Editor: Steve Vinoski • vinoski@ieee.org

Roy

A Statically Typed, Functional Language for JavaScript

Brian McKenna • *Atlassian*

Despite numerous attempts to replace it, JavaScript, for all practical purposes, is the only language for the client-side Web. Functional programmers targeting the Web will almost certainly have to deal with JavaScript at some time. Although writing functional programs in JavaScript is possible, it requires considerable discipline and convention.

Recently, languages that compile to JavaScript have surged in popularity (see http://altjs.org for a list of some). The most popular is CoffeeScript (http://coffeescript.org), which sticks closely to JavaScript's semantics but greatly changes its syntax. These changes make functional programming a bit nicer and require less discipline.

But sticking too closely to JavaScript's semantics keeps many of its warts. Even the biggest JavaScript fans are quick to point out its flaws, many of which float around the loose typing and scoping rules.

After working on browser-based and node.js-based software and being unable to fully reason about typing rules, I decided to look at alternative approaches to writing JavaScript code.

I evaluated compilers that take Haskell or ML and output JavaScript. These systems have large runtimes and unreadable JavaScript output, and make it hard to interoperate with existing JavaScript. From my trials, I had a strong feeling that the JavaScript community would never embrace them.

But CoffeeScript is starting to gain traction in the JavaScript community — in part because it compiles to readable JavaScript. It also includes a distinct compilation step, where it can include a static type-checking phase. CoffeeScript programs can't be completely type-checked due to semantics of the language — type-checking is only feasible in a language with restricted semantics.

So, I was looking for a language that was statically-typed, functional, and had lightweight, readable JavaScript output. No language seemed to satisfy all of these properties, so I started work on Roy.

Roy's Solution

I built Roy specifically to target JavaScript. This means Roy's compiler should know about JavaScript's primitives. Table 1 shows Roy's built-in types, along with their JavaScript representations. The "structure" type uses structural typing as a form of inheritance, which I discuss later.

Roy arrays are variable-length and homogeneous (they can only hold values of a single type), whereas Roy tuples are fixed-length and heterogeneous (each value can have a different type). Roy's type system uses Damas-Milner type inference. This algorithm is global, which means it works on a program without any type annotations. It also tries to be as generic as possible — you only have to write type annotations to restrict what a function accepts.

Roy is also written in JavaScript, which lets it compile source code inside the browser and execute it on the fly. This is particularly useful during development.

Getting Started

Roy runs either on node.js or in the browser. If you want to quickly play with Roy, you can try the online compiler (http://roy.brianmckenna.org). If you want to run Roy on node.js, it's available via npm (http://npmjs.org):

```
$ npm install roy
```


Published by the IEEE Computer Society 1089-7801/12/$31.00 © 2012 IEEE

You can then compile and run a Roy program with the following:

```
$ roy program.roy
$ node program.js
```

Or, you can use Roy's *read-eval-print-loop* (REPL) to start exploring the language:

```
$ roy
Roy: Small functional
  language that compiles to
  JavaScript
Brian McKenna
  <brian@brianmckenna.org>
  (http://brianmckenna.org/)
:? for help
roy> 1 + 1
2 : Number
roy> "Hello world"
Hello world : String
roy>
```

Code Samples

Let's generate some output:

```
roy> console.log (40 + 2)
42
```

The `console.log` function is built into most JavaScript runtimes. Roy doesn't know about the `console` object, so it just treats the `log` access as untyped and emits the call. If we run the previous segment, we'll get 42 printed to the screen.

The type system is strong and removes JavaScript's coercion rules. If we try to add a string and a number, we instead get a compile-time error:

```
roy> console.log ("40" + 2)
Error: Type error: String is
  not Number
```

This is a simple example of how Roy can help us write programs with fewer bugs — it can prove very simple inconsistencies.

We can define a function:

```
roy> let id x = x
```

Table 1. Roy's built-in types, along with their JavaScript representations.

Roy	JavaScript
Boolean	Boolean
Number	Number
String	String
Structure	Object
Array	Array (homogeneous)
Tuple	Array (heterogeneous)
Function	Function

This identity function just returns the value that it receives. We can use the `:t` directive to look at the type:

```
roy> :t id
Function(#a, #a)
```

`id` is polymorphic in the `x` parameter — we could pass in any type. If we want to be more restrictive, we can give an explicit type annotation:

```
roy> let f x: Number = x
roy> f 100
100 : Number
roy> f "100"
Error: Type error: String is
  not Number
```

Each of the valid expressions compile into lightweight, readable JavaScript:

```
console.log(40 + 2);
var id = function(x) {
    return x;
};
var f = function(x) {
    return x;
};
f(100);
```

Structures

Roy implements structural typing. We can view this as a static form of duck typing. The top-most type is `{}`, representing a structure with no properties:

```
roy> let structures (x: {})
  = x
```

All other structures are subtypes. We can pass any structure to the previous function:

```
roy> structures {property: 100}
{"property":100} : {}
```

Type-inference works on these properties:

```
roy> let incrementAge
  o = o.age + 1
roy> :t incrementAge
Function({age: Number},
  Number)
```

We can see that it takes any structure with a numeric age property:

```
roy> incrementAge {name:
  "Brian", age: 21}
22 : Number
```

One downside of structural typing is that you can get very long error messages:

```
roy> let longInput x =
  x.a + x.b + x.c
roy> longInput {a: 100,
  b: 100, d: 200}
Error: Type error: {a: Number,
  b: Number, d: Number} is not
  {a: Number, b: Number,
  c: Number}
```

Roy's workaround is to allow type aliasing:

```
roy> type Person = {firstName:
  String, lastName: String}
```

```
roy> let listIsEmpty lst = match lst
...>   case (Cons _ _) = false
...>   case Nil        = true
...>
roy> let listMap f lst = match lst
...>   case (Cons x r) = Cons (f x) (listMap f r)
...>   case Nil        = Nil
...>
roy> let listFilter p lst = match lst
...>   case (Cons x r) = if (p x) then
...>       Cons x (listFilter p r)
...>     else
...>       listFilter p r
...>   case Nil = Nil
...>
```

Figure 1. Higher-order functions on lists. We define these functions using pattern matching.

Now we can give the alias as an explicit parameter:

```
roy> let getName (x: Person) =
  x.firstName ++ " " ++
  x.lastName
roy> getName {firstName:
  "Brian", lastName:
  "McKenna"}
Brian McKenna : String

roy> getName {}
Error: Type error: {} is not
  Person
```

Roy's structures compile down to plain JavaScript objects:

```
var structures = function(x) {
    return x;
};
structures({
    "property": 100
});
var longInput = function(x) {
    return x.a + x.b + x.c;
};
var getName = function(x) {
    return x.firstName + " " +
       x.lastName;
};
getName({
    "firstName": "Brian",
    "lastName": "McKenna"
});
```

Tagged Unions

Roy supports tagged unions, in which values are tagged with a constructor name. A type can contain many alternative tagged values. To define one, we use the `data` keyword:

```
roy> data ConsList a = Cons a
  (ConsList a) | Nil
```

This generates a `ConsList a` type with values created from two possible tags. From this definition, we can construct a list:

```
roy> let empty = Nil
roy> let listOfOne = Cons 10
  empty
roy> let listOfTwo = Cons 21
  listOfOne
roy> listOfTwo
{"_0":21,"_1":{"_0":10,"_1":
  {}}} : ConsList Number
```

The type parameter `a` is universally quantified. We can't cons a different type:

```
roy> Cons "Hello!" listOfTwo
Error: Type error: String is
  not Number
```

This code converts into JavaScript that uses constructors to create objects:

```
var Cons = function(a_0,
  ConsList_1){
    this._0 = a_0;
    this._1 = ConsList_1;
};
var Nil = function(){};
var empty = new Nil();
var listOfOne = new Cons
  (10, empty);
var listOfTwo = new Cons
  (21, listOfOne);
listOfTwo;
```

Now that we have the data structure, we can write some functions using pattern matching. The code in Figure 1 uses the `match` keyword to detect the value's tag. In `listIsEmpty`, we detect whether the tag is `Cons`. If so, the answer will be false. If the tag is `Nil`, the answer is true.

In `listMap`, we take out the values of each `Cons`. We run the given function on the element and use recursion on the rest of the list. The `listFilter` function is similar but uses the given function as a predicate for reconstructing the element or skipping it.

These combinators use JavaScript's `instanceof` checks for tags (see Figure 2).

Monad Syntax

Many interesting data types and control flows are monads. A monad is something that can satisfy an interface of two functions:

```
bind: Function(Monad #a,
  Function(#a, Monad #b),
  Monad #b)
return: Function(#a, Monad #a)
```

Monads must also satisfy certain mathematical laws to truly be considered monads. Roy doesn't check these rules, so we'll just ignore them for now.

One example of monadic control flow prevalent in the JavaScript world is *continuation passing*. I've read

many JavaScript blog posts in which people reinvent monads for their node.js programs.

It'd be awesome if we had a language that could make asynchronous, continuation passing programs look less like *callback hell*. For those unfamiliar with node.js, callback hell is when code starts to look like that in Figure 3. Notice that the code is getting deeper, traveling toward the right. Some advice is to give names to these functions, but I haven't found this to be greatly effective.

A lot of the same error-handling code is mixed throughout the business logic. It would be good if we had a method for automatically including error-processing rules.

The following code is based on jQuery and won't run in the command-line REPL. You can instead give it a try on the Roy website mentioned previously.

```
let deferred = {
  return: \x -> $.when x
  bind: \x f ->
    let dfd = $.Deferred ()
    x.done (\val ->
      (f val).done (\val2 ->
        dfd.resolve val2)
    )
    dfd.promise ()
}
```

This defines a structure with `return` and `bind` functions. This is how Roy currently represents its monad implementations. The dollar sign (`$`) is defined in jQuery, and we use jQuery's Deferred Object API to create promises.

Now that we have this monad, we can create pipelines:

```
let requests = do deferred
  text1 <- $.ajax "documents/
     hello.txt"
  text2 <- $.ajax "documents/
     world.txt"
  return text1 ++ text2
```

```
var listIsEmpty = function(lst) {
    return (function() {
        if(lst instanceof Cons) {
            return false;
        } else if(lst instanceof Nil) {
            return true;
        }
    })();
};
var listMap = function(f, lst) {
    return (function() {
        if(lst instanceof Cons) {
            var x = lst._0;
            var r = lst._1;
            return new Cons((f(x)), (listMap(f, r)));
        } else if(lst instanceof Nil) {
            return new Nil();
        }
    })();
};
var listFilter = function(p, lst) {
    return (function() {
        if(lst instanceof Cons) {
            var x = lst._0;
            var r = lst._1;
            return (function() {
                if((p(x))) {
                    return new Cons(x, (listFilter(p, r)));
                } else {
                    return listFilter(p, r);
                }
            })();
        } else if(lst instanceof Nil) {
            return new Nil();
        }
    })();
};
```

Figure 2. The compiled higher-order list functions. We can see the lightweight JavaScript output.

```
router.get("/", function(request) {
    Users.get(request.params.name, function(error, user) {
        if(error) return fail(error);
        Posts.find(user.id, function(error, posts) {
            if(error) return fail(error);
            Friends.find(user.id, function(error, friends) {
                if(error) return fail(error);
                render(posts, friends);
            });
        });
    });
});
```

Figure 3. Made-up node.js example. This illustrates "callback hell."

```
var deferred = {
    "return": function(x) {
        return $.when(x);
    },
    "bind": function(x, f) {
        var dfd = $.Deferred();
        x.done((function(val1) {
            return f(val1).done((function(val2) {
                return dfd.resolve(val2);
            }));
        }));
        return dfd.promise();
    }
};
var requests = (function(){
    var __monad__ = deferred;
    return __monad__.bind($.ajax("documents/hello.txt"),
       function(text1) {
        return __monad__.bind($.ajax("documents/world.
          txt"), function(text2) {
            return __monad__.return(text1 + text2);
        });
    });
})();
requests.done((function(result) {
    return console.log(result);
}));
```

Figure 4. Asynchronous, monadic computation compiled to JavaScript callbacks.

This code looks like it would be synchronous in an imperative language. What we've actually made is a sequence of two asynchronous requests. When one finishes, the next will be fired.

With the previous code, we've built up a big promise and can get out the end result by adding a final "done" callback:

```
requests.done (\result ->
  console.log result
)
```

All the previous code will compile into JavaScript that has nested, asynchronous callbacks (see Figure 4).

Modules

Recently, I've been working on modules. The module support is designed to unify the many module standards the JavaScript community has created, including CommonJS Modules/1.0, Asynchronous Module Definitions (AMD), and browser-based globals.

Whenever you compile a module, you generate a `.roym` file. This module descriptor contains all of the type information for the compiled code. For example, the Roy code

```
let obj = {x: 1, y: 2,
  t: "test"}
export obj
```

generates the following module descriptor:

```
obj: {x: Number, y: Number,
  t: String}
```

We can then import the module:

```
import "./module"
module.obj
```

The import built-in loads the external module descriptor without the code and can then type-check the rest of the program.

In node.js mode, the program will compile to

```
var module = require
  ("./module");
module.obj;
```

In browser mode, the program will compile to:

```
(function() {
module.obj;
})();
```

Future

One plan I have for Roy is to implement lenses at the language level. Lenses let you compose immutable getters and setters:

```
.property: Lense {property:
  #a) #a
get: Function(Lense #a #b,
  #a, #b)
```

Languages such as Haskell and Scala don't currently have core support for lenses, so developers must write some amount of boilerplate to use them. Lenses will help developers write immutable programs, but it might be necessary to write mutable code for performance reasons. Roy itself doesn't have any form of mutation — though you can easily call out to JavaScript code. That's a problem, because code with mutation is arguably the hardest type of code to reason about and correctly write. This is an area in which types would be even more important.

To combat performance concerns, Roy will eventually have a reference type allowing mutation:

```
let x: Ref Number = newRef 100
console.log (1 + get x)
```

The JavaScript output will be very lightweight:

```
var x = 100;
console.log(1 + x);
```

Another feature I'd like is the ability to generate type-safe bindings to Web browser specifications. The W3C has worked on providing new specifications with definitions written in the Web Interface Description Language (Web IDL; www.w3.org/TR/WebIDL/). The language looks like the following:

```
// Introduced in DOM Level 2:
[Callback]
interface EventListener
{
  void handleEvent(in Event evt);
};
```

We could eventually convert this into something like this:

```
type EventListener
  = {handleEvent:
  Function(Event, Unit)}
```

Roy could then automatically support static typing of new W3C specifications.

I'd like to get Roy to a stage where I can rewrite Roy in Roy. I've found many bugs that Roy's type system would have easily caught while I've been working on it. It's also particularly important for a compiler to be correct.

I had the chance to talk to Simon Peyton-Jones of Microsoft Research — a key contributor to the design of the Haskell programming language and lead designer of the Glasgow Haskell Compiler (GHC) — in December 2011. One of his ideas was to make a type system that could generate runtime errors for typed code, if asked to. The idea is that sometimes a type error doesn't mean that your whole program is incorrect (for example, code might not be reachable). The compiler shouldn't prevent you from running it.

If I implemented this in Roy, we'd be able to write a Roy program like this:

```
console.log ("Hello!" + 2)
```

We'd get a similar warning on the command-line

```
WARNING: Line 1: Type error:
  String is not Number
```

And the JavaScript would have a similar runtime exception:

```
console.log(function() {
    throw new Error("Line 1:
       Type error: String is
       not Number");
}());
```

Roy's future looks bright. The contributors will be working hard during 2012 to try and get it ready for production systems. You can follow Roy's progress on Twitter at http://twitter.com/roylangjs, and can find Roy repositories at https://bitbucket.org/puffnfresh/roy or https://github.com/pufuwozu/roy. Hopefully you'll join in and help push the statically typed and functional Web forward! ◻

Brian McKenna is a Java Developer with Atlassian. Contact him at brian@brianmckenna.org or via his blog at http://brianmckenna.org.