Context Functions

Context functions are functions with (only) context parameters. Their types are *context function types*. Here is an example of a context function type:

```
type Executable[T] = ExecutionContext ?=> T
```

Context functions are written using `?=>` as the "arrow" sign. They are applied to synthesized arguments, in the same way methods with context parameters are applied. For instance:

```
given ec: ExecutionContext = ...

  def f(x: Int): ExecutionContext ?=> Int = ...

  // could be written as follows with the type alias from above
  // def f(x: Int): Executable[Int] = ...

  f(2)(using ec)   // explicit argument
  f(2)             // argument is inferred
```

Conversely, if the expected type of an expression `E` is a context function type `(T_1, ... , T_n) ?=> U` and `E` is not already an context function literal, `E` is converted to a context function literal by rewriting it to

```
(x_1: T1, ..., x_n: Tn) ?=> E
```

where the names `x_1` , ..., `x_n` are arbitrary. This expansion is performed before the expression `E` is typechecked, which means that `x_1` , ..., `x_n` are available as givens in `E` .

Like their types, context function literals are written using `?=>` as the arrow between parameters and results. They differ from normal function literals in that their types

are context function types.

For example, continuing with the previous definitions,

```
def g(arg: Executable[Int]) = ...

  g(22)      // is expanded to g((ev: ExecutionContext) ?=> 22)

  g(f(2))    // is expanded to g((ev: ExecutionContext) ?=> f(2)(using ev))

  g((ctx: ExecutionContext) ?=> f(3))  // is expanded to g((ctx:
ExecutionContext) ?=> f(3)(using ctx))
  g((ctx: ExecutionContext) ?=> f(3)(using ctx)) // is left as it is
```

Example: Builder Pattern

Context function types have considerable expressive power. For instance, here is how they can support the "builder pattern", where the aim is to construct tables like this:

```
table {
    row {
      cell("top left")
      cell("top right")
    }
    row {
      cell("bottom left")
      cell("bottom right")
    }
  }
```

The idea is to define classes for `Table` and `Row` that allow the addition of elements via `add`:

```
class Table:
    val rows = new ArrayBuffer[Row]
    def add(r: Row): Unit = rows += r
    override def toString = rows.mkString("Table(", ", ", ")")

  class Row:
    val cells = new ArrayBuffer[Cell]
    def add(c: Cell): Unit = cells += c
    override def toString = cells.mkString("Row(", ", ", ")")

  case class Cell(elem: String)
```

Then, the `table`, `row` and `cell` constructor methods can be defined with context function types as parameters to avoid the plumbing boilerplate that would otherwise

be necessary.

```
def table(init: Table ?=> Unit) =
    given t: Table = Table()
    init
    t

  def row(init: Row ?=> Unit)(using t: Table) =
     given r: Row = Row()
     init
     t.add(r)

  def cell(str: String)(using r: Row) =
     r.add(new Cell(str))
```

With that setup, the table construction code above compiles and expands to:

```
table { ($t: Table) ?=>

    row { ($r: Row) ?=>
      cell("top left")(using $r)
      cell("top right")(using $r)
    }(using $t)

    row { ($r: Row) ?=>
      cell("bottom left")(using $r)
      cell("bottom right")(using $r)
    }(using $t)
  }
```

Example: Postconditions

As a larger example, here is a way to define constructs for checking arbitrary postconditions using an extension method `ensuring` so that the checked result can be referred to simply by `result`. The example combines opaque type aliases, context function types, and extension methods to provide a zero-overhead abstraction.

```
object PostConditions:
  opaque type WrappedResult[T] = T

  def result[T](using r: WrappedResult[T]): T = r

  extension [T](x: T)
    def ensuring(condition: WrappedResult[T] ?=> Boolean): T =
      assert(condition(using x))
      x
```

```
end PostConditions
import PostConditions.{ensuring, result}


val s = List(1, 2, 3).sum.ensuring(result == 6)
```

Explanations: We use a context function type `WrappedResult[T] ?=> Boolean` as the type of the condition of `ensuring`. An argument to `ensuring` such as `(result == 6)` will therefore have a given of type `WrappedResult[T]` in scope to pass along to the `result` method. `WrappedResult` is a fresh type, to make sure that we do not get unwanted givens in scope (this is good practice in all cases where context parameters are involved). Since `WrappedResult` is an opaque type alias, its values need not be boxed, and since `ensuring` is added as an extension method, its argument does not need boxing either. Hence, the implementation of `ensuring` is close in efficiency to the best possible code one could write by hand:

```
val s =
  val result = List(1, 2, 3).sum
  assert(result == 6)
  result
```

Reference

For more information, see the blog article, (which uses a different syntax that has been superseded).

More details

‹ Multiv... Contex... ›

Contributors to this page

 pikinier20  BarkingBad  julienrf  odersky  michelou

 smarter  robstoll