Option-less pattern matching

Edit this page on GitHub

The implementation of pattern matching in Scala 3 was greatly simplified compared to Scala 2. From a user perspective, this means that Scala 3 generated patterns are a *lot* easier to debug, as variables all show up in debug modes and positions are correctly preserved.

Scala 3 supports a superset of Scala 2 extractors.

Extractors

Extractors are objects that expose a method `unapply` or `unapplySeq`:

```
def unapply[A](x: T)(implicit x: B): U
def unapplySeq[A](x: T)(implicit x: B): U
```

Extractors that expose the method `unapply` are called fixed-arity extractors, which work with patterns of fixed arity. Extractors that expose the method `unapplySeq` are called variadic extractors, which enables variadic patterns.

Fixed-Arity Extractors

Fixed-arity extractors expose the following signature:

```
def unapply[A](x: T)(implicit x: B): U
```

The type `U` conforms to one of the following matches:

- Boolean match
- Product match

Or `U` conforms to the type `R`:

```
type R = {
```

```
  def isEmpty: Boolean
  def get: S
}
```

and `S` conforms to one of the following matches:

- single match
- name-based match

The former form of `unapply` has higher precedence, and *single match* has higher precedence over *name-based match*.

A usage of a fixed-arity extractor is irrefutable if one of the following condition holds:

- `U = true`
- the extractor is used as a product match
- `U = Some[T]` (for Scala 2 compatibility)
- `U <: R` and `U <: { def isEmpty: false }`

Variadic Extractors

Variadic extractors expose the following signature:

```
def unapplySeq[A](x: T)(implicit x: B): U
```

The type `U` conforms to one of the following matches:

- sequence match
- product-sequence match

Or `U` conforms to the type `R`:

```
type R = {
  def isEmpty: Boolean
  def get: S
}
```

and `S` conforms to one of the two matches above.

The former form of `unapplySeq` has higher priority, and *sequence match* has higher precedence over *product-sequence match*.

A usage of a variadic extractor is irrefutable if one of the following conditions holds:

- the extractor is used directly as a sequence match or product-sequence match

- `U = Some[T]` (for Scala 2 compatibility)
- `U <: R` and `U <: { def isEmpty: false }`

Boolean Match

- `U =:= Boolean`
- Pattern-matching on exactly `0` patterns

For example:

```
object Even:
  def unapply(s: String): Boolean = s.size % 2 == 0

"even" match
  case s @ Even() => println(s"$s has an even number of characters")
  case s          => println(s"$s has an odd number of characters")

// even has an even number of characters
```

Product Match

- `U <: Product`
- `N > 0` is the maximum number of consecutive (parameterless `def` or `val`) `_1: P1` … `_N: PN` members in `U`
- Pattern-matching on exactly `N` patterns with types `P1, P2, ... , PN`

For example:

```
class FirstChars(s: String) extends Product:
  def _1 = s.charAt(0)
  def _2 = s.charAt(1)

   // Not used by pattern matching: Product is only used as a marker trait.
  def canEqual(that: Any): Boolean = ???
  def productArity: Int = ???
  def productElement(n: Int): Any = ???

object FirstChars:
  def unapply(s: String): FirstChars = new FirstChars(s)

"Hi!" match
  case FirstChars(char1, char2) =>
    println(s"First: $char1; Second: $char2")

// First: H; Second: i
```

Single Match

- If there is exactly `1` pattern, pattern-matching on `1` pattern with type `U`

```
class Nat(val x: Int):
  def get: Int = x
  def isEmpty = x < 0

object Nat:
  def unapply(x: Int): Nat = new Nat(x)

5 match
  case Nat(n) => println(s"$n is a natural number")
  case _      => ()

// 5 is a natural number
```

Name-based Match

- `N > 1` is the maximum number of consecutive (parameterless `def` or `val`) `_1: P1 ... _N: PN` members in `U`
- Pattern-matching on exactly `N` patterns with types `P1, P2, ... , PN`

```
object ProdEmpty:
  def _1: Int = ???
  def _2: String = ???
  def isEmpty = true
  def unapply(s: String): this.type = this
  def get = this

"" match
  case ProdEmpty(_, _) => ???
  case _ => ()
```

Sequence Match

- `U <: X`, `T2` and `T3` conform to `T1`

```
type X = {
  def lengthCompare(len: Int): Int // or, `def length: Int`
  def apply(i: Int): T1
  def drop(n: Int): scala.Seq[T2]
  def toSeq: scala.Seq[T3]
}
```

- Pattern-matching on *exactly* `N` simple patterns with types `T1, T1, ... , T1`, where `N` is the runtime size of the sequence, or

where `N` is the runtime size of the sequence, or

- Pattern-matching on `≥ N` simple patterns and *a vararg pattern* (e.g., `xs: _*`) with types `T1, T1, ... , T1, Seq[T1]`, where `N` is the minimum size of the sequence.

```
object CharList:
  def unapplySeq(s: String): Option[Seq[Char]] = Some(s.toList)

"example" match
  case CharList(c1, c2, c3, c4, _, _, _) =>
    println(s"$c1,$c2,$c3,$c4")
  case _ =>
    println("Expected *exactly* 7 characters!")

// e,x,a,m
```

Product-Sequence Match

- `U <: Product`
- `N > 0` is the maximum number of consecutive (parameterless `def` or `val`) `_1: P1` … `_N: PN` members in `U`
- `PN` conforms to the signature `X` defined in Seq Pattern
- Pattern-matching on exactly `≥ N` patterns, the first `N - 1` patterns have types `P1, P2, ... P(N-1)`, the type of the remaining patterns are determined as in Seq Pattern.

```
class Foo(val name: String, val children: Int *)
object Foo:
  def unapplySeq(f: Foo): Option[(String, Seq[Int])] =
    Some((f.name, f.children))

def foo(f: Foo) = f match
  case Foo(name, ns : _*) =>
  case Foo(name, x, y, ns : _*) =>
```

There are plans for further simplification, in particular to factor out *product match* and *name-based match* into a single type of extractor.

Type testing

Abstract type testing with `ClassTag` is replaced with `TypeTest` or the alias `Typeable`.

- pattern `_: X` for an abstract type requires a `TypeTest` in scope
- pattern `x @ X()` for an unapply that takes an abstract type requires a

- pattern `x @ X()` for an unapply that takes an abstract type requires a `TypeTest` in scope

More details on `TypeTest`

Patter... Autom...