Inline

Inline Definitions

`inline` is a new soft modifier that guarantees that a definition will be inlined at the point of use. Example:

```
object Config:
  inline val logging = false

object Logger:

  private var indent = 0

  inline def log[T](msg: String, indentMargin: =>Int)(op: => T): T =
    if Config.logging then
      println(s"${"  " * indent}start $msg")
      indent += indentMargin
      val result = op
      indent -= indentMargin
      println(s"${"  " * indent}$msg = $result")
      result
    else op
end Logger
```

The `Config` object contains a definition of the inline value `logging`. This means that `logging` is treated as a *constant value*, equivalent to its right-hand side `false`. The right-hand side of such an `inline val` must itself be a constant expression. Used in this way, `inline` is equivalent to Java and Scala 2's `final`. Note that `final`, meaning *inlined constant*, is still supported in Scala 3, but will be phased out.

The `Logger` object contains a definition of the inline method `log`. This method will always be inlined at the point of call.

In the inlined code, an `if-then-else` with a constant condition will be rewritten to its `then` - or `else` -part. Consequently, in the `log` method above the `if` `Config.logging` with `Config.logging == true` will get rewritten into its `then` -part.

Here's an example:

```
var indentSetting = 2

def factorial(n: BigInt): BigInt =
  log(s"factorial($n)", indentSetting) {
    if n == 0 then 1
    else n * factorial(n - 1)
  }
```

If `Config.logging == false`, this will be rewritten (simplified) to:

```
def factorial(n: BigInt): BigInt =
  if n == 0 then 1
  else n * factorial(n - 1)
```

As you notice, since neither `msg` or `indentMargin` were used, they do not appear in the generated code for `factorial`. Also note the body of our `log` method: the `else-` part reduces to just an `op`. In the generated code we do not generate any closures because we only refer to a by-name parameter *once*. Consequently, the code was inlined directly and the call was beta-reduced.

In the `true` case the code will be rewritten to:

```
def factorial(n: BigInt): BigInt =
  val msg = s"factorial($n)"
  println(s"${"  " * indent}start $msg")
  Logger.inline$indent_=(indent.+(indentSetting))
  val result =
    if n == 0 then 1
    else n * factorial(n - 1)
  Logger.inline$indent_=(indent.-(indentSetting))
  println(s"${"  " * indent}$msg = $result")
  result
```

Note that the by-value parameter `msg` is evaluated only once, per the usual Scala semantics, by binding the value and reusing the `msg` through the body of `factorial`. Also, note the special handling of the assignment to the private var `indent`. It is achieved by generating a setter method `def inline$indent_=` and calling it instead.

Inline methods always have to be fully applied. For instance, a call to

```
Logger.log[String]("some op", indentSetting)
```

would be ill-formed and the compiler would complain that arguments are missing. However, it is possible to pass wildcard arguments instead. For instance,

```
Logger.log[String]("some op", indentSetting)(_)
```

would typecheck.

Recursive Inline Methods

Inline methods can be recursive. For instance, when called with a constant exponent `n`, the following method for `power` will be implemented by straight inline code without any loop or recursion.

```
inline def power(x: Double, n: Int): Double =
  if n == 0 then 1.0
  else if n == 1 then x
  else
    val y = power(x, n / 2)
    if n % 2 == 0 then y * y else y * y * x

power(expr, 10)
// translates to
//
// val x = expr
// val y1 = x * x // ^2
// val y2 = y1 * y1 // ^4
// val y3 = y2 * x // ^5
// y3 * y3 // ^10
```

Parameters of inline methods can have an `inline` modifier as well. This means that actual arguments to these parameters will be inlined in the body of the `inline def`. `inline` parameters have call semantics equivalent to by-name parameters but allow for duplication of the code in the argument. It is usually useful when constant values need to be propagated to allow further optimizations/reductions.

The following example shows the difference in translation between by-value, by-name and `inline` parameters:

```
inline def funkyAssertEquals(actual: Double, expected: =>Double, inline delta:
  if (actual - expected).abs > delta then
```

```
    throw new AssertionError(s"difference between ${expected} and ${actual} was

funkyAssertEquals(computeActual(), computeExpected(), computeDelta())
// translates to
//
// val actual = computeActual()
// def expected = computeExpected()
// if (actual - expected).abs > computeDelta() then
// throw new AssertionError(s"difference between ${expected} and ${actual}
was larger than ${computeDelta()}")
```

Rules for Overriding

Inline methods can override other non-inline methods. The rules are as follows:

1. If an inline method `f` implements or overrides another, non-inline method, the inline method can also be invoked at runtime. For instance, consider the scenario:

   ```
   abstract class A:
     def f: Int
     def g: Int = f

   class B extends A:
     inline def f = 22
     override inline def g = f + 11

   val b = new B
   val a: A = b
   // inlined invocatons
   assert(b.f == 22)
   assert(b.g == 33)
   // dynamic invocations
   assert(a.f == 22)
   assert(a.g == 33)
   ```

   The inlined invocations and the dynamically dispatched invocations give the same results.

2. Inline methods are effectively final.

3. Inline methods can also be abstract. An abstract inline method can be implemented only by other inline methods. It cannot be invoked directly:

   ```
   abstract class A:
     inline def f: Int
   ```

```
object B extends A:
  inline def f: Int = 22

B.f          // OK
val a: A = B
a.f          // error: cannot inline f in A.
```

Relationship to `@inline`

Scala 2 also defines a `@inline` annotation which is used as a hint for the backend to inline code. The `inline` modifier is a more powerful option:

- expansion is guaranteed instead of best effort,
- expansion happens in the frontend instead of in the backend and
- expansion also applies to recursive methods.

The definition of constant expression

Right-hand sides of inline values and of arguments for inline parameters must be constant expressions in the sense defined by the SLS §6.24, including *platform-specific* extensions such as constant folding of pure numeric computations.

An inline value must have a literal type such as `1` or `true`.

```
inline val four = 4
// equivalent to
inline val four: 4 = 4
```

It is also possible to have inline vals of types that do not have a syntax, such as `Short(4)`.

```
trait InlineConstants:
  inline val myShort: Short

object Constants extends InlineConstants:
  inline val myShort/*: Short(4)*/ = 4
```

Transparent Inline Methods

Inline methods can additionally be declared `transparent`. This means that the return type of the inline method can be specialized to a more precise type upon expansion. Example:

```
class A
class B extends A:
```

```
  def m = true

transparent inline def choose(b: Boolean): A =
  if b then new A else new B

val obj1 = choose(true)  // static type is A
val obj2 = choose(false) // static type is B

// obj1.m // compile-time error: `m` is not defined on `A`
obj2.m    // OK
```

Here, the inline method `choose` returns an instance of either of the two types `A` or `B`. If `choose` had not been declared to be `transparent`, the result of its expansion would always be of type `A`, even though the computed value might be of the subtype `B`. The inline method is a "blackbox" in the sense that details of its implementation do not leak out. But if a `transparent` modifier is given, the expansion is the type of the expanded body. If the argument `b` is `true`, that type is `A`, otherwise it is `B`. Consequently, calling `m` on `obj2` type-checks since `obj2` has the same type as the expansion of `choose(false)`, which is `B`. Transparent inline methods are "whitebox" in the sense that the type of an application of such a method can be more specialized than its declared return type, depending on how the method expands.

In the following example, we see how the return type of `zero` is specialized to the singleton type `0` permitting the addition to be ascribed with the correct type `1`.

```
transparent inline def zero: Int = 0

val one: 1 = zero + 1
```

Transparent vs. non-transparent inline

As we already discussed, transparent inline methods may influence type checking at call site. Technically this implies that transparent inline methods must be expanded during type checking of the program. Other inline methods are inlined later after the program is fully typed.

For example, the following two functions will be typed the same way but will be inlined at different times.

```
inline def f1: T = ...
transparent inline def f2: T = (...): T
```

A noteworthy difference is the behavior of `transparent inline given`. If there is an error reported when inlining that definition, it will be considered as an implicit search mismatch and the search will continue. A `transparent inline given` can add a type ascription in its RHS (as in `f2` from the previous example) to avoid the precise type but keep the search behavior. On the other hand, an `inline given` is taken as an implicit and then inlined after typing. Any error will be emitted as usual.

Inline Conditionals

An if-then-else expression whose condition is a constant expression can be simplified to the selected branch. Prefixing an if-then-else expression with `inline` enforces that the condition has to be a constant expression, and thus guarantees that the conditional will always simplify.

Example:

```
inline def update(delta: Int) =
  inline if delta >= 0 then increaseBy(delta)
  else decreaseBy(-delta)
```

A call `update(22)` would rewrite to `increaseBy(22)`. But if `update` was called with a value that was not a compile-time constant, we would get a compile time error like the one below:

```
|  inline if delta >= 0 then ???
|  ^
|  cannot reduce inline if
|   its condition
|     delta >= 0
|   is not a constant value
| This location is in code that was inlined at ...
```

In a transparent inline, an `inline if` will force the inlining of any inline definition in its condition during type checking.

Inline Matches

A `match` expression in the body of an `inline` method definition may be prefixed by the `inline` modifier. If there is enough static information to unambiguously take a branch, the expression is reduced to that branch and the type of the result is taken. If not, a compile-time error is raised that reports that the match cannot be reduced.

The example below defines an inline method with a single inline match expression that picks a case based on its static type:

```
transparent inline def g(x: Any): Any =
  inline x match
    case x: String => (x, x) // Tuple2[String, String](x, x)
    case x: Double => x

g(1.0d) // Has type 1.0d which is a subtype of Double
g("test") // Has type (String, String)
```

The scrutinee `x` is examined statically and the inline match is reduced accordingly returning the corresponding value (with the type specialized because `g` is declared `transparent`). This example performs a simple type test over the scrutinee. The type can have a richer structure like the simple ADT below. `toInt` matches the structure of a number in Church-encoding and *computes* the corresponding integer.

```
trait Nat
case object Zero extends Nat
case class Succ[N <: Nat](n: N) extends Nat

transparent inline def toInt(n: Nat): Int =
  inline n match
    case Zero     => 0
    case Succ(n1) => toInt(n1) + 1

inline val natTwo = toInt(Succ(Succ(Zero)))
val intTwo: 2 = natTwo
```

`natTwo` is inferred to have the singleton type 2.

Reference

For more information about the semantics of `inline`, see the Scala 2020: Semantics-preserving inlining for metaprogramming paper.

‹ Metap... Compil... ›