When Type Checking Goes WRONG ! But Keeps Going

From **Merlin** to OCaml's Typechecker: **Integrating Typing Recovery**

Tarides - Xavier Van de Woestyne - **xvw.lol**

Some features are **discreetly** integrated into a tool, making their presence almost **unnoticeable**, but their absence **catastrophic**.

Some features are **discreetly** integrated into a tool, making their presence almost **unnoticeable**, but their absence **catastrophic**.

Like **Merlin recoveries mechanisms** (parsing and typing)

Some features are **discreetly** integrated into a tool, making their presence almost **unnoticeable**, but their absence **catastrophic**.

Like **Merlin recoveries mechanisms** (parsing and typing)

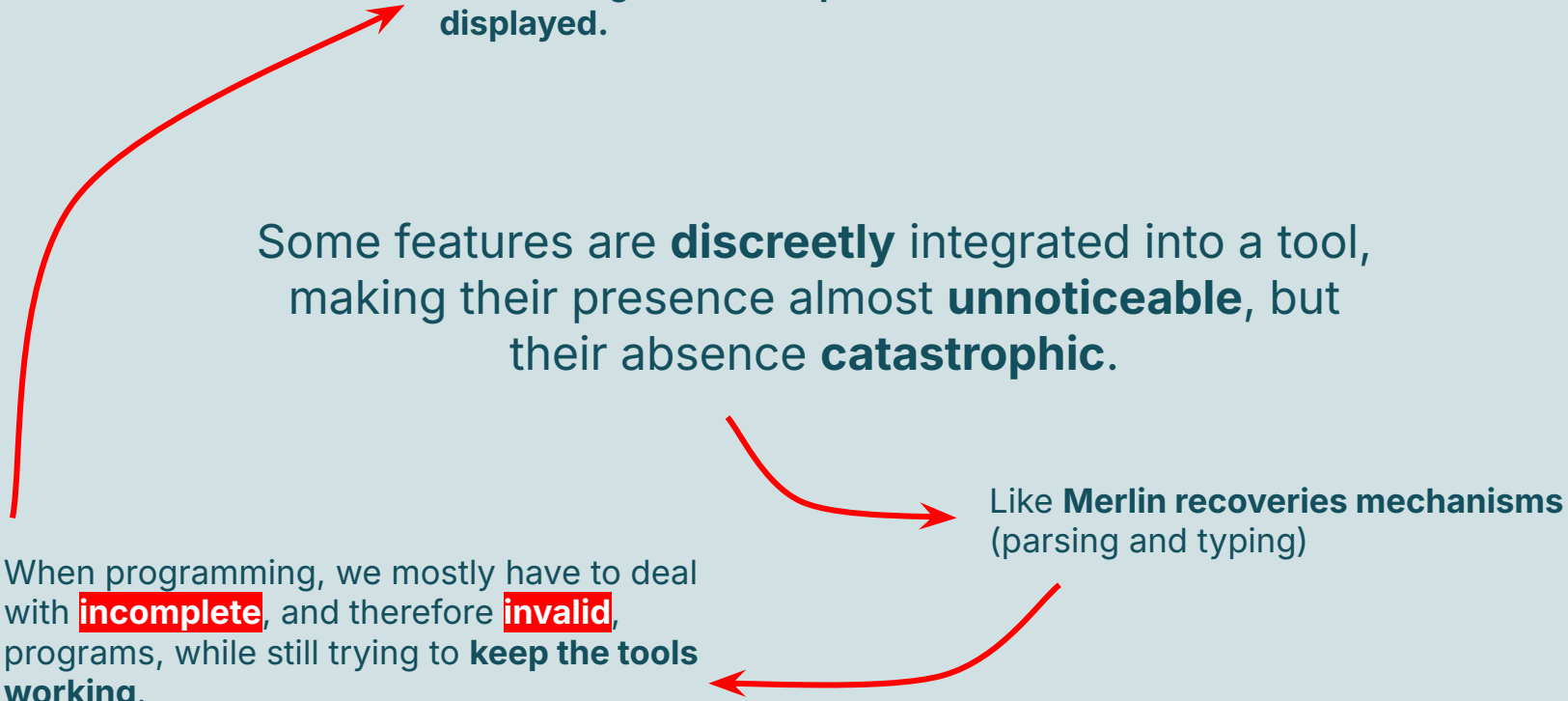
When programming, we mostly have to deal with **incomplete**, and therefore **invalid**, programs, while still trying to **keep the tools working**.

More than having the tool to working (such as completion), we want **a good user experience when errors are displayed.**

Some features are **discreetly** integrated into a tool, making their presence almost **unnoticeable**, but their absence **catastrophic**.

Like **Merlin recoveries mechanisms** (parsing and typing)

When programming, we mostly have to deal with **incomplete**, and therefore **invalid**, programs, while still trying to **keep the tools working**.



More than having the tool to working (such as completion), we want **a good user experience when errors are displayed.**

Some features are **discreetly** integrated into a tool, making their presence almost **unnoticeable**, but their absence **catastrophic**.

Like **Merlin recoveries mechanisms** (parsing and typing)

When programming, we mostly have to deal with **incomplete**, and therefore **invalid**, programs, while still trying to **keep the tools working**.

This presentation will explain the integration of one of these mechanisms, **Typing Recovery**, from **Merlin** to **OCaml** by answering the following questions:

This presentation will explain the integration of one of these mechanisms, **Typing Recovery**, from **Merlin** to **OCaml** by answering the following questions:

This presentation will explain the integration of one of these mechanisms, **Typing Recovery**, from **Merlin** to **OCaml** by answering the following questions:

WHAT?

Whats is Typing Recovery

WHY?

Why not keeping it into Merlin

HOW?

Design, Configuration

And last but not least: **WHERE ARE WE?**

```
let f x =
 x + 1

let g x =
 f x @ [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> int_of_string x

let l = [ 1; 2; 3; "4"; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

Let's imagine the following **program**

```
let f x =
 x + 1

let g x =
 f x @ [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> int_of_string x

let l = [ 1; 2; 3; "4"; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

Let's imagine the following **program**

```
File "test.ml", line 5, characters 2-5:
5 |  f x @ [ 1; 2; 3 ]
     ^^
Error: This expression has type int but an expression
       was expected of type 'a list
```

```
let f x =
 x + 1

let g x =
 f x :: [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> int_of_string x

let l = [ 1; 2; 3; "4"; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

Let's imagine the following **program**

```
File "test.ml", line 9, characters 14-29:
9 |  |`Bar x -> int_of_string x
                ^^^^^^^^^^^^^^
Error: This expression has type int but an expression
       was expected of type string
```

```
let f x =
 x + 1

let g x =
 f x :: [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> string_of_float x

let l = [ 1; 2; 3; "4"; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

Let's imagine the following **program**

```
File "test.ml", line 11, characters 19-22:
11 | let l = [ 1; 2; 3; "4"; 5; bool ]
                       ^^
Error: This constant has type string but an
       expression was expected of type int
```

```
let f x =
 x + 1

let g x =
 f x :: [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> string_of_float x

let l = [ 1; 2; 3; 4; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

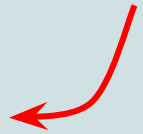
Let's imagine the following **program**

```
File "test.ml", line 11, characters 25-29:
11 | let l = [ 1; 2; 3; "4"; 5; bool ]
                            ^^^^

Error: This constant has type bool but an
       expression was expected of type int
```

```
let f x =
 x + 1

let g x =
 f x :: [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> string_of_float x

let l = [ 1; 2; 3; 4; 5; true]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

This approach is perfectly accurate.

When compiling a program where you want the most precise errors, and stopping type checking as soon as a type error is detected in a compilation unit **is perfectly reasonable for reporting precise errors.**



But when I am editing the code I problably want have **more than one-per-one error.**

```
let f x =
 x + 1

let g x =
 f x @ [ 1; 2; 3 ]

let h = function
 | `Foo x -> string_of_int x
 | `Bar x -> int_of_string x

let l = [ 1; 2; 3; "4"; 5; bool ]

type t = { x: int; y: string }

let i {x; y; z} =
 (x + int_of_string y) ^ z
```

Compiling leads to the **same first error.**
Editing retrieves **more than the first error**
(accepting the price of less precises errors)

```
File "test.ml", line 5, characters 2-5:
5 |  f x @ [ 1; 2; 3 ]
     ^^^
Error: This expression has type int but an expression
was expected of type
         'a list
```

Here is an example using **Emacs** and **OCaml-eglot** (which **uses Merlin under the hood**)

It was working on Merlin **since almost day 1...** so

Merlin uses a patched version of the OCaml Frontend

Short Answer: because it is hard.

Why not just leave things as they are **and keep Typing Recovery in Merlin?**

Merlin uses a patched version of the OCaml Frontend

Short Answer: because it is hard.

Why not just leave things as they are **and keep Typing Recovery in Merlin?**

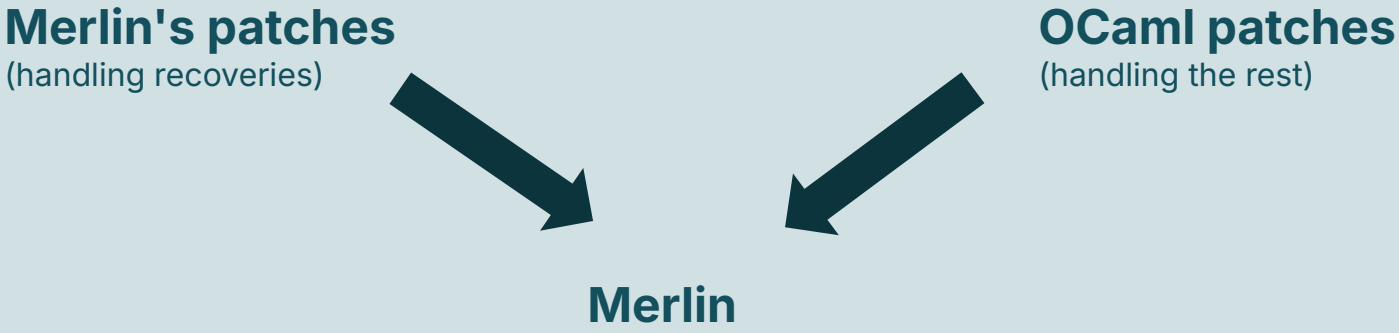
SO AT EVERY NEW OCAML'S VERSION (~6MONTHS)

Merlin's patches
(handling recoveries)

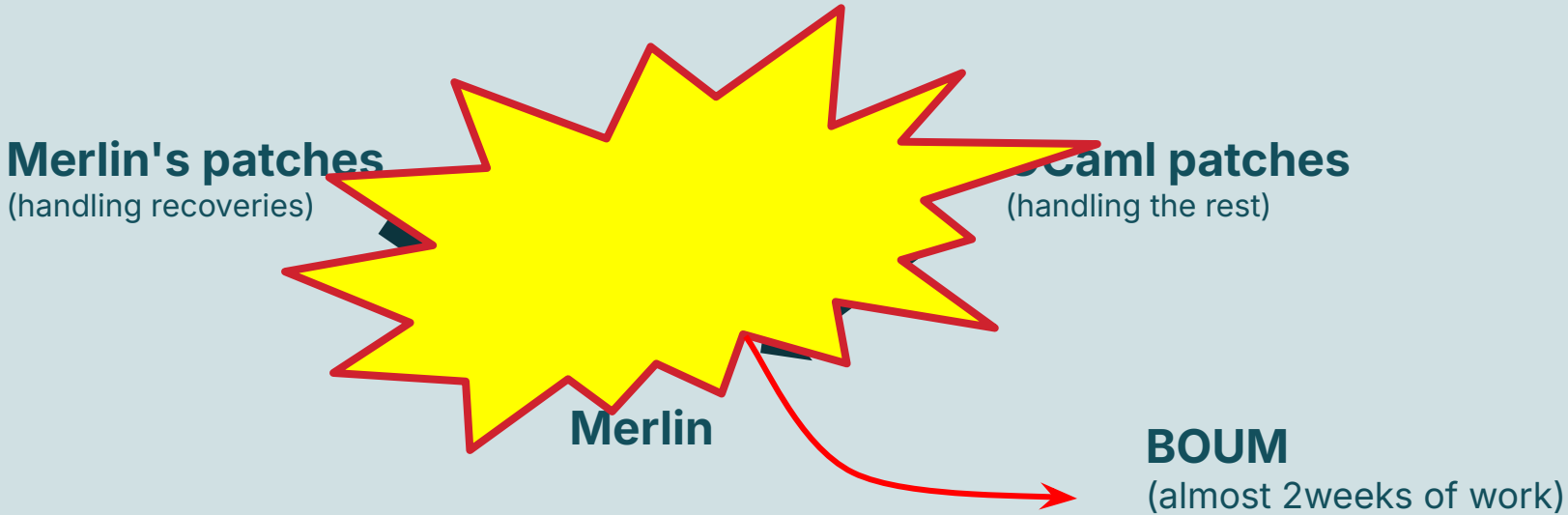
OCaml patches
(handling the rest)

Merlin

SO AT EVERY NEW OCAML'S VERSION (~6MONTHS)



SO AT EVERY NEW OCAML'S VERSION (~6MONTHS)



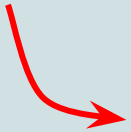
It is nice to reduce the diff between Merlin-OCaml's frontend and the one from OCaml

And usually, maintainers have probably a better idea about how to recover in presence of errors

It is nice to reduce the diff between Merlin-OCaml's frontend and the one from OCaml

And usually, maintainers have probably a better idea about how to recover in presence of errors

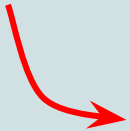
It is nice to reduce the diff between Merlin-OCaml's frontend and the one from OCaml



HOW?

And usually, maintainers have probably a better idea about how to recover in presence of errors

It is nice to reduce the diff between Merlin-OCaml's frontend and the one from OCaml



HOW?

- **Optional**
- **Systematic**
- **Simple**

THE BIG PICTURE

In Merlin, **Parsing** and **Typing** always produce a "valid" result:

- Every source produce a Parsetree (Parsing recovery)
- Every Parsetree produce a Typedtree (Typing recovery)

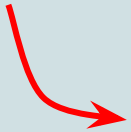
Make the tooling still working even on invalid buffer.

THE BIG PICTURE

The Typechecking is done by some big recursive functions

THE BIG PICTURE

The Typechecking is done by some big recursive functions



ERROR (exception)

we **log** the exception

THE BIG PICTURE

The Typechecking is done by some big recursive functions

ERROR (exception)

we **log** the exception

we **resume** (locally) at the recursive call on the next independent subterm.

So we can report multiple typing errors

THE BIG PICTURE

The Typechecking is done by some big recursive functions

ERROR (exception)

we **log** the exception

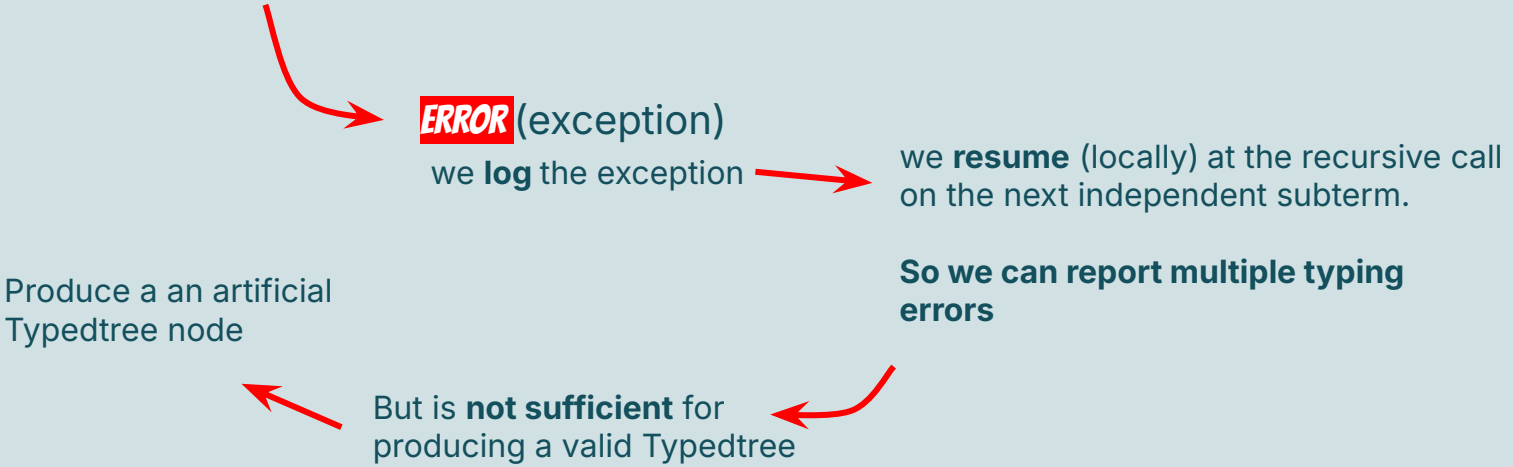
we **resume** (locally) at the recursive call on the next independent subterm.

So we can report multiple typing errors

But is **not sufficient** for producing a valid Typedtree

THE BIG PICTURE

The Typechecking is done by some big recursive functions



This the **Generic Recovery**
There is also **fine-grained cases** (e,g: typing record labels)

HOW TO PROCEED

A new flag **-typing-recovery** (for activating it)

The goal is to be used by Merlin and other tools
(so **not by human users**... *please*)

A SIMPLE API

Replacing Typechecking-related exception **raise** by:

- **log_and_raise** : exn → 'a
- **log_or_raise** : exn → unit

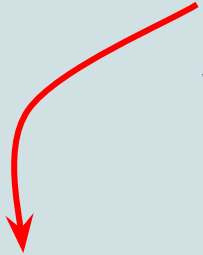
Without **-typing-recovery** they behaves exactly like **raise**

A SIMPLE API

Replacing Typechecking-related exception **raise** by:

- **log_and_raise** : exn → 'a
- **log_or_raise** : exn → unit

Without **-typing-recovery** they behaves exactly like **raise**



Log the error and resume the typing
(When **we can** continue locally)

A SIMPLE API

Log the error and re-raise
(When **we cannot** continue locally)

Replacing Typechecking-related exception **raise** by:

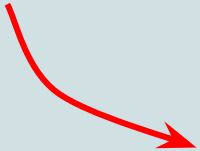
- **log_and_raise** : exn → 'a
- **log_or_raise** : exn → unit

Without **-typing-recovery** they behaves exactly like **raise**

Log the error and resume the typing
(When **we can** continue locally)

WHEN TO RECOVER?

WHEN TO RECOVER?



Not very clear in the historical Merlin implementation

WHEN TO RECOVER?

Not very clear in the historical Merlin implementation

During the re-implementation, we identify a *kind of invariant*:

Only user-facing errors should be loggedby the recovery.

Others, **internal type-checker exception** should ne be logged

WHEN TO RECOVER?

Not very clear in the historical Merlin implementation

During the re-implementation, we identify a *kind of invariant*:

Only user-facing errors should be loggedby the recovery.

Others, **internal type-checker exception** should ne be logged

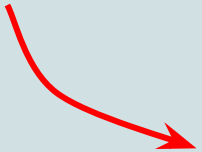
how to enforce the logging of user-facing error ?

```
exception Error of error
module Error : sig
 type exn += private In_context of error
 val log_or_raise : error -> unit
 val log_and_raise : error -> 'a
end = struct
 type exn += In_context of error

 let log_or_raise err =
   Typing_recovery.log_or_raise (In_context err)

 let log_and_raise err =
   Typing_recovery.log_and_raise (In_context err)
end
```

Let's follow the typechecker :)

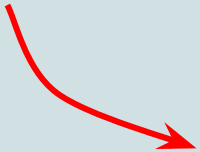
We know **"When perform recovery"** but how to choose between **log_and_raise** and **log_or_raise** ?

We know **"When perform recovery"** but how to choose between **log_and_raise** and **log_or_raise** ?

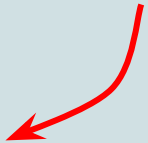


We have **a very precise algorithm !!!**

We know **"When perform recovery"** but how to choose between **log_and_raise** and **log_or_raise** ?

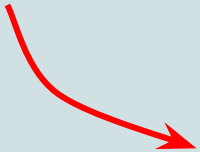


We have **a very precise algorithm !!!**



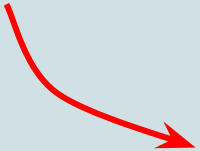
We **write a test** and we observe if the recovery seems fine :)

We know **"When perform recovery"** but how to choose between **log_and_raise** and **log_or_raise** ?



We have **a very precise algorithm !!!**

We **write a test** and we observe if the recovery seems fine :)



We did not find a precise approach...

WHERE ARE WE?

WHERE ARE WE?

the **PR** was merged and it will available in the next release of OCaml (5.6) !

Thanks to **Thomas Refis** for his **huge help and guidance** on the design, the understanding and the deep-dive review.
And to **Florian Angeletti** for a very Quick'N'Precise review before the merge (and help for finalizing the implementation of deep copy of identifier)

WHERE ARE WE?

the **PR** was merged and it will available in the next release of OCaml (5.6) !

Thanks to **Thomas Refis** for his **huge help and guidance** on the design, the understanding and the deep-dive review.
And to **Florian Angeletti** for a very Quick'N'Precise review before the merge (and help for finalizing the implementation of deep copy of identifier)

We have a Merlin Branch that previews 5.6
(with a working test-suite!!!)

(It was a big migration but we know that it will drastically reduce the process for the following ones.

We can discuss more on some implementation details...

BUT THIS IS THE END. THANKS