JSJS - Project Proposal

A general purpose, strongly typed programming language for the web

Jain Bahul	bkj2111
Srivastav Prakhar	ps2894
Jain Ayush	aj2672
Sadekar Gaurang	gss2147

Description

JSJS is a strongly typed language for the web. Taking inspiration from languages such Scala, TypeScript and OCaml, the goal of JSJS is to enable programmers to write type-safe, correct and robust code that compiles down to Javascript. Our target users are programmers who like functional nature of Javascript but want the safety and guarantees that come with a statically typed language without sacrificing its ubiquity.

Motivation

> *Any application that can be written in JavaScript, will eventually be written in JavaScript* - **Jeff Atwood**

Javascript is the *lingua franca* of the web. With its humble beginnings in engine of the browser, Javascript has slowly and steadily has gained wide adoption across the developer community at large. Famed to have gone from conception to a working implementation in just 10 days, Javascript now runs on everything - from massive servers to even the smallest Raspberry Pi. However, Javascript's massive success has often been met with wide bewilderment (even disdain in some cases) from developers.

The members of this group, however, believe that Javascript is an incredibly powerful language. Having spent most of our time programming for the web - both frontend and backend, building apps ranging from trivial browser plugins to standalone cross-platform native ones, we marvel that at the fact that a language could have such a widespread use and applicability. The good news that we are not alone in this assessment. The biggest tech giants of today are spending numerous resources in making the language even better. This includes but is not limited to the monumental advances done by the Google folks on V8, the recent Chakra engine by Microsoft and the numerous tools open-sourced by Facebook.

As a language Javascript is quite interesting. With features such as closures, functions as first class objects, asynchronous programming via callbacks and a prototype based system, Javascript can definitely be touted as a modern language. On the other hand, JS is infamous for weird object rules, global name definitions and hard-to-grok prototype chain. The fact that one of the most popular books on JS is titled "Javascript: The Good Parts" indicates the warts in the language.

Our goal with **JSJS** is to create a functional language that has features such as type safety and immutability but at the same time can be used where Javascript runs. Despite Javascript treating functions as first class objects, Javascript is typically used as an imperative programming language. Its lack of structure has encouraged programmers to think of Javascript objects in terms of state, instead of attempting to transform data. Javascript is a product of rapid evolution, and thus for many people from the functional programming school of thought, it seems broken. Although its core library is small, Javascript's weak typing and object construction system create a broken mix, where functions are first class, yet objects are used imperatively. These are some of the issues that we plan to address with JSJS.

In conclusion, we believe that JSJS's list of features and Javascript's ubiquity is a deadly combination that make it compelling tool for programmers.

Features

- **Strong Type Safety** JavaScript is a dynamically typed language, with no real notion of type safety. A variable in JavaScript can be assigned to data of any type and this assignment can be changed through the course of the program. JSJS is be a strongly typed language with static type checking, like OCaml. Types will not be inferred, so the types must be explicitly stated while declaring values and defining functions.
  A **strong type system** is described as one in which there is no possibility of an unchecked runtime type error and does not permit any implicit type conversion for operations at runtime.
  Since JSJS compiles down to JavaScript, which is a dynamically typed language, types in JSJS merely act as annotations for type safety and will be erased during generation of the JavaScript code.

- **Immutable Values** One of the hallmarks of functional programming is immutability of all values used in a program. Javascript is an imperative language, where state can be changed and variables can be reassigned with new values. JSJS will have no concept of variables and all assignments will be treated as values, and hence cannot be changed or reassigned during the running of the program. We have chosen the keyword `val` for assigning values to identifiers to express the concept of all values being immutable more emphatically.

- **Everything is an Expression** Every statement in JSJS is an expression that evaluates to a result of some type, exactly like in OCaml. We will also provide a Unit type, to allow for side-effects such as printing and logging which don't evaluate to an explicit return type.

- **Immutable Data Structures** An immutable or persistent data structure is one in which previous versions of the data structure are preserved when it is modified. Immutable data structures make programs more robust and easy to reason about. Data structures in JavaScript are completely mutable.
  JSJS will provide immutable implementations for Lists and Maps in the language, with a standard library for functions like `List.hd` and `List.tl`, and `Map.get` and `Map.set`.
- **Polymorphic Types** Support for polymorphic types is another fundamental feature of functional programming. This allows functions to take arguments of generic types and perform some operation on them, without the function being required to explicitly know the underlying types of the argument. For eg., we can have a function that takes as an argument a list of any type, and performs some operation on it. This function can work with both lists of numbers and lists of strings. JSJS supports polymorphism for the built-in data types.
- **Functions as First-Class Objects** Functions in JSJS are first-class objects. Functions can be passed as arguments to other functions, returned by other functions, and can be assigned to variables. JSJS also supports anonymous functions. JavaScript has support for functions as first-class objects, so JSJS provides some syntactic sugar to make their usage sweeter.

Lexical Conventions

Name	Function
Keywords	`if`, `then`, `else`, `num`, `string`, `bool`, `val`, `def`, `list`, `true`, `false`
Comments	`// this is a comment`
Multi-line Blocks	`{ ... ; }`
Conditionals	`if (x == 10) then { 10 * 10; } else { 100; }`

Primitive Data Types

Data Types	Example	Declaration
num	42, -42, 4.2, -0.00042	`val a : num = 42;`
bool	true, false	`val isAwesome? : bool = true;`
string	“jsjs”, “is”, “the”, “best”, “!!!”	`val name : string = "jsjs";`
unit	`unit`	`val _ : unit = println("Hello");`

Type Declaration

Values	Type Declaration	Description
Any	`a: T`, `a : U`	where T and U can be any of the types given below
Number	`a: num`	where num represents the type for Numbers

Values	Type Declaration	Description
Boolean	`a: bool`	where bool represents the type for Logical values like true and false
String	`a: string`	where string represents the type for string literals
Unit	`a: unit`	used for dealing with side-effects
List	`a: list T`	where T is type of elements of the list
Map	`a: <T : U>`	where T is type of the key and U is type of the value
Function	`a: (T1, T2, ..) -> U`	where T1, T2,... are the types of the arguments taken by the function and U is the return type of the function

Operators

Operator Name	Syntax	Applicable Data Types
Assignment	`a = b`	num, bool, string
Integer Arithmetic	`a + b, a - b, a * b, a / b`	num
Modulo	`a % b`	num
Equal to	`a == b`	num, bool, string
Not Equal to	`a != b`	num, bool, string
Comparison	`a <= b, a < b, a >= b, a > b`	num, bool, string
Logical AND	`a && b`	bool
Logical OR	`a || b`	bool
Logical NOT	`!a`	bool
Concat	`a ^ b`	string

Composite Data Types

List

List Type Declaration

- The **list** keyword is used to declare the type of a list
- Type declaration for a list: `a : list T`

List Declaration

```
// List of strings
val names : list string = ["foo", "bar", "baz"];

// List of nums
val nums : list num = [1, 2, 3, 4];
```

```
// List of lists of string
val friends_list : list list string = [["foo"], ["bar"], ["baz"]];
```

List Functions

```
// Returns the first element of the list
val head : num = List.hd([1, 2, 3, 4, 5, 6])

// Returns the list without the head of the list
val tail : list num = List.tl([1, 2, 3, 4, 5, 6])

// Returns the length of the list
val length : num = List.length([1, 2, 3, 4, 5, 6])
```

Map

Map Type Declaration

- The <> construct is used to declare a map
- Map Type Declaration: `a : <T, U>` where T is type of the key and U is the type of the value.

Map Declaration

```
// creates a map with key of string type and value of num type.
val passwords : <string: num> = {
 "foo" : 1245,
 "bar" : 5567,
 "baz" : 5533
};

// creates a map of key string type and value being another map
// that has key of string type and value of num type.
val students : <string: <string: num>> = {
  "foo": { "marks": 97 },
  "bar": { "marks": 23 },
  "baz": { "marks": 47 },
};
```

Map Functions

```
// Returns value associated with key = "foo" in map named passwords
val foo = Map.get(passwords, "foo");

// Sets the value of field "foo" as 12345 in map named passwords
val foo = Map.set(passwords, "foo", 12345);
```

Functions

Function Type Declaration

- JSJS has a unique way of declaring the type or signature of a function.
- This is particularly needed when declaring anonymous functions.
- Function type is declared as: `f : (T, T, ...) -> U` where T are types of the arguments and U is the return type of the function

Function Expressions

```
// defines a function square that takes
// a `num` argument and returns a `num`
def square (x: num) : num = x * x;
```

Multi-line functions

```
// multi-line statements (i.e blocks) are separated by
// a semicolon. The last line in a block is returned automatically.
// Hence, there is no explicit `return` keyword
def cube (x: num): num = {
  val sq = square(x);
  x * sq;
}
```

Functions as values

```
// Assigns the name `sq` to an anonymous function.
// Functionally equivalent to def sq (x: num) : num = x * x;
val sq : (num) -> num = (x: num) : num => x * x;
```

Anonymous Functions

```
// passing an anonymous function using the arrow syntax
val squares : list num = map((x: num) : num => x * x, [1, 2, 3, 4]);
```

Polymorphic Functions

```
// defining a polymorphic function that takes two arguments
// The first argument is a function, the second is a list.
// Finally, the return type of this function is another list
// of (possibly) a different type than the input list.
def map (f: (T) -> U, l: list T): list U {
   if (List.empty?(l))
   then { []; }
   else {
     val res = f(List.hd(l));
     List.cons(res, map(f, List.tl(l)));
   }
}
```

Example Programs

99 bottles of beer

```
def ninetyNineBottles (): unit = {
   def toStr(x:num) : string = {
      if (num == 0)
      then { "no more"; }
      else { String.toString(num); }
   }

   def ninetyNine (x:num) : unit = {
      if(x==0)
      then {
         println("No more bottles of beer on the wall, no more bottles of beer.");
         println("Go to the store and buy some more, 99 bottles of beer on the wall.");
      }
      else {
         println(toStr(num) ^ "bottle of beer on the wall, " ^
                 toStr(num) ^  " bottle of beer.");

         if(x==2)
         then {
            println("Take one down and pass it around, " ^
                    toStr(num - 1) ^ " bottle of beer on the wall.");
            nintetyNine(x-1);
         }
         else{
            println("Take one down and pass it around, " ^
                    toStr(num - 1) ^ " bottles of beer on the wall.");
            nintetyNine(x-1);
         }
      }
   }
   ninetyNine(99);
}
```

Euclid's algorithm for GCD

```
def gcd (a : num, b : num) : bool = {
   if (a == b)
   then { a; }
   else {
      if (a > b)
      then { gcd((a - b), b); }
      else { gcd((b - a), a); }
   }
}
```

Merge Sort

```
def merge (a : list num, b : list num) : list num = {
   if(List.hd(a) < List.hd(b))
   then { List.cons(List.hd(a), merge(List.tl(a), b); }
   else { List.cons(List.hd(b), merge(a, List.tl(b)); }
}

def split (a : list num, start : num, end : num) : list num = {
   if (start != 0)
   then { split(List.tl(num), start - 1, num); }
   else {
      if (end == -1)
      then { a; }
      else { List.cons(List.hd(a), split(List.tl(a), start, end - 1)); }
   }
}

def mergeSort (a : list num) : list num = {
   val size = List.length(a);
   if (size == 0 || size == 1)
   then { a; }
   else { merge(mergeSort(split(a, 0, size/2)),
                mergeSort(split(a, size/2 + 1, size - 1))); }
}
```