A very short introduction to functional programming

Scala and OCaml...

Will Qi

December, 2013

Outline - Scala

- Functional programming through λ-calculus
- Scala syntax in 10 minutes
- Scala object system and pattern matching
- Scala type system and tools for building abstractions
- Reactive programming

Outline - OCaml

- λ-calculus in OCaml
- Lists, variants and pattern matching
- Powerful module system
- Concurrent programming with Async

Functions

- In C++ a typical function application look like:

```
int foo(int a, char b, bool c) {
  // body
}

foo (10, 'x', false);
```

Functions are not first class citizens, it's hard to compose functions, although it's much better with `std::function` and `bind` that came with C++11.

Functions

- In Scala functions can be defined:

```
def foo(a: Int, b: Char, c: Boolean): Int = {
  if (c) a
  else b.toInt
}
```

or in a curried form:

```
def foo(a: Int)(b: Char)(c: Boolean): Int = {
  if (c) a
  else b.toInt
}
```

Observe two things. Scala functions do not use `return` keyword as in C++. Scala functions can be defined as a curried function. These are two distinctive features of functional programming.

Functions

- Currying is the process of transforming a function with multiple arguments into chained partial applications, e.g.

```
val f1: Char => Boolean => Int = foo(10)
val value = f1('x')(true) // 10
```

- In OCaml this is much more concise:

```
let f a b c = if c then a else int_of_char b
> f : int -> char -> bool -> int
let f1 = f 10
let value = f1 'x' true
```