Precise type inference for Scala 3

Matt Bovel @LAMP / LARA, EPFL
Supervised by prof. Martin Odersky
July 7, 2022

ABSTRACT

Scala supports a range of expressive types, including singletons, unions, refinements and type-level operations. Unfortunately the usage of these types can be tedious in practice either because they get *widened*, or because they are not inferred at all from term-level expressions, requiring manual casts instead.

We discuss the problems and challenges of inferring more precise types and compare different solutions. In particular, we present a new precise inference mode that can be enabled using a dedicated keyword. We also introduce *dependent case classes*: data classes with fields precisely typed from the arguments passed to their constructors.

1 BACKGROUND

1.1 Widening

Scala's type system has *literal types* (inhabited by a single value known statically), *term-references* types (inhabited by a single value unknown statically) and *union types* (join between arbitrary types).

When no type annotation is present, these type are *widened* when inferring the type of a variable or of a function, which means that they get approximated by a supertype. For instance, in the following snippet, the three right hand-sides have precise types, but get widened to `Int` when inferring the types of the left hand-sides:

```
val x /*: Int*/ = 3 /*: 3*/
val y /*: Int*/ = x /*: x.type*/
val z /*: Int*/ = if c then 1 else 2 /*: 1 | 2*/
```

Note that this only concerns *inferred* types. In these cases, one can still write an explicit type annotation to preserve the precise types.

There are multiple reasons why the types are widened by default:

- **Usability**: at their core, types are *approximations* meant to help developers verify the shape of data they are dealing with. More often than not, types that are too precise would actually make this harder. In most scenarios, the information "this value is an `Int`" is simply more useful that "this value is either `42`, either the value of the variable `Foo.x.z`, which can be `451` or `1984`". Knowing when a more precise type would actually be useful is not trivial, and the Scala compiler will generally prefer familiar wide types to types that would be too detailed.
- **Performance**: keeping all precise types would make the size of types significantly bigger, and with it the time spent traversing them and their memory footprint.
- **Backward-compatibility**: while it might seem at first that the type of a term might always be replaced by one of its subtypes, this is not the case in Scala because types are not only descriptive but also play a central semantic role and impact the elaboration of programs. In section 1.2, we detail two examples: overload resolutions and implicits search.

1.2 Types precision and program elaboration

Theoretically, sub-typing should allow us to replace a variable with type `T` with a value of type `S <: T`. This the case even when throwing inference into the mix as the typer is able to choose the right precision to instantiate the type variable with respect to the sub-typing constraints. In the following example we can therefore safely assign types `A` or `B` to `B()`—or replace it with any value `x: S` where `S <: T`:

```
class A
class B extends A
def f[T](x: T)(y: T => Int) = 0
val b = B()
f(b: A)((a: A) => 0)
f(b: B)((a: A) => 0) // T still instantiated to A
```

This however does not hold anymore when introducing *givens* (a.k.a *implicits*) because the Scala compiler does not guarantee to find a solution if one exists. Instead, the resolution is *best-effort*, using a procedure seeking the right balance between predictability, performance and expression-power.

Importantly, this procedure starts by instantiating type variables that are constrained by arguments to reduce the search space *before* looking for candidate implicits. This is why in the following example, the code compiles if `b` is typed as `A`, but not when if it gets the more precise type `B`:

```
class Inv[X]
given inv: Inv[A] = Inv()
def g[N](x: N)(using Inv[N]) = 42
val b = B()
g(b: A)
g[A](b)
g(b)(using inv)
g(b) // error: no given instance of type Inv[B]
```

Another area where increasing the precision of inferred types can make a previously correct program fail to compile is *overloads resolution*. As background: in Scala, dispatch is dynamic on `this` but static on all other arguments (including on the first argument of functions that are not methods). The compiler chooses the most precise overload depending on the type of the arguments.

Typing an argument with a more precise type can therefore result in a more specific overload to be chosen.

But even more problematically, precising a type can also lead to an ambiguity in overloads resolution. Consider the following setup: let `h` be a function with two overloads respectively taking two unrelated types `C` and `D` as arguments, and let `y` be a variable that can be typed either as `C` or `C & D`. In the first case, the first overload will be chosen, but in the second the compilation will fail.

The following example shows an instance of the problem with `C = Int` and `D = String | 1 | 2` and `C & D = 1 | 2`:

```
def h(x: Int) = "C"
def h(x: String | 1 | 2) = "D"
val cond = false
val y = if cond then 1 else 2
println(h(y))
val preciseY: 1 | 2 = if cond then 1 else 2
println(h(preciseY)) // error: ambiguous overload
```

The consequence of the way implicits search and overloads resolution work is that we cannot increase the precision of type inference for existing constructs without breaking code that was previously compiling. That is the main reason why we present new constructs in sections 2 and 3.

1.3 Precise typing of constructors and basic operations

In addition to types that are inferred from terms but widened afterwards, there are two kind of terms for which precise types would be useful but are currently not inferred at all.

The first one is classes constructors. Traditionally, the return type of the constructor of a class `C` is trivially `C`. However, when a constructor parameter can directly be mapped to the value of a field in the resulting instance, the return type could be more precise by refining the type of the field in question to the type of the argument. In Scala, this can be achieved using a refinement type, noted `C {val a: A}` and representing a subtype of `C` with the addition or overriding of the member `a` with type `A`. The goal is for the following to work:

```
case class C(a: Int)
val c: C {val a: 2} = C(2) // not working
```

The second kind of terms for which more precise types would be useful is basic operations on primary types. For some of them, Scala provides equivalent at the type-level through the `scala.compiletime.ops` package, but these are currently not inferred from term-level operations. In the following example, the `+` on the left hand-side denotes an infix type in the `scala.compiletime.ops.int` package, while the `+` on the right hand-side denotes the term-level operation on integer. The two are currently not related:

```
import scala.compiletime.ops.int.*
val x: Int = 42
val y: x.type + 2 = x + 2 // not working
```

Together, precise typing of classes constructors and basic operations would ease the writing of dependent code, for instance for the classic example of sized lists:

```
case class Vec(size: Int):
  def concat(that: Vec) = Vec(size + that.size)
val v1: Vec {val size: 3} =
  Vec(1).concat(Vec(2)) // not working
```

As of now, this requires explicit casts to work:

```
def preciseVec(s: Int) =
  Vec2(s).asInstanceOf[Vec {val size: s.type}]
def preciseAdd(a: Int, b: Int) =
  (a + b).asInstanceOf[a.type + b.type]
case class Vec2(size: Int):
  def concat(that: Vec2) =
    preciseVec(preciseAdd(size, that.size))
val v2: Vec2 {val size: 3} =
  preciseVec(1).concat(preciseVec(2)) // works
```

2 PRECISE TYPING MODE

As discussed in section 1.2, always inferring precise types is not an option. Instead, we define a special typing mode where:

1. Singleton types and unions are not widened,
2. Classes constructors applications and basic operations are typed precisely.

We propose to enable this mode either with an explicit keyword, or using some heuristics on the result type.

2.1 Dependent methods and values

The precise inference mode is enables using the `dependent` keyword, applicable both to `def`s and `val`s to mean that the right hand-side should be precisely typed:

```
dependent def precise() =
  val x: /*: (x: Int)*/ = 1
  val y: /*: 2 + x.type*/ = 2 + x
```

The `dependent` keyword was first proposed in [1] and our implementation follows a similar but shallower 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".

2.2 Implementation

In the Scala compiler, typing is always done with respect to a `Context`, which among other things contains a set of flags: `Mode`s. We add a `Precise` mode which is checked from 2 places:

- In `Namer.inferredResultType`, we disable widening if `Precise` is set.
- In `Applications.TypedApply`, if `Precise` is set:
  - If the applied function is either a constructor, or the `apply` method of the companion object of a case class, we refine the result type for each argument that corresponds to a public field.
  - If the applied function is a basic operation that has a type-level equivalent, we use it as the result type of the application.

For the precise types to be persisted in Tasty, they need to be introduced by explicit casts. A drawback of this, is that it introduces many `asInstanceOf` in the type trees. For example, `dependent val z = x + x + 2` is transformed by the typer to:

```
dependent val z: (x: Int) + (x: Int) + (5: Int) =
  x.+(x).$asInstanceOf[(x: Int) + (x: Int)]
   .+(5).$asInstanceOf[(x: Int) + (x: Int) + (5: Int)]
```

While the `asInstanceOf` are erased and therefore do not induce a runtime cost, they significantly increase the size of the trees and can impact compilation time.

2.3 Visibility of arguments to fields mapping

In our prototype, the type of any constructor argument that is declared with the `val` keyword is used to refine the result type in precise mode. However, this yields the important question of whenever this information is public or not. As an example, let's consider the following class and constructor application:

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

The refinement gives the guarantee to the user that the implementation of `Seq` used for the field `items` is the same as for the provided argument. While correct in this version, this could change in a future version where the field `items` would not be directly initialized from the constructor argument:

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

It is currently not precised in the Scala specification if the mapping from constructor parameters to fields is public or not. We suggest that this should be the case for case classes only.

As an alternative design, we could type precisely class constructor only if the class is explicitly annotated by its author. We present such a design in section 3.

3 DEPENDENT CASE CLASSES

In this section, we present an other solution for typing classes constructor precisely which is based on a `dependent` keyword at the class-level. This is similar to the approach taken in [1] and builds upon [2].

The constructor of a class annotated as `dependent` will *always* be typed precisely, without anything specific at the call-site:

```
dependent case class Vec3(size: Int)
val v3: Vec3 {val size: 42} = Vec3(42)
```

In term of implementation, this is a more robust solution as it does not require inspection and casts at the call-site; the constructor can be typed once and for all when completing the denotation for the case class and then be used transparently like any other method type with a dependent result.

Furthermore, typing the constructor ensures that other features like type parameters and multiple parameter lists work out-of-the-box and robustly:

```
dependent case class Vec4[T](size: Int)(foo: Int)
val v4: Vec4 {val size: 42} = Vec4[Int](42)(5)
```

We restrict the `dependent` keyword to case classes for three reasons:

1. Case classes are conceptually similar to structures or data classes; it makes sense to consider arguments and fields as the same thing for them (while it would not for classes in general).
2. Case classes cannot extend other case classes; they are flat by design. This simplifies the implementation, as it avoids the need to map the constructor arguments of a class to the arguments of its super-constructor. This information is currently not recorded in class denotations.
3. This plays well with the syntactic sugar described in section 3.1.

3.1 Syntactic sugar

Refinement syntax can be verbose as every field has to be named. To make it easier to use, we introduce a constructor syntactic sugar that refines the fields in the order they appear in the constructor:

```
v4b: Vec4(42) = Vec4(42)
```

Note that this works for case classes as all arguments are also public fields. The same would not hold for general cases where fields can also be constructor-only or private.

4 FURTHER WORK

4.1 Infer match types

Match types [3] are another kind of types that would gain to be inferred automatically to avoid code duplication:

```
type TypeOrdinal[T] = T match {
  case String => 1
  case Int => 2
}
def typeOrdinal[T](t: T): TypeOrdinal[T] = t match
  case _: String => 1
  case _: Int => 2
```

4.2 Extend widening

As discussed in section 1.2, one can not infer more precise types everywhere without impacting program semantics. However, we could try to follow for operations types and constructors a similar approach as for term references and `if–else`: always type them with a precise type, but widen them after inference. With such a scheme however, extra care should be taken to check that the impact on performance is acceptable.

5 CONCLUSIONS

We presented a precise inference mode for Scala, and an alternative way to type case classes precisely which is more robust for this use-case.

Both solutions suffers from the fact that they require adding a new keyword to the language which is a fundamental source of complexity for the end of user. Before settling on such a solution, we would like to explore further solutions such as the one described in section 4.2.

ACKNOWLEDGEMENTS

We would like thank Sébastien Doeraene, Guillaume Martres for the discussions on program elaboration on which section 1.2 is based, Fengyun Liu for his previous work and comments on implementing dependent classes, and 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

[2] F. Liu, "Implement dependent class type." 2018. Available: https://github.com/lampepfl/dotty/pull/3936

[3] O. Blanvillain, J. Brachthäuser, M. Kjaer, and M. Odersky, "Type-level programming with match types," p. 70, 2021, Available: http://infoscience.epfl.ch/record/290019