

Polymorphic Higher-Order Functions

CS 350

Dr. Joseph Eremondi

Last updated: July 17, 2024

Highly Generic Programming

Polymorphic Functions

- Higher-order functions are even more powerful when combined with type variables

Polymorphic Functions

- Higher-order functions are even more powerful when combined with type variables
- Allows us to say “This works on any type, as long as that type supports this kind of operation”

Polymorphic Functions

- Higher-order functions are even more powerful when combined with type variables
- Allows us to say “This works on any type, as long as that type supports this kind of operation”
- Express ideas like “do this to every element in a list”

Example: Sorting

Example: Sorting

```
(define (sortNumbers [xs : (Listof Number)]) : (Listof Number)
  ....)
```

*;; These implementations are probably doing 99% the same thing
;; except they're using different comparison operators*

```
(define (sortById [xs : (Listof (Number * String))])
  : (Listof (Number * String))
  ....)
```

;; What we really want is this:

```
(define (sortBy [xs : (Listof 'a)]
  [compare : ('a 'a -> Boolean)])
  : (Listof 'a)
  ....)
```

- Sort function that works on any type 'a

Example: Sorting

```
(define (sortNumbers [xs : (Listof Number)]) : (Listof Number)
  ....)
```

*;; These implementations are probably doing 99% the same thing
;; except they're using different comparison operators*

```
(define (sortById [xs : (Listof (Number * String))])
  : (Listof (Number * String))
  ....)
```

;; What we really want is this:

```
(define (sortBy [xs : (Listof 'a)]
  [compare : ('a 'a -> Boolean)])
  : (Listof 'a)
  ....)
```

- Sort function that works on any type 'a
 - So long as we have a comparison function compare that can find if one 'a value is \leq another

Map

- One of the most essential functions on list

Map

- One of the most essential functions on list
- For each element in the list, apply this function to each element

Map

- One of the most essential functions on list
- For each element in the list, apply this function to each element
 - Returns the resulting list, original list is unchanged

Map

- One of the most essential functions on list
- For each element in the list, apply this function to each element
 - Returns the resulting list, original list is unchanged
- If your function takes in type 'a and produces type 'b, then map can turn a (Listof 'a) into (Listof 'b)

Map

- One of the most essential functions on list
- For each element in the list, apply this function to each element
 - Returns the resulting list, original list is unchanged
- If your function takes in type 'a and produces type 'b, then map can turn a (Listof 'a) into (Listof 'b)

Map

- One of the most essential functions on list
- For each element in the list, apply this function to each element
 - Returns the resulting list, original list is unchanged
- If your function takes in type 'a and produces type 'b, then map can turn a (Listof 'a) into (Listof 'b)

```
(define (map [f : ('a -> 'b)]
  [xs : (Listof 'a)]) : (Listof 'b)
  (type-case (Listof 'a) xs
    [empty
     empty]
    [(cons x rest)
     (cons (f x)
            (map f rest))]))
```

Examples

Examples

```
(map (lambda (x) (* x 1001)) '(1 2 3 4))  
(map not '#t #f #f #t))  
(map some '("Hello" "Goodbye"))
```


Examples

```
(map (lambda (x) (* x 1001)) '(1 2 3 4))  
(map not '(#t #f #f #t))  
(map some '("Hello" "Goodbye"))
```

```
'(1001 2002 3003 4004)  
'(#f #t #t #f)  
(list (some "Hello") (some "Goodbye"))
```

Map does recursion so you don't have to

- Lots of times, we were writing code that looked exactly the same

Map does recursion so you don't have to

- Lots of times, we were writing code that looked exactly the same
- Higher-order functions and polymorphism let you turn those patterns into an actual **function**

- Another common function on lists

Filter

- Another common function on lists
- Takes a **predicate** for some type:

Filter

- Another common function on lists
- Takes a **predicate** for some type:
 - Look at two values and return either true or false

Filter

- Another common function on lists
- Takes a **predicate** for some type:
 - Look at two values and return either true or false
 - Defines a property on that type

Filter

- Another common function on lists
- Takes a **predicate** for some type:
 - Look at two values and return either true or false
 - Defines a property on that type
- Returns a new list containing only the elements satisfying the predicate

Filter

- Another common function on lists
- Takes a **predicate** for some type:
 - Look at two values and return either true or false
 - Defines a property on that type
- Returns a new list containing only the elements satisfying the predicate

Filter

- Another common function on lists
- Takes a **predicate** for some type:
 - Look at two values and return either true or false
 - Defines a property on that type
- Returns a new list containing only the elements satisfying the predicate

```
(define (filter [p : ('a 'a -> Boolean)]  
             [xs : (Listof 'a)]) : (Listof 'a)  
  (type-case (Listof 'a) xs  
    [empty  
     empty]  
    [(cons x rest)  
     ;; Check if the first element satisfies p  
     ;; If it does, include it in the results,  
     ;; otherwise omit  
     (if (p x)  
         (cons x (filter p rest))  
         (filter p rest))]))
```

Filter examples

Filter examples

```
(filter (lambda (x) (zero? (modulo x 2)))  
        '(1 2 3 4 5 6))  
(filter some?  
        (list (none) (some "Hello") (none) (some "Goodbye")  
              (none) (none) (some "Cheers") (none)))  
(filter (lambda (x) (> x 1000000))  
        (list 1 2 3 4 (* 100000 100000)) )
```

Filter examples

```
(filter (lambda (x) (zero? (modulo x 2)))  
      '(1 2 3 4 5 6))  
(filter some?  
      (list (none) (some "Hello") (none) (some "Goodbye")  
            (none) (none) (some "Cheers") (none)))  
(filter (lambda (x) (> x 1000000))  
      (list 1 2 3 4 (* 100000 100000)) )
```

```
'(2 4 6)  
(list (some "Hello") (some "Goodbye") (some "Cheers"))  
'(1000000000000)
```

Using Filter: The Functional Quicksort

Using Filter: The Functional Quicksort

```
(define (sortBy [compare : ('a 'a -> Boolean)]
              [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
     empty]
    [(cons first rest)
     (let*
      ([smallers
       (filter (lambda (x) (compare x first))
               rest)]
       [biggers
       (filter (lambda (x) (not (compare x first)))
               rest)])
      (append (sortBy compare smallers)
               (cons first
                     (sortBy compare biggers))))))
```

How Quicksort works

- An empty list is already sorted

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:
 - A sorted list of things smaller than (or equal to) the head

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:
 - A sorted list of things smaller than (or equal to) the head
 - The head

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:
 - A sorted list of things smaller than (or equal to) the head
 - The head
 - A sorted list of things greater than (or equal to) the head

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:
 - A sorted list of things smaller than (or equal to) the head
 - The head
 - A sorted list of things greater than (or equal to) the head
- If we append these together in that order, the result will still be sorted

How Quicksort works

- An empty list is already sorted
- If a list has at least one element, we can partition that list into everything smaller than that element, and greater than that element
- We recursively sort those lists
- This gives us 3 lists:
 - A sorted list of things smaller than (or equal to) the head
 - The head
 - A sorted list of things greater than (or equal to) the head
- If we append these together in that order, the result will still be sorted
 - And contains everything from the original list

Quicksort Examples

Quicksort Examples

```
(sortBy <= '(5 4 1 5 3 9 7))
```

```
(sortBy (lambda (x y) (<= (fst x) (fst y)))  
  (list (pair 5 "a") (pair 4 "b") (pair 1 "c") (pair 9 "d")))
```

```
(sortBy (lambda (s1 s2) (<= (string-length s1) (string-length s2)))  
  (list "goodbye" "hey" "hello" "a" "arithmetic" ))
```

Quicksort Examples

```
(sortBy <= '(5 4 1 5 3 9 7))
```

```
(sortBy (lambda (x y) (<= (fst x) (fst y)))  
  (list (pair 5 "a") (pair 4 "b") (pair 1 "c") (pair 9 "d")))
```

```
(sortBy (lambda (s1 s2) (<= (string-length s1) (string-length s2)))  
  (list "goodbye" "hey" "hello" "a" "arithmetic" ))
```

```
'(1 3 4 5 5 7 9)  
(list (values 1 "c") (values 4 "b") (values 5 "a") (values 9 "d"))  
'("a" "hey" "hello" "goodbye" "arithmetic")
```

- Combinators that are polymorphic are highly general

- Combinators that are polymorphic are highly general
 - Ways to build new functions out of old functions

Polymorphic Combinators

- Combinators that are polymorphic are highly general
 - Ways to build new functions out of old functions
- Often used to build up arguments to map or filter

Function Composition

- For any two functions, we can chain them together

Function Composition

- For any two functions, we can chain them together
 - If their types agree

Function Composition

- For any two functions, we can chain them together
 - If their types agree

Function Composition

- For any two functions, we can chain them together
 - If their types agree

```
;; Written that way to match the symbol in math  
(define (o [g : ('b -> 'c)]  
          [f : ('a -> 'b)]) : ('a -> 'c)  
  (lambda (x)  
    (g (f x))))  
;; (g (f x)) = ((o g f) x) for all x  
;; Arguments in that order so that this equation looks nice
```

Example

Example

```
(map (o (lambda (x) (* x 10)) add1)  
      '(1 2 3 4))
```

```
(filter (o not empty?)  
        (list '() '(1 2) '(3 2 1) '() '(1)))
```

Example

```
(map (o (lambda (x) (* x 10)) add1)  
      '(1 2 3 4))
```

```
(filter (o not empty?)  
        (list '() '(1 2) '(3 2 1) '() '(1)))
```

```
'(20 30 40 50)  
'((1 2) (3 2 1) (1))
```

- See type example on the board

Partial Application

- For functions that take multiple arguments, we can get a new function by providing only one argument

Partial Application

- For functions that take multiple arguments, we can get a new function by providing only one argument

Partial Application

- For functions that take multiple arguments, we can get a new function by providing only one argument

```
(define (curry [f : ('a 'b -> 'c)]  
           [x : 'a])  
  : ('b -> 'c)  
  (lambda (y) (f x y)))
```

- Can reverse order of arguments

Partial Application

- For functions that take multiple arguments, we can get a new function by providing only one argument

```
(define (curry [f : ('a 'b -> 'c)]  
           [x : 'a])  
  : ('b -> 'c)  
  (lambda (y) (f x y)))
```

- Can reverse order of arguments

Partial Application

- For functions that take multiple arguments, we can get a new function by providing only one argument

```
(define (curry [f : ('a 'b -> 'c)]  
             [x : 'a])  
  : ('b -> 'c)  
  (lambda (y) (f x y)))
```

- Can reverse order of arguments

```
(define (flip [f : ('a 'b -> 'c)])  
  : ('b 'a -> 'c)  
  (lambda (bVal aVal) (f aVal bVal)))
```

Example

Example

```
;; Gets (modulo x 2) for each x in the list  
(map (curry (flip modulo) 2)  
      '(1 2 3 4 5 6 7 8)))
```

Example

```
;; Gets (modulo x 2) for each x in the list  
(map (curry (flip modulo) 2)  
      '(1 2 3 4 5 6 7 8)))
```

```
'(1 0 1 0 1 0 1 0)
```

Point Free Programming

- When you build functions using combinators instead of lambda, it's called *point free programming*

Point Free Programming

- When you build functions using combinators instead of lambda, it's called *point free programming*
- Building programs becomes kind of like putting Lego together

Point Free Programming

- When you build functions using combinators instead of lambda, it's called *point free programming*
- Building programs becomes kind of like putting Lego together
- Generally, don't want to always use point-free programming

Point Free Programming

- When you build functions using combinators instead of lambda, it's called *point free programming*
- Building programs becomes kind of like putting Lego together
- Generally, don't want to always use point-free programming
 - Sometimes the lambda is just clearer

Point Free Programming

- When you build functions using combinators instead of lambda, it's called *point free programming*
- Building programs becomes kind of like putting Lego together
- Generally, don't want to always use point-free programming
 - Sometimes the lambda is just clearer
- But can be easier to read in many cases