Pattern-matching with mutable state: danger!

Thomas Refis, Nick Roberts, **Gabriel Scherer**

September 26, 2024

Danger!

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =    (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;    b = None} → 1
  | {a = _;    b = _} when (                    ) → 2
  | {a = true; b = Some y} → y
```

Danger!

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =   (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;     b = None} → 1
  | {a = _;     b = _} when (x.b <- None; false) → 2
  | {a = true; b = Some y} → y
```

Danger!

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =   (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;     b = None} → 1
  | {a = _;     b = _} when (x.b <- None; false) → 2
  | {a = true; b = Some y} → y

let _ = f {a=true; b=Some 5}
```

Danger!

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =   (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;    b = None} → 1
  | {a = _;    b = _} when (x.b <- None; false) → 2
  | {a = true; b = Some y} → y

let _ = f {a=true; b=Some 5}
(* Segmentation fault (core dumped) *)
```

Danger!

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =    (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;    b = None} → 1
  | {a = _;    b = _} when (x.b <- None; false) → 2
  | {a = true; b = Some y} → y

let _ = f {a=true; b=Some 5}
(* Segmentation fault (core dumped) *)
```

Recipe:

- patterns that look into mutable fields
- ability to evaluate code concurrently (**when** guards, allocations, data races)
- optimizing pattern compiler

In this talk

1. Automata/Backtracking/Split-based pattern-matching compilation
2. Optimizations in OCaml
3. Relaxing optimizations for mutable state

Section 1

Automata/Backtracking/Split-based pattern-matching compilation

Pattern-matching compilation

General case: n-ary pattern matrices.

$$
\begin{array}{l}
\texttt{match}\ \langle a_1 \ldots a_n \rangle\ \texttt{with} \\
\mid\ \langle p_1 \ldots p_n \rangle\ \rightarrow\ e_1 \\
\mid\ \langle q_1 \ldots q_n \rangle\ \rightarrow\ e_2 \\
\mid\ \ldots \\
\mid\ \langle r_1 \ldots r_n \rangle\ \rightarrow\ e_m
\end{array}
$$

Pattern-matching compilation

Naive idea: consider all possible constructors for a_1.

```
match ⟨l v⟩ with
| ⟨[] p⟩ → foo
| ⟨_  q⟩ → bar
| ⟨[] r⟩ → bim
```

$\Longrightarrow$

```
switch l with
  | [] →
    match v with
     | p → foo
     | q → bar
     | r → bim
  | _::_ →
    match v with
     | q → bar
```

Pattern-matching compilation

Naive idea: consider all possible constructors for a_1.

```
match ⟨l v⟩ with
| ⟨[] p⟩ → foo
| ⟨_  q⟩ → bar
| ⟨[] r⟩ → bim
```

$\Longrightarrow$

```
switch l with
  | [] →
    match v with
    | p → foo
    | q → bar
    | r → bim
  | _::_ →
    match v with
    | q → bar
```

Problem: the clause `q` → `bar` is duplicated.

Pattern-matching compilation

Naive idea: consider all possible constructors for a_1.

```
match ⟨l v⟩ with
| ⟨[] p⟩ → foo
| ⟨_  q⟩ → bar
| ⟨[] r⟩ → bim
```

$\Longrightarrow$

```
switch l with
  | [] →
    match v with
     | p → foo
     | q → bar
     | r → bim
  | _::_ →
    match v with
     | q → bar
```

Problem: the clause `q → bar` is duplicated.

EXPONENTIAL!

Avoiding code blowup: two approaches

Split-based algorithms (automata/backtracking):
linear code size, but repeated check

Decision trees:
hashconsing strategies to avoid code size blowup

OCaml is split-based. (So are SML implementations; historically first)

Split-based algorithms

```
match ⟨l v⟩ with
| ⟨[] p⟩ → foo
| ⟨_  q⟩ → bar
| ⟨[] r⟩ → bim
```

$\Longrightarrow$

```
try
1: match ⟨l v⟩ with
   | ⟨[] p⟩ → foo
   | _ → fail
2: match ⟨l v⟩ with
   | ⟨_  q⟩ → bar
   | _ → fail
3: match ⟨l v⟩ with
   | ⟨[] r⟩ → bim
   | _ → fail
4: raise Match_failure
```

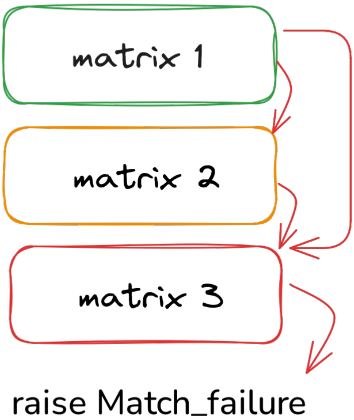
After splitting, each inner match can be compiled to a switch without duplication. `fail` jumps to the next submatrix.

Pros: linear code size.
Cons: some checks (here `[]`) are repeated.

Section 2

Optimizations in OCaml

Static information



- context:
  static knowledge on matched values
- jump summary:
  the context of each jump
  $\Longrightarrow$ optimizes jump targets
- default environment:
  the matrix of each jump target
  $\Longrightarrow$ optimize jumps
- totality
  $\Longrightarrow$ optimize the last matrix

```
compile: totality * env * context * source-matrix
      → compiled-matrix * summary
```

Contexts

```
switch p with
| false → ...
| true →
  switch l with
  | [] → ...
  | x::xs →
    (* HERE *) ...
```

- context at (* HERE *):

Contexts

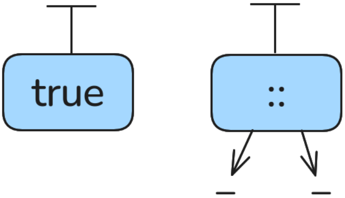
```
switch p with
| false → ...
| true →
  switch l with
  | [] → ...
  | x::xs →
    (* HERE *) ...
```

- context at (* HERE *):

  ⟨true (_::_)⟩

Contexts

```
switch p with
| false → ...
| true →
  switch l with
  | [] → ...
  | x::xs →
    (* HERE *) ...
```



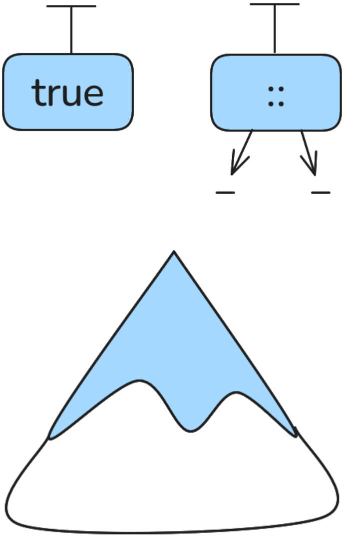
- context at (* HERE *):

  ⟨`true (_::_)`⟩

Contexts

```
switch p with
| false → ...
| true →
  switch l with
  | [] → ...
  | x::xs →
    (* HERE *) ...
```

- context at (* HERE *):
  ⟨true (_::_)⟩



Totality

$$\texttt{\textbf{match}}\ \langle\texttt{l ...}\rangle\ \texttt{\textbf{with}}$$
$$\texttt{|}\ \langle\texttt{x::xs ...}\rangle \rightarrow \texttt{foo}$$

(notice: no | _ $\rightarrow$ `fail` case)

Direct field access.

Totality

```
match ⟨l ...⟩ with
| ⟨x::xs ...⟩ → foo
```

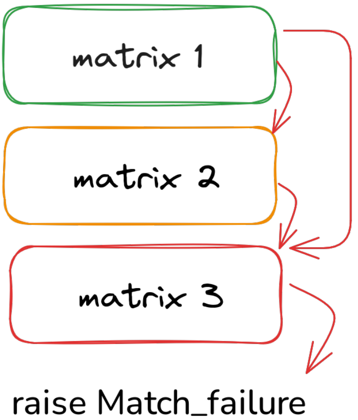
(notice: no `| _ → fail` case)

Direct field access.

Awkward design in OCaml:

- type-checker computes totality information (and checks exhaustivity, usefulness, etc.)
- compiler does not use type information

Big picture (again)



- context:
  static knowledge on matched values
- jump summary:
  the context of each jump
  $\Longrightarrow$ optimizes jump targets
- default environment:
  the matrix of each jump target
  $\Longrightarrow$ optimize jumps
- totality
  $\Longrightarrow$ optimize the last matrix

```
compile: totality * env * context * source-matrix
      → compiled-matrix * summary
```

Section 3

Relaxing optimizations for mutable state

Bug (reminder)

Bug (reminder)

```
type 'a option = None | Some of 'a
type u = {a: bool; mutable b: int option}

let f (x : u) =   (* example by Stephen Dolan, 2017 *)
  match x with
  | {a = false; b = _} → 0
  | {a = _;    b = None} → 1
  | {a = _;    b = _} when (x.b <- None; false) → 2
  | {a = true; b = Some y} → y

let _ = f {a=true; b=Some 5}
(* Segmentation fault (core dumped) *)
```

Bug 1: incorrect contexts

```
try
1: match x with
  | {a = false; b = _} → 0
  | {a = _;     b = None} → 1
  | _ → fail (* HERE *)

2: match x with
   | _ →
     if (x.b <- None; false) then 2
     else fail (* ALSO HERE *)

3: match x with
  | {a = true; b = Some y} → y
```

Context on both `fail`:

$\langle$`a = true; b = Some _`$\rangle$

Bug 1: incorrect contexts

```
1: match x with
  | {a = false; b = _} → 0
  | {a = _;    b = None} → 1
  | _ → fail (* HERE *)
```

Context on `fail`:

⟨`a = true; b = Some _`⟩

Not just about **`when`**.

At the point of `fail`, *any* concurrent mutation can invalidate the context.

Bug 1: incorrect contexts

```
1: match x with
  | {a = false; b = _} → 0
  | {a = _;     b = None} → 1
  | _ → fail (* HERE *)
```

Context on `fail`:

⟨`a = true; b = Some _`⟩

Not just about **`when`**.

At the point of `fail`, *any* concurrent mutation can invalidate the context.

Solution: erase context information in mutable positions.

below: ⟨`a = true; b = _`⟩

Safe!

Bug 2: incorrect totality

```
3: match x with
  | {a = true; b = Some y} → y
```

Notice that there is no `| _ → fail` at the end. Wrong!

Problem: the type-checker believes this program to be total.

```
match x with
| {a = false; b = _} → 0
| {a = _;    b = None} → 1
| {a = _;    b = _} when (x.b <- None; false) → 2
| {a = true; b = Some y} → y
```

Fix 1: forget about totality

Fix: do not trust the type-checker, only the match compiler;
(it can sometimes prove totality)

Problem: many programs are pessimized by this criterion, notably many GADT matches.

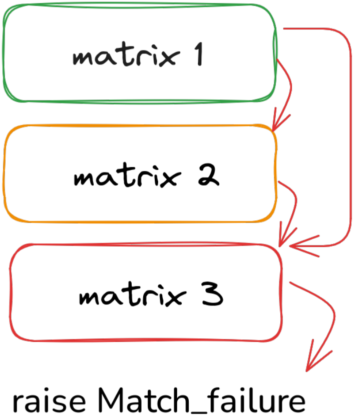
```
type _ t =
  | Int : int → int t
  | Nothing : unit t

let get (Int n : int t) = n
(* (function param : int
    (let (n =a (field_imm 0 param)) n)) *)
```

Fix 2: forget totality in mutable posititions

Only pessimize matches under a mutable field (transitively).

Pessimized programs:

Fix 3: temporality heuristic




```
type temporality =
  First | Following
```

Totality can optimize matrix 3 (outside mutable positions)

Temporality can de-pessimize matrix 1 (at mutable positions)

If the user matching has no split: no pessimization.

Impact analysis

We believe that there were *no* unsound matchings
in real-life OCaml programs.

... but the fix pessimizes more programs

How can we convince everyone to pay the cost of correctness?

Impact analysis

We believe that there were *no* unsound matchings
in real-life OCaml programs.

... but the fix pessimizes more programs

How can we convince everyone to pay the cost of correctness?

1. We implemented a warning to detect pessimization.
2. Nick Roberts compiled the Jane Street codebase with it:

> *I've tested this change and found indeed that it flags only complex matches on mutable fields — I found only 3 instances in a codebase with millions of lines, and it was possible to rewrite them without much trouble.*

Are we sound yet?

Natural specification: each clause is a test.

```
match a with
| p1 → e1
| p2 → e2
...
```

⟹

```
match a with
| p1 → e1
| _ →
  match a with
  | p2 → e2
  | _ →
  match a with
  ...
```

Are we sound yet? OCaml is not.

```
let x = ref 0
let incr () =
  Printf.printf "Observed x=%d\n" !x; x := !x + 1; false

let ret =
  match x with
  | {contents = 0} when incr () → 0
  | {contents = 1} when incr () → 1
  | _ → 2
```

Expected output: `Observed 0. Observed 1.`
Observed output: `Observed 0.`

Note: this issue is *independent* from our work.

Thanks!

Questions?

Secret slide

A scary example from Nick.

```
type 'a myref = { mutable mut : 'a }
type abc = A | B | C
type t = {a: bool; b: abc myref }

let example () =
  let input = { a = true; b = { mut = A } } in
  match input with
  | {a = false; b = _ } → 1
  | {a = _;    b = { mut = B }} → 2
  | {a = _;    b = _} when (input.b.mut <- B; false) → 3
  | {a = true; b = { mut = A }} → 4
  | {a = _;    b = _ } when (input.b.mut <- A; false) → 5
  | {a = true; b = { mut = C }} → 6
```