Implicits

Implicits

The Implicit Modifier

```
LocalModifier  ::= ‘implicit’
ParamClauses   ::= {ParamClause} [nl] ‘(’ ‘implicit’ Params ‘)’
```

Template members and parameters labeled with an `implicit` modifier can be passed to implicit parameters and can be used as implicit conversions called views. The `implicit` modifier is illegal for all type members, as well as for top-level objects.

Example Monoid

The following code defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked implicit.

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
  def add(x: A, y: A): A
}
object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

Implicit Parameters

An *implicit parameter list* `(implicit $p_1$,$\ldots$,$p_n$)` of a method marks the parameters $p_1, \ldots, p_n$ as implicit. A method or constructor can have

only one implicit parameter list, and it must be the last parameter list given.

A method with implicit parameters can be applied to arguments just like a normal method. In this case the `implicit` label has no effect. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided.

The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories. First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition or an implicit parameter. An eligible identifier may thus be a local name, or a member of an enclosing template, or it may be have been made accessible without a prefix through an import clause. If there are no eligible identifiers under this rule, then, second, eligible are also all `implicit` members of some object that belongs to the implicit scope of the implicit parameter's type, T.

The *implicit scope* of a type T consists of all companion modules of classes that are associated with the implicit parameter's type. Here, we say a class C is *associated* with a type T if it is a base class of some part of T.

The *parts* of a type T are:

- if T is a compound type `$T_1$ with $\ldots$ with $T_n$`, the union of the parts of $T_1, \ldots, T_n$, as well as T itself;
- if T is a parameterized type `$S$[$T_1 , \ldots , T_n$]`, the union of the parts of S and $T_1, \ldots, T_n$;
- if T is a singleton type `$p$.type`, the parts of the type of p;
- if T is a type projection `$S$#$U$`, the parts of S as well as T itself;
- if T is a type alias, the parts of its expansion;
- if T is an abstract type, the parts of its upper bound;
- if T denotes an implicit conversion to a type with a method with argument types $T_1, \ldots, T_n$ and result type U, the union of the parts of $T_1, \ldots, T_n$ and U;
- the parts of quantified (existential or universal) and annotated types are defined as the parts of the underlying types (e.g., the parts of `T forSome { ... }` are the parts of `T`);
- in all other cases, just T itself.

Note that packages are internally represented as classes with companion modules to hold the package members. Thus, implicits defined in a package object are part of the implicit scope of a type prefixed by that package.

If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of static overloading resolution. If the parameter has a default argument and no implicit argument can be found the default argument is used.

Example

Assuming the classes from the `Monoid` example, here is a method which computes the sum of a list of elements using the monoid's `add` and `unit` operations.

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

The monoid in question is marked as an implicit parameter, and can therefore be inferred based on the type of the list. Consider for instance the call `sum(List(1, 2, 3))` in a context where `stringMonoid` and `intMonoid` are visible. We know that the formal type parameter `a` of `sum` needs to be instantiated to `Int`. The only eligible object which matches the implicit formal parameter type `Monoid[Int]` is `intMonoid` so this object will be passed as implicit parameter.

This discussion also shows that implicit parameters are inferred after any type arguments are inferred.

Implicit methods can themselves have implicit parameters. An example is the following method from module `scala.List`, which injects lists into the `scala.Ordered` class, provided the element type of the list is also convertible to this type.

```
implicit def list2ordered[A](x: List[A])
  (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
  ...
```

Assume in addition a method

```
implicit def int2ordered(x: Int): Ordered[Int]
```

that injects integers into the `Ordered` class. We can now define a `sort` method over ordered lists:

```
def sort[A](xs: List[A])(implicit a2ordered: A => Ordered[A]) = ...
```

We can apply `sort` to a list of lists of integers `yss: List[List[Int]]` as follows:

```
sort(yss)
```

The call above will be completed by passing two nested implicit arguments:

```
sort(yss)((xs: List[Int]) => list2ordered[Int](xs)(int2ordered))
```

The possibility of passing implicit arguments to implicit arguments raises the possibility of an infinite recursion. For instance, one might try to define the following method, which injects *every* type into the `Ordered` class:

```
implicit def magic[A](x: A)(implicit a2ordered: A => Ordered[A]): Ordered[A] =
  a2ordered(x)
```

Now, if one tried to apply `sort` to an argument `arg` of a type that did not have another injection into the `Ordered` class, one would obtain an infinite expansion:

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

Such infinite expansions should be detected and reported as errors, however to support the deliberate implicit construction of recursive values we allow implicit arguments to be marked as by-name. At call sites recursive uses of implicit values are permitted if they occur in an implicit by-name argument.

Consider the following example,

```
trait Foo {
  def next: Foo
}

object Foo {
  implicit def foo(implicit rec: Foo): Foo =
    new Foo { def next = rec }
}

val foo = implicitly[Foo]
assert(foo eq foo.next)
```

As with the `magic` case above this diverges due to the recursive implicit argument `rec` of method `foo`. If we mark the implicit argument as by-name,

```
trait Foo {
  def next: Foo
}

object Foo {
  implicit def foo(implicit rec: => Foo): Foo =
    new Foo { def next = rec }
}

val foo = implicitly[Foo]
assert(foo eq foo.next)
```

the example compiles with the assertion successful.

When compiled, recursive by-name implicit arguments of this sort are extracted out as val members of a local synthetic object at call sites as follows,

```
val foo: Foo = scala.Predef.implicitly[Foo](
  {
    object LazyDefns$1 {
      val rec$1: Foo = Foo.foo(rec$1)
                       //      ^^^^^
                       // recursive knot tied here
    }
    LazyDefns$1.rec$1
  }
)
assert(foo eq foo.next)
```

Note that the recursive use of `rec$1` occurs within the by-name argument of `foo` and is consequently deferred. The desugaring matches what a programmer would do to construct such a recursive value explicitly.

To prevent infinite expansions, such as the `magic` example above, the compiler keeps track of a stack of "open implicit types" for which implicit arguments are currently being searched. Whenever an implicit argument for type T is searched, T is added to the stack paired with the implicit definition which produces it, and whether it was required to satisfy a by-name implicit argument or not. The type is removed from the stack once the search for the implicit argument either definitely fails or succeeds. Everytime a type is about to be added to the stack, it is checked against existing entries which were produced by the same implicit definition and then,

- if it is equivalent to some type which is already on the stack and there is a by-name argument between that entry and the top of the stack. In this case the search for that type succeeds immediately and the implicit argument is compiled as a recursive reference to the found argument. That argument is added as an entry in the synthesized implicit dictionary if it has not already been added.
- otherwise if the *core* of the type *dominates* the core of a type already on the stack, then the implicit expansion is said to *diverge* and the search for that type fails immediately.
- otherwise it is added to the stack paired with the implicit definition which produces it. Implicit resolution continues with the implicit arguments of that definition (if any).

Here, the *core type* of T is T with aliases expanded, top-level type annotations and refinements removed, and occurrences of top-level existentially bound variables replaced by their upper bounds.

A core type T *dominates* a type U if T is equivalent to U, or if the top-level type constructors of T and U have a common element and T is more complex than U and the *covering sets* of T and U are equal.

The set of *top-level type constructors* $ttcs(T)$ of a type T depends on the form of the type:

- For a type designator, $ttcs(p.c) = \{c\}$;
- For a parameterized type, $ttcs(p.c[targs]) = \{c\}$;
- For a singleton type, $ttcs(p.type) = ttcs(T)$, provided p has type T;
- For a compound type, `$\mathit{ttcs}(T_1$ with $\ldots$ with $T_n)$` $= ttcs(T_1) \cup ... \cup ttcs(T_n)$.

The *complexity* complexity(T) of a core type is an integer which also depends on the form of the type:

- For a type designator, complexity$(p.c) = 1 + \text{complexity}(p)$
- For a parameterized type, complexity$(p.c[targs]) = 1 + \Sigma\,\text{complexity}(targs)$
- For a singleton type denoting a package p, complexity$(p.type) = 0$

- For any other singleton type, complexity($p.type$) $=$ $1 +$ complexity(T), provided p has type T;
- For a compound type, `$\operatorname{complexity}(T_1$ with $\ldots$ with $T_n)$` $= \Sigma$ complexity(T_i)

The *covering set* $cs(T)$ of a type T is the set of type designators mentioned in a type. For example, given the following,

```
type A = List[(Int, Int)]
type B = List[(Int, (Int, Int))]
type C = List[(Int, String)]
```

the corresponding covering sets are:

- $cs(A)$: List, Tuple2, Int
- $cs(B)$: List, Tuple2, Int
- $cs(C)$: List, Tuple2, Int, String

Example

When typing `sort(xs)` for some list `xs` of type `List[List[List[Int]]]`, the sequence of types for which implicit arguments are searched is

```
List[List[Int]] => Ordered[List[List[Int]]],
List[Int] => Ordered[List[Int]],
Int => Ordered[Int]
```

All types share the common type constructor `scala.Function1`, but the complexity of the each new type is lower than the complexity of the previous types. Hence, the code typechecks.

Example

Let `ys` be a list of some type which cannot be converted to `Ordered`. For instance:

```
val ys = List(new IllegalArgumentException, new ClassCastException, new Error)
```

Assume that the definition of `magic` above is in scope. Then the sequence of types for which implicit arguments are searched is

```
Throwable => Ordered[Throwable],
Throwable => Ordered[Throwable],
...
```

Since the second type in the sequence is equal to the first, the compiler will issue an error signalling a divergent implicit expansion.

Views

Implicit parameters and methods can also define implicit conversions called views. A *view* from type S to type T is defined by an implicit value which has

function type `$S$ => $T$` or `(=> $S$) => $T$` or by a method convertible to a value of that type.

Views are applied in three situations:

1. If an expression e is of type T, and T does not conform to the expression's expected type pt. In this case an implicit v is searched which is applicable to e and whose result type conforms to pt. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of `$T$ => $\mathit{pt}$`. If such a view is found, the expression e is converted to `$v$($e$)`.
2. In a selection $e.m$ with e of type T, if the selector m does not denote an accessible member of T. In this case, a view v is searched which is applicable to e and whose result contains a member named m. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection $e.m$ is converted to `$v$($e$).$m$`.
3. In a selection $e.m(args)$ with e of type T, if the selector m denotes some member(s) of T, but none of these members is applicable to the arguments $args$. In this case a view v is searched which is applicable to e and whose result contains a method m which is applicable to $args$. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection $e.m$ is converted to `$v$($e$).$m(\mathit{args})$`.

The implicit view, if it is found, can accept its argument e as a call-by-value or as a call-by-name parameter. However, call-by-value implicits take precedence over call-by-name implicits.

As for implicit parameters, overloading resolution is applied if there are several possible candidates (of either the call-by-value or the call-by-name category).

Example Ordered

Class `scala.Ordered[A]` contains a method

```
def <= [B >: A](that: B)(implicit b2ordered: B => Ordered[B]): Boolean
```

Assume two lists `xs` and `ys` of type `List[Int]` and assume that the `list2ordered` and `int2ordered` methods defined here are in scope. Then the operation

```
xs <= ys
```

is legal, and is expanded to:

```
list2ordered(xs)(int2ordered).<=
  (ys)
  (xs => list2ordered(xs)(int2ordered))
```

The first application of `list2ordered` converts the list `xs` to an instance of class `Ordered`, whereas the second occurrence is part of an implicit parameter passed to the `<=` method.

Context Bounds and View Bounds

```
TypeParam ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:' Type]
              {'<%' Type} {':' Type}
```

A type parameter A of a method or non-trait class may have one or more view bounds `$A$ <% $T$`. In this case the type parameter may be instantiated to any type S which is convertible by application of a view to the bound T.

A type parameter A of a method or non-trait class may also have one or more context bounds `$A$ : $T$`. In this case the type parameter may be instantiated to any type S for which *evidence* exists at the instantiation point that S satisfies the bound T. Such evidence consists of an implicit value with type $T[S]$.

A method or class containing type parameters with view or context bounds is treated as being equivalent to a method with implicit parameters. Consider first the case of a single parameter with view and/or context bounds such as:

```
def $f$[$A$ <% $T_1$ ... <% $T_m$ : $U_1$ : $U_n$]($\mathit{ps}$): $R$ = ...
```

Then the method definition above is expanded to

```
def $f$[$A$]($\mathit{ps}$)(implicit $v_1$: $A$ => $T_1$, ..., $v_m$: $A$ => $T_m$,
                        $w_1$: $U_1$[$A$], ..., $w_n$: $U_n$[$A$]): $R$ = ...
```

where the v_i and w_j are fresh names for the newly introduced implicit parameters. These parameters are called *evidence parameters*.

If a class or method has several view- or context-bounded type parameters, each such type parameter is expanded into evidence parameters in the order they appear and all the resulting evidence parameters are concatenated in one implicit parameter section. Since traits do not take constructor parameters, this translation does not work for them. Consequently, type-parameters in traits may not be view- or context-bounded.

Evidence parameters are prepended to the existing implicit parameter section, if one exists.

For example:

```
def foo[A: M](implicit b: B): C
// expands to:
// def foo[A](implicit evidence$1: M[A], b: B): C
```

Example

The `<=` method from the `Ordered` example can be declared more concisely as follows:

```
def <= [B >: A <% Ordered[B]](that: B): Boolean
```

Manifests

Manifests are type descriptors that can be automatically generated by the Scala compiler as arguments to implicit parameters. The Scala standard library contains a hierarchy of four manifest classes, with `OptManifest` at the top. Their signatures follow the outline below.

```
trait OptManifest[+T]
object NoManifest extends OptManifest[Nothing]
trait ClassManifest[T] extends OptManifest[T]
trait Manifest[T] extends ClassManifest[T]
```

If an implicit parameter of a method or constructor is of a subtype $M[T]$ of class `OptManifest[T]`, *a manifest is determined for* $M[S]$, according to the following rules.

First if there is already an implicit argument that matches $M[T]$, this argument is selected.

Otherwise, let $Mobj$ be the companion object `scala.reflect.Manifest` if M is trait `Manifest`, or be the companion object `scala.reflect.ClassManifest` otherwise. Let M' be the trait `Manifest` if M is trait `Manifest`, or be the trait `OptManifest` otherwise. Then the following rules apply.

1. If T is a value class or one of the classes `Any`, `AnyVal`, `Object`, `Null`, or `Nothing`, a manifest for it is generated by selecting the corresponding manifest value `Manifest.$T$`, which exists in the `Manifest` module.
2. If T is an instance of `Array[$S$]`, a manifest is generated with the invocation `$\mathit{Mobj}$.arrayType[S](m)`, where m is the manifest determined for $M[S]$.
3. If T is some other class type $S\#C[U_1, \ldots, U_n]$ where the prefix type S cannot be statically determined from the class C, a manifest is generated with the invocation `$\mathit{Mobj}$.classType[T]($m_0$, classOf[T], $ms$)` where m_0 is the manifest determined for $M'[S]$ and ms are the manifests determined for $M'[U_1], \ldots, M'[U_n]$.
4. If T is some other class type with type arguments $U_1, \ldots, U_n$, a manifest is generated with the invocation `$\mathit{Mobj}$.classType[T](classOf[T], $ms$)` where ms are the manifests determined for $M'[U_1], \ldots, M'[U_n]$.
5. If T is a singleton type `$p$.type`, a manifest is generated with the invocation `$\mathit{Mobj}$.singleType[T]($p$)`
6. If T is a refined type $T'\{R\}$, a manifest is generated for T'. (That is, refinements are never reflected in manifests).
7. If T is an intersection type `$T_1$ with $, \ldots ,$ with $T_n$` where $n > 1$, the result depends on whether a full manifest is to be determined or not. If M is trait `Manifest`, then a manifest is generated with the invocation `Manifest.intersectionType[T]($ms$)` where ms are the manifests determined for $M[T_1], \ldots, M[T_n]$. Otherwise, if M is trait `ClassManifest`, then a manifest is generated for the intersection dominator of the types

$T_1, \ldots, T_n$.

8. If T is some other type, then if M is trait `OptManifest`, a manifest is generated from the designator `scala.reflect.NoManifest`. If M is a type different from `OptManifest`, a static error results.