Given Instances

Given instances (or, simply, "givens") define "canonical" values of certain types that serve for synthesizing arguments to context parameters. Example:

```
trait Ord[T]:
  def compare(x: T, y: T): Int
  extension (x: T) def < (y: T) = compare(x, y) < 0
  extension (x: T) def > (y: T) = compare(x, y) > 0

given intOrd: Ord[Int] with
  def compare(x: Int, y: Int) =
    if x < y then -1 else if x > y then +1 else 0

given listOrd[T](using ord: Ord[T]): Ord[List[T]] with

  def compare(xs: List[T], ys: List[T]): Int = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _) => -1
    case (_, Nil) => +1
    case (x :: xs1, y :: ys1) =>
      val fst = ord.compare(x, y)
      if fst != 0 then fst else compare(xs1, ys1)
```

This code defines a trait `Ord` with two given instances. `intOrd` defines a given for the type `Ord[Int]` whereas `listOrd[T]` defines givens for `Ord[List[T]]` for all types `T` that come with a given instance for `Ord[T]` themselves. The `using` clause in `listOrd` defines a condition: There must be a given of type `Ord[T]` for a given of type `Ord[List[T]]` to exist. Such conditions are expanded by the compiler to context parameters.

Anonymous Givens

The name of a given can be left out. So the definitions of the last section can also be expressed like this:

expressed like this:

```
given Ord[Int] with
  ...
given [T](using Ord[T]): Ord[List[T]] with
  ...
```

If the name of a given is missing, the compiler will synthesize a name from the implemented type(s).

Note The name synthesized by the compiler is chosen to be readable and reasonably concise. For instance, the two instances above would get the names:

```
given_Ord_Int
given_Ord_List
```

The precise rules for synthesizing names are found here. These rules do not guarantee absence of name conflicts between given instances of types that are "too similar". To avoid conflicts one can use named instances.

Note To ensure robust binary compatibility, publicly available libraries should prefer named instances.

Alias Givens

An alias can be used to define a given instance that is equal to some expression. Example:

```
given global: ExecutionContext = ForkJoinPool()
```

This creates a given `global` of type `ExecutionContext` that resolves to the right hand side `ForkJoinPool()`. The first time `global` is accessed, a new `ForkJoinPool` is created, which is then returned for this and all subsequent accesses to `global`. This operation is thread-safe.

Alias givens can be anonymous as well, e.g.

```
given Position = enclosingTree.position
given (using config: Config): Factory = MemoizingFactory(config)
```

An alias given can have type parameters and context parameters just like any other given, but it can only implement a single type.

Given Macros

Given aliases can have the `inline` and `transparent` modifiers. Example:

```
transparent inline given mkAnnotations[A, T]: Annotations[A, T] = ${
  // code producing a value of a subtype of Annotations
}
```

Since `mkAnnotations` is `transparent`, the type of an application is the type of its right-hand side, which can be a proper subtype of the declared result type `Annotations[A, T]`.

Given instances can have the `inline` but not `transparent` modifiers as their type is already known from the signature. Example:

```
trait Show[T] {
  inline def show(x: T): String
}

inline given Show[Foo] with {
  /*transparent*/ inline def show(x: Foo): String = ${ ... }
}

def app =
  // inlines `show` method call and removes the call to `given Show[Foo]`
  summon[Show[Foo]].show(foo)
```

Note that the inline methods within the given instances may be `transparent`.

The inlining of given instances will not inline/duplicate the implementation of the given, it will just inline the instantiation of that instance. This is used to help dead code elimination of the given instances that are not used after inlining.

Pattern-Bound Given Instances

Given instances can also appear in patterns. Example:

```
for given Context <- applicationContexts do

pair match
  case (ctx @ given Context, y) => ...
```

In the first fragment above, anonymous given instances for class `Context` are established by enumerating over `applicationContexts`. In the second fragment, a

given `Context` instance named `ctx` is established by matching against the first half of the `pair` selector.

In each case, a pattern-bound given instance consists of `given` and a type `T`. The pattern matches exactly the same selectors as the type ascription pattern `_: T`.

Negated Givens

Scala 2's somewhat puzzling behavior with respect to ambiguity has been exploited to implement the analogue of a "negated" search in implicit resolution, where a query Q1 fails if some other query Q2 succeeds and Q1 succeeds if Q2 fails. With the new cleaned up behavior these techniques no longer work. But the new special type `scala.util.NotGiven` now implements negation directly.

For any query type `Q`, `NotGiven[Q]` succeeds if and only if the implicit search for `Q` fails, for example:

```
import scala.util.NotGiven

trait Tagged[A]

case class Foo[A](value: Boolean)
object Foo:
  given fooTagged[A](using Tagged[A]): Foo[A] = Foo(true)
  given fooNotTagged[A](using NotGiven[Tagged[A]]): Foo[A] = Foo(false)

@main def test(): Unit =
  given Tagged[Int]()
  assert(summon[Foo[Int]].value) // fooTagged is found
  assert(!summon[Foo[String]].value) // fooNotTagged is found
```

Given Instance Initialization

A given instance without type or context parameters is initialized on-demand, the first time it is accessed. If a given has type or context parameters, a fresh instance is created for each reference.

Syntax

Here is the syntax for given instances:

```
TmplDef             ::=  ...
                      |  'given' GivenDef
GivenDef            ::=  [GivenSig] StructuralInstance
                      |  [GivenSig] AnnotType '=' Expr
```

```
                    |   [GivenSig] AnnotType
GivenSig            ::=  [id] [DefTypeParamClause] {UsingParamClause} ':'
StructuralInstance  ::=  ConstrApp {'with' ConstrApp} 'with' TemplateBody
```

A given instance starts with the reserved word `given` and an optional *signature*. The signature defines a name and/or parameters for the instance. It is followed by `:`. There are three kinds of given instances:

- A *structural instance* contains one or more types or constructor applications, followed by `with` and a template body that contains member definitions of the instance.
- An *alias instance* contains a type, followed by `=` and a right-hand side expression.
- An *abstract instance* contains just the type, which is not followed by anything.

‹ Contex... Using ... ›

Contributors to this page

 pikinier20  nicolasstucki  michelou  BarkingBad  julienrf

 odersky  ShapelessCat  smarter  amsayk  SrTobi