# PRECISE TYPE INFERENCE IN SCALA 3

Presentation [HTML|PDF], Report [HTML|PDF]

Matt Bovel @LAMP/LARA, EPFL

July 7, 2022

# OUTLINE

- What precise types?
  - Singletons
  - Unions
  - Match types
  - Type-level operations
  - Refinements
- Why not just fit expected result types?
  - Code duplication
  - Not trivial to implement
- Precise mode
  - `dependent` values and functions
  - Example with inferred types
  - Visibility of arguments to fields mapping
  - Dependent case classes
  - Motivating example
- Why not always infer them?
  - Subtyping will save us?
  - Implicits search
  - Overloads resolution
  - Other considerations
  - Why case classes
  - Syntactic Sugar
- Further work
  - Always precise and widen?
  - Distinct term-level constructs?
  - Error types?
  - Type parameters?
- The end


  - Ack
  - Ref

# WHAT PRECISE TYPES?

# SINGLETONS

Singletons are widened.

```
val v1 /*: Int*/ = 3 /*: 3*/
val v2 /*: Int*/ = v1 /*: v1.type*/
```

# UNIONS

If-then-else are typed with unions types, which are then widened.

```
val v3 /*: Int*/ = if c then 1 else 2 /*: 1 | 2*/
```

# MATCH TYPES

By default, the result type of a match is the LUB of the result types of the cases

```
val v4 /*: Boolean */ = x match
  case _: String => true
  case _ => false
```

But we can also type it as the matching match type if we write it explicitly:

```
type IsString[T <: Any] = T match {
  case String => true
  case _ => false }
val v5: IsString[x.type] = x match
  case _: String => true
  case _ => false
```

# TYPE-LEVEL OPERATIONS

```
import scala.compiletime.ops.int.*
val v6: Int = 42
val v7: Int = v6 + 2
val v8: v6.type + 2 = v6 + 2 // error
```

# REFINEMENTS

```
class Foo(val x: Int)
val v9: Foo = Foo(1984)
val v10: Foo {val x: 1984} = Foo(1984) // error
```

See Refine types according to their constructor val's singleton types #1262

# MOTIVATING EXAMPLE

```
import scala.compiletime.ops.int.+
def vec(s: Int) = Vec(s).asInstanceOf[Vec {val size: s.type }]
def add(a: Int, b: Int) = (a + b).asInstanceOf[a.type + b.type]

case class Vec(size: Int):
  def sum(that: Vec {val size: Vec.this.size.type}) = vec(size)
  def concat(that: Vec) = vec(add(size, that.size))

val v11: Vec {val size: 13} = vec(6).concat(vec(7)).sum(vec(13))
```

# WHY NOT ALWAYS INFER THEM?

# SUBTYPING WILL SAVE US?

Thanks to subtyping, we should always be able to replace a type by a more precise type (cf. a subtype). Right?

```
def f1(foo: Foo) = true
val v12 = Foo(1984)
f(v12)
```

```
def f2[T](a: T, b: T) = true
f2(Foo(451), Foo(1984))
```

# IMPLICITS SEARCH

Precising types can break previously working implicits search.

```
class A
class B extends A
class Inv[X]


given inv: Inv[A] = Inv()
def f3[N](x: N)(using Inv[N]) = 1984
```

```
val b = B()
f3(b: A)
f3[A](b)
f3(b)(using inv)
f3(b) // error: no given instance of type Inv[B]
```

# OVERLOADS RESOLUTION

Precising types can break previously working overloads resolution.

```
def f4(x: Int) = "C"
def f4(x: String | 1 | 2) = "D"
val cond = false
val y = if cond then 1 else 2

println(f4(y))
val preciseY: 1 | 2 = if cond then 1 else 2
println(f4(preciseY)) // error: ambiguous overload
```

General problematic setup: let `f` be a function with two overloads respectively taking two unrelated types `C` and `D` as arguments, let `y` be a variable that can be typed either as `C` or `C & D`, consider `f(y)`.

# OTHER CONSIDERATIONS

- **Usability:** types that would just duplicate expressions are generally not useful to help programmers to think.
- **Performance:** bigger types take more time to process.

# WHY NOT JUST FIT EXPECTED RESULT TYPES?

# CODE DUPLICATION

See tf-dotty example.

# NOT TRIVIAL TO IMPLEMENT

When normalization is introduced, we cannot simply fit the structure of the expected type the the right hand-side:

```
val v13: v6.type = (v6 + 1) - 1
```

# PRECISE MODE

# `dependent` VALUES AND FUNCTIONS

Proposition: type everything precisely when a value or a function is annotated with the `dependent` keyword.

```
dependent def precise() =
  val v1 = 1

  val v2 = 2 + v1
  dependent def isString(x: Any) = x match
    case _: String => true
    case _ => false
  val v3 = isString(42)
  val v4 = Foo(42)
```

The `dependent` keyword was first proposed in [1] and our implementation follows a similar but weaker semantic. In our case, `dependent` simply instructs the system to type the body of the function “as precisely as possible”, while in [1] it means “as precise as its implementation”.

# EXAMPLE WITH INFERRED TYPES

```
dependent def precise() =
  val v1 /*: (v1: (1: Int))*/ = 1
  val v2 /*: (v2: (3: Int))**/ = 2 + v1
  dependent def isString(x: Any) /*: (x : Any) match {
    case String => (true : Boolean)
    case Any => (false : Boolean)
  }*/ = x match
    case _: String => true
    case _ => false
  val v3 /*: (false: Boolean) */ = isString(42)
  val v4 /*: Foo {val x = 42} */ = Foo(42)
```

# VISIBILITY OF ARGUMENTS TO FIELDS MAPPING

```
class D(val items: Seq[Int])
dependent val d /*: D {val items: List[Int]}*/ = D(List(1, 2, 3))
```

Can we really type `D(its)` as `D {val items: its.type }`?

```
class D2(its: Seq[Int]):
  val items: Seq[Int] = its.toList
```

# DEPENDENT CASE CLASSES

```
dependent case class Vec3(size: Int)
val v14: Vec3 {val size: 42} = Vec3(42)
```

Similar to Implement Dependent Class Type #3936

# WHY CASE CLASSES

1. Conceptually similar to structures; it makes sense to consider arguments and fields as the same thing for case classes.
2. Case classes cannot extend other case classes.
3. This plays well with the syntactic sugar `D(1, 2, 3)`.

# SYNTACTIC SUGAR

```
dependent case class Vec4(size: Int)
val v15: Vec4(42)= Vec4(42)
```

# FURTHER WORK

# ALWAYS PRECISE AND WIDEN?

Could we follow for constructors and basic operations the same approach as for if-then-else: always infer a precise type but widen it afterward?

# DISTINCT TERM-LEVEL CONSTRUCTS?

Why not provide different term-level constructs with precise return types?

```
import scala.compiletime.ops.int.+!
val v16 /*: v6.type + v6.type*/ = v6 +! v6
```

```
case class E(x: Int)
val v17 /*: E {val x: 2}*/ = E.dependent(2)
```

# ERROR TYPES?

Both example work with the current prototype!

```
dependent def asString(x: Any) = x match
  case x: String => Some(x)
  case _ => None
val v18 /*: Nothing*/ = asString(42).get
```

```
dependent def asString2(x: Any) = x match
  case x: String => x
  case _ => throw new Error()
val v19 /*: Nothing*/ = asString2(42)
```

Could we also get the precise error message?

# TYPE PARAMETERS?

```
class Vec5[S <: Singleton & Int](size: S)
def sum1[S <: Singleton & Int](a: Vec5[S], b: Vec5[S]) = ???
sum1(Vec5(1), Vec5(2))
```

```
class Vec6[S <: Int @Precise](size: S)
def sum2[S <: Int @Precise](a: Vec6[S], b: Vec6[S]) = ???
sum2(Vec6(1), Vec6(2))
```

```
class Vec7[S <: Int @Singleton](size: S)
def sum3[S <: Int @Singleton](a: Vec7[S], b: Vec7[S]) = ???
sum3(Vec7(1), Vec7(2)) // error
```

# THE END

# ACKNOWLEDGEMENTS


- Thanks to Sébastien Doeraene and Guillaume Martres for the discussions on program elaboration on which section “Why not always infer them?” is based.
- Thanks to Fengyun Liu for his previous work and comments on implementing dependent classes.
- Thanks to Martin Odersky and Viktor Kunčak for their precious feedback.

# REFERENCES

[1] G. S. Schmid, O. Blanvillain, J. Hamza, and V. Kuncak, “Coming to terms with your choices: An existential take on dependent types,” *CoRR*, vol. abs/2011.07653, 2020, Available: https://arxiv.org/abs/2011.07653