Pattern-matching on mutable values: danger!

September 5, 2024

Abstract

The OCaml pattern-matching compiler is unsound, it generates incorrect code in the obscure corner case where we match on a value with mutable fields, and those fields are mutated during pattern-matching – from `when` clauses, allocation callbacks, or an access race in concurrent execution.

We recall the overall compilation strategy of the OCaml pattern-matching compiler, and explain how to weaken its optimization information to remain correct in the face of mutation.

Pattern matching on mutable values

The problem

The following program segfaults on all released versions of OCaml:

```
type u = {a: bool; mutable b: int option}

let f x =
  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}
```

The function `f` is pattern-matching on a record with two fields `a : bool` and `b : int option`, and the field `b` is mutable (this could be written with an immutable field of type `int option ref`). The pattern-matching clauses first check whether `a = false`, then whether `b = None`. If these two clauses failed, then of course it must be the case that `a = true` and `b <> None`, so `b` must be of the form `Some _`. But then the third clause executes a `when` guard that sets `x.b <- None`. When the compiler checks the fourth and last clause, it is still convinced (wrongly) that `x.b` must be of the form `Some _`, and it compiles the pattern `Some y` as a direct to the first argument, without checking for the

possibility of `None` first. This tries to dereference `None` as a pointer and we get a segfault.

In other words, the OCaml pattern-matching compiler is unsound in presence of mutable state, it generates incorrect code.

The general problem is as follows:

1. The pattern-matching compiler relies on optimizations to generate better code by reasoning about what values may flow in certain sub-patterns.
2. Code execution can happen while the pattern-matching program runs, mutate the value that is being matched, and violate these assumptions.

We implemented a comprehensive fix for this issue and we are in the process of getting it merged in the OCaml compiler. (See https://github.com/ocaml/ocaml/issues/7241#issuecomment-1722226025 for the full details of the implementation progress.) The fix touches subtle, advanced parts of the pattern-matching compiler, a little-known part of the codebase. Reviewing the changes and thinking about their performance implications is non-trivial, and finding the right reviewers was difficult – the original author, Luc Maranget, is happy to give feedback from the sidelines but was not available for a full review.

In the rest of this document: we start with a reminder/tutorial on pattern-matching compilation and its optimizations as performed by the OCaml compiler; we explain why some of those optimizations are unsound in presence of mutable state; and we describe a mitigation strategy – how to weaken those optimizations – to stop generating incorrect code in this situation.

Are other languages concerned?

Our example uses a `when` guard which is arguably a dubious feature of OCaml, but there are other ways to run effectful code simultaneously with pattern-matching control flow:

- Certain pattern-matching constructs allocate (for example reading a boxed double-float-point variable from an unboxed record or array), and allocations may trigger context switches and/or call asynchronous callbacks.
- Since OCaml 5, the runtime supports parallel execution of OCaml programs. The matched value `x` could be shared with another thread/domain that mutates it in parallel.

This suggests that other ML-family languages with pattern-matching and mutable data structures may be affected by this issue as well, even if they don't offer `when`-guards or more general view patterns.

This example also relies on the fact that:

1. Pattern-matching can inspect mutable positions of the scrutinee. Haskell would not be affected as it does not allow mutable-read operations to be

composed within pattern-clauses. (Mutable reads must have a monadic type, and pattern guards do not perform monadic binding.)

2. The OCaml type discipline allows two different parts of the program to race on access to a mutable position (two accesses, one of them being a write). A direct transcription of this program in Rust would fail to satisfy the ownership discipline and be rejected at compile-time.

OCaml Pattern-matching compilation and optimizations

The OCaml compiler uses a standard "backtracking" compilation scheme, also called "compilation to automata", as described in Augustsson (1985).

A reminder on backtracking compilation

In its general form the pattern-matching compiler takes a list of argument expressions to match, of length n, and a list of "clauses", which each a list of n patterns, an optional `when`-guard, and the expression to return if the pattern and guard match.

For example (in pseudo-code, using `[a b c]` for a three-element list):

```
match [x.a x.b] with
| [false _] -> 0
| [_  None] -> 1
end
```

Let us call this input (the arguments and clauses) a *sub-matching*. This is often called a *sub-matrix*, in reference to the core part of this input data, a rectangular matrix of patterns:

```
[false _]
[_  None]
```

The core operation of the pattern-matching compiler is to *decompose* a sub-matching into a switch on the head constructor of one of the matched values, with a specialized sub-matching for each switch case. For example,

```
match [li v] with
| [[]    p] -> foo
| [x::xs q] -> bar
end
```

becomes:

```
switch li with
| [] ->
  match [v] with
  | [p] -> foo
  end
```

```
| x::xs ->
  match [v] with
  | [q] -> bar
  end
end
```

We could generate a switch for this sub-matching because the patterns in its first column each start with a head constructor for lists datatype – we can partition matching into disjoint sub-matchings, such that each possible input values may only match one of them. We say that such a sub-matching is "decomposable". (Precisely: a matrix is decomposable along a colum if, for any two patterns in the column, either they have the same head pattern constructor, or their heads match disjoint sets of values.)

Not all pattern matrices are decomposable, so the first step of pattern-matching compilation is to *split* the input sub-matching into a sequence of decomposable sub-matchings.

Consider for example the variation:

```
match [li v] with
| [[] p] -> foo
| [_  q] -> bar
end
```

The `[_ q]` clause may match whether `li` is an empty list or a cons cell. If we wanted to generate a switch right away, we would have to include this clause in both sub-matchings, resulting in code duplication – in the worst case, there may be an exponential blowup in code size, which happens on realistic examples. "Decision trees"-based compilers use sharing / hash-consing strategies to reduce code blowup, while "backtracking"/"automata"-based compilers use splits to guarantee linear-sized output – each decomposition step preserves the total number of clauses.

Here the result of the split would be

```
try
  match [li v] with
  | [[] p] -> foo
  | _ -> fail
  end
or
  match [li v] with
  | [_  q] -> bar
  end
end
```

where `fail` is a new (pseudo-code) keyword indicating that the current sub-matching failed to match the input, and `try m1 or m2 end` means: first try to see if `m1` matches the input, and if it fails match it using `m2` instead. This splits

give its names to the compilation technique ("backtracking compiler" or "matrix automata"): we can see `try m1 or m2 end` as backtracking, or we can see each sub-matching as a state ("we are in the state of checking the input against this sub-matching"), and the fallback to `m1` to `m2` as a "jump", or transition from one state to another.

In general the result of the pattern-matching compiler looks like a control-flow tree that alternates between `try` nodes, a list of sub-matchings to attempt in sequence (each of them may match the value or fail), and `switch` nodes, which decompose one of the input values.

The total number of clauses is preserved by each compilation step, and `switch` nodes result in structurally smaller patterns. Eventually at the leaves we encounter matrices of empty rows (matching on no value at all), which gets compiled to just taking the first action in the matrix:

```
match [] with
| [] -> foo
| [] -> bar
end
```

becomes just the expression `foo`.

Optimizing pattern-matching compilation

Totality Before pattern-matching compilation, the OCaml type-checker checks that the pattern-matching is well-typed, and also checks whether the clauses are exhaustive, whether they cover all possible inputs. If a pattern-matching is *not* exhaustive, a final default clause `| _ -> raise (Match_failure ...)` is implicitly added at the end.

During compilation, the pattern-matching compiler tracks whether the sub-matching it is compiling is known to be total, or whether it is partial. A sub-matching is *total* if all values that flow to this part of the generated code will be matched by some clause of the sub-matching. Otherwise it is partial, it may fail to another sub-matching in the current split, or to the final default clause. On `switch`, the totality of each decomposed sub-matching is the totality of the input sub-matching. On `try`, all sub-matchings except the last one are considered partial, and the last one is total if the input sub-matching was total.

This lets us generate better code, consider for example:

```
match [li ...] with
| [x::xs ...] -> foo
end
```

There is no `| _ -> fail` catch-all clause when we know that this sub-matching is total (for example because we are under a `switch` that also handled the empty-list case). When generating code for this matching, we do not need to check that

the head constructor is `x::xs`, we can generate direct field accesses to compute `x` and `xs` without a branch. This substantially improves the generated code.

Contexts

Within each branch of a `switch` node, we have learned some information on the scrutinee values. For example in the `| x::xs ->` case we know that the first value is of the form `_::_`. The OCaml compiler tracks this information about the global input of the pattern-matching in a "context".

For example if we started from the input vector `[o li]` and have done the following decompositions so far

```
switch p with
| false -> ...
| true ->
  switch li with
  | [] -> ...
  | x::xs -> (* we are here *)
  end
end
```

then the corresponding sub-matching on inputs `[x xs]` will carry a context relating this input vector to the original input vector of the outer matching `[p li]`, which we here know must be of the form `[true x::xs]`.

The context at any point in the generated control-flow tree is made of two part: - information about what is "above" the current scrutinees, how they relate to the scrutinees of outer sub-matchings; in this example this information can be represented as a stack of head constructors `[true (_::_)]` (growing to the right). - information about what is "below", a known prefix shape for the current scrutinees; in this example we know nothing about the scrutinees `[x xs]`, so the information is just `[_ _]`.

We can "shift" a context, changing the perspective from the current sub-matching to an outer sub-matching – while preserving the same static information. Shifting once pops the `_::_` constructor from the "context above", and we get the same context from the perspective of the parent sub-matching: above we now have `[true]`, and below we have `[_::_]`. If we shift again we have the empty list `[]` above and the two-element list `[true (_::_)]` below, representing this static context from the perspective of the outermost matching on the list `[p li]`.

Using contexts for failure optimization

In general `try` nodes represent a sequential choice between an arbitrary number of non-disjoint sub-matchings. It is useful to name them, for example with a number from `1` to `n`:

```
try
1: <sub-matching>
2: <sub-matching>
3: <sub-matching>
...
end
```

Default environments are another piece of static context information that maps the name/number of sub-matchings in a `try` sequence to its sub-matrix, seen as a description of the set of values that could possibly be succesfully matched by this sub-matching.

Consider the following example:

```
try
1: switch p with
   | true ->
     match [li ...] with
     | [x::xs ...] -> ... (* we are here *)
     | _ -> fail
     end
   end
2: match [p li ...] with
   | [_ [] ...] -> ...
   | _ -> fail
   end
3: match [p li ...] with
   | [true x::xs ...] -> ...
   | _ -> fail
   end
...
end
```

When we compile the first sub-matching, failure has the effect of jumping to a following sub-matching in the `try` sequence. Naively one would jump to the next sub-matching (here the one numbered `2:`). But we can see from the context information within the sub-matching, which is at least as precise as `[true  x::xs]` that the current values cannot match the non-failure clauses of `2:`, so we can jump right ahead at `3:` instead. The generic `fail` keyword would thus be transformed into a more precise `exit 3` instruction, jumping to a specific sub-matching in the sequence. If we are deeper down in the matching of `1:` we could have even more precise context information and jump to a later sub-matching.

Jump summaries

When we compile a sub-matching, we generate a `switch/try` tree that contains `exit <i>` instructions jumping to subsequent sub-matchings – not bound in the

specific sub-matching we are compiled, but by an outer `try` node, they are free variables. The pattern-matching compiler collects the context at the point of each such `exit i` instruction: a mapping from those free exit variables to a union of contexts describing the known shape of the current scrutinee for any jump included in the generated code. We call this mapping a "jump summary".

The jump summary of `try` node is the union of the jump summaries of each component. The jump summary of a `switch` node requires shifting all contexts in the jump summary of each sub-matching, and then taking the union.

When we compile a sub-matching that comes next in the sequence, we start from a more precise context, given by the union of the jump summaries of earlier sub-matchings. In particular, if this context is empty, we know that there is not in fact any `exit` jumping to this sub-matching, and we can remove it entirely as dead code. Even when it is non-empty, it can let us discover that certain clauses (or sub-clauses obtained by further decomposition) are useless / dead code, as none of the possible scrutinee values can reach them, and we can improve the generated code. Finally, we can sometimes discover that a switch that was assumed to be Partial is in fact Total, if its jump summaries indicates that it never jumps to an outer exit, so we can improve the generated code again.

Mutability ruins the party

When we analyzed the causes of the unsound compilation, we discovered that there are two orthogonal issues: - totality information can be incorrect - context information can be incorrect

Incorrect totality information

We mentioned that the OCaml type-checker computes totality information. The pattern-matching compiler itself does not rely on type information, and this separation was made for reasons of simplicity – typing GADTs in particular is complex, generating good code for pattern-matching is non-trivial, we don't want to interleave the two logics in the same piece of code.

In particular, the type-checker has information about the fact that certain branches would be unreachable coming from type information, that is not available to the pattern-matching compiler. This occurs with polymorphic variants and GADTs:

```
let foo (x : [< `A | `B ]) =
  match x with
  | `A -> 1
  | `B - >2

type _ gadt = Int : int gadt | Bool : bool gadt
let bar (x : int gadt) =
```

```
  match x with
  | Int -> 1
```

In both case, we need type-checking knowledge to tell that these clauses are exhaustive, and the pattern-matching compiler itself does not maintain/propagate/check such typing information, it blindly trusts the [Partial | Total] information given by the type-checker.

Unfortunately this information is sometimes wrong, because the type-checker does *not* consider the possibility of the value being mutated while it is matched. Consider our original example:

```
let f x =
  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
```

The type-checker cannot guess whether a clause `p when foo` will be taken or not, so it conservatively assumes that it never gets taken. This does not account for the possibility that the evaluation of `foo` will mutate the value being matched. The sub-matching is split in three matrices (one for clauses 1, 2, one for clause 3, one for clause 4), and the last one is compiled as Total because the type-checker determined that the whole matching is Total. When we reach a `switch x.b with Some y -> ...`, we optimize the check to a direct field access, leading to a segfault.

Mitigation – and re-optimizations

The simplest mitigation strategy is that each time we go under a *mutable* field (or a reference pattern) during the pattern-matching decomposition, we degrade the partiality information from Total to Partial.

From this description one may assume that the following example gets de-optimized:

```
type 'a ref = {mutable v: 'a}

let f : bool ref -> int = function
  | {v = true} -> 1
  | {v = false} -> 2
```

This outer matching is determined to be Total, but when going under the mutable `v` field we would pessimize to `Partial`, generating a silly `Match_failure` clause after checking both `true` and `false`.

In fact, the pattern-matching compiler knows about the number of declared constructors of algebraic types (including booleans). (This is a very restricted form of typing information.) So even if it compiles the `true | false` switch in

Partial mode it will notice that all constructors are covered and not generate a fallback clause.

On the other hand, the following example would get de-optimized:

```
type _ gadt = Int : int gadt | Bool : bool gadt
let bar (x : int gadt ref) =
  match x with
  | {v = Int} -> 1
```

We implement a "re-optimization" to avoid degrading the code in this case: in addition to the Total|Partial information coming from the context, we propagate a new information which we call "temporality information", which is either `First` or `Following`. `Following` means that we are under a `try` node in a *following* sub-matching, not its first sub-matching. `First` means that we haven't gone under a `try` at all, or only in their first sub-matching. When we are in `First` position, we are not coming from any other match, so in particular our totality information does not depend on an analysis of previous cases that could be falsified by side-effects. In this case we can preserve `Total`, even under mutable positions.

Some other innocuous examples remain degraded by this change, for example:

```
type 'a ref = { mutable v: 'a }
let foo (p : (int option ref * int option ref)) =
  match p with
  | {v = Some a}, {v = Some b} -> Some (a + b)
  | {v = None}, _ -> None
  | _, {v = None} -> None
```

In this example, the pattern-matching will split the sub-matching in a `try` node, checking the two first clauses first (their first column are disjoint) and the third clause second. Matching on the `None` pattern in the third clause will now generate an explicit check with a `Match_failure` case instead of returning `None` unconditionally.

The following variants are exactly equivalent in term of intended meaning, but they are not pessimized in this way, and thus result in better code:

```
let foo1 (p : (int option ref * int option ref)) =
  match p with
  | {v = Some a}, {v = Some b} -> Some (a + b)
  | _ -> None

let foo2 (p : (int option ref * int option ref)) =
  match p with
  | {v = Some a}, {v = Some b} -> Some (a + b)
  | {v = None}, _ -> None
  | {v = Some _}, {v = None} -> None
```

Note that there is a good reason to include a `Match_failure` clause in the first version: due to the way it is compiled, the `v` field of the second element of the pair will get read and checked several times, so it *is* possible that a concurrent thread would mutate the value in-between, making the `Match_failure` result observable. The three variants look equivalent at the source level, but they are compiled in different control-flow trees, some of which make the mutability observable, some which do not.

Incorrect context information

Consider the following variant of our running example

```
let f (x : bool * int option ref) =
  match x with
  | false, _ -> 0
  | _, {v = None} -> 1
  | _, _ when (x.b <- None; false) -> 2
  | true, {v = Some y} -> y
```

If only downgrade the partiality of the second third sub-matching (the one of `_, {v = Some y} -> y`) to partial, we still get unsound code.

This comes from the fact that the jump summary of the first sub-matching (the first two clauses) gives us precise, incorrect information about the shape of the value at this point. The first sub-matching corresponds to the following matrix:

```
[false _]
[_ {v = None}]
```

This will generate a switch on the first constructor, and it can only fail in the `true` case, if the second column fails to match `None`. In this case, we must have a `Some _` value. The pattern-matching compiler will compute this (correct!) information, and the jump summary will tell us that the second and third clauses are only reached for values matching the context `[true (Some _)]`. At this point, the compiler (incorrectly) knows that the `[true {v = Some y}]` clause will always match the input, so it does not need to check the `Some` constructor and generates a direct access to `y`.

To get a sound compiler, in addition to the downgrade of totality information on mutable fields, we need to erase mutable information when we "shift" contexts from inner to outer sub-matchings. In this example, when we compute the jump summary for the exit in the first clause, we have the above context `[true {v = _}]`, and `[None]` as the context below. When we "shift" this context up, instead of `[true]` above and `[{v = None}]` below as OCaml currently computes, we weaken the result into `[true]` above and `[{v = _}]` below. In other words, we erase any context information below mutable fields/references, which may get invalidated by a side-effect during matching.

Are we sound yet?

A natural reference semantics for pattern-matching construct is the clause-by-clause semantics, where each clause is interpreted independently as a conditional test. In other words:

```
match a with
| p1 -> e1
| p2 -> e2
...
```

should be equivalent to

```
match a with
| p1 -> e1
| _ ->
  match a with
  | p2 -> e2
  | _ ->
  match a with
  ...
```

We believe that this equivalence holds for OCaml pattern-matching when no concurrent mutation of values is intended. It certainly did not hold before our work for programs that contain concurrent mutation of the scrutinee – as crashing cannot occur in the clause-by-clause translation. We do *not* claim that the pessimization we performed suffice to restore this equivalence. We believe that the changes we made suffice to regain memory-safety and type-safety (because we are systematically more conservative in all places where direct field accesses would be generated), which was our goal. But they do not suffice to recover the equivalence. For example, consider the following program:

```
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
```

The clause-by-clause semantics would expect the following output (and a return value of 2):

```
Observed 0
Observed 1
```

but in fact the compiler (before and after our fixes) prints the following output

(and returns `2`):

```
Observed 0
```

After checking the first `incr ()` guard, the compiler wrongly assumes that the second clause can be skipped. This does not endanger type-safety, but it does not respect the natural clause-by-clause specification.

Final words

Side-effects during pattern-matching make classic compiler optimizations unsound. We describe how to weak the two key optimizations of the OCaml compiler (totality information and static contexts) to remain sound in presence of arbitrary mutation of sub-values of the scrutinee. We have implemented this fix for the OCaml compiler.

Note that the desirability of this fix is not obvious:

- In practice, people do *not* write code that mutates a scrutinee during its matching. Side-effects in when-guards are a very dubious idea, and writing a concurrent race on a complex mutable structure is equally unreasonable. We know of no instance of this compiler bug affecting code in the wild, only of tests specifically crafted to hit this issue.
- On the other hand, the mitigation we propose is reasonably simple and makes the compiler correct again, but it does degrade the compiled code slightly for some matchings that people do write today in practice, which are now slightly slower.

We believe that the change of generated code will *not* noticeably decrease the perofrmance of user programs – it will be *slightly* worse in a few places. If they notice a performance degradation in their code, they can rewrite their pattern-matching clauses to avoid splits. But it is always hard to convince people to *maybe* make their code slightly slower to fix bugs that they will probably never encounter in practice.

References

Lennart Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, 1985.