The `mixin` phase

© Miguel Garcia, LAMP, EPFL
http://lamp.epfl.ch/~magarcia

October 10th, 2011

Abstract

Now that we understand the workings of `AddInterfaces`, how difficult can it be to understand `mixin`? As a warm-up, Sec. 1 gives a bird's eye view of this phase (in terms of input and output AST shapes, and a summary of the transformation). A detailed description comes next, structured along the main stages that the transformation comprises (pre and post transforms, as well as type rewriting in `transformInfo`). In order to ease things *even further*, Sec. 3 checkpoints the status of ASTs in-between pre and post transforms.

```
     phase name id  description
     ---------- --  -----------
         parser  1  parse source into ASTs, perform simple desugaring
          namer  2  resolve names, attach symbols to named trees
 packageobjects  3  load package objects
          typer  4  the meat and potatoes: type the trees
 superaccessors  5  add super accessors in traits and nested classes
        pickler  6  serialize symbol tables
      refchecks  7  reference/override checking, translate nested objects
       liftcode  8  reify trees
        uncurry  9  uncurry, translate function values to anonymous classes
      tailcalls 10  replace tail calls by jumps
     specialize 11  @specialized-driven class and method specialization
  explicitouter 12  this refs to outer pointers, translate patterns
        erasure 13  erase types, add interfaces for traits
       lazyvals 14  allocate bitmaps, translate lazy vals into lazified defs
     lambdalift 15  move nested functions to top level
   constructors 16  move field definitions into constructors
        flatten 17  eliminate inner classes
/*-----------------------------------------------------------------------*/
          mixin 18  mixin composition
/*-----------------------------------------------------------------------*/
        cleanup 19  platform-specific cleanups, generate reflective calls
          icode 20  generate portable intermediate code
                 . . .
```

Contents

1 Overview

1.1 A note on terminlogy

Adding to the terminology inherited from `AddInterfaces`, some more:

- **composite class** is short for "*a non-trait class with some non-interface traits among its `mixinClasses`*". A finer distinction is *class-to-be-composed* (before the `mixin` transform) vs. *composed-class* (after `mixin`). In these notes, "composite class" refers to both.

- **implementation module**: what an impl-class becomes after `mixin`.

1.2 Input shapes

In essence, `mixin` assumes that `AddInterfaces` has splitted non-interface traits and that `constructors` has rephrased template-initialization in terms of VM-level fields + constructors. In particular, `AddInterfaces` has transformed:

- Non-interface traits, making them look as described in Sec. 1.2.1.

- Composite classes, making them look as described in Sec. 1.2.2

1.2.1 What used to be non-interface traits

Non-interface traits have vanished, each has been split by `AddInterfaces` into:

1. an interface-only facet, containing abstract methods (for super-accessors and for public trait-members) and nested `ClassDef`s unless `flatten` has run (in particular module classes resulting from `refchecks` desugaring of objects)

2. an implementation class containing:

   (a) an `$init$` constructor (that name is reserved for *the* constructor of an implementation class),

   (b) all non-public members that used to reside in the non-interface trait, i.e. private fields and private methods.

   (c) so called "implementation methods", which have an abstract counterpart in the iface-facet. Once rewritten, they will be targeted by forwarder methods in composed classes.

Regarding parents and subtypes:

- both interface and implementation facets have `Object` as super-class and `ScalaObject` as super-interface.

- an implementation class may also extend other implementation classes (this will be fixed up by `mixin`), it extends its interface-facet counterpart (implementing zero, some, or all of that interface's methods other than super-accessors), and may also extend any interface-only traits that the non-interface trait extended;

- an interface facet may extend other interface facets as well as any interface-only traits that the non-interface trait extended. A class (whether trait or not) that used to extend a non-interface trait now extends its interface facet.

1.2.2 What composed-classes will look like

A composite class was rewritten by `AddInterfaces` to some extent. Besides being made to inherit interface-facets, its primary constructor got trait-init-calls added (precisely for those non-interface traits among its `mixinClasses`). However, these callsites to trait-inits don't yet pass a self-param (nor do trait-inits have a formal param for it).

A note on typing: although at this point a composite class C lacks in general implementations for methods in interfaces it extends (unless C happens to override them), it still type checks because what we call "interface facets" are still internally `ClassSymbols` with the `TRAIT` flag set (albeit also with the `ABSTRACT` flag). Therefore the typechecker doesn't complain C doesn't support all methods in all inherited interface facets. But the VM would.

1.3 Output shapes

There's work to do before classes can be handed over to `cleanup`:

- **The "breaking impl-class apart" perspective**: all implementation methods end up in the implementation module (an impl-module contains only VM-level static methods, with an extra self-param as first argument, and with their bodies rewritten to access private data not on `This` but via the self-param.). And finally, some private members will be copied to each composed class (private fields) while others end up in the implementation module (helper methods invoked from implementation methods).

- **The "pasting into a composed-class" perspective**: in a composite class, forwarders to implementation methods are missing, as well as own copies of private fields.

1.4 End-to-end transformation

The division of labor between `preTransform` and `postTransform` revolves around limiting AST typing (i.e., the invocation of `localTyper.typed` or `typedPos`) to `postTransform` (which runs under `atPhase(phase.next)`), once certain `info.decls` have grown by a few symbols appended in-place during `preTransform` (Sec. 2.1 to Sec. 2.3).

Other than that, the division of labor involves a limited amount of term rewriting during `preTransform` (Sec. 2.4 and Sec. 2.5), with the heavy part of term rewriting done in `postTransform`.

The same mnemonics as for `AddInterfaces`: rather than sticking with `clazz`, `sym`, and so on, you may try instead with `compoClazz`, `ifaceFacet`, `implClazz`, `compoMember`, etc. Makes wonders for readability (less context to keep in mind).

```
TODO a brief overview of postTransform
```

2 At current phase ("preTransform")

At this point in the pipeline, classes can be classified into future-VM-classes (those for which `!csym.isTrait`, including in particular implementation classes) and those ready as VM-interfaces (`csym.isTrait`). Among the latter, the `lateINTERFACE` ones resulted from trait splitting. The others were interface-only traits to start with.

In a nutshell, the processing during `preTransform` comprises:

1. in-place additions to `info.decls`, for interface facets (Sec. 2.1) and for composite classes (Sec. 2.2 and Sec. 2.3).

   The symbols thus added correspond to `Tree` terms that are yet to be added by `addNewDefs` in the `case Template` of `postTransform` (Sec. 5). Moreover, `addNewDefs` decides what trees to paste based on these symbols, which can be recognized by their `Flags.MIXEDIN`.

2. term rewriting focuses mostly on implementation classes (Sec. 2.4) but other rewritings are also carried out (Sec. 2.5).

2.1 Type rewriting for interface facets: Adding symbols for accessors to composite-owned fields

```
/** Add getters and setters for all non-module fields of an implementation
 *  class to its interface unless they are already present. This is done
 *  only once per class. The mixedin flag is used to remember whether late
 *  members have been added to an interface.
 *    - lazy fields don't get a setter.
 */
def addLateInterfaceMembers(clazz: Symbol) {
```

Implementation methods require in general access to fields of a composed class (including private ones) via a self-param (because all an impl-module knows about a self-param is its type, an iface-facet). For that, `MethodSymbols` have to be appended to the interface facet's `info.decls` (at the current phase). Operationally, `addLateInterfaceMembers` visits the val/var symbols in the impl-class (skipping module-vars) and creates `lateDEFERRED` getters and setters in the interface facet (actually, a setter isn't added for val/vars of lazy fields or for those having constant type).

There's no map tracking "which compo-class-field for this iface-accessor" because that lookup can be performed via "`member.getter(ifaceFacet)`".

2.2 Type rewriting for composites (1 of 2): Completing support for interface facets

```
/** Mix in members of trait ifaceFacet into class clazz. Also,
 *  for each lazy field in ifaceFacet, add a link from its mixed in member to its
 *  initializer method inside the implclass.
 */
def mixinTraitMembers(ifaceFacet: Symbol) {
```

This sub-step is part of `addMixedinMembers` (the other sub-step is described in Sec. 2.3). Taken together, they add `MethodSymbols` to the `info.decls` of a composite class (based on those in an interface-facet and those in an implementation class, resp.)

In this first sub-step, symbols for concrete accessors, super-accessors, and modules are created and added after inspecting an interface-facet. As a sidenote: "`ifaceFacet.info.decls`" already contains accessors for composite-owned fields (Sec. 2.1).

1. **Mixing-in a concrete accessor (including back-up field and lazy-initializer-tracking, if needed)** consists in creating a composite-owned `MethodSymbol` by "cloning before erasure", which sets the type history of the resulting clone at two points: erasure and current phase. As with other cloning, the original symbol becomes alias of the resulting clone (in the other direction, the alias of the original symbol remains unchanged).

   That's the basic procedure for all concrete accessors. For lazy accessors and for getters, additional work is needed as described next.

   - For a lazy getter, the initializer is looked up in the impl-class and tracked as a pair *(new-compo-member, impl-owned-init)* in the map:

```
/** Map a lazy, mixedin field accessor to it's trait member accessor */
val initializer = perRunCaches.newMap[Symbol, Symbol]
```

   - For getters, its return type hints at whether we should mix-in the back-up field as well (constant or `Unit`: don't try to mix it in). Otherwise add a new value symbol to the composite class's `info.decls`.

2. **Mixing-in a super-accessor** consists in creating a composite-owned symbol by cloning the interface-member in question. To recap from `superaccessors`, the alias of an interface-owned super-selector symbol records the symbol of the super-ref expression `Select(sup @ Super(_,mix), name)`. Now that the actual super-type is known ("`base`" below is the composed class) that alias is used to find out the symbol that is accessed by a super-accessor in a mixin composition. That symbol becomes the alias of the composite-owned super-accessor.

```
/** Returns the symbol that is accessed by a super-accessor in a mixin composition.
 *
 *  @param base       The class in which everything is mixed together
 *  @param member     The symbol statically referred to by the superaccessor in the trait
 *  @param mixinClass The mixin class that produced the superaccessor
 */
private def rebindSuper(base: Symbol, member: Symbol, mixinClass: Symbol): Symbol =
```

3. **Mixing-in a module getter**: Unlike for concrete accessors, no "cloning before erasure" nor special processing for lazy val nor back-up fields is done. Unlike for super-accessors no alias bookkeeping is needed. Instead, the symbol of an interface-owned module getter is cloned and added to the composed class's `info.decls`.

2.3 Type rewriting for composites (2 of 2): Adding symbols for forwarders

```
/** Mix in members of implementation class mixinClass into class clazz */
def mixinImplClassMembers(impl: Symbol, iface: Symbol) {
```

The alias of the composite-owned forwarder is made to be the impl-owned forwarded `MethodSymbol`.

```
for (implMember <- implClazz.info.decls) {
  if (isForwarded(implMember)) {
    val ifaceMember = implMember.overriddenSymbol(ifaceFacet)
    if (ifaceMember.overridingSymbol(compoClazz) == NoSymbol &&
        compoClazz.info.findMember(implMember.name, 0, lateDEFERRED, false).alternatives.contains(ifaceMember)) {
          val compoMember = addMember(
            compoClazz,
            cloneBeforeErasure(ifaceFacet, compoClazz, ifaceMember)
              setPos compoClazz.pos
              resetFlag (DEFERRED | lateDEFERRED))
          compoMember.asInstanceOf[TermSymbol] setAlias implMember;
        }
  }
}
```

2.4 Term rewriting (1 of 2): Implementation classes

Not every method in the impl-class will end up in the implementation module, and those that do will undergo rewriting. Most of the `transformInfo` has to do with these adjustments (Sec. 4).

The following rewritings affect different aspects of the `ClassDef` of an implementation class, they are performed in the `match` expression at `preTransform()`:

First, what gets elided:

- all fields are elided, because they're composite-owned after `mixin`. Symbols for its accessors were added to the iface-facet (Sec. 2.1) and to the composite class (including back-up field, Sec. 2.2).

```
TODO what happens with the trees of accessors that contain references to their symbols.
    addNewDefs can reconstruct their bodies based only on symbols, right?
```

- methods that belong in the composite class are elided (such methods "aren't statically implemented").

```
TODO addNewDefs must add them, as part of case Template in postTransform.
```

Definition of "*Not statically implemented*":

- *non-private modules: these are implemented directly in the mixin composition class (private modules, on the other hand, are implemented statically, but their module variable is not. all such private modules are lifted, because non-lifted private modules have been eliminated in* ***`ExplicitOuter`****)*

- *field accessors and super-accessors, except for lazy value accessors which become initializer methods in the impl class (because they can have arbitrary initializers)*

But, not everything is elided from the implementation class. Those `DefDef`s whose `msym` satisfies `isImplementedStatically(msym)` stay all the way to the implementation module (those are all the members an impl-module will have). The `DefDef`s in question have their formal params adjusted, to accomodate the `nme.SELF` param (i.e., "`$this`") that is the hallmark of forwarded methods. The type of this param is the interface facet (thus the symbols for concrete accessors added to that interface, Sec. 2.2). The body of impl-module-methods is not transformed at this point, but `transformInfo` doesn't overlook updating the corresponding `MethodSymbol.info` (Sec. 4).

2.5 Term rewriting (2 of 2): The rest

1. In an interface-facet, a setter that was concrete in the non-interface trait is marked with the `scala.runtime.TraitSetter` annotation.

```
TODO why?
```

2. Type tests and casts are adjusted from impl-class to iface-facet. Actually, `tpe`s are adjusted in-place, `symbol.info`s will take care of themselves:

```
case Apply(tapp @ TypeApply(fn, List(arg)), List()) =>
  if (arg.tpe.typeSymbol.isImplClass) {
    val ifacetpe = toInterface(arg.tpe)
    arg.tpe = ifacetpe
    tapp.tpe = MethodType(List(), ifacetpe)
    tree.tpe = ifacetpe
  }
  tree
```

```
TODO Details about arg.symbol
```

3 AST changes so far: a checklist

- **Interface facet**: its `info.decls` contains symbols for super-accessors and for public trait-members (already there on entry to `mixin`, Sec. 1.2.1), as well as symbols for accessors to composite-owned fields (private or not), lazy-getters (TODO: what about module-getters) (added in `preTransform`, Sec. 2.1).
- **Implementation class**:
  - The `info.decls` of an impl-class symbol contained a number of things on entry to `mixin` (as summarized in Sec. 1.2.1: `$init$` constructor and what used to reside in the non-interface trait, which can be classified into non-public members and "implementation methods") but most of that is gone due to `transformInfo` (however, the `$init$` constructor stays, Sec. 4) which also removes all parents:

```
parents1 = List()
decls1 = new Scope(decls.toList filter isImplementedStatically)
```

- The template of an impl-class `ClassDef` contained on entry what's summarized above, but after `preTransform` it contains only the `$init$` constructor, impl-module methods (Sec. 2.4), and nested `ClassDef`s (unless `flatten` has run). Regarding the template's parents list, so far it hasn't changed, but `postTransform` will rebuild it to match what the `ClassInfoType` states:

```
val parents1 = currentOwner.info.parents map (t => TypeTree(t) setPos tree.pos)
```

- **Composite class**:
  - The `info.decls` of a composite-class contains what it already had on entry to `mixin` (Sec. 1.2.2) as well as symbols for field-accessors, lazy-getters, module-getters, back-up fields, super-accessors, and forwarders (all of them concrete, added in `preTransform` for each `lateINTERFACE` parent among `mixinClasses`, Sec. 2.2 and Sec. 2.3).
  - The `ClassDef` of a composite-class hasn't changed so far.

4 transformInfo

The effects on type history of `transformInfo()` are most noticeable during `postTransform()` (Sec. 5). Given that some type rewriting was performed in-place during `preTransform()` (Sec. 2.1 to Sec. 2.3) there's not much left to do here.

A sweeping change affects the parents of all non-interface classes (except implementation classes): any impl-class parents they might have are replaced with their interface-facet counterpart.

```
parents1 = parents.head :: (parents.tail map toInterface)
```

```
TODO Does this change any parent at all?
     I.e. can an impl-class be parent to any class other than an impl-class?

     In case it does result in updated parents,
     the corresponding term rewriting "will be done" in the case Template of postTransform.
```

The remaining changes involve implementation classes only:

- the signature of a forwarded method needs a self-param:

```
case MethodType(params, restp) =>
  toInterfaceMap(
    if (isImplementedStatically(sym)) {
      val ownerParam = sym.newSyntheticValueParam(toInterface(sym.owner.typeOfThis))
      MethodType(ownerParam :: params, restp)
    } else
      tp)
```

- the `Scope` of an impl-class keeps only forwarded methods

```
decls1 = new Scope(decls.toList filter isImplementedStatically)
```

- the impl-class is turned into a module (more precisely, a `lateMODULE`, but this flag is never checked), all parents are elided (no more `Object`, no more `ScalaObject`, either). `sourceModule` may have to be fabricated too.

```
parents1 = List()
```

The term-rewriting counterpart for Items 1 and 2 can be found in Sec. 2.4. That for Item 3 is performed in the `case Template` of `postTransform`, Sec. 5.

5 At next phase ("postTransform")

This step is the last chance to do term rewriting on a node: the input AST comes from `super.transform`-ing a `preTransform`-ed tree (i.e., all children have been visited) and neither `postTransform` nor its helpers trigger recursion into children (i.e., children have been visited for good).

```
TODO Confirm that postTransform does not recurses via transform() into children
      (this implies that its helpers also don't).
```

Most of the screen real state associated to this step has to do with lazy values (these notes don't cover that).

```
TODO Is this step only about term rewriting?
      If so, then one can assume symbol.info to remain stable during postTransform.
```

```
TODO Summary of subsections.
```

5.1 Super-refs

Static super-refs stay as-is, trait-level super-refs are callsites by now (thanks to `superaccessors`), other super-refs handled by Sec. 5.4.

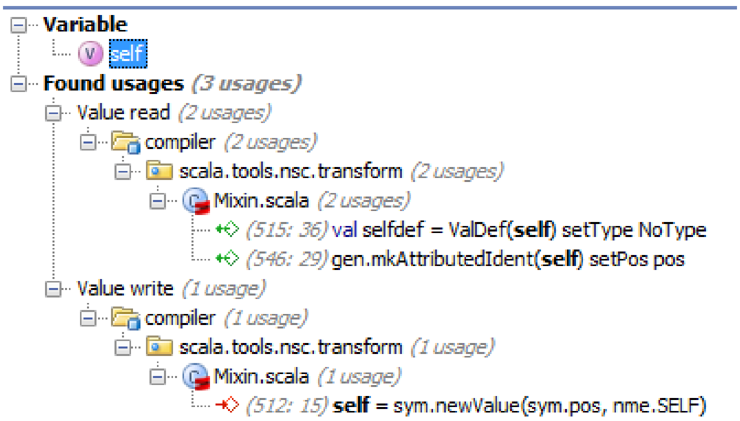
```
case Select(Super(_, _), name) =>
  tree
```

5.2 When in implementation-module, detour `This` to current self-param

```
case This(_) =>
  transformThis(tree)
```

```
/** Replace a this reference to the current implementation class by the self
 *  parameter. Leave all other trees unchanged.
```

```
 */
private def transformThis(tree: Tree) = tree match {
  case This(_) if tree.symbol.isImplClass =>
    assert(tree.symbol == currentOwner.enclClass)
    selfRef(tree.pos) /*- i.e. gen.mkAttributedIdent(self) setPos pos */
  case _ =>
    tree
}
```



5.3 Accesses to composite-owned fields over self-param

The `case` handlers below rewrite a `Select` or `Assign` into a callsite instead. The callsite is pieced together from a `MethodSymbol` that is looked up on the iface-facet, while the receiver is taken to be the `Select`'s qualifier. That qualifier, for the self-case, was already rewritten as discussed in Sec. 5.2.

- reference to composite-owned-field via abstract iface-facet-owned getter

```
case Select(qual, name) if sym.owner.isImplClass && !isStaticOnly(sym) =>
  assert(!sym.isMethod, "no method allowed here: %s%s %s".format(sym, sym.isImplOnly, flagsToString(sym.fl

  // refer to fields in some implementation class via an abstract
  // getter in the interface.
  val iface = toInterface(sym.owner.tpe).typeSymbol
  val getter = sym.getter(iface)
  assert(getter != NoSymbol)
  typedPos(tree.pos)((qual DOT getter)())
```

- assignment to composite-owned-field via abstract iface-facet-owned setter

```
case Assign(Apply(lhs @ Select(qual, _), List()), rhs) =>
  // assign to fields in some implementation class via an abstract
  // setter in the interface.
  def setter = lhs.symbol.setter(
    toInterface(lhs.symbol.owner.tpe).typeSymbol,
    needsExpandedSetterName(lhs.symbol)
  ) setPos lhs.pos

  typedPos(tree.pos) { (qual DOT setter)(rhs) }
```

5.4 Callsites re-targeted to impl-module-methods

```
case Apply(Select(qual, _), args) =>
  /** Changes `qual.m(args)` where m refers to an implementation class method
   *  to `Q.m(S, args)` where Q is the implementation module of `m`
   *  and S is the self parameter for the call, which is determined as follows:
   *     - if qual != super, qual itself
   *     - if qual == super,
   *         - if we are in an implementation class, the current self parameter.
   *         - otherwise, `this`
   */
```

An invocation to a trait-method can appear in two places: a composed class or the impl-module itself.

- Callsites in a composed class
  - those were there on entry to `mixin` are part of user-written code, they don't result from any `mixin` reshuffling. They target already the forwarded method. Therefore, they are left as-is.
  - those in the body of a forwarder ...

    ```
    TODO
    ```

- Callsites in the impl-module itself are rewritten as described in the source comment above.

5.5 Appending term trees to interface facets and composites

```
/** Add all new definitions to a non-trait class
*  These fall into the following categories:
*    - for a trait interface:
*       - abstract accessors for all fields in the implementation class
*    - for a non-trait class:
*       - A field for every in a mixin class
*       - Setters and getters for such fields
*           - getters for mixed in lazy fields are completed
*       - module variables and module creators for every module in a mixin class
*         (except if module is lifted -- in this case the module variable
*          is local to some function, and the creator method is static.)
*       - A super accessor for every super accessor in a mixin class
*       - Forwarders for all methods that are implemented statically
*  All superaccessors are completed with right-hand sides (@see completeSuperAccessor)
*  @param clazz The class to which definitions are added
*/
private def addNewDefs(clazz: Symbol, stats: List[Tree]): List[Tree] = {
```

`addNewDefs` stretches over 500 lines, and appends term trees to templates:

- It doesn't add anything to an impl-module, which `preTransform` already transformed to contain only impl-methods (albeit their bodies need rewriting).

- Instead, `addNewDefs` has to create trees (mostly `DefDefs` but some `ValDefs` too) for the symbols added by `preTransform`, i.e. `addNewDefs` adds terms to templates of iface-facets and composite classes.

Saying that "`addNewDefs` appends terms" is an oversimplification, because the template's `stats` may already contain abstract defs that should be replaced (for example, TODO). Therefore, a tree from the original `stats` stays only if no signature-equivalent term was created by `addNewDefs`:

```
/** Add ‘newdefs‘ to ‘stats‘, removing any abstract method definitions
 *  in ‘stats‘ that are matched by some symbol defined in ‘newDefs‘.
 */
def add(stats: List[Tree], newDefs: List[Tree]) = {
```

The utility method above receives almost all concrete methods there's to add, except trees for super-accessors. In one more pass, `completeSuperAccessor` is used to replace their abstract counterparts, and that's the new body of the template:

```
stats1 = add(stats1, newDefs.toList)
if (!clazz.isTrait) stats1 = stats1 map completeSuperAccessor
stats1
```

5.5.1 Trees for interface facets

Just an abstract `DefDef` is added:

```
for (sym <- clazz.info.decls) {
  if (sym hasFlag MIXEDIN) {
    if (clazz hasFlag lateINTERFACE) {
      // if current class is a trait interface, add an abstract method for accessor ‘sym‘
      addDefDef(sym, vparamss => EmptyTree)
```

5.5.2 Trees for composites: Accessors

These notes do not cover mixing-in of lazy values, other than saying that it's done here.

```
TODO
```

5.5.3 Trees for composites: Modules

```
else if (sym.isModule && !(sym hasFlag LIFTED | BRIDGE)) {
  // add modules
  val vdef = gen.mkModuleVarDef(sym)
  addDef(position(sym), vdef)

  val rhs = gen.newModule(sym, vdef.symbol.tpe)
  val assignAndRet = gen.mkAssignAndReturn(vdef.symbol, rhs)
  val attrThis = gen.mkAttributedThis(clazz)
  val rhs1 = mkInnerClassAccessorDoubleChecked(attrThis, assignAndRet)
  addDef(position(sym), DefDef(sym, rhs1))
}
```

5.5.4 Trees for composites: Fields

```
  else if (!sym.isMethod) {
  // add fields
  addDef(position(sym), ValDef(sym))
}
```

5.5.5 Trees for composites: Superaccessors

```
  else if (sym.isSuperAccessor) {
  // add superaccessors
  addDefDef(sym, vparams => EmptyTree)
}
```

As shown in the introduction of Sec. 5.5, the bodies of super-accessors are completed just before returning from `addNewDefs()`:

```
stats1 = add(stats1, newDefs.toList)
if (!clazz.isTrait) stats1 = stats1 map completeSuperAccessor
stats1
```

5.5.6 Trees for composites: Forwarders

```
  else {
  // add forwarders
  assert(sym.alias != NoSymbol, sym)
  addDefDef(sym, vparams =>
    Apply(staticRef(sym.alias), gen.mkAttributedThis(clazz) :: (vparams map Ident)))
}
```