Roy Documentation

Release 0.1

Brian McKenna

March 16, 2014

Contents

Contents:

CHAPTER 1

Introduction

Roy is a programming language that targets JavaScript. It has a few main features:

- Damas-Hindley-Milner type inference
- Whitespace significant syntax
- Simple tagged unions
- Pattern matching
- Structural typing
- Monad syntax
- Not-horrible JS output

Most of these features are common in statically-typed, functional languages, such as:

- Haskell
- OCaml
- Scala

1.1 Why JavaScript?

JavaScript is a necessity for web applications. It is the only feasible language you can natively run in your web browser.

A lot of developers are now using JavaScript outside of the browser; you can use node.js and Rhino to write server-side JavaScript.

1.2 What not just write JavaScript?

As universal as JavaScript has become, the language itself has problems:

1.2.1 Boilerplate

Creating functions is something that functional programmers want to do but JavaScript makes this a bit too verbose:

```
var x = function(a, b, c){
  return a + b + c;
};
```

One solution would be to come up with a shorthand syntax for functions and have implicit returns. We might also be able to get rid of those braces.

1.2.2 Tolerance

JavaScript tries to guess what the developer is trying to do:

```
var two = 1 + true;
console.log(two == "2"); // true
```

It doesn't really make sense to add the number `1` to the Boolean `true` value. It doesn't really make sense to compare the number `2` to the string `"2"`.

For correctness, we should show an error instead of trying to guess what the programmer wants.

1.2.3 Complexity

JavaScript allows multiple ways to do the same thing. Which is the correct way to construct a Person?

```
var p1 = Person();
var p2 = new Person();
```

It depends on the library and can be a source of confusion.

1.2.4 Dangerous

It can be easy to do things with unintentional side-effects. JavaScript has the `var` keyword to create local variables but unqualified assignments write to the global object:

```
var x = 10;

function getX() {
  x = 100; // forgot 'var'
  return x;
}

console.log(getX()); // 100
console.log(x); // 100
```

1.3 What can we do?

We've identified some problems with JavaScript, what can we do to fix it?

- Try to replace JavaScript in the browser with another language
- Try to replace JavaScript in the browser with a general purpose bytecode
- Change the JavaScript standard
- Compile from another language to JavaScript

The last option is the path of least resistance. In fact, there's already quite a few languages that compile to JavaScript, the most popular being CoffeeScript, haXe and Objective-J.

There also ways to compile Haskell, OCaml and Scala to JavaScript. These can help with writing statically-typed, functional code for the browser but they usually have a few downsides:

- Hard/impossible to interoperate with JavaScript libraries
- Generate a lot of code
- Generate code that requires a hefty runtime
- Must be compiled on the server-side (not in the browser)

1.4 The Roy solution

After trying to write correct programs in JavaScript and languages that compile to JavaScript, Roy was created. Roy tries to keep close to JavaScript semantics for ease of interoperability and code generation. It's also written in JavaScript so that it can compile code from the browser.

One of the biggest ideas when coming from JavaScript is the use of compile-time type-checking to remove type errors. We'll cover that in the next chapter.

CHAPTER 2

Types

The Curry-Howard isomorphism states that types are theorems and programs are proofs. Roy takes the stance of not generating programs (proofs) if the types (theorems) don't make sense.

This is called static type-checking and is a useful tool for writing correct programs.

2.1 Primitives

Roy implements the same primitive types as JavaScript:

- Number
- Boolean
- String
- Array
- Object

In Roy, Arrays and Objects have special semantics and are composed of multiple types. We'll separately cover typing of *Arrays* and typing of *Objects*.

2.2 The Read Evaluate Print Loop

Roy has an interactive mode which allows compilation and execution of code.

The simplest example is a value. Putting in the number `1` will respond back with that same value and its type:

```
roy> 1
1 : Number
```

We could also give a string of characters:

```
roy> "Hello world!"
Hello world! : String
```

Or a Boolean:

```
roy> true
true : Boolean
```

Evaluating expressions will also tell you the type of the result:

```
roy> 1 + 1
2 : Number
```

Let's make the type-checking fail:

```
roy> 1 + "Test"
Error: Type error: Number is not String
```

Notice that it doesn't give you an answer to the expression. JavaScript at this point would instead guess what you meant and give you an answer of `"1Test"`.

2.3 Strings

These behave similar to a Javascript String, but they won't have methods attached to them. String concatentation is done with the '++' operator.

2.4 Arrays

Arrays are homogeneous, meaning that they can only hold elements of the same type. For example, we can make an Array of Numbers:

```
roy> [1, 2, 3]
1,2,3 : [Number]
```

Trying to treat it as a heterogeneous collection will result in a type error:

```
roy> [1, true, 3]
Error: Type error: Number is not Boolean
```

Access array elements with the `@` operator:

```
roy> [1,2,3] @ 1
2 : Number
```

2.5 Objects

Objects use structural subtyping. An object is a "subtype" of another object if it satisfies all of the properties that the other object has.

An empty object is the supertype of all objects:

```
roy> {}
[object Object] : {}
```

An object containing a single property is a subtype of only the empty object:

```
roy> {a: 100}
[object Object] : {a: Number}
```

This property can be used to write well-typed code that works on object properties:

```
roy> let a = {a:100}
roy> let b = {a:5, b:5}
roy> let f o = o.a + 6
roy> f a
106 : Number
roy> f b
11 : Number
roy> let d = {b:100}
roy> f d
Error: Type error: {b: Number} is not {a: Number}
```

2.6 Interoperating with JavaScript

Referring to unknown identifier will assume that the identifier refers to a native JavaScript global.

For example, you can refer to `console.log`, something not known natively to Roy:

```
roy> console.log "Hello!"
Hello!
```

2.7 Using Native types

Given Roy's current limitations, you may want to use a Native type sometimes:

```
roy> "abc".length
Error: Parse error on line 2: Unexpected '.'

roy> (String "abc")
abc : Native
roy> (String "abc").length
3 : Native
```

2.8 Regular Expressions

Roy does not have direct support for regular expressions, including literals like /exp/

To use a regular expression in Roy you need one of the following approaches:

- Have an existing RegExp
- Create a native RegExp using the RegExp constructor
- Invoke match on a Native String, which converts the matching String to a RegExp

```
roy> (String "abcd").match "a.c"
["abc"] : Native

roy> (RegExp("a.c")).exec 'abcd'
["abc"] : Native
```

If you want, you can try and shorten up RegExp construction:

```
roy> let r s = RegExp s
roy> r "a.c"
/a.c/ : Native
roy> r"a.c"
/a.c/ : Native

roy> (r"a.c").exec "abcd"
["abc"] : Native
```

Access array elements with the @ operator

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*