Match Types

Edit this page on GitHub

A match type reduces to one of its right-hand sides, depending on the type of its scrutinee. For example:

```
type Elem[X] = X match
  case String => Char
  case Array[t] => t
  case Iterable[t] => t
```

This defines a type that reduces as follows:

```
Elem[String]       =:=  Char
Elem[Array[Int]]   =:=  Int
Elem[List[Float]]  =:=  Float
Elem[Nil.type]     =:=  Nothing
```

Here `=:=` is understood to mean that left and right-hand sides are mutually subtypes of each other.

In general, a match type is of the form

```
S match { P1 => T1 ... Pn => Tn }
```

where `S`, `T1`, ..., `Tn` are types and `P1`, ..., `Pn` are type patterns. Type variables in patterns start with a lower case letter, as usual.

Match types can form part of recursive type definitions. Example:

```
type LeafElem[X] = X match
  case String => Char
  case Array[t] => LeafElem[t]
  case Iterable[t] => LeafElem[t]
  case AnyVal => X
```

Recursive match type definitions can also be given an upper bound, like this:

```
type Concat[Xs <: Tuple, +Ys <: Tuple] <: Tuple = Xs match
  case EmptyTuple => Ys
  case x *: xs => x *: Concat[xs, Ys]
```

In this definition, every instance of `Concat[A, B]`, whether reducible or not, is known to be a subtype of `Tuple`. This is necessary to make the recursive invocation `x *: Concat[xs, Ys]` type check, since `*:` demands a `Tuple` as its right operand.

Dependent Typing

Match types can be used to define dependently typed methods. For instance, here is the value level counterpart to the `LeafElem` type defined above (note the use of the match type as the return type):

```
def leafElem[X](x: X): LeafElem[X] = x match
  case x: String      => x.charAt(0)
  case x: Array[t]    => leafElem(x(0))
  case x: Iterable[t] => leafElem(x.head)
  case x: AnyVal      => x
```

This special mode of typing for match expressions is only used when the following conditions are met:

1. The match expression patterns do not have guards
2. The match expression scrutinee's type is a subtype of the match type scrutinee's type
3. The match expression and the match type have the same number of cases
4. The match expression patterns are all Typed Patterns, and these types are `=:=` to their corresponding type patterns in the match type

Representation of Match Types

The internal representation of a match type

```
S match { P1 => T1 ... Pn => Tn }
```

is `Match(S, C1, …, Cn) <: B` where each case `Ci` is of the form

```
[Xs] =>> P => T
```

Here, `[Xs]` is a type parameter clause of the variables bound in pattern `Pi`. If there are no bound type variables in a case, the type parameter clause is omitted and only the function type `P => T` is kept. So each case is either a unary function type or a type lambda over a unary function type.

`B` is the declared upper bound of the match type, or `Any` if no such bound is given. We will leave it out in places where it does not matter for the discussion. The scrutinee, bound, and pattern types must all be first-order types.

Match Type Reduction

Match type reduction follows the semantics of match expressions, that is, a match type of the form `S match { P1 => T1 ... Pn => Tn }` reduces to `Ti` if and only if `s: S match { _: P1 => T1 ... _: Pn => Tn }` evaluates to a value of type `Ti` for all `s: S`.

The compiler implements the following reduction algorithm:

- If the scrutinee type `S` is an empty set of values (such as `Nothing` or `String & Int`), do not reduce.
- Sequentially consider each pattern `Pi`
  - If `S <: Pi` reduce to `Ti`.
  - Otherwise, try constructing a proof that `S` and `Pi` are disjoint, or, in other words, that no value `s` of type `S` is also of type `Pi`.
  - If such proof is found, proceed to the next case (`Pi+1`), otherwise, do not reduce.

Disjointness proofs rely on the following properties of Scala types:

1. Single inheritance of classes
2. Final classes cannot be extended
3. Constant types with distinct values are nonintersecting
4. Singleton paths to distinct values are nonintersecting, such as `object` definitions or singleton enum cases.

Type parameters in patterns are minimally instantiated when computing `S <: Pi`. An instantiation `Is` is *minimal* for `Xs` if all type variables in `Xs` that appear covariantly and nonvariantly in `Is` are as small as possible and all type variables in `Xs` that appear contravariantly in `Is` are as large as possible. Here, "small" and "large" are understood with respect to `<:`.

For simplicity, we have omitted constraint handling so far. The full formulation of subtyping tests describes them as a function from a constraint and a pair of types to either *success* and a new constraint or *failure*. In the context of reduction, the subtyping test `S <: [Xs := Is] P` is understood to leave the bounds of all variables in the input constraint unchanged, i.e. existing variables in the constraint cannot be instantiated by matching the scrutinee against the patterns.

Subtyping Rules for Match Types

The following rules apply to match types. For simplicity, we omit environments and constraints.

1. The first rule is a structural comparison between two match types:

   ```
   S match { P1 => T1 ... Pm => Tm }  <:  T match { Q1 => U1 ... Qn => Un }
   ```

   if

   ```
   S =:= T,  m >= n,  Pi =:= Qi and Ti <: Ui for i in 1..n
   ```

   I.e. scrutinees and patterns must be equal and the corresponding bodies must be subtypes. No case re-ordering is allowed, but the subtype can have more cases than the supertype.

2. The second rule states that a match type and its redux are mutual subtypes.

   ```
   S match { P1 => T1 ... Pn => Tn }  <:  U
   U  <:  S match { P1 => T1 ... Pn => Tn }
   ```

   if

   `S match { P1 ⇒ T1 ... Pn ⇒ Tn }` reduces to `U`

3. The third rule states that a match type conforms to its upper bound:

   ```
   (S match { P1 => T1 ... Pn => Tn } <: B)  <:  B
   ```

Termination

Match type definitions can be recursive, which means that it's possible to run into an infinite loop while reducing match types.

Since reduction is linked to subtyping, we already have a cycle detection mechanism in place. As a result, the following will already give a reasonable error message:

```
type L[X] = X match
  case Int => L[X]

def g[X]: L[X] = ???
```

```
|  val x: Int = g[Int]
   |                 ^
   |Recursion limit exceeded.
   |Maybe there is an illegal cyclic reference?
   |If that's not the case, you could also try to
   |increase the stacksize using the -Xss JVM option.
   |A recurring operation is (inner to outer):
   |
   |  subtype LazyRef(Test.L[Int]) <:< Int
```

Internally, the Scala compiler detects these cycles by turning selected stack overflows into type errors. If there is a stack overflow during subtyping, the exception will be caught and turned into a compile-time error that indicates a trace of the subtype tests that caused the overflow without showing a full stack trace.

Match Types Variance

All type positions in a match type (scrutinee, patterns, bodies) are considered invariant.

Related Work

Match types have similarities with closed type families in Haskell. Some differences are:

- Subtyping instead of type equalities.
- Match type reduction does not tighten the underlying constraint, whereas type family reduction does unify. This difference in approach mirrors the difference between local type inference in Scala and global type inference in Haskell.

Match types are also similar to Typescript's conditional types. The main differences here are:

- Conditional types only reduce if both the scrutinee and pattern are ground, whereas match types also work for type parameters and abstract types.
- Match types support direct recursion.

- Conditional types distribute through union types.

Type L... Depen...

Contributors to this page

pikinier20 BarkingBad dwijnand OlivierBlanvillain
julienrf odersky michelou bishabosha
anatoliykmetyuk felher iamrecursion robstoll

Scaladoc Copyright (c) 2002-2022, LAMP/EPFL