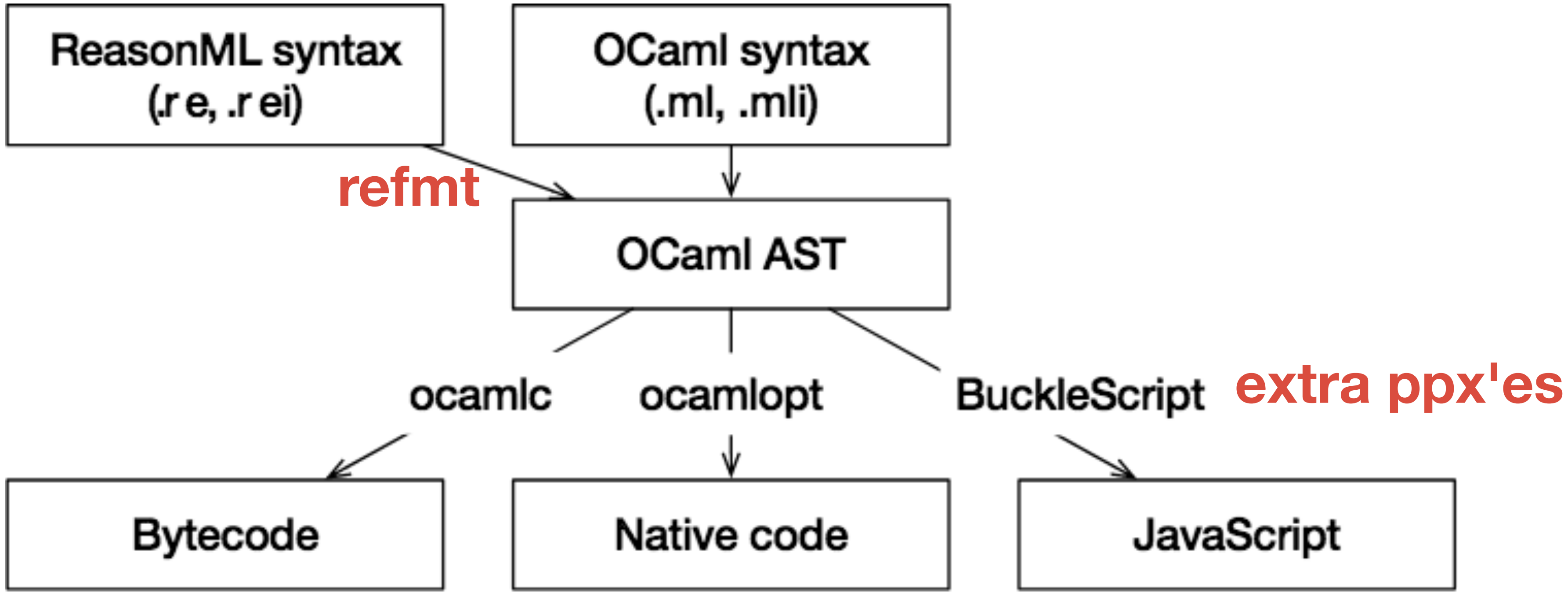
REASON WORKSHOP

Revision 1.3

Patrick Stapfer / @ryyppy

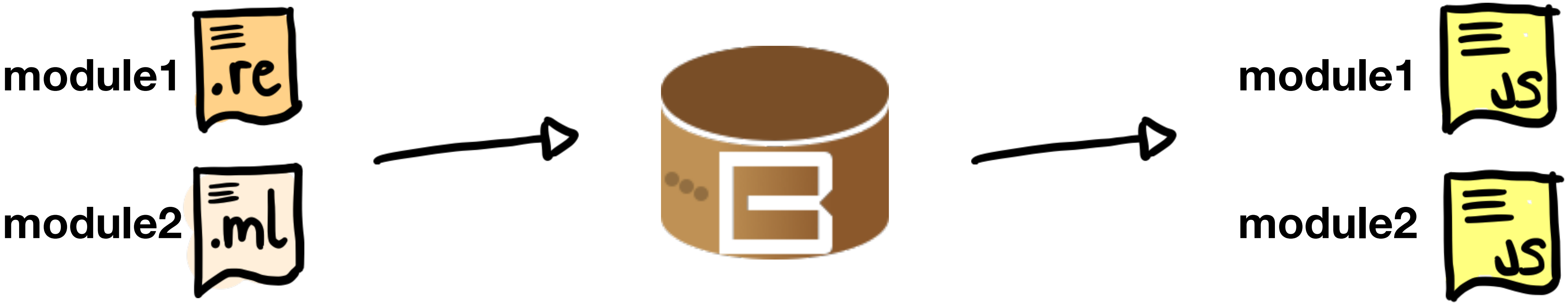
About Reason

About Reason



- Reason = Syntax (menhir + ppx) + refmt + npm-tooling
- Reason AST == OCaml AST (**A**bstract **S**yntax **T**ree)
- Is able to use OCaml build tools to compile to **native**
- Leverages the **BuckleScript platform** to compile to **JS**
- Every compiler works **exclusively with OCaml ASTs**

About Reason



module1

module2

module1

module2

```
$ npm install -g bs-platform
```

- **BuckleScript** compiles OCaml to JS
- It offers **additional JS** related **modules** for **interop**
- It is yielding very **efficient** JS code + does aggressive **dead-code elimination**
- Understanding BS' **interop layer** is the **most challenging part** of this workshop
- This tool is "kinda like the TypeScript-compiler", but for the OCaml ecosystem

About Reason

Detailed Info about the relations between these projects can be found on the related website:

- **Reason**
  https://reasonml.github.io/docs/en/what-and-why.html

- **BuckleScript**
  https://bucklescript.github.io/docs/en/what-why.html

About Reason

Major differences between the OCaml <--> **Reason** ecosystem:

- **Reason** is commonly used in tandem with BuckleScript
- **Reason** is focusing the **npm / yarn** workflow
- **Reason** tries to **unify tools** and optimizes them for the **JS** use-case
- **Reason is just an alternative Syntax to OCaml**, therefore it can be used with every major OCaml build tool

About Reason

Major differences between the **OCaml** <--> Reason ecosystem:

- **OCaml** is using the **opam** package manager & dune (prev. jbuilder) build-tool to target native binaries

- **OCaml** native binaries are **quite fast** and **efficient**

About Reason

Goals of the Reason Project:

- Offer an **alternative syntax**, which makes it easier for JavaScript developers to get into OCaml
- Leverage the **OCaml type system** to build **type-safe webapps**
- **Modernize docs** of the OCaml ecosystem during the process

Note: It **doesn't want to replace OCaml nor JavaScript** (all languages even can be mixed inside a project)

About Reason

Why should we care?

- Reason & OCaml is a **pragmatic functional programming language** by allowing **OOP features and mutability**
- It **maps really well to JavaScript** (multiparadigm "functional language")
- The **type-system** has been around for more than 20 years and offers really **strong type guarantees** without writing too many annotations (think: **Flow on steroids**)
- 1st level support for **immutability, functional composition**
- Domain Driven Development with **Variants & Pattern Matching**!

Disclosure

Word of Caution

- Reason is something you are probably not used to
- Some type errors will be confusing
- As soon as you get over the first big hurdles and understand the basic concepts, it will gradually be more enlightening
- Don't overthink it! Go slow and ask questions if something is unclear
- We will only scratch the surface in 4 hours

Data Types

```
/* unit */
let nothing = ();

let str = "Some string";

/* Int is its own data type */
let someInt = 1;

/* The dot signals a floating point number */
let someFloat = 1.;

/* Yeah, Reason also supports single characters */
let someChar = 'c';

/* List is immutable, good for small number of entries */
let someList = [1, 2, 3];

/* Arrays are quicker and mutable... good for JS interop */
let someArray = [|1, 2, 3|];

/* Tuples always contain a strict fixed number of elements */
let someTuple = (1, 2);

/* You can annotate variables as well */
let someAnnotated: string = "";

/* Some record (needs type definition of given record) */
let someRecord = {test: "test", good: true };
```

Defining Types

```
type aa = int;
type bb = string;
type cc = float;
type tupleT = (int, int);

/* You can do type aliases */
type someAlias = aa;

/* A record type for structured data */
type user = {
  name: string,
  friendly: bool
};

/* Closed JS object type */
type jsUser = {.
    "name": string,
    "friendly": bool,
};

/* Open JS object type */
type openUser('a) = {..
    "fullname": string
} as 'a;
```

Variants

```
/* This is a variant type `color` with 3 tags */
type color = Red | Green | Blue;

/* Tags don't have any concrete value.
Note that we never have to annotate `myColor` */
let myColor = Red;

/* You can define type constructors, which
can attach data to provided Tags */
type distance = int;
type movement =
    | Up(distance)
    | Down(distance)
    | Left(distance)
    | Right(distance);


/* When we want to use `Up`, we need to provide a value */
let myMove = Up(10);
```

Variants: Option Type

Special variant type: `option`

```
/* Option Type */
let maybeString = Some("test");
let notAString = None;
```

- A global type constructor provided by OCaml

- **It's a simple variant type definition:**

  ```
  type option('a) = Some('a) | None;
  ```

- The only way to express "Nullability" in OCaml

- There is **no null** in OCaml!

- Options are handled like any other variant (pattern matching)

Pattern Matching

```
let lamp =
  switch (1) {
  | 0 => "off"
  | 1 => "on"
  | _ => "off"
  };

switch(myMove) {
    | Up(distance) => Js.log({j|Walked $distance upwards|j})
    | Down(distance) => Js.log({j|Walked $distance downwards|j})
    | Left(distance) => Js.log({j|Walked $distance to the left|j})
    | Right(_) => Js.log({j|We don't really walk to the right|j})
};

switch someList {
    | [] => Js.log("Empty list")
    | [a] => Js.log("First value: " ++ string_of_int(a))
    | [_, ...b] => {
        let sum = List.fold_left((+), 0, b) |> string_of_int;
        Js.log("Sum: " ++ sum)
      }
};
```

Function Value & Types

Basic definitions:

```
/* A function is just a value */
let add = (a, b) => a + b;

/* We can define types of a function */
type addFn = (int, int) => int;
```

Generic version:

```
/* We can define generic placeholders for function types as well */
type genericAdd('a) = ('a, 'a) => 'a;

/* Here we are giving type hints to make addFloat complain
if we use the + operator instead of +. */
let addFloat: genericAdd(float) = (a, b) => a +. b;
```

Currying & Application

In Reason and OCaml, functions are automatically curried until all parameters for a call are in place:

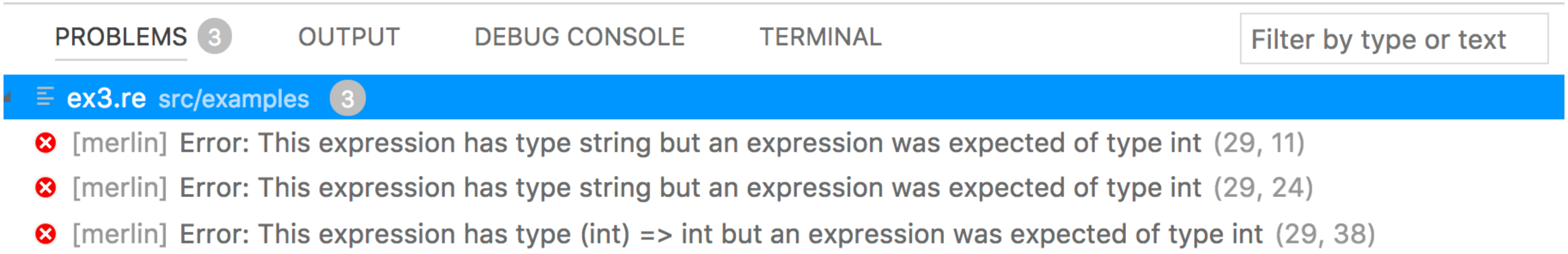
```
/* We bind the first argument (a) to 3, which
will return a new function (int) => int called add3 */
let add3 = add(3);

/* returns 5 */
add3(2);
```

- A function which is returned through currying is also called a "partially applied function"
- This can cause confusing type errors whenever you forget to provide all parameters and assume a certain type inside a variable

Currying & Application

This example shows how currying can produce type-errors, because the developer partially applied the **add** function by accident

```
let result = add(3);
let str = "Result: " + string_of_int(result);
```



ex3.re src/examples 3

```
[merlin] Error: This expression has type string but an expression was expected of type int (29, 11)
[merlin] Error: This expression has type string but an expression was expected of type int (29, 24)
[merlin] Error: This expression has type (int) => int but an expression was expected of type int (29, 38)
```

Pipe Operator / Composition

```
let convertMtoF = (ch) => switch(ch) {
    | 'M' => 'F'
    | v => v
};

/* Trivia: How to optimize this for JS ?
Tip: Look in the BS JS-Api for another interop function */
let repeatString = (n, str) =>
  Array.fold_left((++), "", Array.make(n, str));

let result =
    "moo"
    |> String.capitalize
    |> String.map(convertMtoF)
    |> repeatString(2);

let repeatFoo3times = 3 |> repeatString(_, "foo");

/* Equivalent: Fast-Pipe operator to inject the left side value as the
   first position parameter of the right side function: */
let repeatFoo3times_fastpipe = 3->repeatString("foo");
```

- |> is the "pipe operator", it feeds the outcome of the left-hand function into the last argument position of the right-hand function

- This is the reason why auto-currying can be found in all major functional programming languages!

- Prefer the new fast-pipe / "_ placeholder" syntax for better optimized JS output (prevent currying for JS)

Modules

A few words about Modules

- Every **.re** file is a module of the **same name**
- All module names are automatically capitalized (ex1.re --> Ex1)
- Module filenames should not contain any special characters (e.g. no \`-\` or multiple \`.\` allowed)
- Modules can be nested & parametrized (Functors)
- Module names are globally unique inside a project (requires some small workarounds for JS, like index.js files)
- Use `open MyModule;` to get access to types & values in the current module scope ("for using the declarations")
- Use `include MyModule;` for making types & values part of the module ("for copying the declarations")

Modules

```
/* Nested module inside ch01.re */

module MyValidator {
  type t('a) = Validated('a) | NotValidated;

  let validate = (a) => Validated(a);

  /* Needs to be implemented */
  let isValidated = (_a) => false;
};

let validatedInt = MyValidator.validate(1);
let isActuallyValidated = MyValidator.isValidated(validatedInt);
```

Notes:

- 'a is a generic placeholder for a specific type

- type t is a common convention for module specific types, sometimes they are abstract (type t;)

Labeled Arguments

Unlike in JS, function arguments can be labeled (assigned by name):

```
/* One labeled argument, all parameters required */
let processFilepath = (~ext, filepath: string) : string => {
  filepath ++ "." ++ ext;
};

/* Labeled arguments can have a default value */
let processFilePathWithDefaultExt = (~ext="txt", filepath) => {
  /* handy shorthand, longversion: ~ext=ext */
  processFilepath(~ext, filepath);
};
```

- Label args **without default value** are treated as **option** type (require pattern-matching)

- For easier auto-curry, if all arguments are labeled, it is recommended by convention to add a () as a last argument

Labeled Arguments & Currying

Example why a last unnamed argument is important:

```
let processFullPath = (~name, ~dir, ~ext, ()) => {
  {j|$dir/$name.$ext|j};
};

/* We can now apply all labeled arguments without applying the
function */
let runProcessFullPath = processFullPath(
 ~name="test",
 ~dir="test",
 ~ext="txt"
 );

/* This calls the function */
runProcessFullPath();
```

Note: ( ) is the value of type **`unit`** and represents "nothing" (this is not the same as null!)