High-Level Functional Programming

Bob Keller
September 2010

First, A Few Words About Bindings, Scope, Environment

- A **binding** is the association of a **value** to an **identifier** (aka "variable").
- The **purpose** of binding is to be able to use the identifier in place of the value.
  - The value may be messy to write/read.
  - The value may be determined by some computation.

'let in Racket

- One means of establishing a binding is to use a **'let** expression:

(The red parens are explained shortly.)

```
(let (
      (freq  (/ 1 (* 2 pi)))
     )
    (* 3 freq) )
```

This expression is evaluated to get the result of the let. It may **use** the binding.

Think of this as an **equation** establishing a **binding**:
LHS = RHS
freq = (/ 1 (* 2 pi))

Special Forms

- A **special form** in Racket is not a function being applied.
- Certain identifiers are have a special built-in meaning, that signals a special form.
- Examples of such identifiers are:
  define, let, let*, if, cond, case, lambda

The 'let Special Form

(let (
  … *0 or more equations* …
)
 … *result expression* …)

The reason for the red parens is that there can be more than one equation.
These parens group the equations, separating them from the result expression.

Environments

- An **environment** is a **set** of bindings.
- The let expression effectively establishes an environment for evaluation of the result expression.

*bindings determined in equations **augment** those in the **outer** environmnent of the **let** expression*

(let (
  … *0 or more equations* …
)
 … *result expression* …)

*result expression can use bindings in the **inner** enviroment*

Outer vs. Inner Environments

Suppose that a is bound to 5 outside.

```
(let (
        (b 3)
        (c (* 2 a))
     )
   (+ a b c)
)                         Result is:
```

A Tricky Thing

- The right-hand sides of the equations in a 'let expression are evaluated in the **outer** environment.
- Any **shadowing** of a binding (re-binding an identifier) are not seen in that environment.

Tricky Environments Illustrated

Suppose that a is bound to 5 outside.

```
(let (
        (a 3)
        (c (* 2 a))
     )                    evaluated in outer environment
   (+ a c)
)                         Result is:
```

Scope

- By the scope of an identifier, we mean the set of places where that identifier has its given meaning.

The **scope** of this 'a is the **result expression** only, not the RHS of the next equation.

```
(let (
        (a 3)
        (c (* 2 a))
     )
   (+ a c)
)
```

let* vs. let

- let* is another special form. It has the same syntax as let, except for let* vs. let.
- In let* the environment is augmented with each new equation.
- The scope of a variable includes the RHS of equations that follow and the result expression (unless shadowing occurs).

let* example

The **scope** of this 'a is the **result expression *and*** the RHS of the next equation.

```
(let* (
        (a 3)
        (c (* 2 a))
      )
   (+ a c)
)
```

The environment here is not the outer environment, but rather the outer enviroment with a rebound to 3.

Result is:

Shadowing is Non-Destructive

- Although a variable may be shadowed, by re-binding in an inner environment, it retains its original value in the outer environment.
- Returning to that environment finds the variable to have the same binding.
- This is called "referential transparency".
- This is why **definitions**, such as those provided by let, are **not regarded as assignment statements.**

let* as nested lets

- (let* ((v1 e1) (v2 e2) … (vn en)) result)

  is same as

- ```
  (let ( (v1 e1) )
     (let ( (v2 e2) )
         …
          (let ( (vn en) ) result) …) )
  ```

Why do I start with high-level programming rather than recursion?

- Recursion is not needed to understand the **effect** of a function.
- High-level programming is often a simpler way to achieve a given goal.
- We'll get to recursion soon enough.

Two Ways to Define Factorial

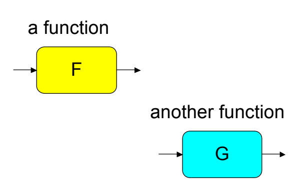
- ```
  (define (fac n)
     (if (< n 2)
          1
          (* n (fac (- n 1)))))
  ```
  Must understand this definition as a whole.
- ```
  (define (fac n)
     (foldl * 1 (range 1 n)))
  ```
  Can understand this definition in two pieces: **foldl**, **range**.

  [Assuming range is already defined.]
- This is just for comparison. Neither of these is the most efficient way.

Why Functional Programming is So Important

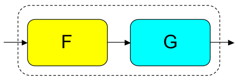
- No side effects:
  - Easier to debug
  - Easier to get parallel execution (e.g. on a "multi-core" system)
- Composability: Easier to compose complex functions from simpler ones

Composabilty


  ```

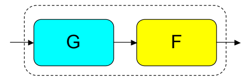
Composabilty

a third function



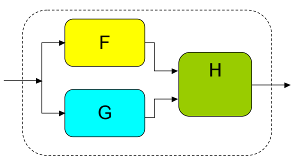
Composabilty

a fourth function

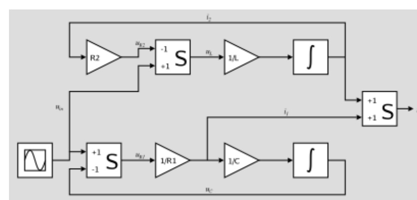


More Ways to Compose

to get still more functions



Engineering Applications



"But can I get a job?"

from various recruiting websites

Google groups

Boston Area Scala Enthusiasts

Functional programming job fair

LinkedIn Home Profile Contacts

Functional Programming

FP Roles - Are you a competent functional programmer?
Posted 6 days ago by James Wood, Consultant at Kaizen Partnership
Follow discussion | Add comment »

Don Stewart
R&D Lead at Galois Inc
See all Don's activity »
Follow Don

Galois is Hiring Functional Programmers
Galois is hiring motivated software engineers with very strong functional programming skills, in particular in Haskell.

Please see the full announcement http://www.haskell.org/pipermail/haskell/2009-June/021401.html

Posted 7 months ago | Reply Privately

OCaml News

Putting the fun in functional programming since 2005!

Saturday, 19 May 2007

Functional programmer stole my Job

Middle-class white-collar American computer programmers are feeling the squeeze again today, and its not from Sharon in accounts.

Only months after many programmers lost their jobs when they were offshored to third world countries like Europe, it seems that programming jobs are now being stolen by a second crowd.

Gangs of ruthless functional programmers are overwhelming interviewers by listing programming languages like OCaml, Haskell and even Lisp on their CVs.

The horrifying trend is thought to have begun in Germany, with a growing number of singles citing "OCaml" among their hobbies. In the US, starving unemployed programmers were spotted on street corners holding signs saying "Will code for Food (but only in OCaml)".

The effect is being compounded by the number of employers who now regard unenlightened programmers as second-class citizens, driving a trend of employing functional programmers in the interests of productivity. The result: functional programming is the new black.

The only programmers unaffected by this revolution are the self-employed, who have known for years that functional programming offers order of magnitude improvements in development speed, maitainability, reliability and mojo.

You have to ask yourself one question: is OCaml on your CV?
Posted by Flying Frog Consultancy Ltd. at 06:41
Labels: haskell, lisp, ocaml, ruby, scheme

(Increasingly) Popular Functional Languages

- Haskell
- Erlang
- OCaml
- Scala
- F#
- Clojure

So should I become a functional programmer?

- Yes, but ***don't stop there***.
- Have functional skills in your toolkit, but be able to work outside that domain as well.
- Be "amphibious" and "agnostic" and exploit the best of what the community has to offer.

mapping over a list

- **map** applies a 1-ary function to each element of a list, returning a list of the same length, with the results of the applications in order

```
> (define (cube x) (* x x x))

> (map cube '(1 2 3 4 5))
(1 8 27 64 125)
```

More mapping examples

```
> (map symbol->string '(I should care))
("I" "should" "care")

> (map string-length (map symbol->string '(I should care)))
(1 6 4)

> (map reverse '( (Washington George) (Lincoln Abraham)
                  (Jefferson Thomas) (Obama Barack)))
((George Washington) (Abraham Lincoln)
 (Thomas Jefferson) (Barack Obama))
```

The *Type* of map

- (map Function List) List
$A \rightarrow B \times A^* \rightarrow B^*$
- So map:$(A \rightarrow B) \times A^* \rightarrow B^*$
- A and B could be the same type
- map **preserves the length** of its argument list

n-ary map

- map will apply an n-ary function to n equal-length lists "pointwise"

```
> (map + '(1 2 3 4) '(9 8 7 6))
(10 10 10 10)

> (map list '(1 2 3 4) '(9 8 7 6))
((1 9) (2 8) (3 7) (4 6))
```

The type of n-ary map

- map:$(A^n \rightarrow B) \times (A^*)^n \rightarrow B^*$
- All argument lists must be the same length in Racket

foldr and foldl

- These functions “fold” a list into something like an element of the list.
- The first argument is a 2-ary function.
- The second argument is the result for an empty list.
- The third argument is the list being folded.

```
> (define (demo x y) (list '+ x y))

> (foldl demo 0 '(1 2 3 4 5))
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))

> (foldr demo 0 '(1 2 3 4 5))
(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))
```

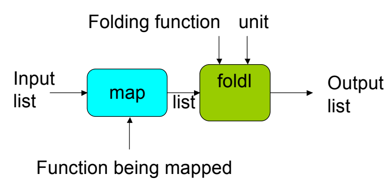
```
> (foldl + 0 '(1 2 3 4 5))
15
> (foldl * 1 '(1 2 3 4 5))
120
```

The type of foldl/foldr

(foldl	Function	Unit	List)	Element
	$A \times A \rightarrow A$	A	$A^* \rightarrow$	A

- So foldl:$\underbrace{(A \times A \rightarrow A)}_{\text{meaning the set of 2-ary functions on A}} \times A \times A^* \rightarrow A$

Composing map and foldl



The famous **mapReduce**!

fold vs. reduce

- ***reduce*** is the functional programming identifier for folding when the folding direction is non-specific.
- reduce is not in Racket with that name.
- **map/reduce** in combination achieved fame from Google’s many uses of it.
- Google recently *patented* its use of map/reduce!?!

On Google’s Patent

Google's patent on MapReduce could potentially pose a problem for those using third-party open source implementations. Patent #7,650,331, which was granted to Google on Tuesday, defines a **system and method for efficient large-scale data processing**:

> **A large-scale data processing system and method includes one or more application-independent map modules configured to read input data and to apply at least one application-specific map operation to the input data to produce intermediate data values, wherein the map operation is automatically parallelized across multiple processors in the parallel processing environment. A plurality of intermediate data structures are used to store the intermediate data values. One or more application-independent reduce modules are configured to retrieve the intermediate data values and to apply at least one application-specific reduce operation to the intermediate data values to provide output data.**

Reactions

› Open Ended # Open source and programming

Google's MapReduce patent: what does it mean for Hadoop?

By Ryan Paul | Last updated January 20, 2010 10:10 AM

The USPTO awarded search giant Google a software method patent that covers the principle of distributed MapReduce, a strategy for parallel processing that is used by the search giant. If Google chooses to aggressively enforce the patent, it could have significant implications for some open source software projects that use the technique, including the Apache Foundation's popular Hadoop software framework.

Han Soete

"Map" and "reduce" are functional programming primitives that have been used in software development for decades. A "map" operation allows you to apply a function to every item in a sequence, returning a sequence of equal size with the processed values. A "reduce" operation, also called "fold," accumulates the contents of a sequence into a single return value by performing a function that combines each item in the sequence with the return value of the previous iteration.

Hadoop

Hadoop

From Wikipedia, the free encyclopedia

Apache Hadoop is a Java software framework that supports data-intensive distributed applications under a free license.[1] It enables applications to work with thousands of nodes and petabytes of data. Hadoop was inspired by Google's MapReduce and Google File System (GFS) papers.

Hadoop is a top-level Apache project, being built and used by a community of contributors from all over the world.[2] Yahoo! has been the largest contributor[3] to the project and uses Hadoop extensively in its web search and advertising businesses.[4] IBM and Google have announced a major initiative to use Hadoop to support university courses in distributed computer programming.[5]

Hadoop was created by Doug Cutting (now a Cloudera employee),[6] who named it after his son's stuffed elephant.[7] It was originally developed to support distribution for the Nutch search engine project.[8]

Our own map-reduce

```
> (define (map-reduce binary unit unary L)
          (foldr binary unit (map unary L)))

> (map-reduce + 0 length '((1 2 3) (4 5) (6) ()))
6
```

Averaging a Non-Empty List

```
> (define (average L) (/ (foldl + 0 L) (length L)))

> (average '(1 2 3 4 5 6 7 8 9))
5

> (map average '((1 2 3) (2 3 4) (3 4 5) (5 6 7)))
(2 3 4 6)

> (average (map length '((1 2 3) (2 3 4) (3 4 5) (5 6 7))))
3

> (average (map average '((1 2 3) (2 3 4) (3 4 5) (5 6 7))))
3 3/4
```

Filtering a List

- Function **filter** keeps elements that satisfy a predicate argument.

```
>  (define (even x) (= 0 (modulo x 2))) ;; = is a numeric test

> (filter even '(1 2 3 4 5 7 9 10))
(2 4 10)
```

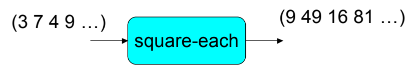
Example: Generalized Anagram

- Suppose spaces "don't count".
- How would you define function anagram?
- Filter out the spaces.

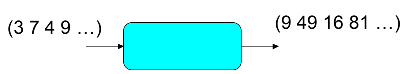
Anonymous Functions

- We functions don't need **names** to do our job.
- You can define us *anonymously* to:
  - Avoid having to think up names
  - Avoid cluttering up the namespace with temporary functions

A Named Function



An Anonymous Function Doing the Same Job



Functions as "First-Class Citizens"

Disposition of data types in a typical language

Type	Need name?	Use as argument	Return as result	Put in structure
Number	No	Yes	Yes	Yes
String	No	Yes	Yes	Yes
Function	?	?	?	?

In Racket, the answer to these is Yes

Lambda Expressions

- One way to specify an anonymous function uses the idea of "lambda expression"

  (lambda (x) …)

  means

  "the function that, with argument x, returns the value computed by …"

Lambda Expression Examples

- (lambda (x) (+ 5 x))
  "the function that adds 5 to its argument"
- (lambda (x y) (expt y x))
  "the function that raises its second argument y to the power of the first argument x"
- (lambda (x) (list x x))
  "the function that makes a 2-element list of its argument twice in succession"

Lambda Expressions are *applied* just like any other function

```
> ( (lambda (x) (+ 5 x))  99)
104

> ( (lambda (x y) (expt y x)) 10 2)
1024

> ( (lambda (x) (list x x))  "foobar")
("foobar" "foobar")
```

mapping and filtering with lambda expressions

- This kind of usage is very common:

```
> (map (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)

> (filter (lambda (x) (> x 3)) '(1 2 3 4 5 6))
(4 5 6)
```

Imported Variables in Lambda Expressions

- By "imported" I mean variables that are not arguments. These are sometimes called "free variables" in the lambda expression.
- These variables retain the meaning they had at the time of the function's definition. This is called **static scope**.
- They do not change their meaning based on context (which would be **dynamic scope**).

Example of Imported Variable

- Below, b is ***imported*** into the lambda expression.

imported

```
> (let (
        (b 99)
       )
   (map (lambda (x) (+ x b)) '(1 2 3 4 5))
  )
(100 101 102 103 104)
```

not imported (an *argument*)

How to Spot Imported Variables

- They are not arguments.
- They are not defined locally inside the lambda expression (e.g. in a **let** form).

Imported Values Bind Statically in Racket

Functions should not be chameleons.

```
> (let* (
         (b 99)
         (f (lambda (x) (+ b x)))
        )
    ( (lambda (b) (f 1)) 1000)
   )
                 more recent
                 binding of b
100
```

Early implementations of Lisp tended to get this wrong. It was called the "Funarg problem". [Google it]

How Static Binding is Implemented

- The compiler turns a function into a "closure".
- Racket shows closures **cryptically**, as <procedure> ☹

```
> (lambda (x) (+ 5 x))
#<procedure>
```

- A closure contains:
  - Reference to imported values
  - Code for evaluating the function, given the arguments
- Closures live "on the heap". They do not disappear when the stack shrinks.

Racket tries to do "the right thing" for the user's convenience

y not bound yet

```
> (define f (lambda (x) ( + x y)))

> (define y 5)
> (f 10)
15

> (let ( (y 100) )
   (f 10)
   )
15
```

Does not re-bind y dynamically.

Functions Returning Functions

imported

```
> (define (add n)  (lambda (x) (+ x n)))

> (map (add 5) '(1 2 3 4 5))
(6 7 8 9 10)

> (map (add 10) '(1 2 3 4 5))
(11 12 13 14 15)
```

(add n) returns a function, for any argument n

The Same Definition *Not* Using Lambda

```
> (define ((add n) x) (+ x n))

> (map (add 5) '(1 2 3 4 5))
(6 7 8 9 10)
```

This is called "Currying" the add function [Google this],

in honor of logician Haskell B. Curry.

The idea is due to Moses Schönfinkel .
Linguists called it "Schönfinkelisation".

The Type of (add 5) in Racket

```
> (add 5)

#<procedure>
```

Functions that both take and give functions as arguments

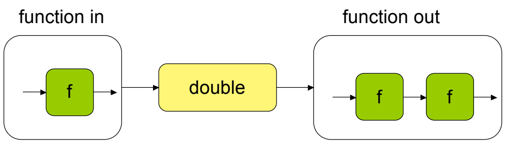
Returns a function that applies its argument twice in succession. (Not related to numeric doubling.)

```
> (define (double f) (lambda (x) (f (f x))))

> (define (square x) (* x x))
> ((double square) 5)

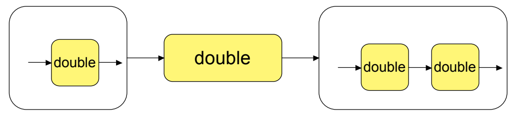
625
```

In Pictures



(double double)

Is this meaningful?



What would it do?

(double double)

```
> (double double)
#<procedure>

> (((double double) square) 5)
152587890625

> (square (square (square (square 5))))
152587890625
```

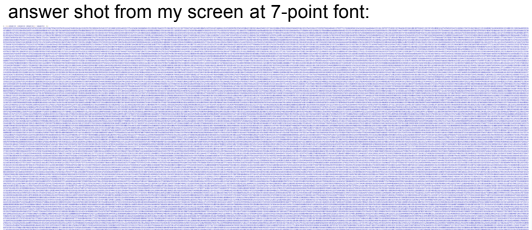
How Big?

(((double (double double)) square) 5)

How Big?

(((double (double double)) square) 5)

answer shot from my screen at 7-point font:



```
> (/ (log (((double (double double)) square) 5)) (log 10))
45,807 ; decimal digits
```

Composing Functions Using Functions

```
> (define (compose f g) (lambda (x) (f (g x))))

> (define (cube x) (* x x x))
> ((compose cube square) 5)
15625
> ((compose square cube) 5)
15625
> (define (add2 n) (+ 2 n))
> ((compose add2 square) 5)
27
> ((compose square add2) 5)
49
```

Draw a picture