Type Inference and ReasonML

- What is Type Inference?
- Type inference programming language examples
- What is ReasonML?
- A Taste of ReasonML

Static, Dynamic, Gradual Typing

Typing Strategy	Description	Languages
Static	detect type constraints violations before runtime	C, C++, Java, GO, Haskell, Reasonml
Dynamic	detect type errors during runtime	Javascript, Python, Ruby
Gradual	optional type constraints that will caught before runtime	Typescript, Python MyPy, Php hack

Gradual Typing and Soft Typing

Mixing type annotations with any is called Gradual Typing, defaulting/prefering type any is called Soft Typing

Strong vs Weak Typing

I have seen many people confuse Static Typing with Strong Typing, but they are independant C is statically typed but has weak typing, Python is Dynamically typed but is strongly type.

C Example of Weak Typing

```c
#include <stdio.h>

int main() {
    /* treat int as char */
    char c = 8 + '0';
    /* treat char as int */
    printf("%d\n", c);
    /* an array, string and pointer */
    char label[] = "foo";
    /* should be a type error */
    printf("%s\n", label + 1);
    return 0;
}
```

Javascript Example of Weak Typing

```
var cost = 2;
var total = cost + 'USD';
var weird1 = [] + [] === ''; // true
var w2 = [] + {} === '[object Object]'; // true
var weird3 = {} + [] === 0; // true
var weird4 = isNaN({} + {}); // true
```

Type Inference or Type Reconstruction

Type inference is automatic deduction of types for unannoted code for the purposes of type checking. Applicalbe to static and gradual typing.

Type Inference Scopes

Typing Inference Scope	Description	Languages
Local/Unidirectional	deduce variable declaration, and return type	C++(11), Typescript, Java(10), Go
Global/Non Local/Bidirectional	as above, plus argument types and recursive functions	SML, F#, Ocaml, ReasonML, Haskell

Typescript Local Inference

```
// simple inference
let x = 5;

// local type inference of
// function return type
let foo = () => {
    return 42;
}

// infer type from destructuring
let [c, d, ...rest] = [1, 2, 3];
```

Typescript Local Inference Limitations

```
// local type inference
// cannot infer parameters
let bar = (x: number, y: number) => {
    return x + y;
}

// local type inference cannot
// infer return type of recursive
// functions
function fac(n: number): number {
    return n >= 1 ? 1 : n * fac(n - 1);
}
```

Haskell Mostly Unannoted

```haskell
module Main where

import System.IO (isEOF)
import Text.Printf

squaredDist mean num = (num - mean)**2

stddev len mean nums =
    let sq = squaredDist mean in
        sqrt(sum(map sq nums) / (fromIntegral len))

nonOutlier mean sd num =
    (num < mean + sd) && (num > mean - sd)

readFloats nums = do
    done <- isEOF
    if done
        then return nums
    else do
        num <- readLn :: IO Float
        readFloats (num:nums)

main = do
    nums <- readFloats([])
    let len = length nums
    let total = sum nums
    let avg = total / (fromIntegral len)
    let sd = stddev len avg nums
    let fno = nonOutlier avg sd
    let no = filter fno nums
    putStrLn . unlines $ printf "%.2f" <$> no
```

ReasonML Mostly Unannoted

```
module Tc = Tablecloth;

let readFloats: unit => array(float) = [%bs.raw {|
    () => {
        let {readFileSync} = require('fs');
        let lines = readFileSync(0).toString().split('\n');
        return lines.map(line => parseFloat(line))
                    .filter(n => !isNaN(n));
    }
|}];

let stats = (nums) => {
    let sum = Tc.Array.floatSum(nums);
    let len = Array.length(nums);
    let avg = sum /. float_of_int(len);
    let ss = Array.map((e) => (e -. avg)**2.0, nums);
    (avg, sqrt(Tc.Array.floatSum(ss) /. float_of_int(len)));
};

let main = () => {
    let nums = readFloats();
    let (mean, stddev) = stats(nums);
    let nonOutlier = (num) => {
        num < mean +. stddev && num > mean -. stddev;
    }
    let nonOutliers = Tc.Array.filter(~f=nonOutlier, nums);
    Array.map(Printf.printf("%f\n"), nonOutliers);
};

main();
```

Structural, Nominal and Duck Typing

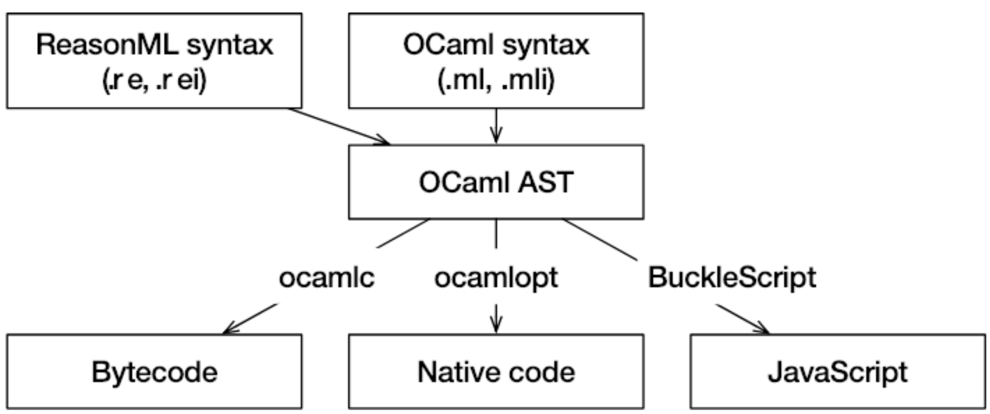
Typing Strategy	Description	Languages
Nominal	Typing based on name not data shape, better error messages	Java
Structural	Typing based on shape of data, more powerful and flexible, bad error messages	C++ Templates, GO Interfaces, Typescript, Haskell, ReasonML
Duck	Flexible like Structural typing but no compile time safety doesn't suffer from bad error messages	Python, Javascript

Typescript Structural Typing

```
interface Named {
    firstName: string
}

class Person {
    firstName: string
    constructor(firstName: string) {
        this.firstName = firstName;
    }
}

// structural type shapes match
let p1: Named = new Person("Troy");
let p2: Named = { firstName: "Travis" };

// infer type from destructuring
let {firstName} = p1;
```

What is Reason/ReasonML?

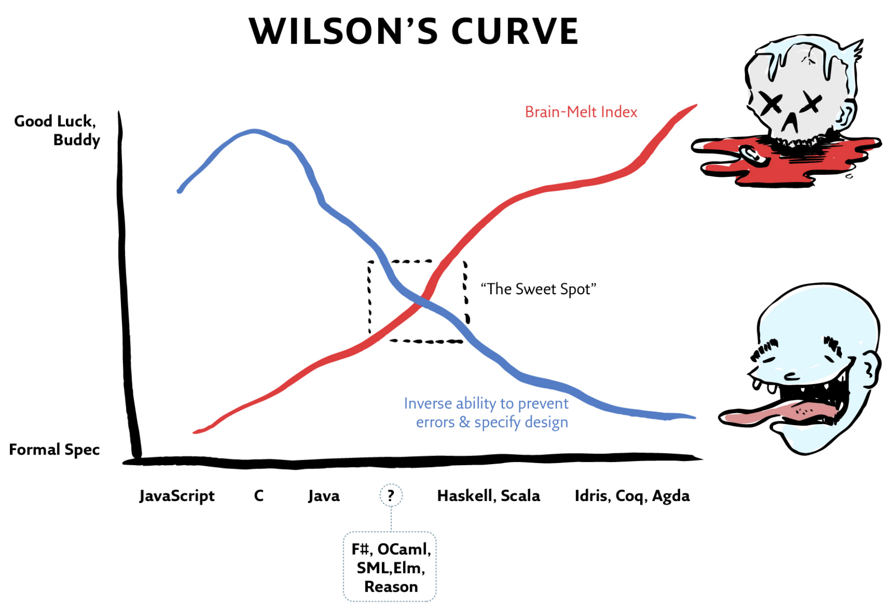
- Tries to make Ocaml more like Javascript/Typescript also provides interop with javascript echo system
- Similar to Elixir which tries to make Erlang more like Ruby



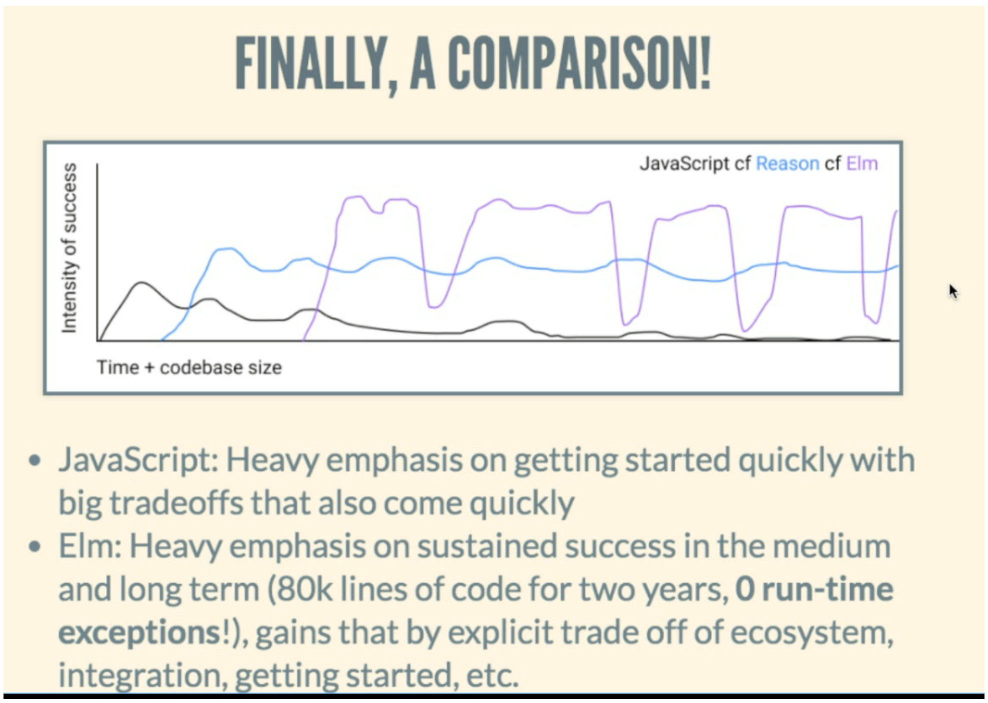
Reasonml History

- Started at facebook 2016 by Jordan Walke
- Jordan Walke also created React
- Original React prototype was done in SML a close relative of OCaml
- Reasonml has very close ties to react supporting inline JSX
- [Reason] is the best way to take React to the next level --Jordan Walke

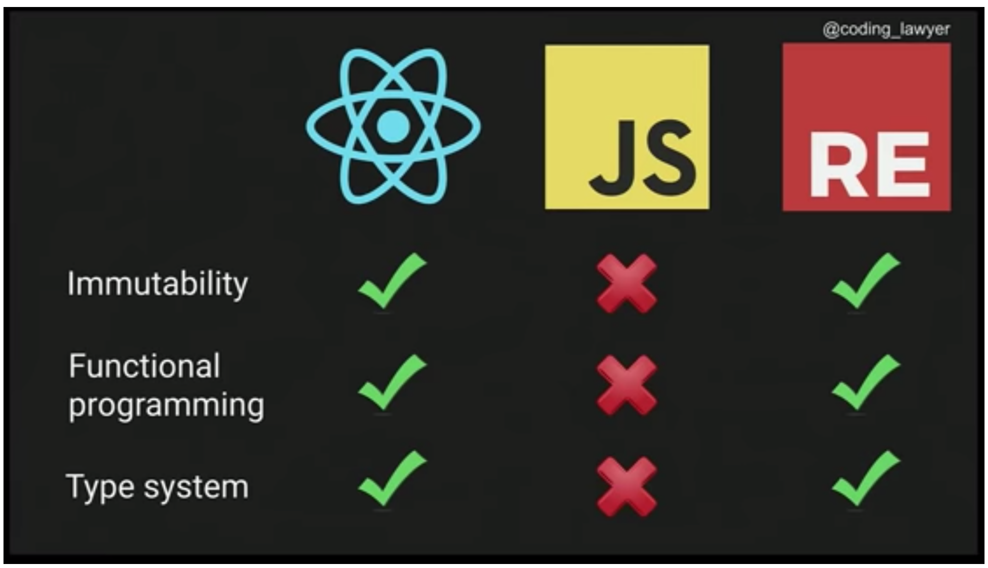
Reasons for Reason



More Reasons for Reason



Reasons for Reason



A taste of Reason

```
let listSrc = {js|/* lists are generic types also key in fp *
let favNums : list(int) = [31, 37, 42];
/* destructure we use underscore to say we don't care */
let [first, ..._] = favNums;
switch(favNums) {
    | [] => print_string("empty\n")
    | [x] => print_string("one\n")
    | [head, ...rest] => print_string("many\n")
};
/* modules List, ListLabels, Belt.List Tablecloth.List */
```

Reason powerful Variant Sum types

```
type option('a) = Some('a) | None;
type result('a, 'b) = Ok('a) | Error('b);

let ndiv = (x, y) => {
    switch(y) {
    | 0 => None
    | _ => Some(x/y)
    }
};

let ediv = (x, y) => {
    switch(y) {
    | 0 => Error("Can not divide by zero")
    | _ => Ok(x/y)
    }
};

let e = ediv(1, 0);

switch(e) {
| Error(err) => print_endline(err)
| Ok(quotient) => {
        print_int(quotient);
        print_newline();
    }
}
```

Compose Results Variants

```
let map = (r, f) => {
    switch(r) {
    | Error(err) => Error(err)
    | Ok(a) => Ok(f(a))
    }
};

let flatMap = (r, f) => {
    switch(r) {
    | Error(err) => Error(err)
    | Ok(a) => f(a)
    }
};

let lift1 = (f) => {
    (a) => {
        switch(a) {
        | Error(err) => Error(err)
        | Ok(b) => Ok(f(b))
        }
    }
};

let r1 = map(Ok(41), a => a + 1);
let r2 = flatMap(Ok(41), a => Ok(string_of_int(a)));
let f1: result(int, string) => result(int, string) = lift1(a
let r3 = f1(Ok(1));
```

Reason Arrays

```
/* arrays are also generic */
/* fast random access but fixed size */
let brothers : array(string) =
    [| "Troy", "Chris", "Travis" |];
/* destructure exact size */
let [| b1, b2, b3 |] = brothers;
/* access mutate cell */
a[2] = ""; /* replace Travis with empty string */
/* modules Array, ArrayLabels, Belt.Array Js.Array
Tablecloth.Array */
```

Reason records and structural typing

```
/* must create record types to use them*/
type person = {
    name: string,
    age: int
};
let olivia = { name: "Olivia", age: 1 };
/* destructure */
let {name, age} = olivia;
let { name: n, age: a } = olivia; /* n = "Olivia", a = 1 */
/* punning */
let oli = { name, age }; /* - : person = {name: "Olivia", age
```

Reason Tuples

```
/* keyword type lets you create types or type aliases */
type intPair = (int, int);
let favPrimes : intPair = (31, 37);
/* destructure */
let (first, second) = favPrimes;
/* Tablecloth.Tuple2, Tablecloth.Tuple3 */
```

Reason Opt in Mutability

```
// mutable values
let count = ref(0);
count := 1; // mutate cell update
// use suffix ^ operator to get value of ref
print_string(string_of_int(count^) ++ "\n");
// mutable records
type person = {
    name: string,
    mutable age: int
};
let troy = {name: "Troy", "age": 45};
let isBirthday = true;
troy.age = isBirthday ? troy.age + 1 : troy.age;
```

Summary

- Reasonml occupies a sweet spot in terms of language features
- Global Type Inference eliminates lot of verbosity
- React development is better in Reasonml
- Transpilation speed Reasonml >> Typescript >> Babel(Flow)
- Reasonml will scale to larger code bases
- Typescript >> Reasonml in terms of javascript interop
- Reason javascript interop is still good which solves the lack of library problem
- Reasonml does optimizations on the javascript code, Typescript cannot because it is unsound
- Reasonml can target native, mobile, web and nodejs