Functional Imperative Style

Jan van Eijck
CWI & ILLC, Amsterdam

August 7, 2012

Literate Programming, Again

```
module FunctionalImperative

where

import List
```

Loops in Imperative Programming

```
x := 0;
n := 0;
while n < y do
    {
      x := x + 2*n + 1;
      n := n + 1;
    }
return x;
```

What a Functional Programmer Might Write

```
f :: Int -> Int
f y = f' y 0 0

f' :: Int -> Int -> Int -> Int
f' y x n = if n < y then
             let
               x' = x + 2*n + 1
               n' = n + 1
             in f' y x' n'
             else x
```

This replaces a while loop by a recursive function call.

Reasoning About While Loops

To show that the imperative version computes the value of y^2 in x, the key is to show that the loop invariant $x = n^2$ holds for the while loop:

```
{ x == n^2 }
  x := x + 2*n + 1;
  n := n + 1;
{ x == n^2 }
```

Reasoning About Recursion

Recursive procedures suggest inductive proofs. In this case we can use induction on y to show that f' returns the square of y, for non-negative y, as follows.

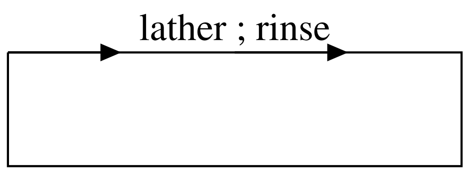
Base case If $y = 0$, then $f'\ 0\ 0\ 0$ returns 0, by the definition of f'. This is correct, for $0^2 = 0$.

Induction step Assume for $y = m$ the function call $f'\ m\ x\ m$ returns x with $x = m^2$. We have to show that for $y = m + 1$, the function call $f'(m+1)\ x\ m$ returns $(m+1)^2$.

$\ldots$

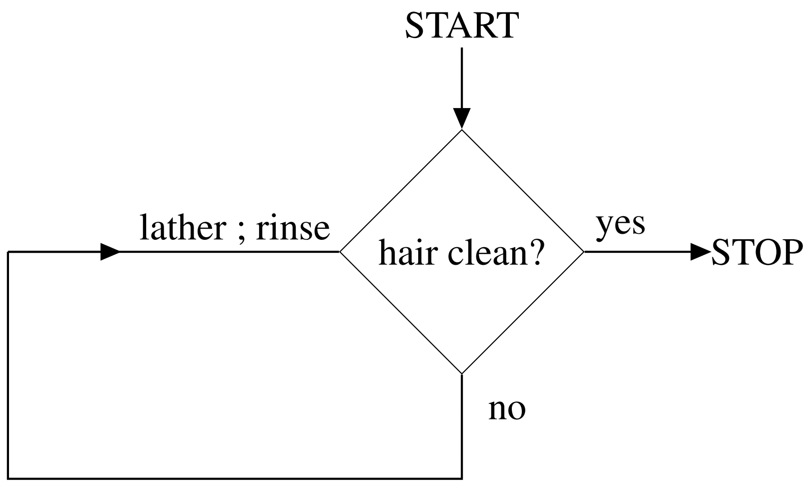
The Essence of a While Loop

If taken literally, the compound action ‘lather, rinse, repeat’ would look like this:



Repeated Actions With Stop Condition

repeat the lather rinse sequence until your hair is clean. This gives a more sensible interpretation of the repetition instruction:



Written as an Algorithm

Hair wash algorithm

- while hair not clean do:

  1. lather;
  2. rinse.

While in Haskell

The two ingredients are:

- a test for loop termination;
- a step function that determines the parameters for the next step in the loop.

The termination test takes a number of parameters and returns a boolean, the step function takes the same parameters and computes new values for those parameters.

While with a Single Parameter

Suppose for simplicity that there is just one parameter. Here is an example loop:

- while even x do
  $x := x \div 2$.

Here $\div$ is the 'div' operator for integer division. The result of $x \div y$ is the integer you get if you divide x by y and throw away the remainder. Thus, $9 \div 2 = 4$.

Functional Version

The functional version has the loop replaced by a recursive call:

```
g x = if even x then
      let
        x' = x `div` 2
      in g x'
      else x
```

Combination of Test and Step

g has a single parameter, and one can think of its definition as a combination of a test p and a step h, as follows:

```
p x = even x
h x = x `div` 2

g1 x = if p x then g1 (h x)
       else x
```

Let's make this explicit . . .

While Loop With Single Parameter

Here is a definition of the general form of a while loop with a single parameter:

```
while1 :: (a -> Bool) -> (a -> a) -> a -> a
while1 p f x
  | p x       = while1 p f (f x)
  | otherwise = x
```

While Defined in Terms of Until

Another way to express this is in terms of the built-in Haskell function `until`:

```
neg :: (a -> Bool) -> a -> Bool
neg p = \x -> not (p x)

while1 = until . neg
```

This allows us to write the function g as follows:

```
g2 = while1 p h
```

Reformulation

It looks like the parameters have disappeared, but we can write out the test and step functions explicitly:

```
g3 = while1 (\x -> even x) (\x -> x `div` 2)
```

But this can be abbreviated again:

```
g3' = while1 even (`div` 2)
```

This is the functional version of the loop. This is how close functional programming really is to imperative programming.

Example: Least Fixpoint Algorithm

Least fixpoint algorithm

- while $x \neq f(x)$ do
  $x := f(x)$.

Least Fixpoint, Functional Style

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x   = x
        | otherwise = lfp f (f x)
```

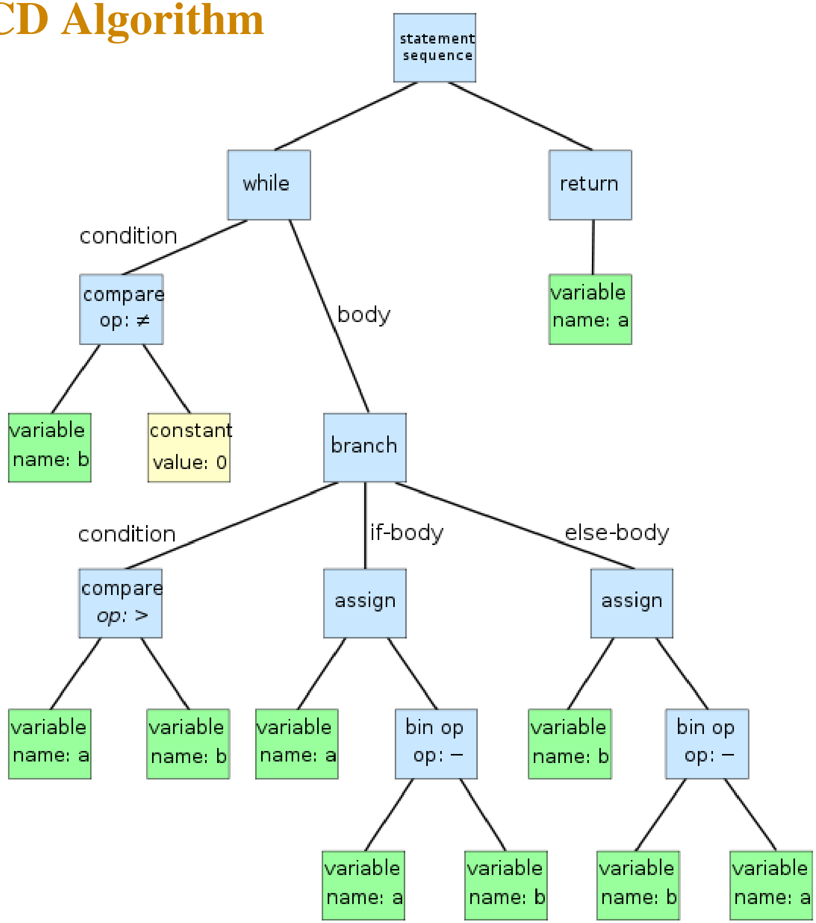
Least Fixpoint, Functional Imperative Style

```
lfp' :: Eq a => (a -> a) -> a -> a
lfp' f = while1 (\x -> x /= f x) (\x -> f x)
```

While With Two Parameters

```
while2 :: (a -> b -> Bool)
       -> (a -> b -> (a,b))
       -> a -> b -> b
while2 p f x y
  | p x y     = let (x',y') = f x y in
                              while2 p f x' y'
  | otherwise = y
```

Euclid's GCD Algorithm



GCD Algorithm in Imperative Pseudo-code

Euclid's GCD algorithm

1. while $x \neq y$ do
   if $x > y$ then $x := x - y$ else $y := y - x$;
2. return y.

GCD Algorithm in Functional Imperative Style

```
euclidGCD :: Integer -> Integer -> Integer
euclidGCD = while2
             (\ x y -> x /= y)
             (\ x y -> if x > y
                       then (x-y,y)
                       else (x,y-x))
```

Squaring Function in Functional Imperative Style

```
sqr :: Int -> Int
sqr y = let
          n = 0
          x = 0
        in sqr' y n x

sqr' y = while2
           (\ n _ -> n < y)
           (\ n x -> (n+1, x + 2*n + 1))
```

Note the use of an anonymous variable _.

While With Three Parameters

```
while3 :: (a -> b -> c -> Bool)
       -> (a -> b -> c -> (a,b,c))
       -> a -> b -> c -> c
while3 p f x y z
  | p x y z     = let
                    (x',y',z') = f x y z
                  in while3 p f x' y' z'
  | otherwise = z
```

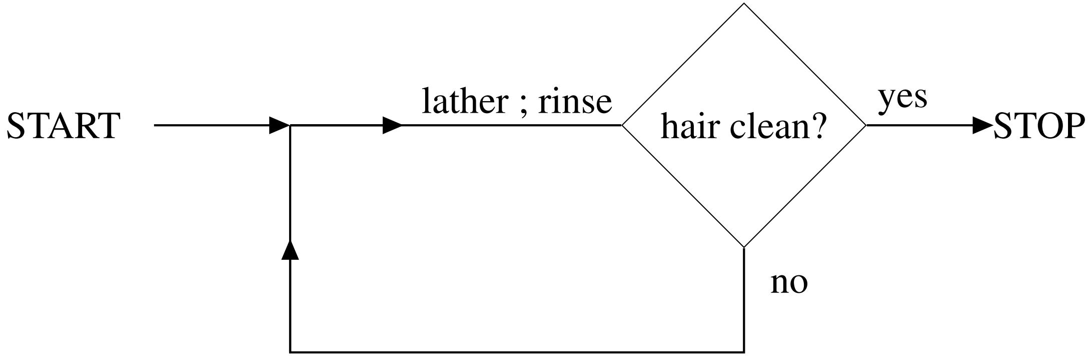
Repeat Loops in Functional Imperative Style

Another hair wash algorithm

- repeat
  1. lather;
  2. rinse;

  until hair clean.

In a Picture



Repeat in Terms of While

$$\text{repeat } P \text{ until } C$$

is equivalent to:

$$P; \text{ while } \neg C \text{ do } P.$$

This gives us a recipe for repeat loops in functional imperative style, using repeat wrappers such as the following …

Repeat1

```
repeat1 :: (a -> a) -> (a -> Bool) -> a -> a
repeat1 f p = while1 (\ x -> not (p x)) f . f
```

Repeat2

```
repeat2 :: (a -> b -> (a,b))
        -> (a -> b -> Bool) -> a -> b -> b
repeat2 f p x y = let
     (x1,y1) = f x y
     negp    = (\ x y -> not (p x y))
  in while2 negp f x1 y1
```

Repeat3

```
repeat3 :: (a -> b -> c -> (a,b,c))
        -> (a -> b -> c -> Bool) -> a -> b -> c -> c
repeat3 f p x y z = let
     (x1,y1,z1) = f x y z
     negp       = (\ x y z -> not (p x y z))
  in while3 negp f x1 y1 z1
```

For Loops in Functional Imperative Style

A natural way to express an algorithm for computing the factorial function, in imperative style, is in terms of a "for" loop:

Factorial Algorithm

1. $t := 1$;
2. for i in $1 \ldots n$ do $t := i * t$;
3. return t.

For Wrapper in Haskell

For a faithful rendering of this in Haskell, we define a function for the “for” loop:

```
for :: [a] -> (a -> b -> b) -> b -> b
for [] f y = y
for (x:xs) f y = for xs f (f x y)
```

This gives the following Haskell version of the algorithm:

```
fact :: Integer -> Integer
fact n = for [1..n] (\ i t -> i*t) 1
```

With Initialisation

If we wish, we can spell out the initialisation, as follows:

```
fact :: Integer -> Integer
fact n = let
           t = 1
         in fact' n t

fact' :: Integer -> Integer -> Integer
fact' n = for [1..n] (\ i t -> i*t)
```

Version With While Loop

Let's contrast this with a version of the algorithm that uses a "while" loop:

Another Factorial Algorithm

1. $t := 1$;
2. while $n \neq 0$ do
   (a) $t := n * t$;
   (b) $n := n - 1$;
3. return t.

Functional Imperative Version

In functional imperative style, this becomes:

```
factorial :: Integer -> Integer
factorial n = let
                t = 1
              in factorial' n t

factorial' = while2 (\ n _ -> n /= 0)
                    (\ n t ->  let
                      t' = n*t
                      n' = n-1
                    in (n',t'))
```

Digression on Fold

Wherever imperative programmers use “for” loops, functional programmers tend to use `fold` constructions.

The pattern of recursive definitions over lists consists of matching the empty list `[]` for the base case, and matching the non-empty list `(x:xs)` for the recursive case. Witness:

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Foldr

This occurs so often that Haskell provides a standard higher-order function that captures the essence of what goes on in this kind of definition:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []       = b
foldr f b (x:xs) = f x (foldr f b xs)
```

Foldr in Action

Here is what happens if you call `foldr` with a function f, and identity element z, and a list $[x_1, x_2, x_3, \ldots, x_n]$:

$$\text{foldr } f\ z\ [x_1, x_2, ..., x_n] = (f\ x_1\ (f\ x_2\ (f\ x_3\ \ldots\ (f\ x_n\ z)\ldots).$$

And the same thing using infix notation:

$$\text{foldr } f\ z\ [x_1, x_2, ..., x_n] = (x_1\ `f`\ (x_2\ `f`\ (x_3\ `f`\ (\ldots(x_n\ `f`\ z)\ldots).$$

For instance, the `and` function can be defined using `foldr` as follows:

```
and = foldr (&&) True
```

Foldl

While `foldr` folds to the right, the following built-in function folds to the left:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z0 xs0 = lgo z0 xs0
             where
                lgo z []       =  z
                lgo z (x:xs) = lgo (f z x) xs
```

Foldl in Action

If you apply `foldl` to a function $f :: \alpha \rightarrow \beta \rightarrow \alpha$, a left identity element $z :: \alpha$ for the function, and a list of arguments of type β, then we get:

$$\text{foldl } f \; z \; [x_1, x_2, ..., x_n] = (f \ldots (f(f(f \; z \; x_1) \; x_2) \; x_3) \ldots x_n)$$

Or, if you write f as an infix operator:

$$\text{foldl } f \; z \; [x_1, x_2, ..., x_n] = (\ldots(((z \text{ `}f\text{` } x_1) \text{ `}f\text{` } x_2) \text{ `}f\text{` } x_3) \; \ldots \; \text{`}f\text{` } x_n)$$

Factorial in Terms of Product

The standard way to define the factorial function in functional programming is:

```
factorial n = product [1..n]
```

Sum and Product in Terms of Foldl

The function `product` is predefined. If we look up the definition of `sum` and `product` in the Haskell prelude, we find:

```
sum, product        :: (Num a) => [a] -> a
sum                 =  foldl (+) 0
product             =  foldl (*) 1
```

For2

Here is a version of “for” where the step function has an additional argument:

```
for2 :: [a] -> (a -> b -> c -> (b,c))
           -> b -> c -> c
for2 [] f _ z = z
for2 (x:xs) f y z = let
    (y',z') = f x y z
  in
    for2 xs f y' z'
```

For3

With two additional arguments:

```
for3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
           -> b -> c -> d -> d
for3 [] f _ _ u = u
for3 (x:xs) f y z u = let
    (y',z',u') = f x y z u
  in
    for3 xs f y' z' u'
```

And so on.

Fordown

We can also count down instead of up:

```
fordown :: [a] -> (a -> b -> b) -> b -> b
fordown = for . reverse
```

```
fordown2 :: [a] -> (a -> b -> c -> (b,c))
                -> b -> c -> c
fordown2 = for2 . reverse
```

```
fordown3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
                      -> b -> c -> d -> d
fordown3 = for3 . reverse
```

Summary, and Further Reading

This lecture has introduced you to programming in functional imperative style.

Iteration versus recursion is the topic of chapter 2 of the classic [1]. This book is freely available on internet, from address `http://infolab.stanford.edu/~ullman/focs.html`.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. Foundations of Computer Science — C edition. W. H. Freeman, 1994.