Functional Programming

Robert Bieber

March 1, 2011
Outline

1 Preliminaries
   • John Backus
   • Issues With von Neumann Languages

2 A Functional Programming System
   • Foundations
   • Components of an FP System
   • The Algebra of Programs

3 Practical Functional Programming
   • Haskell
   • Ruby
Born 1924.
Died 2007.
Directed development of FORTRAN, released in 1957.
Backus-Naur Form.
Received Turing Award in 1977 for his work on programming languages.
The von Neumann Architecture

This diagram illustrates the von Neumann Architecture, which consists of a memory and a CPU connected by a data path. The diagram highlights the von Neumann Bottleneck, indicating a potential bottleneck in data transfer between these components.
Excessive "Housekeeping" Code

/*
 * Computes dot product of two vectors
 */

class SimpleVectorOperations {
    public int dotProduct(int[] a, int[] b) {
        int res = 0;
        for (int i = 0; i < a.length; i++)
            res += a[i] * b[i];
        return res;
    }
}
Lack of Useful Algebraic Properties

- Example:

\[ f(x) + 2f(x) + 3f(x) \]
\[ = (1 + 2 + 3)f(x) \]
\[ = 6f(x) \] (1)

- Compare to:

```plaintext
int y = 1*f(x) + 2*f(x) + 3*f(x);
```

- Is \( y == 6*f(x) \)?
Potential Problems

- Not if `f` references global variables.

  ```c
  int f(int x)
  {
    return someGlobalVariable++;
  }
  ```

- Or if `f` interacts with the user.

  ```c
  int f(int x)
  {
    int ret;
    scanf("%d", &ret);
    return ret;
  }
  ```
Characteristics of an FP System

- The system is not history-sensitive.
- Functions have no side-effects.
- Functions are formed by combining other functions.
- Programs built in an FP system can be transformed by useful algebraic properties.
Structure of an FP System

- A set $O$ of objects.
- A set $F$ of primitive functions.
- The application operation.
- A set $F$ of functional forms.
- A set $D$ of defined functions.
Objects are the data on which our programs operate.

- Atoms are numbers and combinations of characters: 12, T, F, FOO
- Lists are any combination of atoms: <3, 4, 5>, <A, B, <C, D>>, φ
- ⊥ represents “bottom,” or “undefined” Any list containing ⊥ is equivalent to ⊥, and any function that operates on ⊥ returns ⊥.
Primitive Functions

- A function is a mapping from the set of objects to the set of objects.
- Functions accept a single argument and return a single result.
- The primitive functions are those “built-in” to the system.
- Functions are applied with the application operator :.
- For any function $f$ and any object $x$, $f : x \equiv$ the result of applying $f$ to $x$. 
Function Application

input → function → output
Examples of Function Application

- \( id : A = A \)
- \( + : < 3, 4 > = 7 \)
- \( 2 : < 5, 6, 7 > = 6 \)
- \( - : 1 = \bot \)
- \( tl : \bot = \bot \) (Remember that all functions are \( \bot \) preserving).
- \( trans : < < 1, 2 >, < 5, 6 > > = < < 1, 5 >, < 2, 6 > > \)
Functional Forms

- Functional forms are expressions that represent functions.
- Functional forms are operators with functions as their operands.
- We can use functional forms to construct more useful functions from the supplied primitives.
Examples of Functional Forms

- Composition: \( f \circ g : x \equiv f : (g : x) \)
- Apply to All: \( \alpha f :< x_1, x_2, ...x_n > \equiv < f : x_1, f : x_2, ...f : x_n > \)
- Insert: \( / f :< x_1, x_2, ...x_n > \equiv f :< x_1, / f : x_2, ...x_n > > \)
- Construction: \([f_1, f_2, ...f_n] : x \equiv < f_1 : x, f_2 : x, ...f_n : x > \)
- Condition: \( (p \rightarrow f; g) \equiv (p : x) = T \rightarrow f : x; g : x \)
- Constant: \( \overline{x} : y \equiv x \)
Composition

\[ f \circ g \]

Diagram:
- Input
- Transformation through functions \( g \) and \( f \)
- Output
Construction

\[ [f, g] \]
Conditional

\[ p \rightarrow f; g \]

Diagram:
- Input
- Condition `p`
- Conditional: `if T then f` \( \quad \) `if F then g` \( \quad \) `else ⊥`
- Output
Definitions

- A definition, or defined function, associates a name with a functional form.
- Definitions are analogous to function definitions in von Neumann languages.
- A Definition consists of a name on the left side and an expression consisting only of functional forms involving primitive and other defined functions on the right side.
- Takes the form \( \text{Def name} \equiv \text{expression} \)
Examples of Functional Programs

- $\text{Def } sub1 \equiv - \circ [id, 1]$
  $\text{Def } eq0 \equiv eq \circ [id, 0]$
  $\text{Def } fact \equiv eq0 \rightarrow 1; \times \circ [id, fact \circ sub1]$

- Collatz Sequence:
  $\text{Def } eq1 \equiv eq \circ [id, 1]$
  $\text{Def } div2 \equiv \div \circ [id, 2]$
  $\text{Def } mult3add1 \equiv + \circ [1, \times \circ [id, 3]]$
  $\text{Def } collatzstep \equiv \text{even} \rightarrow div2; mult3add1$
  $\text{Def } collatz \equiv eq1 \rightarrow <1>; \text{apndl} \circ [id, collatz \circ collatzstep]$
Algebraic Properties of FP Systems

- Unlike von Neumann languages, FP systems exhibit useful algebraic properties.
- Using these properties, we can manipulate programs algebraically the way we manipulate mathematical expressions in high school algebra.
- We can prove properties of programs using the programming language itself.
Examples of Algebraic Laws of the FP System

- \([f_1, \ldots, f_n] \circ g \equiv [f_1 \circ g, \ldots, f_n \circ g]\)
- \(\alpha f \circ [g_1, \ldots, g_n] \equiv [f \circ g_1, \ldots, f \circ g_2]\)
- \((p \rightarrow f; g) \circ h \equiv p \circ h \rightarrow f \circ h; g \circ h\)
- \([..., \perp, ...] \equiv \perp\)
- \(f \circ \perp \equiv \perp\)
- \(f \circ id \equiv f\)
Properties of Haskell

- Pure functional language: no concept of state, functions have no side-effects (except for I/O).
- Unlike Backus’ FP system, handles multiple argument functions by currying.
- User writes program by specifying type signatures and function definitions.
- Functions are first class objects and can be passed to other functions.
Example Haskell Program

collatzStep :: Int -> Int
collatzStep x = if x `mod` 2 == 0 then x `div` 2 else 3 * x + 1

collatz :: Int -> [Int]
collatz x = if x == 1 then [1] else x : (collatz $ collatzStep x)

main = do
  x <- getLine
  putStrLn $ show $ collatz $ read x
Execution

```bash
bieber@bieber-laptop:~$ ./collatz
1
[1]
bieber@bieber-laptop:~$ ./collatz
2
[2,1]
bieber@bieber-laptop:~$ ./collatz
3
[3,10,5,16,8,4,2,1]
bieber@bieber-laptop:~$ ./collatz
20
[20,10,5,16,8,4,2,1]
bieber@bieber-laptop:~$
```
Pointfree Style

- Avoid use of variable names in function definitions.
- Favor function composition and partial application.
- Very similar to Backus’ style.
Examples

A function, `sumx2`, which multiplies every element of a list by 2 and sums the resultant list.

- Normal:
  \[\text{sumx2 } xs = \text{sum } (\text{map } (*2) \ xs)\]

- Pointfree:
  \[\text{sumx2 } = \text{sum} \ . \ \text{map } (*2)\]
Properties of Ruby

- Dynamic object-oriented scripting language.
- Allows methods to accept "blocks" as their final argument: basically anonymous functions.
- Blocks minimize "housekeeping code" that Backus cited.
File I/O Example

- C Example:
  ```c
  FILE* fin = fopen("test.txt", "r");
  while(!feof(fin))
  {
    fscanf(fin, "%s\n", text);
    printf("%s\n", text);
  }
  fclose(fin);
  ```

- Ruby Example:
  ```ruby
  IO.foreach('test.txt') do |line|
    puts line
  end
  ```
Summary

- Imperative programming languages are inflexible, and don’t exhibit useful algebraic properties.
- FP systems allow the user to create programs by combining simpler functions, and these programs can be manipulated algebraically.
- In modern languages we have both purely functional languages and imperative/object-oriented languages which incorporate functional features.
J. Backus.  
*Communications of the ACM, 21(8):*, 1978.

Haskell Wiki.  
Pointfree  
http://www.haskell.org/haskellwiki/Pointfree

Ruby Documentation  
Guides and API Documentation  
http://www.ruby-doc.org/