Reflection

Reflection enables inspection and construction of Typed Abstract Syntax Trees (Typed-AST). It may be used on quoted expressions (`quoted.Expr`) and quoted types (`quoted.Type`) from Macros or on full TASTy files.

If you are writing macros, please first read Macros. You may find all you need without using quote reflection.

API: From quotes and splices to TASTy reflect trees and back

With `quoted.Expr` and `quoted.Type` we can compute code but also analyze code by inspecting the ASTs. Macros provide the guarantee that the generation of code will be type-correct. Using quote reflection will break these guarantees and may fail at macro expansion time, hence additional explicit checks must be done.

To provide reflection capabilities in macros we need to add an implicit parameter of type `scala.quoted.Quotes` and import `quotes.reflect.*` from it in the scope where it is used.

```
import scala.quoted.*

inline def natConst(inline x: Int): Int = ${natConstImpl('{x})}

def natConstImpl(x: Expr[Int])(using Quotes): Expr[Int] =
  import quotes.reflect.*
  ...
```

Extractors

`import quotes.reflect.*` will provide all extractors and methods on `quotes.reflect.Tree` s. For example the `Literal(_)` extractor used below.

```
def natConstImpl(x: Expr[Int])(using Quotes): Expr[Int] =
  import quotes.reflect.*
  val tree: Term = x.asTerm
  tree match
    case Inlined(_, _, Literal(IntConstant(n))) =>
      if n <= 0 then
        report.error("Parameter must be natural number")
        '{0}
      else
        tree.asExprOf[Int]
    case _ =>
      report.error("Parameter must be a known constant")
      '{0}
```

We can easily know which extractors are needed using `Printer.TreeStructure.show`, which returns the string representation the structure of the tree. Other printers can also be found in the `Printer` module.

```
tree.show(using Printer.TreeStructure)
// or
Printer.TreeStructure.show(tree)
```

The methods `quotes.reflect.Term.{asExpr, asExprOf}` provide a way to go back to a `quoted.Expr`. Note that `asExpr` returns a `Expr[Any]`. On the other hand `asExprOf[T]` returns a `Expr[T]`, if the type does not conform to it an exception will be thrown at runtime.

Positions

The `Position` in the context provides an `ofMacroExpansion` value. It corresponds to the expansion site for macros. The macro authors can obtain various information about that expansion site. The example below shows how we can obtain position information such as the start line, the end line or even the source code at the expansion point.

```
def macroImpl()(quotes: Quotes): Expr[Unit] =
  import quotes.reflect.*
  val pos = Position.ofMacroExpansion

  val path = pos.sourceFile.jpath.toString
  val start = pos.start
  val end = pos.end
  val startLine = pos.startLine
  val endLine = pos.endLine
  val startColumn = pos.startColumn
```

```
  val endColumn = pos.endColumn
  val sourceCode = pos.sourceCode
  ...
```

Tree Utilities

`quotes.reflect` contains three facilities for tree traversal and transformation.

`TreeAccumulator` ties the knot of a traversal. By calling `foldOver(x, tree)(owner)` we can dive into the `tree` node and start accumulating values of type `X` (e.g., of type `List[Symbol]` if we want to collect symbols). The code below, for example, collects the `val` definitions in the tree.

```
def collectPatternVariables(tree: Tree)(using ctx: Context): List[Symbol] =
  val acc = new TreeAccumulator[List[Symbol]]:
    def foldTree(syms: List[Symbol], tree: Tree)(owner: Symbol): List[Symbol] =
      case ValDef(_, _, rhs) =>
        val newSyms = tree.symbol :: syms
        foldTree(newSyms, body)(tree.symbol)
      case _ =>
        foldOverTree(syms, tree)(owner)
  acc(Nil, tree)
```

A `TreeTraverser` extends a `TreeAccumulator` and performs the same traversal but without returning any value. Finally, a `TreeMap` performs a transformation.

ValDef.let

`quotes.reflect.ValDef` also offers a method `let` that allows us to bind the `rhs` (right-hand side) to a `val` and use it in `body`. Additionally, `lets` binds the given `terms` to names and allows to use them in the `body`. Their type definitions are shown below:

```
def let(rhs: Term)(body: Ident => Term): Term = ...

def lets(terms: List[Term])(body: List[Term] => Term): Term = ...
```

‹ Runtim... TASTy I... ›