Note: This SIP is not up to date. For a better alternative to specialization, check out miniboxing: scala-miniboxing.org

SIP: Changes to the Specialization Phase

This SIP consists of several proposed improvements which are conceptually orthogonal:

- the mechanism for generating the specialized classes (now traits)
- selective specialization (@specializeOn, Group, SameAs, FewerBytes, MoreBytes)

Selective Specialization (@specializeOn, Group, SameAs, FewerBytes, MoreBytes)

Value Type Specialization

Instead of using the `@specialized` parameters, we should use the following construct:

```
@specializeOn(Int, Int)
@specializeOn((Int, Long, Float, Double), (Unit, AnyRef))
@noSpecializeOn(Int, Unit)
class C[@specialized T, @specialized U] { … }
```

The following annotations are introduced:

- `@specializeOn(typeGroup1, … typeGroupN)` -- specialize the following definition's N type parameters with the cartesian product of typeGroup1 x typeGroup2 x … x typeGroup N
- `@noSpecializeOn(typeGroup1, … typeGroupN)` -- do not specialize the following definition's N type parameters on any of the elements in the cartesian product of the groups

Group specification:

- groups are implemented by tupling types
  - `(type1, type2, … typeK)` is a group of K value types
- special groups are:
  - `SameAs(M)` - denotes a parameter that takes the same value as the M-th parameter

    - `FewerBytes(M1, M2, … Mn)` - denotes a group of all type parameters that have fewer bytes in their representation[1] then any of the M1, M2, … Mn-th parameters. Note the relation is strict, so FewerBytes((Int)) is does not include Int itself.
    - `MoreBytes(M1, M2, … Mn)` - the opposite of FewerBytes, again strict.
- Examples
    - `@specializeOn((Int, Double), (SameAs(1), Unit))`
        - `(Int, Int)`
        - `(Int, Unit)`
        - `(Double, Double)`
        - `(Double, Int)`
    - `@specializeOn(Int, FewerBytes(1))`
        - `(Int, Short)`
        - `(Int, Char)`
        - `(Int, Byte)`
        - `(Int, Boolean)`
        - `(Int, Unit)`

Operator precedence:

- all specializeOn groups are computed - P(specializeOn)
- all noSpecializeOn groups are computed - P(noSpecializeOn)
- the specialization will be done on the set P(specializeOn) / P(noSpecializeOn)

Interaction with previous specialized types definition:

- `@specialize(types*)` becomes deprecated, with warning
- `@specializeOn` takes precedence over `@specialize(types*)`
- `@noSpecializeOn` takes precedence over `@specialize(types*)`

Specialized member resolution

If multiple specializations are available but none of them fit the current types, there are multiple options on resolving the specialization:

- call the generic (AnyRef everywhere) implementation - current strategy
- call the first match, considering boxing some of the types
- call the best match, which boxes the minimum number of types
    - example:
        - given specializations:
            - `Int, AnyRef, AnyRef`
            - `Int, Int, AnyRef`
            - `Int, Int, Int`

[1] As defined by AnyRef < Double < Float < Long < Int < Short < Char < Byte < Boolean < Unit, to suggest less information contained in the value type

        - for the `(Int, Int, Byte)` the specializer should choose
            - `Int, Int, AnyRef`, as a single element is not boxed
    - breaking ties:
        - heuristic: usually the first type parameters represent input types while the last ones represent outputs. The best output is obtained if computation takes place with unboxed elements, thus it is best to unbox the first type parameters as much as possible
        - resolution:
            - step1: take the specialized implementation that matches the most type parameters, not counting Unit, for which boxing does not cost anything
            - step2: for the same number of matched type parameters, the implementation picked should be the one with the minimum sum of indices of the matched type parameters, excluding Unit
            - step3: should multiple such specialized implementations exist, pick the first one in the list

References:

- scala-internals discussion

Related bugs:

- https://issues.scala-lang.org/browse/SI-5507 - should be fixed by @noSpecializeOn
- https://issues.scala-lang.org/browse/SI-5442 - avoid nonsensical values for the type params (Function2[Unit, Unit, *])

New Mechanism for Generating Specialized Classes

<Description>

Specialized Member Specification (@specialize/@nospecialize)

Specialization of class members should be overridden by @specialize and @nospecialize method/field annotations.

TODO: Describe how to implement this.

References:

- scala-debate discussion
- scala-internals discussion

Related bugs: