f \ (a_1 \ldots a_n)) \equiv (f(a_1) \ldots f(a_n))\]

\[(\text{map } 1+ \ (10 \ 20 \ 30 \ 40))\]
\[\rightarrow (11 \ 21 \ 31 \ 41)\]

\[(\text{map } (\lambda x \ (* \ x \ x)) \ (10 \ 20 \ 30 \ 40))\]
\[\rightarrow (100 \ 400 \ 900 \ 1600)\]
mapcan

\[
(\text{define } (\text{mapcan } f \text{ ls}) \n  \text{ (if } (\text{null? } \text{ls}) \n    '() \n    \text{ (append } (f (\text{car } \text{ls})) (\text{mapcan } f \text{ (cdr } \text{ls}))) \n  ) \n)
\]

\[
(\text{mapcan } \text{cdr } '((1 2 3) (4 5 6) (7))) \n\Rightarrow (2 3 5 6)
\]
fold-left

Higher order functions
Paliath Narendran
Functional Programming
The Lambda Calculus
Scope
Nested definitions
Higher-order functions
fold-left

Higher order functions
Paliath Narendran

Functional Programming
The Lambda Calculus
Scope
Nested definitions
Higher-order functions

\[ \text{fold-left} \]

\[
f \quad \sigma \quad a_n \quad a_{n-1} \quad \ldots \quad a_1 \quad x \quad \text{list}
\]

The diagram shows a recursive structure for the fold-left operation, where each application of the function unfolds the list until an element is reached.
(define (fold-left f x ls)
  (define (lrhelp y L)
    (if (null? L) y
        (lrhelp (f y (car L)) (cdr L)))
    )
  )
  (lrhelp x ls)
)

(fold-left + 0 '(1 3 5 7 9 11))
⇒ 36
(fold-left
   (lambda (x y) (cons y x))
   '()
   '(1 3 5 7)
)

⇒ (7 5 3 1)

(fold-left (lambda (x y) (cons (1+ y) x))
   '()
   '(10 20 30)
)

⇒ (31 21 11)
fold-right

\[
\text{fold-right} \quad f \quad a_1 \quad f \quad a_2 \quad f \quad a_n \quad x
\]
fold-right

(define (fold-right f x ls)
  (define (frhelp ls)
    (if (null? ls)
      x
      (f (car ls) (frhelp (cdr ls)))))
  (frhelp ls))

(fold-right - 0 '(7 11 13))
⇒ 9
(fold-right + 0 '(1 3 5 7 9 11))
⇒ 36
Higher order functions

Paliath Narendran

Functional Programming

The Lambda Calculus

Scope

Nested definitions

Higher-order functions

(fold-right cons '() '(1 3 5 7))

⇒ (1 3 5 7)

(define (I x) x)

(define (myf x f) (lambda (y) (f (cons x y))))

((fold-right myf I '(1 3 5 7)) '())

⇒ (7 5 3 1)
Higher order functions

Paliath Narendran

Functional Programming

The Lambda Calculus

Scope

Nested definitions

Higher-order functions

\( \lambda x. (\text{cons} \ 7 \ (\text{cons} \ 5 \ (\text{cons} \ 3 \ (\text{cons} \ 1 \ x))) \)

\( \lambda x. (\text{cons} \ 7 \ (\text{cons} \ 5 \ x)) \)

\( \lambda x. (\text{cons} \ 7 \ x) \)

\( \lambda x. x \)