idea
基础软件中心

ReScript crash course

基础软件理论与实践公开课

黎伟坚

Introduction

Why ReScript ?

- Compiles to readable & efficient JavaScript
- No null pointers/No undefined errors
- When it compiles, it mostly works
- Fearless refactoring
- Model data accurately
- Unit tests easy to maintain

How to run and debug ReScript ?

```
git clone https://github.com/rescript-lang/rescript-project-template
cd rescript-project-template
npm install
npm run res:start
node src/Demo.bs.js
```

Or https://rescript-lang.org/try

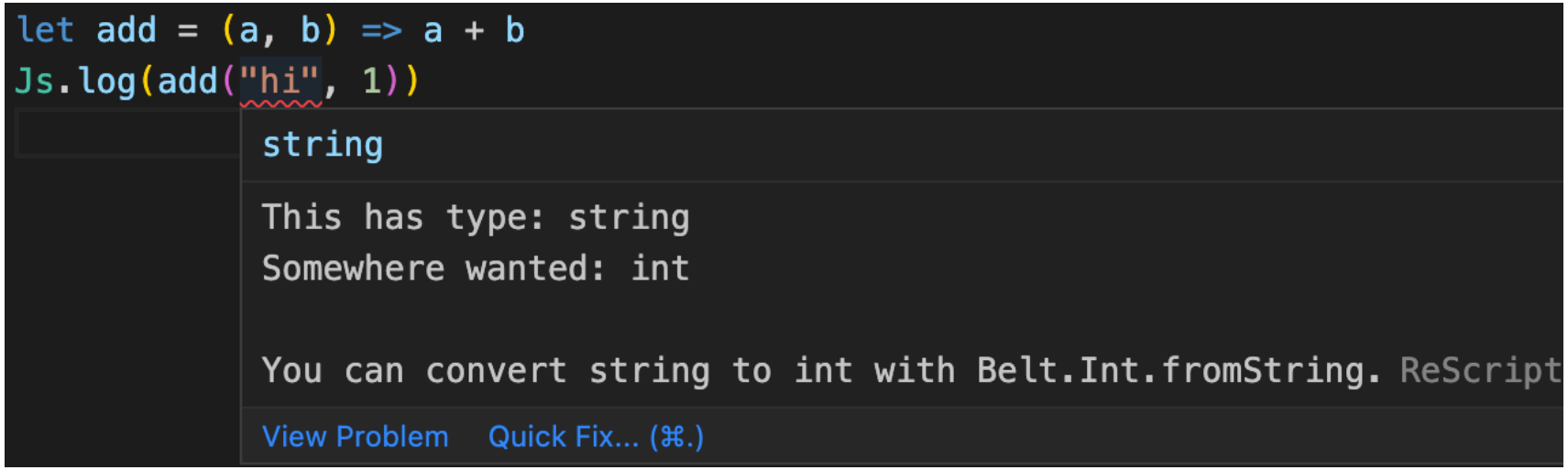
Primitive Types

- ReScript comes with: `string`, `int`, `float`, `char`, `bool`
- ReScript is a **Strong** & **Soundly** typed language
- Different operators on different types(`++`, `+`, `+.`)

Primitive Types - example 1

```
// JavaScript, got "hi1"
const add = (a,b) => a + b
console.log(add("hi", 1))
```

In ReScript:



Primitive Types - example 2

```
// JavaScript
const add = (a,b) => a + b
console.log(add(1,3)) // 4
console.log(add(1.5, 4.2)) // 5.7
console.log(add("he","llo")) // hello
```

```
// ReScript
let addInt = (a, b) => a + b
Js.log(addInt(4, 1)) // 5

let addFloat = (a, b) => a +. b
Js.log(addFloat(2., 3.3)) // 5.3

let addString = (a, b) => a ++ b
Js.log(addString("he", "llo")) // hello
```

Record

- Record immutable by default
- Record have fixed fields

Record example

```
// Type Declaration
type org = {
  name: string,
  age: int,
}

// Create a record
let idea = {
  name: "idae",
  age: 4,
}

// Access record's field
Js.log(idea.name) //idae
```

Record - immutable update

```
type org = {
  name: string,
  age: int,
}

let idea = {
  name: "idae",
  age: 4,
}

let newIdea = {
  ...idea,
  name: "idea",
}
Js.log(idea.name) //idae
Js.log(newIdea.name) //idea
```

Record - mutable update

```
// mutable: efficiently update fields inplace
type org = {
  mutable name: string,
  age: int,
}

let idea = {
  name: "idae",
  age: 4,
}

idea.name = "idea"
Js.log(idea.name) //idea
```

Record - mutate let binding

- Let-bindings are immutable
- Use `ref` to expose as a record with a single mutable field

```
let myValue = ref(5)

let five = myValue.contents // 5

myValue.contents = 6 // update myValue
myValue := 6 // or use syntax sugar

let six = myValue.contents // 6
```

List

- immutable
- fast at prepending items
- fast at getting the head

List - internal

```
// Demo.res
let myList = list{1, 2, 3}
```

```
// Demo.bs.js
var myList = {
  hd: 1,
  tl: {
    hd: 2,
    tl: {
      hd: 3,
      tl: /* [] */0
    }
  }
};
```

List - pattern matching 1

`switch` is usually used to access list items

```
let message = switch myList {
| list{} => "This list is empty"
| list{a, ..._} => "Head of the list:" ++ Js.Int.toString(a)
}

Js.log(message) //Head of the list:1
```

List - pattern matching 2

We could use `switch` to go through the list

```
// Demo.res
let rec printList = list2print => {
  switch list2print {
  | list{} => ()
  | list{hd, ...tail} => {
      Js.log(hd)
      printList(tail)
    }
  }
}
```

List - immutable prepend

```
let myList = list{1, 2, 3}
let anotherList = list{0, ...myList}

printList(myList) // 1 2 3
printList(anotherList) // 0 1 2 3
```

PS: `list{a, ...b, ...c}` syntax is coming

Array

- ordered data structure
- random accessed
- dynamic resized
- mutable update

Array - internal

```
// Demo.res
let myArray = ["hello", "world", ":)"]
```

```
// Demo.bs.js
var myArray = [
  "hello",
  "world",
  ":)"
];
```

Array - usage

```
let myArray = ["hello", "world", ":)"]

Js.log(myArray[0]) // hello

myArray[0] = "hey"

Js.log(myArray[0]) // hey

let _ = Js.Array2.push(myArray, "bye")

Js.log(myArray[3]) // bye
```

Variant and Switch

- Record: express this **and** that
- Variant: express this **or** that
- Represent complex data
- Organizing complex case-analysis

Variant - usage

```
type status = Succ | Fail | Maybe

let handleStatus = st => {
  if st == Succ {
    "nice"
  } else if st == Fail {
    "take a look"
  } else {
    "try again"
  }
}

Js.log(handleStatus(Succ)) // nice
```

Variant - with switch

```
let matchStatus = st =>
  switch st {
  | Succ => "nice"
  | Fail => "take a look"
  | Maybe => "try again"
  }

Js.log(matchStatus(Fail)) // take a look
```

Variant - constructor arguments

```
type argStatus = Succ | Fail(string) | Maybe(int)

let matchArgStatus = st =>
  switch st {
  | Succ => "nice"
  | Fail(msg) => msg
  | Maybe(n) => "Try " ++ Belt.Int.toString(n) ++ " times"
  }

Js.log(matchArgStatus(Maybe(3))) // Try 3 times
Js.log(matchArgStatus(Fail("Permission error"))) //Permission error
```

For loop

Iterate from a starting value to the ending value(**include**).

```
for x in 1 to 3 {
  Js.log(x) // 1 2 3
}

for x in 10 downto 7 {
  Js.log(x) // 10 9 8 7
}
```

While loop

While loops execute its body code block while its condition is true.

```
let break = ref(false)

while !break.contents {
  if Js.Math.random() > 0.3 {
    break := true
  } else {
    Js.log("Still running")
  }
}
```

Function - declaration

idea
基础软件中心

ReScript functions are declared with an arrow and return an expression

```
// Demo.res
let greetMore = () => "hi"
let add1 = a => a + 1
let addMore = (x, y, z) => x + y + z
```

```
// Demo.bs.js
function greetMore(param) {
  return "hi";
}

function add1(a) {
  return a + 1 | 0;
}

function addMore(x, y, z) {
  return (x + y | 0) + z | 0;
}
```

Function - labeled arguments

Labeled arguments are useful, especially when arguments are of the same type

```
let addCoordinates = (x, y) => {
  // use x and y here
}
// ...
addCoordinates(5, 6) // which is x, which is y?

let addCoordinatesNew = (~x, ~y) => {
  // use x and y here
}
// ...
addCoordinatesNew(~x=5, ~y=6)
```

Function - currying

```
// Demo.res
let add = (a, b) => a + b
let add1 = add(1)
let n1 = add1(5)
```

```
// Demo.bs.js
function add(a, b) {
  return a + b | 0;
}

function add1(param) {
  return 1 + param | 0;
}

var n1 = 6;
```

Function - recursive

Use `rec` to declare a recursive function

```
let rec listHas = (lst, item) =>
  switch lst {
  | list{} => false
  | list{a, ...rest} => a === item || listHas(rest, item)
  }

Js.log(listHas(list{1, 2, 3}, 3)) // true
```

Function - mutually recursive

idea
基础软件中心

Use `rec` and `and` to declare a recursive functions

```
let rec is_even = x => {
  if x == 0 {
    true
  } else {
    is_odd(x - 1)
  }
}
and is_odd = x => {
  if x == 0 {
    false
  } else {
    is_even(x - 1)
  }
}

Js.log(is_even(10)) // true
Js.log(is_even(1)) // false
```

Function - pipe

```
let add = (a, b) => a + b
let add1 = add(1)
let add10 = add(10)
let add100 = add(100)

let sum1 = add1(5)
let sum2 = add10(add1(5))
let sum3 = add100(add10(add1(5))) // hard to read

let sum3a = 5->add1->add10->add100 // pipe to rescue
```

Modules - files

- Files correspond to modules

```
// Aux.res
let add = (a, b) => a + b
```

```
// Demo.res
let n = Aux.add(2, 3)
```

Modules - module

- Use the `module` keyword to create a module

```
// Demo.res
module Helper = {
  let mul = (a, b) => a * b
}

Js.log(Helper.mul(2, 3))
```