Functional Programming in Typed Racket

Florian Florian
fbie@itu.dk

IT University of Copenhagen & UCAS

2016-05-26

Who am I?

Just call me Florian!

Who am I?

Just call me Florian!

- I am from Germany and live in Denmark.

Who am I?

Just call me Florian!

- I am from Germany and live in Denmark.
- Second year PhD student at IT University of Copenhagen and UCAS.

Who am I?

Just call me Florian!

- I am from Germany and live in Denmark.
- Second year PhD student at IT University of Copenhagen and UCAS.
- My Chinese is really bad (but I am trying to learn!)

Who am I?

Just call me Florian!

- I am from Germany and live in Denmark.
- Second year PhD student at IT University of Copenhagen and UCAS.
- My Chinese is really bad (but I am trying to learn!)
- If you do not understand what I say or if I speak too fast, **please tell me right away!**

Why Functional Programming?

Functional programming is old but recently it has become very popular!

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- Makes it easier to think about what a program does.

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- Makes it easier to think about what a program does.
- Good for parallel programming, performance (see next lecture).

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- Makes it easier to think about what a program does.
- Good for parallel programming, performance (see next lecture).
- Used in financial industry, modern compilers and more.

Why Functional Programming?

Functional programming is old but recently it has become very popular!

- Makes it easier to think about what a program does.
- Good for parallel programming, performance (see next lecture).
- Used in financial industry, modern compilers and more.
- Jobs in functional programming pay better!

Typed Racket

Racket and its sister-language **Typed Racket** are Scheme-dialects.

Typed Racket

Racket and its sister-language **Typed Racket** are Scheme-dialects.

- As the names say, Racket is untyped, Typed Racket has types.

Typed Racket

Racket and its sister-language **Typed Racket** are Scheme-dialects.

- As the names say, Racket is untyped, Typed Racket has types.
- Functional language, no side-effects (mostly).

Typed Racket

Racket and its sister-language **Typed Racket** are Scheme-dialects.

- As the names say, Racket is untyped, Typed Racket has types.
- Functional language, no side-effects (mostly).
- Great meta-programming (we won't look at that).

Hello World in Typed Racket

```racket
;; Tell run-time which language to use.
#lang typed/racket

;; Now print something.
(print "Nihao!")
```

A More Interesting Program

```
#lang typed/racket

;; Define x as the result of the computation.
(define x (* (+ 2 4) (+ 42 9)))
```

A More Interesting Program

```
#lang typed/racket

;; Define x as the result of the computation.
(define x (* (+ 2 4) (+ 42 9)))
```

We can check its value in the interactive mode:

```
> x
59
```

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Operators are also just functions:

`(+ x y)`	$\Rightarrow$	$x + y$
`(> x y)`	$\Rightarrow$	$x > y$
`(/ x y)`	$\Rightarrow$	$\frac{x}{y}$
`(f x y)`	$\Rightarrow$	$f(x, y)$

Do You Think This Looks Strange?

In Racket, all expressions are written like this:

```
(function arg1 arg2 ... argN)
```

Operators are also just functions:

`(+ x y)`	$\Rightarrow$	$x + y$
`(> x y)`	$\Rightarrow$	$x > y$
`(/ x y)`	$\Rightarrow$	$\frac{x}{y}$
`(f x y)`	$\Rightarrow$	$f(x, y)$

Some functions are *variadic*:

`(+ x y z)` $\Rightarrow$ $x + y + z$

Local Bindings

Just like local variables in Java, but you can never change them!

```
#lang typed/racket

(let ([x (* 2 16)]
      [y (* 3 17)])
    (+ x y))
```

What does this program do?

Note: You cannot reference `x` and `y` after the last closing parenthesis of the `let` expression!

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- Functions need no return statement. Their return value is the last executed statement!

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- Functions need no return statement. Their return value is the last executed statement!
- Type annotations start with `:` and describe the type of a symbol.

A First Function

```
#lang typed/racket

(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

Several new things on this slide:

- Functions need no return statement. Their return value is the last executed statement!
- Type annotations start with `:` and describe the type of a symbol.
- The type of `times-two` is $Number \to Number$.

Types in Typed Racket

We write the function type $A \to B$ as:

```
(-> A B)
```

where `A` is the parameter type and `B` is the return type.

Types in Typed Racket

We write the function type $A \to B$ as:

```
(-> A B)
```

where `A` is the parameter type and `B` is the return type.

```
(-> A B C)
```

Which types are parameter types, which ones are return types?

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))
(define (twice a)
  (pair a a))
```

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))
(define (twice a)
  (pair a a))
```

"For all types `A`, the type of `twice` is such that iff you pass it a value of some type `A` it will return a pair of values of type `A`."

Polymorphic Types

Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))
(define (twice a)
  (pair a a))
```

“For all types `A`, the type of `twice` is such that iff you pass it a value of some type `A` it will return a pair of values of type `A`.”

It’s just like generic types in Java!

Local Bindings, Revisited

Let’s turn our local binding example into a function!

```
(: two-three-sum (-> Number Number Number))
(define (two-three-sum a b)
  (let ([x (* 2 a)]
        [y (* 3 b)])
    (+ x y)))
```

Local Bindings, Revisited

Let's turn our local binding example into a function!

```
(: two-three-sum (-> Number Number Number))
(define (two-three-sum a b)
  (let ([x (* 2 a)]
        [y (* 3 b)])
      (+ x y)))
```

Now, we can call it like any other function:

```
> (two-three-sum 1 1)
5
> (two-three-sum 20 20)
100
```

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)
2
```

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)
2
```

Here, we have defined a lambda expression and applied it to the value `2`. We call this function the *identity function*.

Anonymous Functions

You can define functions without giving them a name. Such functions are called *lambda expressions*:

```
> ((lambda (x) x) 2)
2
```

Here, we have defined a lambda expression and applied it to the value `2`. We call this function the *identity function*.

We will often come across lambda expressions in functional programming!

Structs in Typed Racket

Structs are containers for values.

```
(struct (myNumberBox [value : Number]))
(struct (myStringBox [value : String]))
```

Structs in Typed Racket

Structs are containers for values.

```
(struct (myNumberBox [value : Number]))
(struct (myStringBox [value : String]))
```

We can also use type polymorphism here:

```
(struct (A) (myPolyBox [value : A]))
```

That Was a Quick Intro

Next, we will do a live coding session!

That Was a Quick Intro

Next, we will do a live coding session!

- I code on the big screen and show you how to do functional programming in Racket.

That Was a Quick Intro

Next, we will do a live coding session!

- I code on the big screen and show you how to do functional programming in Racket.
- You help with ideas and suggestions and experiment yourselves!

That Was a Quick Intro

Next, we will do a live coding session!

- I code on the big screen and show you how to do functional programming in Racket.
- You help with ideas and suggestions and experiment yourselves!
- All code we write will be available on
  - `github.com/fbie/parallel-functional-programming`

That Was a Quick Intro

Next, we will do a live coding session!

- I code on the big screen and show you how to do functional programming in Racket.
- You help with ideas and suggestions and experiment yourselves!
- All code we write will be available on
  - `github.com/fbie/parallel-functional-programming`
- You can download Racket from
  - `racket-lang.org`

Java Equivalent to Maybe

```java
public abstract class Maybe<A> {}

public class None<A> extends Maybe<A> {
  public None() {}
}
public class Some<A> extends Maybe<A> {
  public final A a;
  public Some(A a) {
    this.a = a;
  }
}
```

Java Equivalent to Cons List

```java
public abstract class LinkedList<A> {}

public class Nil<A> extends LinkedList<A> {
  public Nil() {}
}
public class Cons<A> extends LinkedList<A> {
  public final A a;
  public final LinkedList<A> tail;

  public Cons(A a, LinkedList<A> tail) {
    this.a = a;
    this.tail = tail;
  }
}
```

Java Equivalent to Binary Tree

```
public abstract class BinaryTree<A> {}

public class Leaf<A> extends BinaryTree<A> {
  public final A a;
  public Leaf(A a) {
    this.a = a;
  }
}
public class Node<A> extends BinaryTree<A> {
  public final BinaryTree<A> left, right;
  public Node(BinaryTree<A> left,
              BinaryTree<A> right) {
    this.left = left; this.right = right;
  }
}
```