Just another day in the Javascript world

Christian Rebischke
@Shibumi

The reason why javascript sucks..

```javascript
> '5' - 3
2  // weak typing + implicit conversions * headaches
> '5' + 3
's3'  // Because we all love consistency
> '5' - '4'
1  // string - string * integer. What?
> '5' + + '5'
'55'
> 'foo' + + 'foo'
'fooNaN' // Marvelous.
> '5' + - '2'
'5-2'
> '5' + - - - - - - - - -  // Apparently it's ok
's2'
> var x = 3;
> '5' + x - x
50
> '5' - x + x
5  // Because fuck math
```
## Why JSJS?

*How does JSJS compare to JS?*

<table>
<thead>
<tr>
<th>Feature</th>
<th>JSJS</th>
<th>JS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Safety</td>
<td>📘</td>
<td>🍂</td>
</tr>
<tr>
<td>Type Inference</td>
<td>📘</td>
<td>🍂</td>
</tr>
<tr>
<td>Immutable values</td>
<td>📘</td>
<td>🍂</td>
</tr>
<tr>
<td>Immutable Collections</td>
<td>📘</td>
<td>🍂</td>
</tr>
<tr>
<td>Programming Style</td>
<td>Functional</td>
<td>Imperative</td>
</tr>
<tr>
<td>Works with Node</td>
<td>📘</td>
<td>🍂</td>
</tr>
<tr>
<td>Works in the browser</td>
<td>📘</td>
<td>🍂</td>
</tr>
</tbody>
</table>
A Sneak Peak - Syntax

val sort = /\(xs: list T): list T => {
  if xs == [] then []
  else {
    val smaller = List.filter(/\(x) => x < hd(xs), tl(xs));
    val greater = List.filter(/\(x) => x >= hd(xs), tl(xs));
    List.concat(sort(smaller), hd(xs) :: sort(greater));
  }
};

// let's test it
val sorted = sort([10, 5, 0, 3, 8]);
val sorted_strs = sort(["c", "z", "a", "e", "y"]);

// printing...
List.iter(print, sorted);
List.iter(print, sorted_strs);
val sort = \(xs: \text{list } T): \text{list } T \Rightarrow \{
  \text{if } xs == [] \text{ then } [] \\
  \text{else } \{
    \text{val smaller} = \text{List.filter}(\(x) \Rightarrow x < \text{hd}(xs), \text{tl}(xs)); \\
    \text{val greater} = \text{List.filter}(\(x) \Rightarrow x >= \text{hd}(xs), \text{tl}(xs)); \\
    \text{List.concat(sort(smaller), hd(xs) :: sort(greater))};
  \};
\};

// let's test it
val sorted = sort([10, 5, 0, 3, 8]);
val sorted_strs = sort(["c", "z", "a", "e", "y"]);

// printing...
List.iter(print, sorted);
List.iter(print, sorted_strs);
val map = λ(fn: (T) → U, xs: list T): list U => {
  if empty?(xs) then []
  else fn(hd(xs)) :: map(fn, tl(xs));
};

val sq = λ(x) => x * x;

print(map(sq, [1,2,3,4,5,6]));
Closures

Local value for a function - kept alive after function has returned

```scala
val sayHelloTo = /\(name\) => {
  val text = "Hello " ^ name;
  /\() => print(text);
};

val sayTo = sayHelloTo("Bob");
sayTo(); // prints "Hello Bob"
```
Error Handling

Error Reporting

```scala
val x = 1 && 2;
```
Type error: expected value of type 'num', got a value of type 'bool' instead

Exceptions and Exception Handling

```scala
try {
  throw "You have failed this program...";
}
catch (msg) {
  print("Maybe not \o/");
};
```
Maybe not \o/
Immutability

- Immutability makes it easier to reason about code
- The `val` keyword defines a value. No `variables` in JSJS.
- Names cannot be redefined in the current scope

```javascript
let locals, globals = env in
if NameMap.mem id locals then raise (AlreadyDefined(id))
else if KeywordsSet.mem id js_keywords_set then raise (CannotRedefineKeyword(id))
else
```
Immutable Collections - Lists & Maps

- Uses Facebook’s Immutable.js library to enforce immutability.
- All library functions written is JSJS.
Immutable Collections - Lists

```scala
val xs = [1, 2, 3, 4, 5];

// using cons operator
val _ = "foo" :: ["bar", "baz"];

// heterogenous collections not allowed
val _ = [true, "foo", 10];
```

All functions return new lists (do not modify the list in place)

List Collection Library

hd, tl, empty?, filter, map, fold_left, rev, iter, range, concat, insert, remove, sort, nth, length
Immutable Collections - Map

```scala
val moneyOwedByFriends = {
    "ben": 10,
    "mary": 20,
    "mark": 43,
    "alice": 54
};

// map annotations
val getTotal = /(m: <string: num>): num => {
    val values = Map.values(m);
    List.fold_left(/\(x, y) => x + y, 0, values);
};

getTotal(moneyOwedByFriends);
```

All functions return a new map (do not modify the map in place)

Map Collection Library

get, set, has?, length, values, keys, count, merge, del
JSJS Type System

T( _) is a generic type that helps with type Inference or generic user annotations.

TAny is only used for type inference.

TNum, TBool, TUnit, TString are data types.

TFun is the function type comprising of args (list of primitives) and a return type.

TList and TMap are composite List and Map types.
Type Inference - Our ‘Inspiration’

Guillermo Rauch
@rauchg

JSJS is a very cool proposal for type-safe JS. Digging the syntax a lot.
cs.columbia.edu/~%20sedwards/c ...

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

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

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

WHAT YEAR IS IT?
Type Inference - JSJS vs OCaml

JSJS

gaurang@dyn-160-39-206-186 ~/C/s/p/JSJS> ./jsjs.out -s
\( f \) \( \Rightarrow \) \( f(42) \);
\( f : (\text{num}) \rightarrow A \) : A = \{ \{ f : (\text{num}) \rightarrow A \}(42.0 : \text{num}) : A \} \}

OCaml

utop # fun f -> f 42;;
- : (int -> 'a) -> 'a = <fun>
Type Inference - JSJS vs OCaml

**JSJS**

\( \forall (f) \Rightarrow f(f(42)) \);
\( \forall (f : (\text{num}) \rightarrow \text{num}) : \text{num} = \{ (f : (\text{num}) \rightarrow \text{num})(f : (\text{num}) \rightarrow \text{num})(42 : \text{num}) : \text{num} \rightarrow \text{num} \} \)

**OCaml**

```
utorp # fun f -> f (f 42);;
- : (int -> int) -> int = <fun>
```
Type Inference with user annotations

gaurang@dyn-160-39-206-186 ~/C/s/p/JSJS> ./jsjs.out -s
∧(x, y) ⇒ x < y;
∧ (x : A, y : A) : bool = { { ((x: A) < (y: A): bool) } }
Type Errors - JSJS vs OCaml

JSJS

\((x, y) => \{ \text{print_string}(y); x + y; \}\);
Type error: expected value of type 'string', got a value of type 'num' instead

OCaml

utop # fun x y -> print_endline y; x + y;;
Error: This expression has type bytes but an expression was expected of type int
Type Errors - JSJS vs OCaml

JSJS

\( A(z) \Rightarrow z(z); \)
Type error: expected value of type 'A', got a value of type '(A) -> B' instead

OCaml

```
$ utop # fun z -> z z;;
Error: This expression has type 'a -> 'b
   but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```

Demo

[Image of a webpage with examples and code]

**EXAMPLES**

Click on one of the examples to load and freely edit the code for it in the editor.

- GREATEST COMMON DIVISOR
- PRIME SIEVE
- 99 BOTTLES OF BEER
- DATA STRUCTURES: MAPS
- QUICKSORT

**TRY IT OUT!**

```javascript
// let's define a function
val sq = \(x) => x * x;

// values and functions both
// can be optionally annotated
val sum: num = List.fold_left(
  \(x, y) => x + y, 0,
  // lambda expressions
  List.map(square, List.range(1, 10))
);

// show result
print(sum);
```
What next?

➔ Tuples
➔ Option Type
➔ Pattern Matching
➔ Javascript FFI
➔ Line Number Error Reporting
Thank You

So long, and thanks for all the fish.