



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Functions and State

Principles of Reactive Programming

Martin Odersky

Functions and State

Until now, our programs have been side-effect free.

Therefore, the concept of *time* wasn't important.

For all programs that terminate, any sequence of actions would have given the same results.

This was also reflected in the substitution model of computation.

Reminder: Substitution Model

Programs can be evaluated by *rewriting*.

The most important rewrite rule covers function applications:

$$\begin{array}{l} \text{def } f(x_1, \dots, x_n) = B; \dots f(v_1, \dots, v_n) \\ \rightarrow \\ \text{def } f(x_1, \dots, x_n) = B; \dots [v_1/x_1, \dots, v_n/x_n] B \end{array}$$

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

→ `if (1 == 0) 3 else iterate(1-1, square, square(3))`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

→ `if (1 == 0) 3 else iterate(1-1, square, square(3))`

→ `iterate(0, square, square(3))`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

→ `if (1 == 0) 3 else iterate(1-1, square, square(3))`

→ `iterate(0, square, square(3))`

→ `iterate(0, square, 3 * 3)`

Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

```
→ if (1 == 0) 3 else iterate(1-1, square, square(3))  
→ iterate(0, square, square(3))  
→ iterate(0, square, 3 * 3)  
→ iterate(0, square, 9)
```


Rewriting Example:

Say you have the following two functions `iterate` and `square`:

```
def iterate(n: Int, f: Int => Int, x: Int) =  
  if (n == 0) x else iterate(n-1, f, f(x))  
def square(x: Int) = x * x
```

Then the call `iterate(1, square, 3)` gets rewritten as follows:

→ `if (1 == 0) 3 else iterate(1-1, square, square(3))`

→ `iterate(0, square, square(3))`

→ `iterate(0, square, 3 * 3)`

→ `iterate(0, square, 9)`

→ `if (0 == 0) 9 else iterate(0-1, square, square(9))` → 9

Observation:

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ -calculus, the theory behind functional programming.

Example:

```
if (1 == 0) 3 else iterate(1 - 1, square, square(3))
```

Observation:

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ -calculus, the theory behind functional programming.

Example:

```
if (1 == 0) 3 else iterate(1 - 1, square, square(3))
```

```
iterate(0, square, square(3))
```

Observation:

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ -calculus, the theory behind functional programming.

Example:

```
if (1 == 0) 3 else iterate(1 - 1, square, square(3))
```

[illegible]

Stateful Objects

One normally describes the world as a set of objects, some of which have state that *changes* over the course of time.

An object *has a state* if its behavior is influenced by its history.

Example: a bank account has a state, because the answer to the question

“can I withdraw 100 CHF ?”

may vary over the course of the lifetime of the account.

Implementation of State

Every form of mutable state is constructed from variables.

A variable definition is written like a value definition, but with the keyword `var` in place of `val`:

```
var x: String = "abc"  
var count = 111
```

Just like a value definition, a variable definition associates a value with a name.

However, in the case of variable definitions, this association can be changed later through an *assignment*, like in Java:

```
x = "hi"  
count = count + 1
```

State in Objects

In practice, objects with state are usually represented by objects that have some variable members. **Example:** Here is a class modeling a bank account.

```
class BankAccount {  
  private var balance = 0  
  def deposit(amount: Int): Unit = {  
    if (amount > 0) balance = balance + amount  
  }  
  def withdraw(amount: Int): Int =  
    if (0 < amount && amount <= balance) {  
      balance = balance - amount  
      balance  
    } else throw new Error("insufficient funds")  
}
```

State in Objects (2)

The class `BankAccount` defines a variable `balance` that contains the current balance of the account.

The methods `deposit` and `withdraw` change the value of the `balance` through assignments.

Note that `balance` is private in the `BankAccount` class, it therefore cannot be accessed from outside the class.

To create bank accounts, we use the usual notation for object creation:

```
val account = new BankAccount
```


Working with Mutable Objects

Here is a worksheet that manipulates bank accounts.

```
val account = new BankAccount    // account: BankAccount = BankAccount
account deposit 50                 //
account withdraw 20               // res1: Int = 30
account withdraw 20               // res2: Int = 10
account withdraw 15               // java.lang.Error: insufficient funds
```

Applying the same operation to an account twice in a row produces different results. Clearly, accounts are stateful objects.

Statefulness and Variables

Remember the definition of streams (lazy sequences) in [week 7, #progfun](#). Instead of using a lazy val, we could also implement non-empty streams using a mutable variable:

```
def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {  
  def head = hd  
  private var tlOpt: Option[Stream[T]] = None  
  def tail: T = tlOpt match {  
    case Some(x) => x  
    case None => tlOpt = Some(tl); tail  
  }}  

```

Question: Is the result of cons a stateful object?

☐ Yes

☐ No

Statefulness and Variables (2)

Consider the following class:

```
class BankAccountProxy(ba: BankAccount) {  
  def deposit(amount: Int): Unit = ba.deposit(amount)  
  def withdraw(amount: Int): Int = ba.withdraw(amount)  
}
```

Question: Are instances of BankAccountProxy stateful objects?

☐ Yes

☐ No