Parameter Untupling

Say you have a list of pairs

```
val xs: List[(Int, Int)]
```

and you want to map `xs` to a list of `Int`s so that each pair of numbers is mapped to their sum. Previously, the best way to do this was with a pattern-matching decomposition:

```
xs map {
  case (x, y) => x + y
}
```

While correct, this is also inconvenient and confusing, since the `case` suggests that the pattern match could fail. As a shorter and clearer alternative Scala 3 now allows

```
xs.map {
  (x, y) => x + y
}
```

or, equivalently:

```
xs.map(_ + _)
```

and

```
def combine(i: Int, j: Int) = i + j
xs.map(combine)
```

Generally, a function value with `n > 1` parameters is wrapped in a function type of the form `((T_1, ... , T_n)) ⇒ U` if that is the expected type. The tuple parameter

is decomposed and its elements are passed directly to the underlying function.

More specifically, the adaptation is applied to the mismatching formal parameter list. In particular, the adaptation is not a conversion between function types. That is why the following is not accepted:

```
val combiner: (Int, Int) => Int = _ + _
xs.map(combiner)     // Type Mismatch
```

The function value must be explicitly tupled, rather than the parameters untupled:

```
xs.map(combiner.tupled)
```

A conversion may be provided in user code:

```
import scala.language.implicitConversions
transparent inline implicit def `fallback untupling`(f: (Int, Int) => Int): ((
  p => f(p._1, p._2)     // use specialized apply instead of unspecialized `tup
xs.map(combiner)
```

Parameter untupling is attempted before conversions are applied, so that a conversion in scope cannot subvert untupling.

Reference

For more information see:

- More details
- Issue #897.

‹ Open ... Param... ›