Pattern Matching

Pattern Matching

Patterns

```
Pattern         ::=  Pattern1 { '|' Pattern1 }
Pattern1        ::=  boundvarid ':' TypePat
                  |  '_' ':' TypePat
                  |  Pattern2
Pattern2        ::=  id ['@' Pattern3]
                  |  Pattern3
Pattern3        ::=  SimplePattern
                  |  SimplePattern {id [nl] SimplePattern}
SimplePattern   ::=  '_'
                  |  varid
                  |  Literal
                  |  StableId
                  |  StableId '(' [Patterns] ')'
               |  StableId '(' [Patterns ','] [id '@'] '_' '*' ')'
                  |  '(' [Patterns] ')'
                  |  XmlPattern
Patterns        ::=  Pattern {',' Patterns}
```

A pattern is built from constants, constructors, variables and type tests. Pattern matching tests whether a given value (or sequence of values) has the shape defined by a pattern, and, if it does, binds the variables in the pattern to the corresponding components of the value (or sequence of values). The same variable name may not be bound more than once in a pattern.

Example

Some examples of patterns are: 1. The pattern `ex: IOException` matches all instances of class `IOException`, binding variable `ex` to the instance. 1. The pattern `Some(x)` matches values of the form `Some($v$)`, binding `x` to the argument value v of the `Some` constructor. 1. The pattern `(x, _)` matches pairs of values, binding `x` to the first component of the pair. The second component is matched with a wildcard pattern. 1. The pattern `x :: y :: xs` matches lists of length ≥ 2, binding `x` to the list's first element, `y` to the list's second element, and `xs`

to the remainder. 1. The pattern `1 | 2 | 3` matches the integers between 1 and 3.

Pattern matching is always done in a context which supplies an expected type of the pattern. We distinguish the following kinds of patterns.

Variable Patterns

```
SimplePattern   ::=  '_'
                  |  varid
```

A *variable pattern* x is a simple identifier which starts with a lower case letter. It matches any value, and binds the variable name to that value. The type of x is the expected type of the pattern as given from outside. A special case is the wild-card pattern `_` which is treated as if it was a fresh variable on each occurrence.

Typed Patterns

```
Pattern1        ::=  varid ':' TypePat
                  |  '_' ':' TypePat
```

A *typed pattern* $x : T$ consists of a pattern variable x and a type pattern T. The type of x is the type pattern T, where each type variable and wildcard is replaced by a fresh, unknown type. This pattern matches any value matched by the type pattern T; it binds the variable name to that value.

Pattern Binders

```
Pattern2        ::=  varid '@' Pattern3
```

A *pattern binder* `$x$@$p$` consists of a pattern variable x and a pattern p. The type of the variable x is the static type T implied by the pattern p. This pattern matches any value v matched by the pattern p, and it binds the variable name to that value.

A pattern p *implies* a type T if the pattern matches only values of the type T.

Literal Patterns

```
SimplePattern   ::=  Literal
```

A *literal pattern* L matches any value that is equal (in terms of `==`) to the literal L. The type of L must conform to the expected type of the pattern.

Interpolated string patterns

```
Literal  ::=  interpolatedString
```

The expansion of interpolated string literals in patterns is the same as in expressions. If it occurs in a pattern, a interpolated string literal of either of the forms

```
id"text0{ pat1 }text1 … { patn }textn"
id"""text0{ pat1 }text1 … { patn }textn"""
```

is equivalent to:

```
StringContext("""text0""", …, """textn""").id(pat1, …, patn)
```

You could define your own `StringContext` to shadow the default one that's in the `scala` package.

This expansion is well-typed if the member `id` evaluates to an extractor object. If the extractor object has `apply` as well as `unapply` or `unapplySeq` methods, processed strings can be used as either expressions or patterns.

Taking XML as an example

```
implicit class XMLinterpolation(s: StringContext) = {
    object xml {
        def apply(exprs: Any*) =
            // parse ‘s’ and build an XML tree with ‘exprs’
            //in the holes
        def unapplySeq(xml: Node): Option[Seq[Node]] =
          // match `s’ against `xml’ tree and produce
          //subtrees in holes
    }
}
```

Then, XML pattern matching could be expressed like this:

```
case xml"""
      <body>
        <a href = "some link"> \$linktext </a>
      </body>
     """ => ...
```

where linktext is a variable bound by the pattern.

Stable Identifier Patterns

```
SimplePattern   ::=  StableId
```

A *stable identifier pattern* is a stable identifier r. The type of r must conform to the expected type of the pattern. The pattern matches any value v such that `$r$ == $v$` (see here).

To resolve the syntactic overlap with a variable pattern, a stable identifier pattern may not be a simple name starting with a lower-case letter. However, it is possible to enclose such a variable name in backquotes; then it is treated as a stable identifier pattern.

Example

Consider the following function definition:

```
def f(x: Int, y: Int) = x match {
  case y => ...
}
```

Here, `y` is a variable pattern, which matches any value. If we wanted to turn the pattern into a stable identifier pattern, this can be achieved as follows:

```
def f(x: Int, y: Int) = x match {
  case `y` => ...
}
```

Now, the pattern matches the `y` parameter of the enclosing function `f`. That is, the match succeeds only if the `x` argument and the `y` argument of `f` are equal.

Constructor Patterns

```
SimplePattern   ::=  StableId ‘(’ [Patterns] ‘)’
```

A *constructor pattern* is of the form $c(p_1, \ldots, p_n)$ where $n \geq 0$. It consists of a stable identifier c, followed by element patterns $p_1, \ldots, p_n$. The constructor c is a simple or qualified name which denotes a case class. If the case class is monomorphic, then it must conform to the expected type of the pattern, and the formal parameter types of x's primary constructor are taken as the expected types of the element patterns $p_1, \ldots, p_n$. If the case class is polymorphic, then its type parameters are instantiated so that the instantiation of c conforms to the expected type of the pattern. The instantiated formal parameter types of c's primary constructor are then taken as the expected types of the component patterns $p_1, \ldots, p_n$. The pattern matches all objects created from constructor invocations $c(v_1, \ldots, v_n)$ where each element pattern p_i matches the corresponding value v_i.

A special case arises when c's formal parameter types end in a repeated parameter. This is further discussed here.

Tuple Patterns

```
SimplePattern   ::=  ‘(’ [Patterns] ‘)’
```

A *tuple pattern* `($p_1 , \ldots , p_n$)` is an alias for the constructor pattern `scala.Tuple$n$($p_1 , \ldots , p_n$)`, where $n \geq 2$. The empty tuple `()` is the unique value of type `scala.Unit`.

Extractor Patterns

```
SimplePattern   ::=  StableId '(' [Patterns] ')'
```

An *extractor pattern* $x(p_1, \ldots, p_n)$ where $n \geq 0$ is of the same syntactic form as a constructor pattern. However, instead of a case class, the stable identifier x denotes an object which has a member method named `unapply` or `unapplySeq` that matches the pattern.

An extractor pattern cannot match the value `null`. The implementation ensures that the `unapply`/`unapplySeq` method is not applied to `null`.

An `unapply` method in an object x *matches* the pattern $x(p_1, \ldots, p_n)$ if it has a single parameter (and, optionally, an implicit parameter list) and one of the following applies:

- $n = 0$ and `unapply`'s result type is `Boolean`. In this case the extractor pattern matches all values v for which `$x$.unapply($v$)` yields `true`.
- $n = 1$ and `unapply`'s result type is `Option[$T$]`, for some type T. In this case, the (only) argument pattern p_1 is typed in turn with expected type T. The extractor pattern matches then all values v for which `$x$.unapply($v$)` yields a value of form `Some($v_1$)`, and p_1 matches v_1.
- $n > 1$ and `unapply`'s result type is `Option[($T_1 , \ldots , T_n$)]`, for some types $T_1, \ldots, T_n$. In this case, the argument patterns $p_1, \ldots, p_n$ are typed in turn with expected types $T_1, \ldots, T_n$. The extractor pattern matches then all values v for which `$x$.unapply($v$)` yields a value of form `Some(($v_1 , \ldots , v_n$))`, and each pattern p_i matches the corresponding value v_i.

An `unapplySeq` method in an object x matches the pattern $x(q_1, \ldots, q_m, p_1, \ldots, p_n)$ if it takes exactly one argument and its result type is of the form `Option[($T_1 , \ldots , T_m$, Seq[S])]` (if `m = 0`, the type `Option[Seq[S]]` is also accepted). This case is further discussed below.

Example

If we define an extractor object `Pair`:

```
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

This means that the name `Pair` can be used in place of `Tuple2` for tuple formation as well as for deconstruction of tuples in patterns. Hence, the following is possible:

```
val x = (1, 2)
val y = x match {
```

```
  case Pair(i, s) => Pair(s + i, i * i)
}
```

Pattern Sequences

```
SimplePattern ::= StableId '(' [Patterns ','] [varid '@'] '_' '*' ')'
```

A *pattern sequence* $p_1, \ldots, p_n$ appears in two contexts. First, in a constructor pattern $c(q_1, \ldots, q_m, p_1, \ldots, p_n)$, where c is a case class which has $m+1$ primary constructor parameters, ending in a repeated parameter of type `S*`. Second, in an extractor pattern $x(q_1, \ldots, q_m, p_1, \ldots, p_n)$ if the extractor object x does not have an `unapply` method, but it does define an `unapplySeq` method with a result type conforming to `Option[(T_1, ... , T_m, Seq[S])]` (if `m = 0`, the type `Option[Seq[S]]` is also accepted). The expected type for the patterns p_i is S.

The last pattern in a pattern sequence may be a *sequence wildcard* `_*`. Each element pattern p_i is type-checked with S as expected type, unless it is a sequence wildcard. If a final sequence wildcard is present, the pattern matches all values v that are sequences which start with elements matching patterns $p_1, \ldots, p_{n-1}$. If no final sequence wildcard is given, the pattern matches all values v that are sequences of length n which consist of elements matching patterns $p_1, \ldots, p_n$.

Infix Operation Patterns

```
  Pattern3  ::=  SimplePattern {id [nl] SimplePattern}
```

An *infix operation pattern* $p; op; q$ is a shorthand for the constructor or extractor pattern $op(p, q)$. The precedence and associativity of operators in patterns is the same as in expressions.

An infix operation pattern $p; op; (q_1, \ldots, q_n)$ is a shorthand for the constructor or extractor pattern $op(p, q_1, \ldots, q_n)$.

Pattern Alternatives

```
  Pattern   ::=  Pattern1 { '|' Pattern1 }
```

A *pattern alternative* `$p_1$ | $\ldots$ | $p_n$` consists of a number of alternative patterns p_i. All alternative patterns are type checked with the expected type of the pattern. They may not bind variables other than wildcards. The alternative pattern matches a value v if at least one its alternatives matches v.

XML Patterns

XML patterns are treated here.

Regular Expression Patterns

Regular expression patterns have been discontinued in Scala from version 2.0.

Later version of Scala provide a much simplified version of regular expression patterns that cover most scenarios of non-text sequence processing. A *sequence pattern* is a pattern that stands in a position where either (1) a pattern of a type `T` which is conforming to `Seq[A]` for some `A` is expected, or (2) a case class constructor that has an iterated formal parameter `A*`. A wildcard star pattern `_*` in the rightmost position stands for arbitrary long sequences. It can be bound to variables using `@`, as usual, in which case the variable will have the type `Seq[A]`.

Irrefutable Patterns

A pattern p is *irrefutable* for a type T, if one of the following applies:

1. p is a variable pattern,
2. p is a typed pattern $x : T'$, and $T <: T'$,
3. p is a constructor pattern $c(p_1, \ldots, p_n)$, the type T is an instance of class c, the primary constructor of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for T_i.

Type Patterns

```
TypePat            ::=  Type
```

Type patterns consist of types, type variables, and wildcards. A type pattern T is of one of the following forms:

- A reference to a class C, $p.C$, or `$T$#$C$`. This type pattern matches any non-null instance of the given class. Note that the prefix of the class, if it exists, is relevant for determining class instances. For instance, the pattern $p.C$ matches only instances of classes C which were created with the path p as prefix. This also applies to prefixes which are not given syntactically. For example, if C refers to a class defined in the nearest enclosing class and is thus equivalent to $this.C$, it is considered to have a prefix.

  The bottom types `scala.Nothing` and `scala.Null` cannot be used as type patterns, because they would match nothing in any case.

- A singleton type `$p$.type`. This type pattern matches only the value denoted by the path p (the `eq` method is used to compare the matched value to p).

- A literal type `$lit$`. This type pattern matches only the value denoted by the literal *lit* (the `==` method is used to compare the matched value to *lit*).

- A compound type pattern `$T_1$ with $\ldots$ with $T_n$` where each T_i is a type pattern. This type pattern matches all values that are matched by each of the type patterns T_i.
- A parameterized type pattern $T[a_1, \ldots, a_n]$, where the a_i are type variable patterns or wildcards `_`. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards. The bounds or alias type of these type variable are determined as described here.
- A parameterized type pattern `scala.Array$[T_1]$`, where T_1 is a type pattern. This type pattern matches any non-null instance of type `scala.Array$[U_1]$`, where U_1 is a type matched by T_1.

Types which are not of one of the forms described above are also accepted as type patterns. However, such type patterns will be translated to their erasure. The Scala compiler will issue an "unchecked" warning for these patterns to flag the possible loss of type-safety.

A *type variable pattern* is a simple identifier which starts with a lower case letter.

Type Parameter Inference in Patterns

Type parameter inference is the process of finding bounds for the bound type variables in a typed pattern or constructor pattern. Inference takes into account the expected type of the pattern.

Type parameter inference for typed patterns

Assume a typed pattern $p : T'$. Let T result from T' where all wildcards in T' are renamed to fresh variable names. Let $a_1, \ldots, a_n$ be the type variables in T. These type variables are considered bound in the pattern. Let the expected type of the pattern be pt.

Type parameter inference constructs first a set of subtype constraints over the type variables a_i. The initial constraints set $\mathcal{C}\_0$ reflects just the bounds of these type variables. That is, assuming T has bound type variables $a_1, \ldots, a_n$ which correspond to class type parameters $a'_1, \ldots, a'_n$ with lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$, $\mathcal{C}_0$ contains the constraints

$$\begin{cases} a_i & <: \sigma U_i \quad (i = 1, \ldots, n) \\ \\ \sigma L_i & <: a_i \quad (i = 1, \ldots, n) \end{cases}$$

where σ is the substitution $[a'_1 := a_1, \ldots, a'_n := a_n]$.

The set $\mathcal{C}_0$ is then augmented by further subtype constraints. There are two cases.

Case 1

If there exists a substitution σ over the type variables $a_i, \ldots, a_n$ such that σT conforms to pt, one determines the weakest subtype constraints $\mathcal{C}\_1$ over the type variables $a_1, \ldots, a_n$ such that $\mathcal{C}\_0 \wedge \mathcal{C}_1$ implies that T conforms to pt.

Case 2

Otherwise, if T can not be made to conform to pt by instantiating its type variables, one determines all type variables in pt which are defined as type parameters of a method enclosing the pattern. Let the set of such type parameters be $b_1, \ldots, b_m$. Let $\mathcal{C}\_0'$ be the subtype constraints reflecting the bounds of the type variables b_i. If T denotes an instance type of a final class, let $\mathcal{C}\_2$ be the weakest set of subtype constraints over the type variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ such that $\mathcal{C}\_0 \wedge \mathcal{C}\_0' \wedge \mathcal{C}\_2$ implies that T conforms to pt. If T does not denote an instance type of a final class, let $\mathcal{C}\_2$ be the weakest set of subtype constraints over the type variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ such that $\mathcal{C}\_0 \wedge \mathcal{C}\_0' \wedge \mathcal{C}\_2$ implies that it is possible to construct a type T' which conforms to both T and pt. It is a static error if there is no satisfiable set of constraints $\mathcal{C}\_2$ with this property.

The final step consists in choosing type bounds for the type variables which imply the established constraint system. The process is different for the two cases above.

Case 1

We take $a_i >: L_i <: U_i$ where each L_i is minimal and each U_i is maximal wrt $<:$ such that $a_i >: L_i <: U_i$ for $i = 1, \ldots, n$ implies $\mathcal{C}\_0 \wedge \mathcal{C}\_1$.

Case 2

We take $a_i >: L_i <: U_i$ and \$b_i >: L_i' <: U_i' \$ where each L_i and L_j' is minimal and each U_i and U_j' is maximal such that $a_i >: L_i <: U_i$ for $i = 1, \ldots, n$ and $b_j >: L_j' <: U_j'$ for $j = 1, \ldots, m$ implies $\mathcal{C}\_0 \wedge \mathcal{C}\_0' \wedge \mathcal{C}_2$.

In both cases, local type inference is permitted to limit the complexity of inferred bounds. Minimality and maximality of types have to be understood relative to the set of types of acceptable complexity.

Type parameter inference for constructor patterns

Assume a constructor pattern $C(p_1, \ldots, p_n)$ where class C has type parameters $a_1, \ldots, a_n$. These type parameters are inferred in the same way as for the typed pattern `(_: $C[a_1 , \ldots , a_n]$)`.

Example

Consider the program fragment:

```
val x: Any
x match {
  case y: List[a] => ...
}
```

Here, the type pattern `List[a]` is matched against the expected type `Any`. The pattern binds the type variable `a`. Since `List[a]` conforms to `Any` for every type argument, there are no constraints on `a`. Hence, `a` is introduced as an abstract type with no bounds. The scope of `a` is right-hand side of its case clause.

On the other hand, if `x` is declared as

`val x: List[List[String]]`,

this generates the constraint `List[a] <: List[List[String]]`, which simplifies to `a <: List[String]`, because `List` is covariant. Hence, `a` is introduced with upper bound `List[String]`.

Example

Consider the program fragment:

```
val x: Any
x match {
  case y: List[String] => ...
}
```

Scala does not maintain information about type arguments at run-time, so there is no way to check that `x` is a list of strings. Instead, the Scala compiler will erase the pattern to `List[_]`; that is, it will only test whether the top-level runtime-class of the value `x` conforms to `List`, and the pattern match will succeed if it does. This might lead to a class cast exception later on, in the case where the list `x` contains elements other than strings. The Scala compiler will flag this potential loss of type-safety with an "unchecked" warning message.

Example

Consider the program fragment

```
class Term[A]
class Number(val n: Int) extends Term[Int]
def f[B](t: Term[B]): B = t match {
  case y: Number => y.n
}
```

The expected type of the pattern `y: Number` is `Term[B]`. The type `Number` does not conform to `Term[B]`; hence Case 2 of the rules above applies. This means that `B` is treated as another type variable for which subtype constraints are inferred. In our case the applicable constraint is `Number <: Term[B]`, which entails `B = Int`. Hence, `B` is treated in the case clause as an abstract type with lower and upper bound `Int`. Therefore, the right hand side of the case clause, `y.n`, of type `Int`, is found to conform to the function's declared result type, `Number`.

Pattern Matching Expressions

```
Expr            ::=  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
CaseClauses     ::=  CaseClause {CaseClause}
CaseClause      ::=  ‘case’ Pattern [Guard] ‘=>’ Block
```

A *pattern matching expression*

```
e match { case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

consists of a selector expression e and a number $n > 0$ of cases. Each case consists of a (possibly guarded) pattern p_i and a block b_i. Each p_i might be complemented by a guard `if $e$` where e is a boolean expression. The scope of the pattern variables in p_i comprises the pattern's guard and the corresponding block b_i.

Let T be the type of the selector expression e and let $a_1, \ldots, a_m$ be the type parameters of all methods enclosing the pattern matching expression. For every a_i, let L_i be its lower bound and U_i be its higher bound. Every pattern $p \in \{p_1, , \ldots, p_n\}$ can be typed in two ways. First, it is attempted to type p with T as its expected type. If this fails, p is instead typed with a modified expected type T' which results from T by replacing every occurrence of a type parameter a_i by *undefined*. If this second step fails also, a compile-time error results. If the second step succeeds, let T_p be the type of pattern p seen as an expression. One then determines minimal bounds $L_1 1, \ldots, L'_m$ and maximal bounds $U'_1, \ldots, U'_m$ such that for all i, $L_i <: L'_i$ and $U'_i <: U_i$ and the following constraint system is satisfied:

$$L_1 <: a_1 <: U_1 \ \wedge \ \ldots \ \wedge \ L_m <: a_m <: U_m \ \Rightarrow \ T_p <: T$$

If no such bounds can be found, a compile time error results. If such bounds are found, the pattern matching clause starting with p is then typed under the assumption that each a_i has lower bound L'_i instead of L_i and has upper bound U'_i instead of U_i.

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound of the types of all blocks b_i.

When applying a pattern matching expression to a selector value, patterns are tried in sequence until one is found which matches the selector value. Say this case is `case $p_i \Rightarrow b_i$`. The result of the whole expression is the result of evaluating b_i, where all pattern variables of p_i are bound to the corresponding parts of the selector value. If no matching pattern is found, a `scala.MatchError` exception is thrown.

The pattern in a case may also be followed by a guard suffix `if e` with a boolean expression e. The guard expression is evaluated if the preceding pattern in the case matches. If the guard expression evaluates to `true`, the pattern match

succeeds as normal. If the guard expression evaluates to `false`, the pattern in the case is considered not to match and the search for a matching pattern continues.

In the interest of efficiency the evaluation of a pattern matching expression may try patterns in some other order than textual sequence. This might affect evaluation through side effects in guards. However, it is guaranteed that a guard expression is evaluated only if the pattern it guards matches.

If the selector of a pattern match is an instance of a `sealed` class, the compilation of pattern matching can emit warnings which diagnose that a given set of patterns is not exhaustive, i.e. that there is a possibility of a `MatchError` being raised at run-time.

Example

Consider the following definitions of arithmetic terms:

```
abstract class Term[T]
case class Lit(x: Int) extends Term[Int]
case class Succ(t: Term[Int]) extends Term[Int]
case class IsZero(t: Term[Int]) extends Term[Boolean]
case class If[T](c: Term[Boolean],
                 t1: Term[T],
                 t2: Term[T]) extends Term[T]
```

There are terms to represent numeric literals, incrementation, a zero test, and a conditional. Every term carries as a type parameter the type of the expression it represents (either `Int` or `Boolean`).

A type-safe evaluator for such terms can be written as follows.

```
def eval[T](t: Term[T]): T = t match {
  case Lit(n)        => n
  case Succ(u)       => eval(u) + 1
  case IsZero(u)     => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

Note that the evaluator makes crucial use of the fact that type parameters of enclosing methods can acquire new bounds through pattern matching.

For instance, the type of the pattern in the second case, `Succ(u)`, is `Int`. It conforms to the selector type `T` only if we assume an upper and lower bound of `Int` for `T`. Under the assumption `Int <: T <: Int` we can also verify that the type right hand side of the second case, `Int` conforms to its expected type, `T`.

Pattern Matching Anonymous Functions

```
BlockExpr ::= '{' CaseClauses '}'
```

An anonymous function can be defined by a sequence of cases

```
{ case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

which appear as an expression without a prior `match`. The expected type of such an expression must in part be defined. It must be either `scala.Function$k$[$S_1 , \ldots , S_k$, $R$]` for some $k > 0$, or `scala.PartialFunction[$S_1$, $R$]`, where the argument type(s) $S_1, \ldots, S_k$ must be fully determined, but the result type R may be undetermined.

If the expected type is SAM-convertible to `scala.Function$k$[$S_1 , \ldots , S_k$, $R$]`, the expression is taken to be equivalent to the anonymous function:

```
($x_1: S_1 , \ldots , x_k: S_k$) => ($x_1 , \ldots , x_k$) match {
  case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
}
```

Here, each x_i is a fresh name. As was shown here, this anonymous function is in turn equivalent to the following instance creation expression, where T is the weak least upper bound of the types of all b_i.

```
new scala.Function$k$[$S_1 , \ldots , S_k$, $T$] {
  def apply($x_1: S_1 , \ldots , x_k: S_k$): $T$ = ($x_1 , \ldots , x_k$) match {
    case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
  }
}
```

If the expected type is `scala.PartialFunction[$S$, $R$]`, the expression is taken to be equivalent to the following instance creation expression:

```
new scala.PartialFunction[$S$, $T$] {
  def apply($x$: $S$): $T$ = x match {
    case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
  }
  def isDefinedAt($x$: $S$): Boolean = {
    case $p_1$ => true $\ldots$ case $p_n$ => true
    case _ => false
  }
}
```

Here, x is a fresh name and T is the weak least upper bound of the types of all b_i. The final default case in the `isDefinedAt` method is omitted if one of the patterns $p_1, \ldots, p_n$ is already a variable or wildcard pattern.

Example

Here is a method which uses a fold-left operation `/:` to compute the scalar product of two vectors:

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
```

```
    case (a, (b, c)) => a + b * c
  }
```

The case clauses in this code are equivalent to the following anonymous function:

```
(x, y) => (x, y) match {
  case (a, (b, c)) => a + b * c
}
```