Pattern Matching

Programmieren in Rust

F. Klopfer

07. Juni 2019

Gliederung

Patterns

- May introduce shadowing
- irrefutable: always matches the value it's matched against
  ⇒ only contains identifiers or wildcards _
  ```
  let (x, y) = (1, 2);
  ```
- refutable: possibly not matching the value it's matched against
  ⇒ contains literals
  ```
  if let (a, 2) = (1, 2) {...}
  else if let Some(x) = 12 {...}
  ```
- combine them with a pipe |
  ```
  if let Msg::Move | Msg::Stop = message => {...}
  ```

Where they can be used: Irrefutable

`let`-statements:

```
let (x, y, z) = (1, 2, 3);
```

`for`-loops:

```
for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

`fn` parameters:

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}
```

Closures:

```
let closure_inferred  = | (x, y) |    x * y  ;
```

Where they can be used: Refutable

`match`-Arms:

```
match x {
    1 => println!("one"),
    _ => println!("something else"),
}
```

`if let`-expressions:

```
if let Some(color) = favorite_color {
    println!("Using your favorite color, {}", color);
}
```

`while let`-loops:

```
while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

Overview

Syntax

Pattern :
LiteralPattern
| *IdentifierPattern*
| *WildcardPattern*
| *RangePattern*
| *ReferencePattern*
| *StructPattern*
| *TupleStructPattern*
| *TuplePattern*
| *GroupedPattern*
| *SlicePattern*
| *PathPattern*
| *MacroInvocation*

Won't go into details on macro invocation
(Would be another presentation)

Literal & Wildcard Patterns

Syntax

LiteralPattern :
BOOLEAN_LITERAL
| CHAR_LITERAL
| BYTE_LITERAL
| STRING_LITERAL
| RAW_STRING_LITERAL
| BYTE_STRING_LITERAL
| RAW_BYTE_STRING_LITERAL
| -? INTEGER_LITERAL
| -? FLOAT_LITERAL

- match same value (literals) or everything (wildcards)

- _ for single, .. for multiple ignored literals
- always refutable
- Floating-point literals are going to be forbidden in a future version

```
match i {
    -1 | 1 => {...},
    _ => {...}),
}
```

Range Patterns

Syntax

RangePattern :
RangePatternBound `..=` *RangePatternBound*
| *RangePatternBound* `...` *RangePatternBound*

RangePatternBound :
CHAR_LITERAL
| BYTE_LITERAL
| `-`? INTEGER_LITERAL
| `-`? FLOAT_LITERAL
| *PathInExpression*
| *QualifiedPathInExpression*

- work for integers, characters & floating-points (deprecated)
- `a..=b` means from a to (inclusive) b with $a \leq b$
- irrefutable when spanning the whole type domain, else refutable

```
println!("{}", match ph {
    0..=6 => "acid",
    7 => "neutral",
    8..=14 => "base",
    _ => unreachable!(),
});
```

Grouped & Slice Patterns I

Syntax

GroupedPattern :
`(` *Pattern* `)`

- used to control operator precedence where'd be ambigous

```
let int_reference = &3;
match int_reference {
    &(0..=5) => (),
    _ => (),
}
```

Syntax

SlicePattern :
`[` *Pattern* (`,` *Pattern*)* `,`? `]`

- fixed size arrays or dynamic collections
- subslicing of slices to be stabilized e.g. `[a, ..]` will not work

Grouped & Slice Patterns II

```
// Fixed size
let arr = [1, 2, 3];
match arr {
    [1, _, _] => "starts with one",
    [a, b, c] => "starts with something else",
};
```

```
// Dynamic size
let v = vec![1, 2, 3];
match v[..] {
    [a, b] => { /* length doesn't match */ }
    [a, b, c] => { /* will apply */ }
    _ => { /* wildcard required, since length not known*/ }
};
```

Identifier Patterns I

Syntax

IdentifierPattern :
`ref`? `mut`? IDENTIFIER (`@` *Pattern*)?

- identifier patterns bind value they match to a variable
  `let mut variable = 10;`
- @ syntax binds what matched a pattern to an identifier

```
let x = 2;
match x {
    e @ 1 ..= 5 => println!("{}", e), // 2
    _ => println!("anything else"),
}
```

Identifier Patterns II

- identifier patterns bind by default to copy or move depending on presence of `Copy`-Trait
- use `ref` and `mut ref` to bind identifier to reference to the value's memory location

```
match a {
    None => (),
    Some(value) => (),
} // copy or move

match a {
    None => (),
    Some(ref value) => (),
} // reference
```

Identifier Patterns III

- require `ref` due to destructing patterns not allowing `&` to be

```
if let Person{name: &person_name, age: 18..=150} = value { }
```

```
error[E0308]: mismatched types
  |
9 | if let Person{name: &person_name, age: 18..=150} = value { }
  |                     ^^^^^^^^^^^^
  |       expected struct `std::string::String`, found reference
  |
  = note: expected type `std::string::String`
             found type `&_`
```

Solution:

```
if let Person{name: ref person_name, age: 18..=150} = value { }
```

Identifier patterns: Binding modes

Automatically convert non-references to `mut ref` or `ref`

```
let x: &Option<i32> = &Some(3);
if let Some(y) = x {
    // y was converted to `ref y` and its type is &i32
}
```

- Default binding mode: move semantics
- on match of reference and non-reference patterns; deref and update binding mode:
  - move
  - ref or ref mut
  - if ref was reached it stays in ref.

Reference Patterns

Syntax

ReferencePattern :

(`&` | `&&`) `mut`? *Pattern*

- deref pointer that is beeing matched
  ⇒ Borrow them
- always refutable

```
let int_reference = &3;
let a = match *int_reference { 0 => "zero", _ => "some" };
let b = match int_reference { &0 => "zero", _ => "some" };
assert_eq!(a, b);
```

Struct Patterns

Syntax

StructPattern :
PathInExpression `{`
StructPatternElements?
`}`

StructPatternElements :
StructPatternFields (`,` | `,` *StructPatternEtCetera*)?
| *StructPatternEtCetera*

StructPatternFields :
StructPatternField (`,` *StructPatternField*)*

StructPatternField :
*OuterAttribute**
(
TUPLE_INDEX `:` *Pattern*
| IDENTIFIER `:` *Pattern*
| `ref`? `mut`? IDENTIFIER
)

StructPatternEtCetera :
*OuterAttribute**
`..`

- can be used to destructure a data type
- when destructuring, all fields need to be addressed or ignored by `_` or `..`
- is refutable when one of its subpatterns is refutable.

```
match s {
    Point {x: 10, y: 20} => (),
    Point {y: 10, x: 20} => (),
    Point {x: 10, ..} => (),
    Point {..} => (),
}
```

Tuple struct & Tuple Patterns

Syntax

TupleStructPattern :
PathInExpression (*TupleStructItems*)

TupleStructItems :
Pattern (, *Pattern*)* , ?
| (*Pattern* ,)* .. ((, *Pattern*)+ , ?)?

Syntax

TuplePattern :
(*TuplePatternItems*?)

TuplePatternItems :
Pattern ,
| *Pattern* (, *Pattern*)+ , ?
| (*Pattern* ,)* .. ((, *Pattern*)+ , ?)?

both follow struct patterns analogous

```
match t {
    PointTuple {0: 10, 1: 20} => (),
    PointTuple {1: 10, 0: 20} => (),
    PointTuple {0: 10, ..} => (),
    PointTuple {..} => (),
}
```

Path Patterns

Syntax

PathPattern :
PathInExpression
| *QualifiedPathInExpression*

May refer to:

- enum variants
- structs
- constants
- associated constants

Irrefutable for structs and enums with one variant, else refutable

```
local_var;
globals::STATIC_VAR;
let some_constructor = Some::<i32>;
let push_integer = Vec::<i32>::push;
let slice_reverse = <[i32]>::reverse;
```

The match-Expression

- branches on a expression compared to patterns
- Comparable to switch and complicated if else

```
match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!("{}", s),
    };
```

Syntax

Syntax

MatchExpression :
`match` *Expression*$_{\text{except struct expression}}$ `{`
InnerAttribute*
MatchArms$^?$
`}`

MatchArms :
(*MatchArm* `=>` (*BlockExpression* `,`$^?$ | *Expression* `,`))*
MatchArm `=>` (*BlockExpression* | *Expression*) `,`$^?$

MatchArm :
OuterAttribute* *MatchArmPatterns* *MatchArmGuard*$^?$

MatchArmPatterns :
`|`$^?$ *Pattern* (`|` *Pattern*)*

MatchArmGuard :
`if` *Expression*

Match Guards

- allow further refinement of the matching criteria
- `if` condition after the pattern

```
match pair {
    (x, y) if x == y => println!("These are twins"),
    // The ^ `if condition` part is a guard
    (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
    (x, _) if x % 2 == 1 => println!("The first one is odd"),
    _ => println!("No correlation..."),
}
```

if let/while let

- syntactic sugar for special match

```
if let Pattern = Expr {
    Body
}
```

```
match Expr {
    Pattern => { Body }
    _ => break,
}
```

```
while let Pattern = Expr {
    Body
}
```

```
loop {
    match Expr {
        Pattern => { Body}
        _ => break,
    }
}
```

References I

S. Klabnik, C. Nichols, and R. Community, *The Rust Programming Language*. No Starch Press, Incorporated, 2018, ISBN: 9781593278281. [Online]. Available: https://books.google.de/books?id=jgHoAQAACAAJ.

(Apr. 12, 2016). Patterns, [Online]. Available: https://doc.rust-lang.org/1.8.0/book (visited on 06/04/2019).

(May 23, 2019). Match expressions - the rust reference, [Online]. Available: https://doc.rust-lang.org/reference/expressions/match-expr.html (visited on 06/04/2019).