Making a matchmaker: Exhaustiveness checking in Lisette



Iván Ovejero · n8n
November 2, 2026
RustLab Italy

Do you have an idea?
Or does the idea have you?

On people, papers,
and passion projects

Submitted to ML'08

Compiling Pattern Matching to good Decision Trees

Primary sources have no mercy

Abstract

We address the issue of compiling ML pattern matching to efficient decisions trees. Traditionally, compilation to decision trees is optimized by (1) implementing decision trees as dags with maximal sharing; (2) guiding a simple compiler with heuristics. We first design new heuristics that are inspired by *necessity*, a notion from lazy pattern matching that we rephrase in terms of decision tree semantics. Thereby, we simplify previous semantical frameworks and demonstrate a direct connection between necessity and decision tree runtime efficiency. We complete our study by experiments, showing that optimized compilation to decision trees is competitive. We also suggest some heuristics precisely.

Categories and Subject Descriptors D 3. 3 [*Programming Languages*]: Language Constructs and Features—Patterns

In this paper we study compilation to decision tree, whose primary advantage is never testing a given subterm of the subject value more than once (and whose primary drawback is potential code size explosion). Our aim is to refine naive compilation to decision trees, and to compare the output of such an optimizing compiler with optimized backtracking automata.

Compilation to decision can be very sensitive to the testing order of subject value subterms. The situation can be explained by the example of an human programmer attempting to translate a ML program into a lower-level language without pattern matching. Let `f` be the following function[1] defined on triples of booleans :

```
let f x y z = match x,y,z with
| _,F,T -> 1
| F,T,_ -> 2
| _,_,F -> 3
```

```rust
enum Severity { Low, High, Critical }

fn label(s: Option<Severity>) -> &'static str {
    match s {
        Some(Severity::Low) => "low",
        None => "none",
    }
}
```

```
error[E0004]: non-exhaustive patterns: `Some(Severity::High)` and
  `Some(Severity::Critical)` not covered

help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern, a match arm with
  multiple or-patterns as shown, or multiple match arms
    |
  6 ~         None => "none",
  7 ~         Some(Severity::High) | Some(Severity::Critical) =>
    todo!(),
    |
```

```go
type Severity int

const (
    Low Severity = iota
    High
    Critical
)

func shouldAlert(s Severity) bool {
    switch s {
    case Low:
        return false
    case High:
        return true
    }
    return false
}

func main() {
    fmt.Println(shouldAlert(Critical)) // false
}
```

CAMBRIDGE UNIVERSITY PRESS

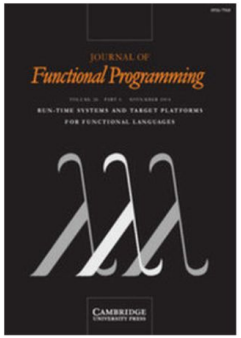
Discover Content

Produc

Cambridge Core | Browse Services Open research

Home > Journals > Journal of Functional Programming > Volume 17 Issue 3 > Warnings for pattern matching

Match expressions



Exhaustiveness check + Usefulness check

Warnings for pattern matching

Published online by Cambridge University Press: **15 February 2007**

LUC MARANGET

Show author details

Article Discussions Metrics

Save PDF Share Cite Rights & Permissions

Abstract

Core share and HTML view are not available for this content. However, as you have access to this content, a full PDF is available via the 'Save PDF' action button.

We examine the ML pattern-matching anomalies of useless clauses and non-exhaustive matches. We state the definition of these anomalies, building upon pattern matching semantics, and propose a simple algorithm to detect them. We have integrated the algorithm in the Objective Caml compiler, but we show that the same algorithm is also usable in a non-strict language such as Haskell. Or-patterns are considered for both strict and non-strict languages.

Files

main

Go to file T

.github
LICENSES
compiler
rustc
rustc_abi
rustc_arena
rustc_ast
rustc_ast_ir
rustc_ast_lowering
rustc_ast_passes
rustc_ast_pretty
rustc_attr_ir
rustc_attr_parsing
rustc_baked_icu_data
rustc_borrowck
rustc_builtin_macros
rustc_codegen_cranelift

ChrisDenton Remove ensure_sufficient_stack 4641990 · last month History

Code Blame 1882 lines (1789 loc) · 75.5 KB Raw

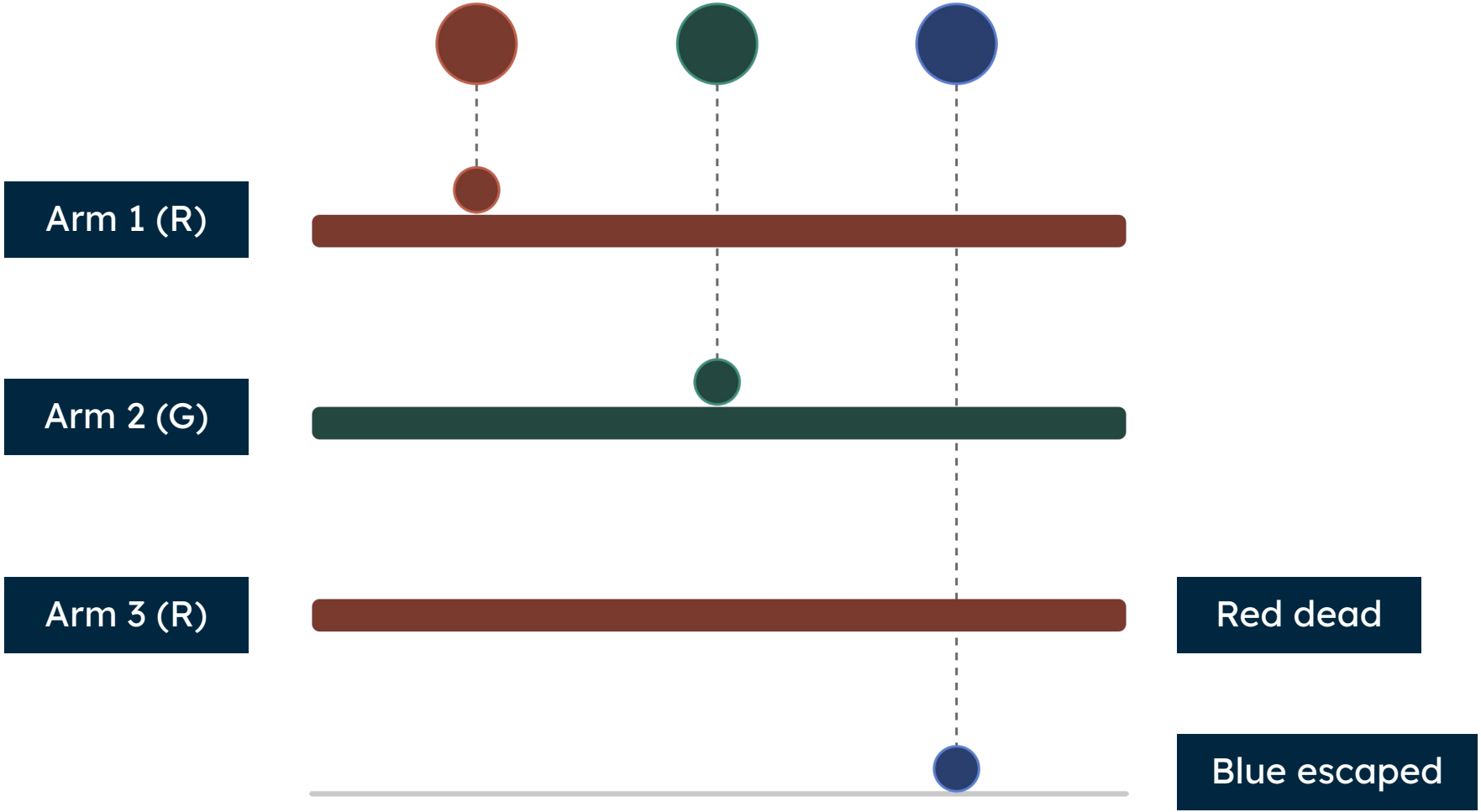
```
1    //! # Match exhaustiveness and redundancy algorithm
2    //!
3    //! This file contains the logic for exhaustiveness and usefulness checking for pattern-matching.
4    //! Specifically, given a list of patterns in a match, we can tell whether:
5    //! (a) a given pattern is redundant
6    //! (b) the patterns cover every possible value for the type (exhaustiveness)
7    //!
8    //! The algorithm implemented here is inspired from the one described in [this
9    //! paper](http://moscova.inria.fr/~maranget/papers/warn/index.html). We have however changed it in
10   //! various ways to accommodate the variety of patterns that Rust supports. We thus explain our
11   //! version here, without being as precise.
12   //!
13   //! Fun fact: computing exhaustiveness is NP-complete, because we can encode a SAT problem as an
14   //! exhaustiveness problem. See [here](https://niedzejkob.p4.team/rust-np) for the fun details.
15   //!
16   //!
17   //! # Summary
18   //!
19   //! The algorithm is given as input a list of patterns, one for each arm of a match, and computes
20   //! the following:
21   //! - a set of values that match none of the patterns (if any),
22   //! - for each subpattern (taking into account or-patterns), whether removing it would change
23   //!     anything about how the match executes, i.e. whether it is useful/not redundant.
24   //!
25   //! To a first approximation, the algorithm works by exploring all possible values for the type
```

Conceptual
starting point:
A match as a manifold?



```rust
enum Color { Red, Green, Blue }
use Color::*;

fn describe(c: Color) -> &'static str {
    match c {
        Red   => "stop",
        Green => "go",
        Red   => "halt",
    }
}
```


Arm 1 (R)
Arm 2 (G)
Arm 3 (R)
Red dead
Blue escaped

```
error[E0004]: non-exhaustive patterns: `Color::Blue` not covered
  |
5 |     match c {
  |           ^ pattern `Color::Blue` not covered
  |
help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern or an explicit pattern as
  shown
  |
8 ~         Red   => "halt",
9 ~         Color::Blue => todo!(),
  |
```

```
warning: unreachable pattern
  |
6 |         Red   => "stop",
  |         --- matches all the relevant values
7 |         Green => "go",
8 |         Red   => "halt",
  |         ^^^ no value can reach this
  |
```

```rust
enum Color { Red, Green, Blue }
use Color::*;

fn describe(c: Color) -> &'static str {
    match c {
        Red   => "stop",
        Green => "go",
        Red   => "halt",
        _     => todo!(), // does anything ever reach this?
    }
}
```

```rust
fn name(n: u32) -> &'static str {
    match n {
        0 => "zero",
        7 => "seven",
        _ => "other", // stranger
    }
}
```

u32:
4,294,967,296
values

shortlist

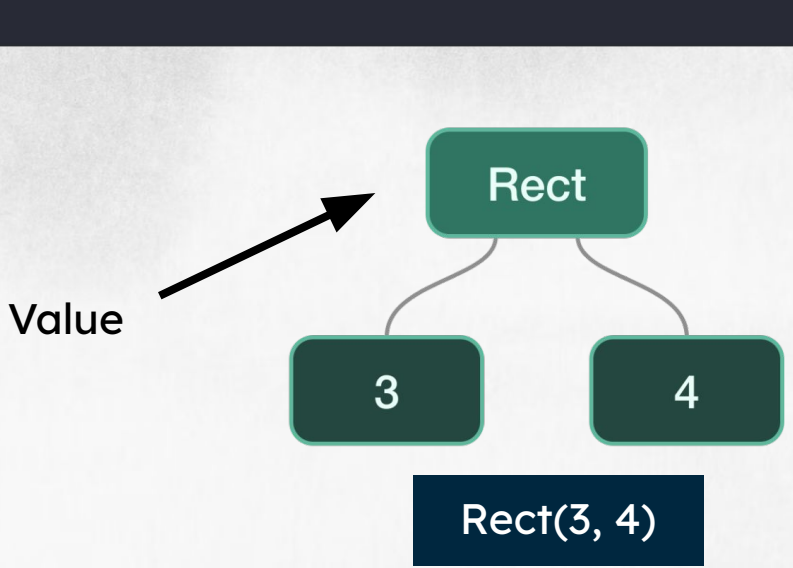


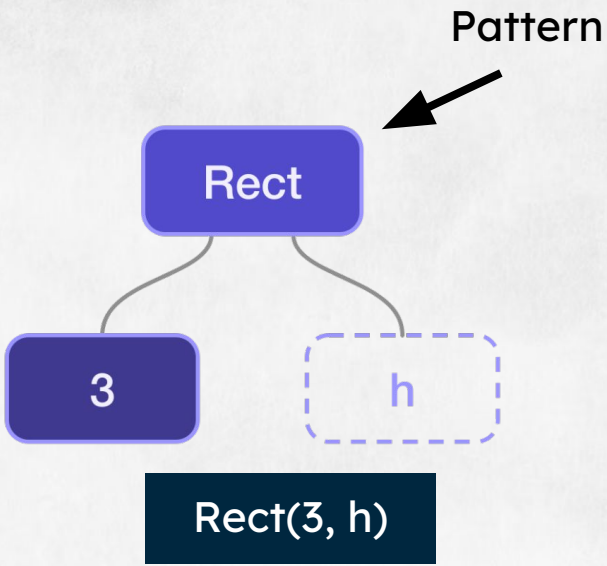
stranger

```
error[E0004]: non-exhaustive patterns: `1_u32..=6_u32` and
  `8_u32..=u32::MAX` not covered
  |
2 |     match n {
  |           ^ patterns `1_u32..=6_u32` and `8_u32..=u32::MAX`
    not covered
  |
help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern, a match arm with
  multiple or-patterns as shown, or multiple match arms
  |
4 ~         7 => "seven",
5 ~         1_u32..=6_u32 | 8_u32..=u32::MAX => todo!(),
  |
```

```rust
enum Shape { Circle(u32), Rect(u32, u32) }
use Shape::*;

fn height_of_width_3(s: Shape) -> Option<u32> {
    match s {
        Rect(3, h) => Some(h),
        _ => None,
    }
}
```





Re-representation: A match as a matrix?

Example 2

Let P and Q be the following two pattern matrices.

$$P = \begin{pmatrix} \texttt{Nil} & _ \\ _ & \texttt{Nil} \end{pmatrix} \qquad Q = \begin{pmatrix} \texttt{Nil} & _ \\ _ & \texttt{Nil} \\ \texttt{One}(_) & _ \\ _ & \texttt{One}(_) \\ \texttt{Cons}\ (_,_) & _ \\ _ & \texttt{Cons}(_,_) \end{pmatrix}$$

Matrix P is not exhaustive, since, for instance, vector $\vec{v} = (\texttt{One}\ \ (0)\ \texttt{One}\ \ (0))$ does not match any row of P.

By contrast, matrix Q is exhaustive. Let us consider any value vector $\vec{v}$ of the appropriate type. Then, v_1 and v_2 are instances of the patterns `Nil`, `One(_)`, or `Cons(_,_)`. That is, we may partition values into nine sets denoted by nine different pattern vectors. It turns out that this partition is precise enough to apply Definition 2.

If $\vec{v}$ is an instance of...	then, $\vec{v}$ matches row number...
(Nil Nil) (Nil One(_)) (Nil Cons(_,_))	1
(One(_) Nil) (Cons(_,_) Nil)	2
(One(_) One(_)) (One(_) Cons(_,_))	3
(Cons(_,_) One(_))	4
(Cons(_,_) Cons(_,_))	5

```rust
enum Color { Red, Green, Blue }
use Color::*;

fn describe(c: Color) -> &'static str {
    match c {
        Red => "stop",
        Green => "go",
        Blue => "wait",
    }
}
```

```rust
enum Color { Red, Green, Blue }
use Color::*;

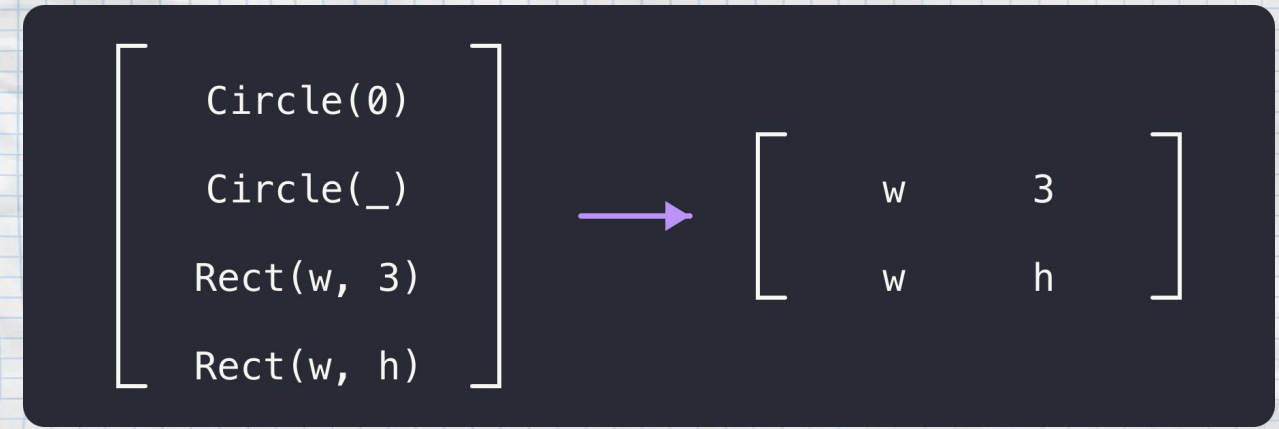
fn describe(c: Color, urgent: bool) -> &'static str {
    match (c, urgent) {
        (Red, true) => "stop now",
        (Red, false) => "stop",
        (_, _) => "go",
    }
}
```

$$P = \begin{bmatrix} \text{Red} \\ \text{Green} \\ \text{Blue} \end{bmatrix}$$

$$P = \begin{bmatrix} \text{Red} & \text{true} \\ \text{Red} & \text{false} \\ _ & _ \end{bmatrix}$$

```rust
enum Shape { Circle(u32), Rect(u32, u32) }
use Shape::*;

fn area(s: Shape) -> u32 {
    match s {
        Circle(0) => 0,
        Circle(_) => 1,
        Rect(w, 3) => w * 3,
        Rect(w, h) => w * h,
    }
}
```



```
match s {
    Circle(0) ⇒ 0,
    Circle(_) ⇒ 1,
    Rect(w, 3) ⇒ w * 3,
    Rect(w, h) ⇒ w * h,
}
```

1

```
Circle(0)
Circle(_)
Rect(w, 3)
Rect(w, h)
```

2

~~Circle(0)~~

~~Circle(_)~~

Rect(w, 3)

Rect(w, h)

```
match s {
    Circle(0) ⇒ 0,
    Circle(_) ⇒ 1,
    Rect(w, 3) ⇒ w * 3,
    Rect(w, h) ⇒ w * h,
}
```

3

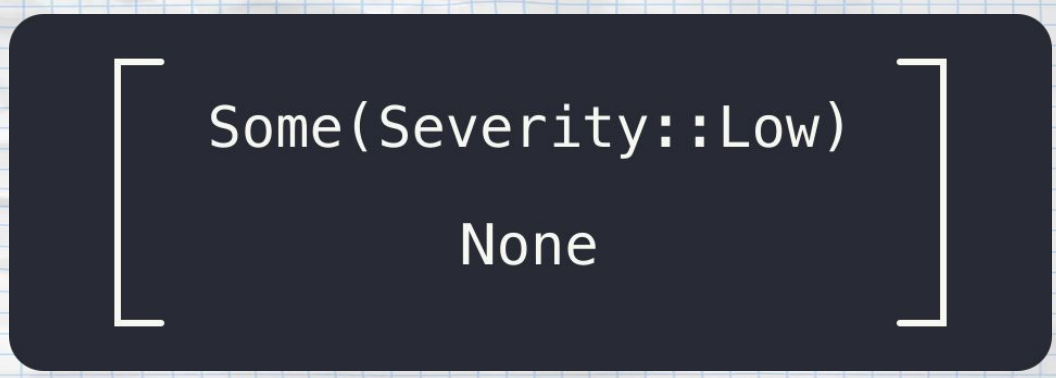
$$\begin{bmatrix} w & 3 \\ w & h \end{bmatrix}$$

4

$$\begin{bmatrix} w & 3 \\ w & h \end{bmatrix}$$

```
enum Severity { Low, High, Critical }

fn label(s: Option<Severity>) → &'static str {
    match s {
        Some(Severity::Low) ⇒ "low",
        None ⇒ "none",
    }
}
```



```rust
match s {
    Some(Severity::Low) => "low",
    None => "none",
}
```

1

~~Some(Severity::Low)~~

None

2

✓

3

Some(Severity::Low)

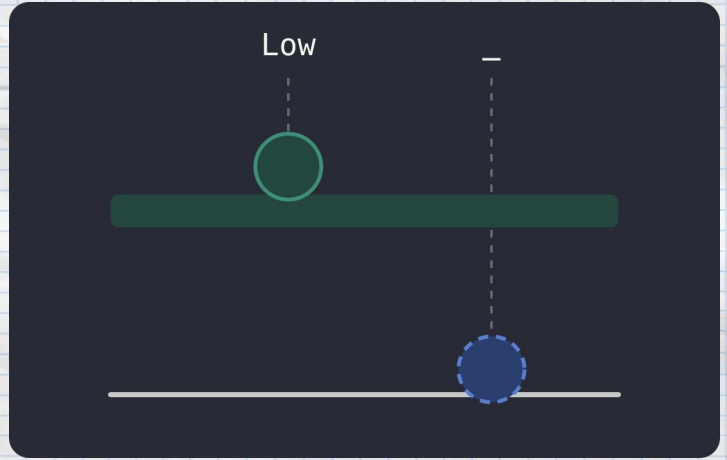
~~None~~

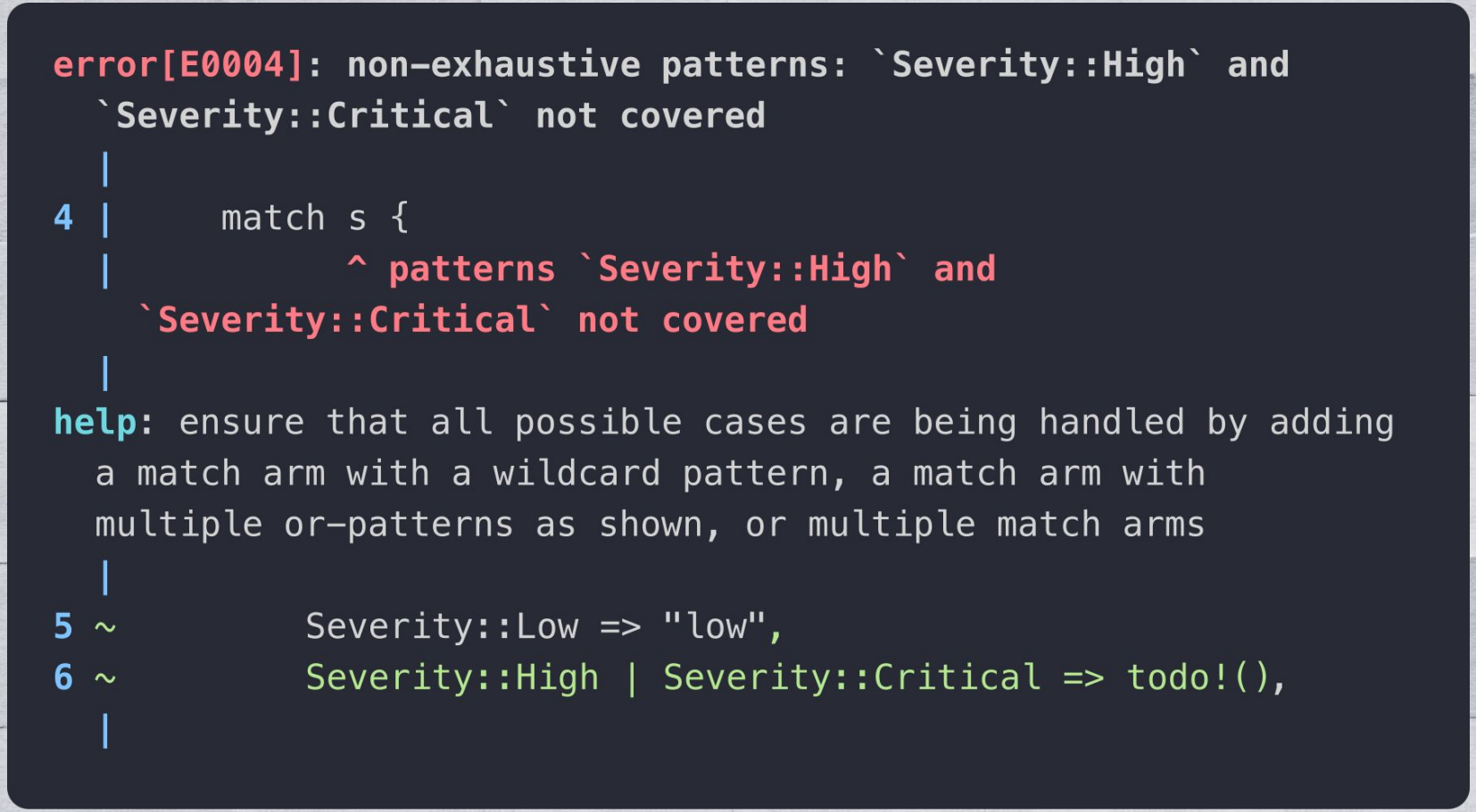
4

Severity::Low

```rust
enum Severity { Low, High, Critical }

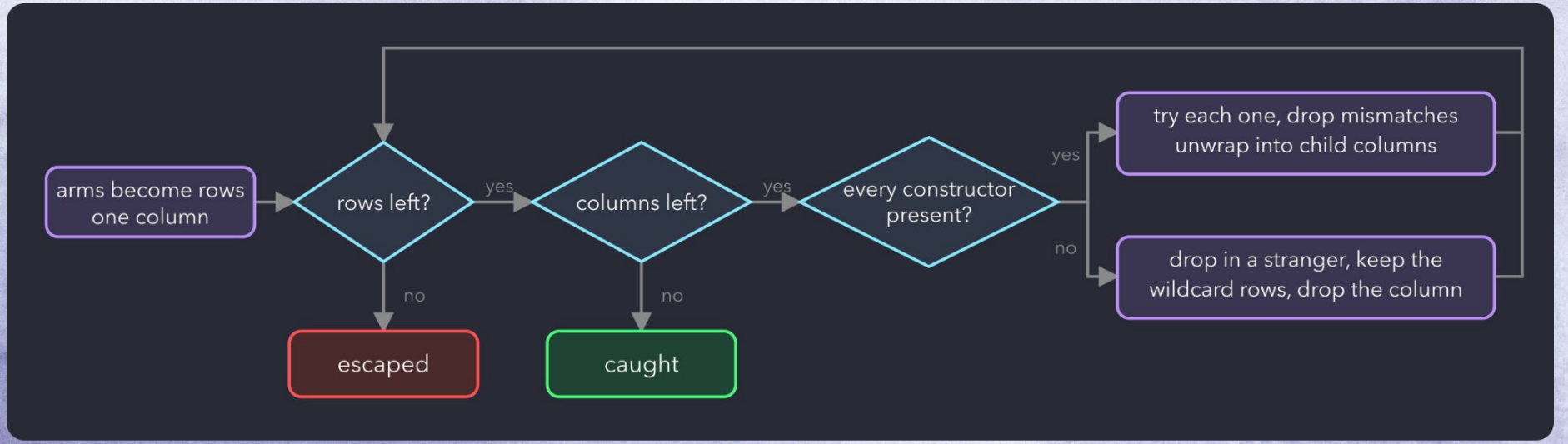
fn label_inner(s: Severity) -> &'static str {
    match s {
        Severity::Low => "low",
    }
}
```




error[E0004]: non-exhaustive patterns: `Severity::High` and
`Severity::Critical` not covered
|
4 | match s {
| ^ patterns `Severity::High` and
`Severity::Critical` not covered
|
help: ensure that all possible cases are being handled by adding
a match arm with a wildcard pattern, a match arm with
multiple or-patterns as shown, or multiple match arms
|
5 ~ Severity::Low => "low",
6 ~ Severity::High | Severity::Critical => todo!(),
|

```rust
enum Severity { Low, High, Critical }

fn label(s: Option<Severity>) -> &'static str {
    match s {
        Some(Severity::Low) => "low",
        None => "none",
    }
}
```



Housekeeping: Naming the witnesses

PART TWO

Implementation

5 Specializing $\mathscr{U}$ for exhaustiveness check

Programmers sometimes are quite upset in front of "non-exhaustive match" warnings. An example of a "non-matching value" helps a lot not only in convincing them that they indeed wrote a non-exhaustive match, but also in correcting their code.

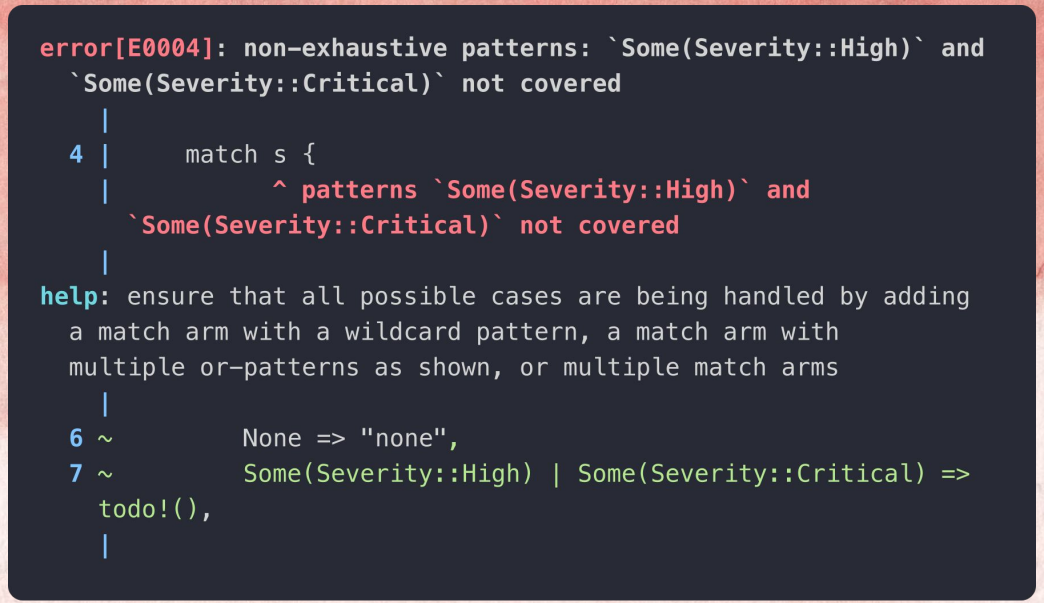
Such an example (or counter-example) is best expressed as a pattern representing a set of non-matching instances. Then, programmers can add this "counter-example" at the end of their matching, hoping this will make it exhaustive. Consider an easy example.

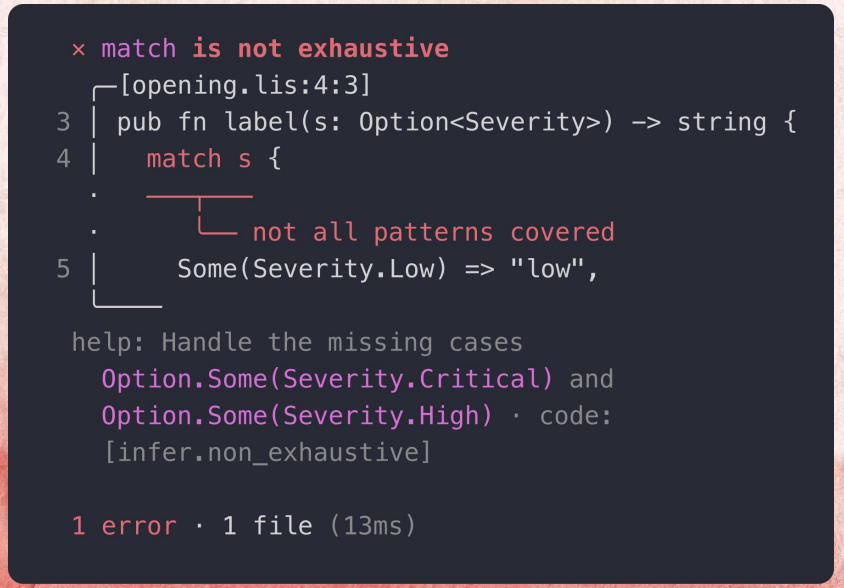
```
let nilp = function [] -> true
Warning: this pattern-matching is not exhaustive.
Here is an example of a non-matching value:
_::_
```

The given pattern matching is not exhaustive and all instances of the pattern `_::_` (a list cell) are non-matching. Here, one achieves exhaustive match by adding a clause with pattern `_::_`.

Examples of non-matching values are easily computed by a slight extension of algorithm $\mathscr{U}$. Indeed, algorithm $\mathscr{U}$ shows that $\mathscr{M}(P, \vec{q})$ is not empty by implicitly computing a witness of that fact.

The new algorithm $\mathscr{I}$ takes a matrix P and an integer n as arguments, since $\mathscr{I}$ is used in a context where the pattern vector $\vec{q}$ of Section 3.1 is a vector of n wildcards. Algorithm $\mathscr{I}$ normally returns a pattern vector $\vec{p}$ of size n such that all the instances


error[E0004]: non-exhaustive patterns: `Some(Severity::High)` and `Some(Severity::Critical)` not covered
4 | match s {
^ patterns `Some(Severity::High)` and `Some(Severity::Critical)` not covered
help: ensure that all possible cases are being handled by adding a match arm with a wildcard pattern, a match arm with multiple or-patterns as shown, or multiple match arms
6 ~ None => "none",
7 ~ Some(Severity::High) | Some(Severity::Critical) => todo!(),


× match is not exhaustive
[opening.lis:4:3]
3 | pub fn label(s: Option<Severity>) -> string {
4 | match s {
not all patterns covered
5 | Some(Severity.Low) => "low",
help: Handle the missing cases
Option.Some(Severity.Critical) and
Option.Some(Severity.High) · code:
[infer.non_exhaustive]
1 error · 1 file (13ms)

```rust
enum Severity { Low, High, Critical }
use Severity::*;

fn code(s: Severity, urgent: bool) -> u8 {
    match (s, urgent) {
        (Low, _) => 1,
        (High, _) => 2,
        (Critical, _) => 3,
        (_, true) => 4,
    }
}
```

```
warning: unreachable pattern
  |
9 |         (_, true) => 4,
  |         ^^^^^^^^^ no value can reach this
  |
note: multiple earlier patterns match some of the same values
  |
6 |         (Low, _) => 1,
  |         -------- matches some of the same values
7 |         (High, _) => 2,
  |         --------- matches some of the same values
8 |         (Critical, _) => 3,
  |         ------------- matches some of the same values
9 |         (_, true) => 4,
  |         ^^^^^^^^^ collectively making this unreachable
```



Complications?

- Or-patterns
- Match guards
- Slice patterns
- Uninhabited types

```rust
fn score(x: Option<u32>) -> u8 {
    match x {
        Some(_) | Some(0) => 1,
        None => 2,
    }
}
```

```
warning: unreachable pattern
   |
3  |         Some(_) | Some(0) => 1,
   |         -------   ^^^^^^^ no value can reach this
   |         |
   |         matches all the relevant values
   |
```

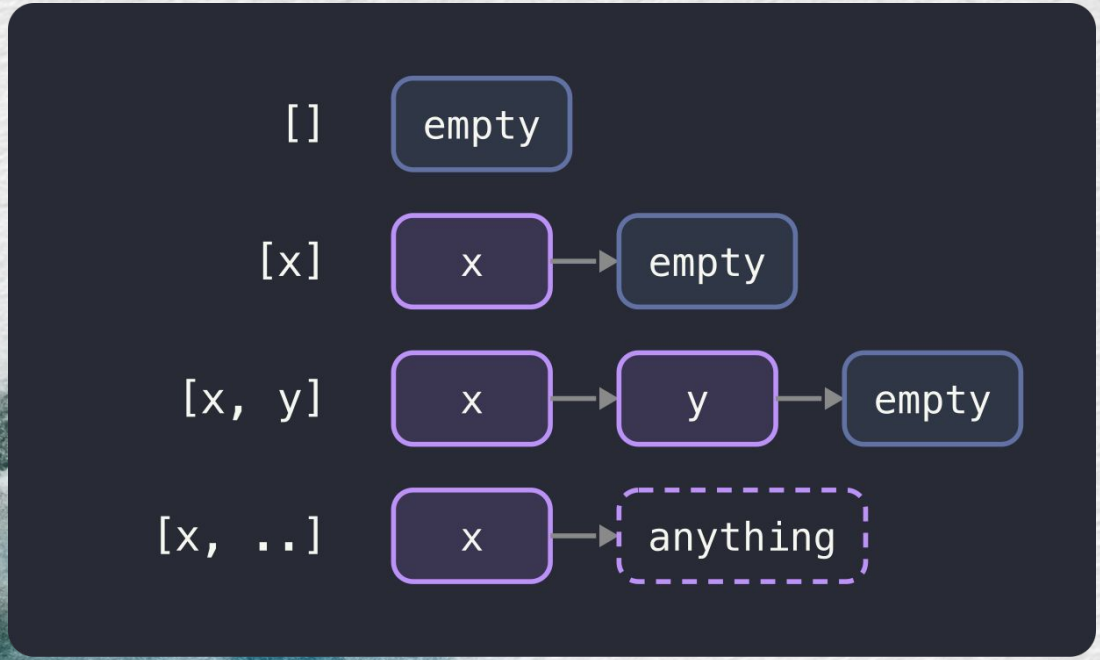
```
fn size(n: u32) → &'static str {
    match n {
        x if x > 5 ⇒ "big",
        x if x ⩽ 5 ⇒ "small",
    }
}
```

```
error[E0004]: non-exhaustive patterns: `0_u32..=u32::MAX` not
  covered
  |
2 |     match n {
  |           ^ pattern `0_u32..=u32::MAX` not covered
  |
  = note: match arms with guards don't count towards exhaustivity
help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern or an explicit pattern as
  shown
  |
4 ~         x if x <= 5 => "small",
5 ~         0_u32..=u32::MAX => todo!(),
  |
```

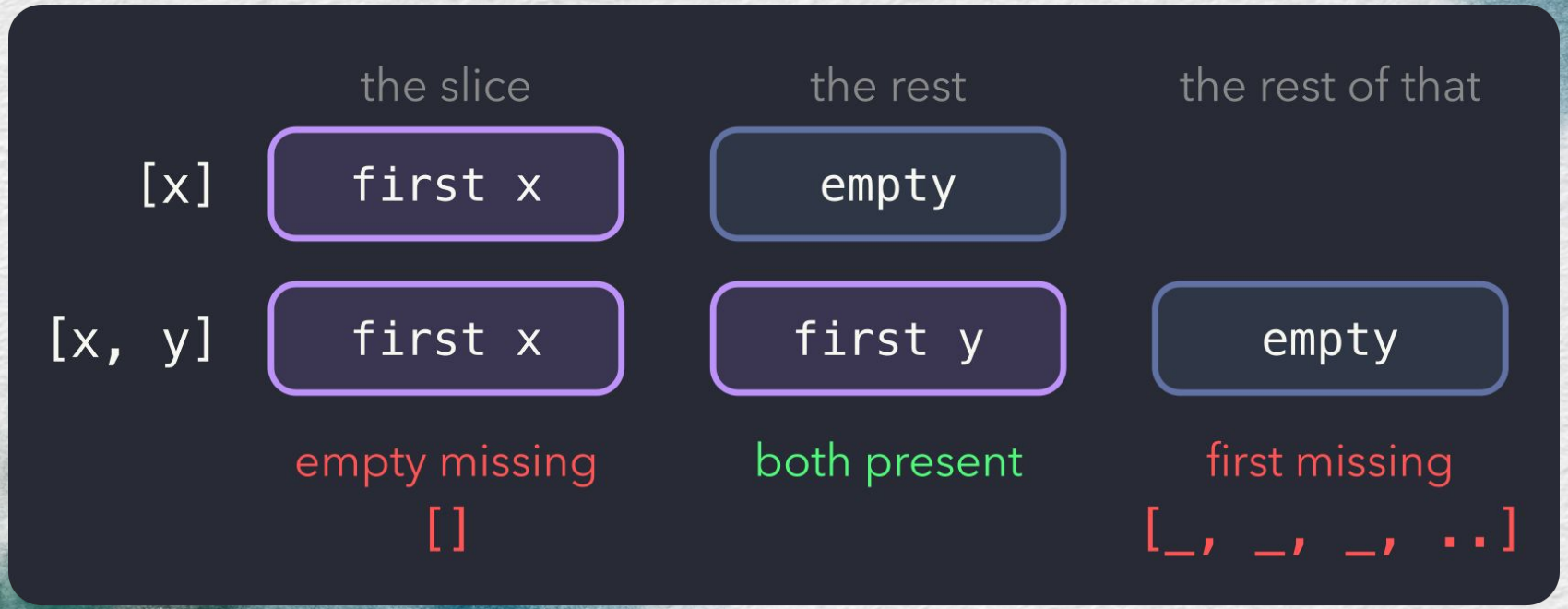
```rust
fn describe(items: &[i32]) -> String {
    match items {
        [x] => format!("one: {x}"),
        [x, y] => format!("two: {x}, {y}"),
    }
}
```

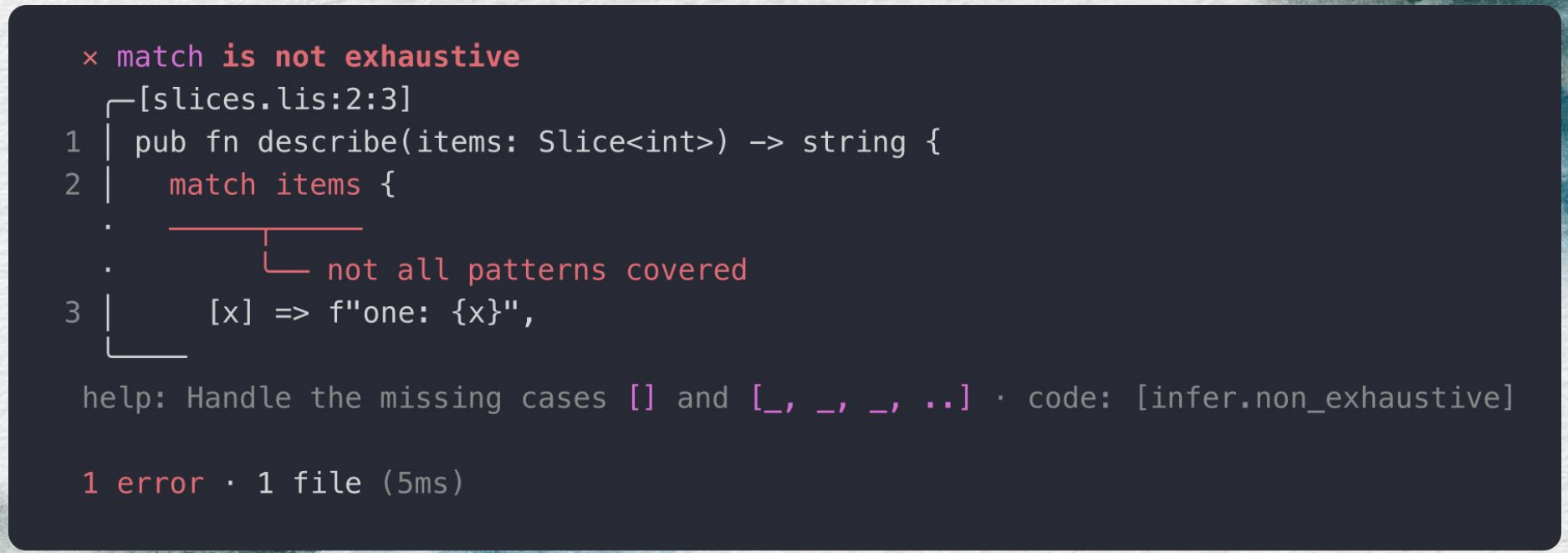
```
error[E0004]: non-exhaustive patterns: `&[]` and `&[_, _, _, ..]`
  not covered
   |
2  |     match items {
   |           ^^^^^ patterns `&[]` and `&[_, _, _, ..]` not
     covered
   |
help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern, a match arm with
  multiple or-patterns as shown, or multiple match arms
   |
4  ~         [x, y] => format!("two: {x}, {y}"),
5  ~         &[] | &[_, _, _, ..] => todo!(), // non-empty(x, y)
   |
```

```
match items {
    [x] ⇒ format!("one: {x}"),
    [x, y] ⇒ format!("two: {x}, {y}"),
}
```



```rust
match items {
    [x] ⇒ format!("one: {x}"),
    [x, y] ⇒ format!("two: {x}, {y}"),
}
```

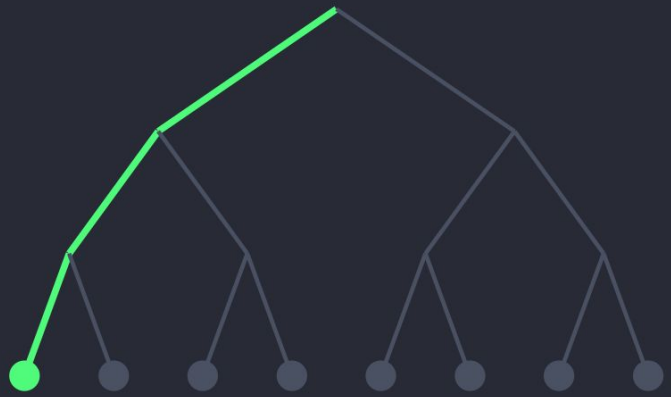



× match is not exhaustive
╭─[slices.lis:2:3]
1 │ pub fn describe(items: Slice<int>) -> string {
2 │   match items {
. ──────┬──────
. └── not all patterns covered
3 │   [x] => f"one: {x}",
╰────
help: Handle the missing cases [] and [_, _, _, ..] · code: [infer.non_exhaustive]
1 error · 1 file (5ms)

```rust
use std::convert::Infallible;

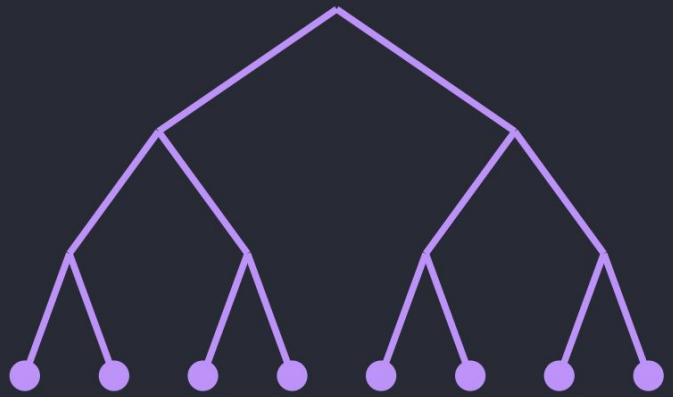
fn unwrap_ok(r: Result<u32, Infallible>) -> u32 {
    match r {
        Ok(n) => n,
    }
}
```

(true, true, true)



one branch of eight

(_, _, _)



all eight

3 columns → 8 branches (2^3)
20 columns → ~1m branches (2^20)

Satisfiability puzzle

a or b

not b or not c

not a or c

Values of a and b so that all rules are true?



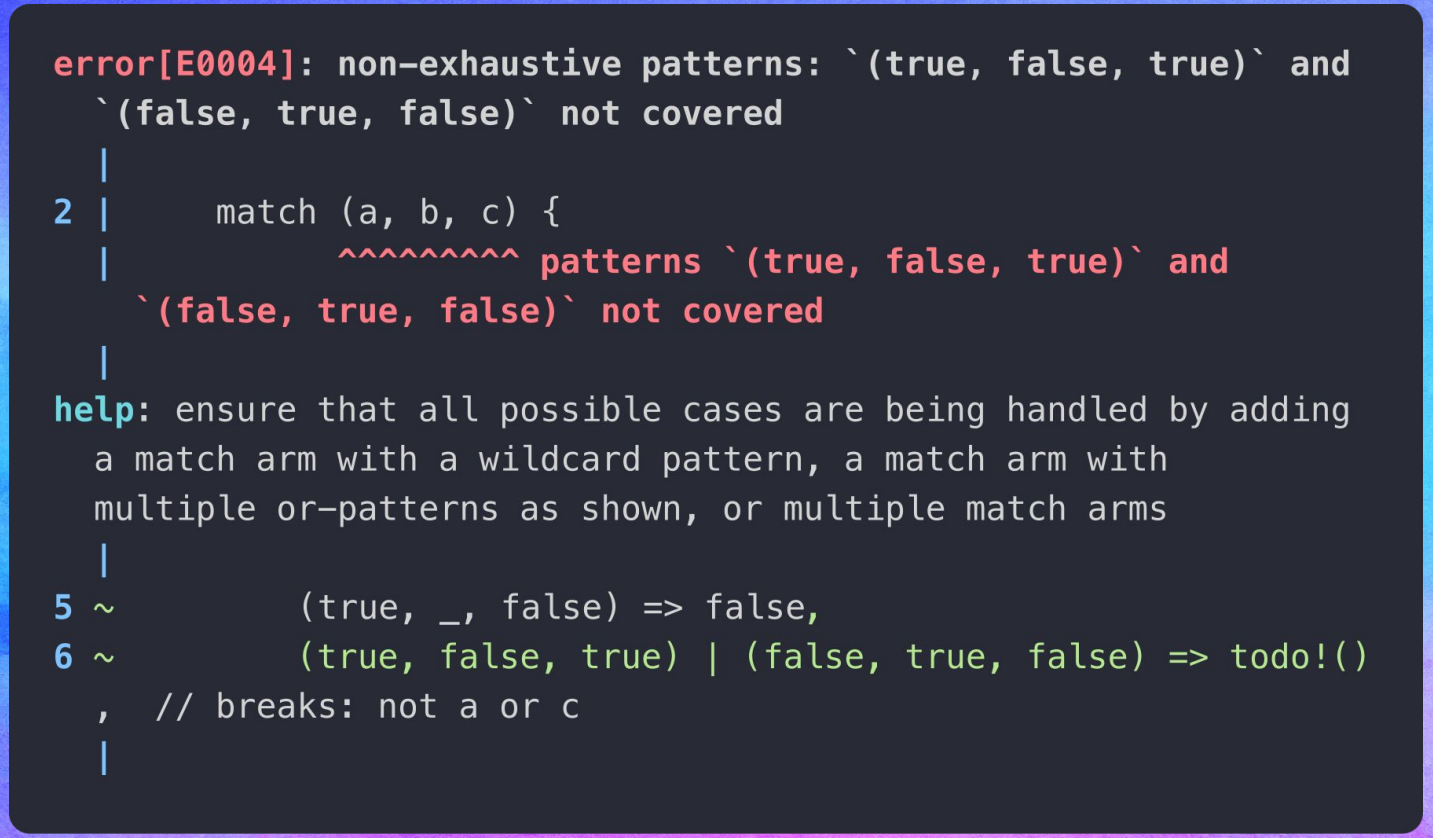
Match as a SAT problem?

a or b

not b or not c

not a or c

```
fn puzzle(a: bool, b: bool, c: bool) → bool {
    match (a, b, c) {
        (false, false, _) ⇒ false, // breaks: a or b
        (_, true, true) ⇒ false,   // breaks: not b or not c
        (true, _, false) ⇒ false,  // breaks: not a or c
    }
}
```


error[E0004]: non-exhaustive patterns: `(true, false, true)` and `(false, true, false)` not covered
|
2 | match (a, b, c) {
| ^^^^^^^^^ patterns `(true, false, true)` and `(false, true, false)` not covered
|
help: ensure that all possible cases are being handled by adding a match arm with a wildcard pattern, a match arm with multiple or-patterns as shown, or multiple match arms
|
5 ~ (true, _, false) => false,
6 ~ (true, false, true) | (false, true, false) => todo!()
, // breaks: not a or c
|

```
fn t(v: (bool, bool, bool, bool, ...)) → u8 {
    match v {
        (true,  true,  true,  true,  ...) ⇒ 0,
        (false, false, false, false, ...) ⇒ 0,
        (_,     true,  true,  true,  ...) ⇒ 1,
        (_,     false, false, false, ...) ⇒ 1,
        (_,     _,     true,  true,  ...) ⇒ 2,
        (_,     _,     false, false, ...) ⇒ 2,
        ...
    }
}
```

columns	arms	Lisette	rustc
18	36	0.4 s	0.3 s
22	44	6.0 s	3.4 s
26	52	101 s	55 s

Apple M1 · rustc 1.95.0 · lis 0.12.2

Match Expressions: 1990s-present

```
error[E0004]: non-exhaustive patterns: `Some(Severity::High)` and
  `Some(Severity::Critical)` not covered

help: ensure that all possible cases are being handled by adding
  a match arm with a wildcard pattern, a match arm with
  multiple or-patterns as shown, or multiple match arms
    |
  6 ~         None => "none",
  7 ~         Some(Severity::High) | Some(Severity::Critical) =>
    todo!(),
    |
```

Warnings for pattern matching

LUC MARANGET

Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
(*e-mail:* Luc.Maranget@inria.fr)

Abstract

We examine the ML pattern-matching anomalies of useless clauses and non-exhaustive
lies, building upon pattern matching semantics,
em. We have integrated the algorithm in the
e same algorithm is also usable in a non-strict
idered for both strict and non-strict languages.

duction

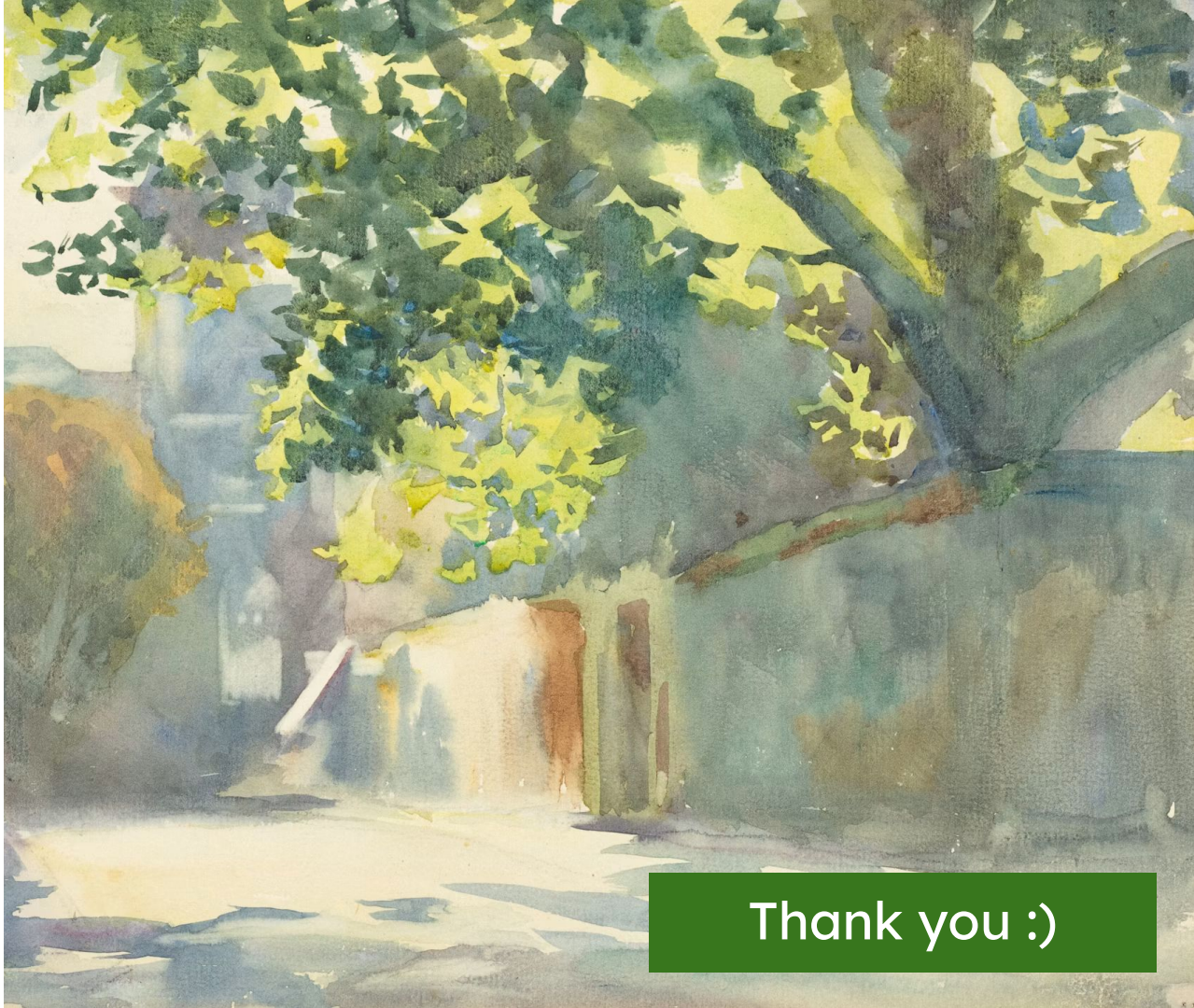
tures of the ML family of programing
ning (and programming) on a case by case
o basic checks: Are all cases considered?
rs? In pattern matching terms, one wishes
xpressions whose clauses all are useful.
n matching expressions that do not comply
with those two basic assumptions or, in other words, expressions that exhibit such
anomalies. By doing so, compilers provide an important help to programmers in

```
pub enum Shape {
  Circle(int),
  Square(int),
  Triangle(int, int),
}

pub fn area(shape: Shape) -> int {
  match shape {
    Circle(r) => 3 * r * r,
    Square(s) => s * s,
    Triangle(b, h) => b * h / 2,
  }
}
```

```
func Area(shape Shape) int {
    switch shape.Tag {
    case ShapeCircle:
        r := shape.Circle
        return 3 * r * r
    case ShapeSquare:
        s := shape.Square
        return s * s
    default:
        return shape.Triangle0 * shape.Triangle1 / 2
    }
}
```

Exhaustiveness checking, and relearning how to learn



Thank you :)