Splitting traits into implementation and interface facets (aka `AddInterfaces`)

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

September 30th, 2011

Abstract

`AddInterfaces` is a phase in disguise:

1. it runs on trees pre-transformed by `erasure` and
2. transforms them further assuming erased types (because it runs under `atPhase(erasure.next)`) although `erasure` isn't over yet.
3. After receiving from `AddInterfaces` "splitted trees", it only remains for `erasure` to run its "custom modifier-typer" on them (i.e., retyping accounts for the impl-classes and trait-ifaces now in place).

The transformation in Step 2 operates on traits:

- an interface-trait (i.e., a trait lacking concrete methods) is left as is.
- each non-interface-trait is split into an implementation-class (to be shared across all non-trait classes where the original trait is mixed-in) and a trait-interface (capturing the non-private members of the original trait, i.e. the contract that its subtypes must support).

```
      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 The peaceful coexistence of class linearization, trait splitting, mixing, and VM-level types

At the end of the day the compiler has to emit VM-level classes and interfaces which must abide by subtyping and overriding rules less expressive than those of Scala.

This section tries to convey the intuition why VM-level programs with translated traits "don't go wrong" (a colophon on this in Sec. 4.3). That intuition helps in convincing oneself about the purpose of the rewritings described in Sec. 3 and Sec. 4. The other piece of the puzzle, `mixin`, is summarized as necessary. In the rest of this section we focus on a non-trait class (affectionally called "the non-trait class C") because the other cases fall naturally from it.

The contract of a Scala non-trait class C is given by (a) its template; and (b) its linearization. Jumping ahead, the VM-level counterpart to C (which has to fulfill the VM-level counterpart to C's contract) is a VM-level class `C`, whose class-inheritance and interface-extends chains can be visualized as a stack of:

```
(VM-level-class, List[VM-level-interface])
```

Giving names to individual types in that stack allows discussing how they are inter-related:

```
C     ,  I(N+1, 1),  ...  , I(N+1, M(N+1))
S(N)  ,  I(N  , 1),  ...  , I(N  , M(N)  )
S(N-1),  I(N-1, 1),  ...  , I(N-1, M(N-1))
...
S(1)  ,  I(1  , 1),  ...  , I(1  , M(1)  )
Object
```

The superclasses of `C` are shown above as `S(N)` to `S(1)`, with `S(N)` the direct superclass of `C`, `S(N-1)` the direct superclass of `S(N)`, and so on. The VM-level interfaces that a VM-level class at level `i` supports are depicted as `I(i, 1)` to `I(i, M(i))`.

For VM-level types "not to go wrong" there should be a mapping between the "stacked types" shown above and the contract of C in Scala. First we show the structure of that mapping, without arguing just yet about the contract-preservation property.

Informally speaking, the VM-level types above correspond to the Scala types in the linearization of C:

C, mixinClasses-for-C,
S(N), mixinClasses-for-S(N),
. . . ,
S(1), mixinClasses-for-S(1)
AnyRef
Any

As usual, the linearization of C starts with C and ends with *Any*. To recap (Sec. 2.3) `mixinClasses` returns "*The directly or indirectly inherited mixins of this class except for the superclass and mixin classes inherited by the superclass. Mixin classes appear in linearization order.*"

The following differences between "Scala linearization" and "VM-level stacked-types" hint at the transformations necessary to shoehorn ASTs under the former into ASTs for the latter:

1. in the original linearization, "*mixinClasses*" may contain both interface-only and non-interface traits. While the former remain as-is, splitting is required for the latter (so that the "new linearization" contains only the interface-facets).

2. in the original linearization, C and the *S(i)* need not re-declare methods they inherit from a type appearing later in their respective linearizations. After trait-splitting, `C` and its superclasses can't assume anymore that those methods have been inherited, and concrete methods (delegators) should be pasted into the AST of `C`.

3. Something similar occurs for *mixinClasses*, but in this case any super-access in one of them was rewritten (by `superaccessors`, Sec. 2.1) to target a synthetic abstract method (which `mixin` should fill with a method body).

The rest of these notes focus on the Step 1 (that's what `AddInterfaces` does). Steps 2 and 3 are covered in the write-up on `mixin`.

2 Preliminaries to rewriting

2.1 Input shapes

A few highlights about AST shapes that `AddInterfaces` is about to transform:

1. The classes arriving at `AddInterfaces` can be trait or non-trait (for the former, either interface-only or non-interface; and for the latter either abstract or concrete).

2. After `superaccessors`, all traits (whether interface-only or not) lack super-refs. Because `superaccessors` rewrites them into invocations of private trait-level methods syntehsized to that effect. Same thing for super-refs targeting members of the super-class of an outer class.

3. Given that:

   (a) `refchecks` lowers `ModuleDef` nodes into module-class, module-variable, and module-accessor;

   (b) *general instance creation expression* were expanded (by `parser`) into blocks containing a `ClassDef` and a *simple instance creation expression*, { `class a extends t; new a` }

   trait-splitting and trait-mixing can focus on `ClassDef` nodes only.

4. Non-interface `trait` classes contain a primary constructor, i.e. a single no-args constructor named `nme.MIXIN_CONSTRUCTOR` that was fabricated back in `parser` (by calling the `ast.Trees.Template()` factory method, Figure 1). The only possible contents of that constructor are early initializers for `val`/`var` in a super-trait, for example:

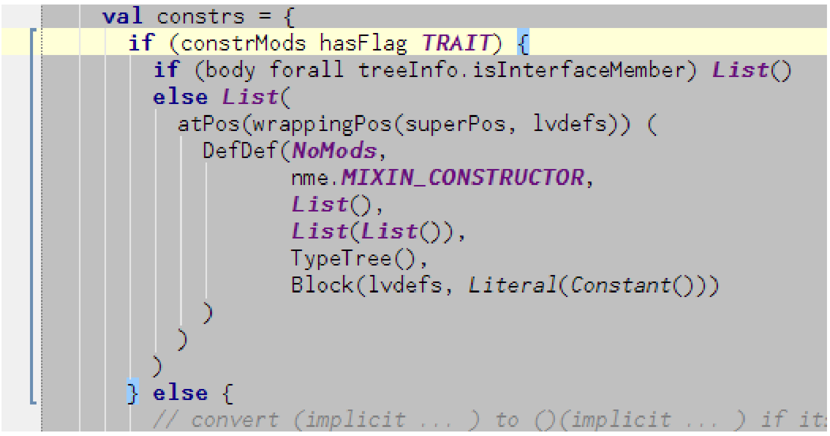
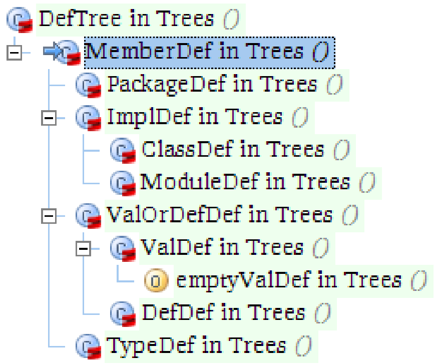


Figure 1: Sec. 2.1

```
trait Person { val age: Int }

trait TenYearOld extends { val age = 10 } with Person
```

5. Given that `constructors` hasn't run yet, the template of the incoming trait may contain executable statements.

6. *Some* of the possible contents of a template at this point in the pipeline (actually, `PackageDef` and `ModuleDef` can't show up):



2.2 Naming convention

Specially on a first reading, names like the following are useful: `traitClazz` (the symbol of the trait being splitted) vs. `implClazz` (a new symbol), i.e. using the name to convey information about the splitting role, akin to Hungarian notation. Hungarian notation is enough because there are long stretches of code where a symbol can't possibly be anything other than, say, a `traitClazz`, thus it's not misleading to call it like that.

Listing 1: Sec. 2.3

```
/** Is this a symbol which exists only in the implementation class, not in its trait? */
final def isImplOnly: Boolean =
  hasFlag(PRIVATE) ||
  (owner.isImplClass || owner.isTrait) &&
  ((hasFlag(notPRIVATE | LIFTED) && !hasFlag(ACCESSOR | SUPERACCESSOR | MODULE) || isConstructor) ||
   (hasFlag(LIFTED) && isModule && isMethod))


override final def isTrait: Boolean =
  isClass && hasFlag(TRAIT | notDEFERRED) // A virtual class becomes a trait (part of DEVIRTUALIZE)

def isVirtualTrait =
  hasFlag(DEFERRED) && isTrait

/** Is this symbol a trait which needs an implementation class? */
final def needsImplClass: Boolean =
  isTrait && (!isInterface || hasFlag(lateINTERFACE)) && !isImplClass
```

Similarly for `traitMember`, `implMember`, and `traitDecls` (also symbols) and `traitTempl` and `traitTemplStat` (`Trees`).

The suggestions above are useful because `sym` can denote an `implClazz`, `traitMember`, or `implMember` depending on context. In other cases, different names are used at different locations for the same concept (e.g., `implClazz` goes by the name of `impl`, `implClass`, or `sym`, depending on whether the naming context is `implClass` (the single point of access to the `implClassMap` map), `implDecls`, or `LazyImplClassType`.

Regarding `Trees`, `tree` stands most of the time for `traitTemplStat`, however in one occassion `primaryConstrBody` is more descriptive (in `addMixinConstructorCalls()`).

Another suggestion:

```
private def implMethodDef(traitDefDef: Tree): Tree = {
  val traitMethod = traitDefDef.symbol
  implMethodMap.get(traitMethod) match {
    case Some(implMethod) =>
      traitDefDef.symbol = implMethod
      new ChangeOwnerAndReturnTraverser(traitMethod, implMethod)(traitDefDef)
    case None =>
      abort("implMethod missing for " + traitMethod)
  }
}
```

2.3 Trait-related helpers in `Symbols.scala`

Several query-methods directly about traits:

- Yes/No (Listing 1): `isImplOnly`, `isTrait`, `isVirtualTrait`, `needsImplClass`
- Filters (Listing 2): `toInterface`, `mixinClasses`, `primaryConstructor`

Listing 2: Sec. 2.3

```
/** If this symbol is an implementation class, its interface, otherwise the symbol itself
 *  The method follows two strategies to determine the interface.
 *   - during or after erasure, it takes the last parent of the implementation class
 *     (which is always the interface, by convention)
 *   - before erasure, it looks up the interface name in the scope of the owner of the class.
 *     This only works for implementation classes owned by other classes or traits.
 */
final def toInterface: Symbol =
  . . .

/** The directly or indirectly inherited mixins of this class
 *  except for mixin classes inherited by the superclass. Mixin classes appear
 *  in linearization order.
 */
def mixinClasses: List[Symbol] = {
  val sc = superClass
  ancestors takeWhile (sc ne)
}

/** The primary constructor of a class. */
def primaryConstructor: Symbol = {
  var c = info.decl(
    if (isTrait || isImplClass) nme.MIXIN_CONSTRUCTOR
    else nme.CONSTRUCTOR)
  c = if (c hasFlag OVERLOADED) c.alternatives.head else c
  //assert(c != NoSymbol)
  c
}

/** The implementation class of a trait. */
final def implClass: Symbol = owner.info.decl(nme.implClassName(name))
```

3 Term rewriting

3.1 Shape of splitted `ClassDefs`

As part of splitting, the contents of the incoming trait are separated between the resulting facets. It's easier to describe what is allowed in the interface facet (Sec. 3.1.1) and from there figure out why "the rest" has to end up in the implementation fact (Sec. 3.1.2). With these "recipes" in place, it's possible to stand back and see why the resulting program remains well-typed after trait splitting (Sec. 4.3).

3.1.1 Interface facet

There's already a `ClassDef` for the interface facet (the incoming non-interface trait itself). On exit, this `ClassDef` will have been pruned as follows:

1. **What stays as is (carrying their original symbols)**: the `DefDef` trees of public abstract methods, as well as those of super-accessors (although they are private).

2. **What gets added, minus body**: a non-abstract method in the incoming trait belonging to its contract can't be added as-is to the interface facet.

Instead, a method signature is fabricated, including the original symbol. The incoming method body belongs instead in the implementation facet.

3. **The rest is elided**

Two sidenotes:

- A super-accessor passes neither the `isInterfaceMember()` nor the `needsImplMethod()` tests. It "stays as is" by virtue of falling off into the last branch of chained `if-else`s.

- The counterpart to one of the added "non-abstract methods, minus body" can be found via `implMethodMap(traitMethod)`.

None of the methods (nor their symbols) arriving in the trait's template "gets lost". The following snippet shows they are separated evenly between both facets:

```
private def ifaceMemberDef(traitTemplStat: Tree): Tree =
  if (    !traitTemplStat.isDef
       || !isInterfaceMember(traitTemplStat.symbol))
    EmptyTree
  else if (needsImplMethod(traitTemplStat.symbol))
    DefDef(traitTemplStat.symbol, EmptyTree)
  else
    traitTemplStat
```

```
private def implMemberDef(traitTemplStat: Tree): Tree =
  if (    !traitTemplStat.isDef
       || !isInterfaceMember(traitTemplStat.symbol))
    traitTemplStat
  else if (needsImplMethod(traitTemplStat.symbol))
    implMethodDef(traitTemplStat, traitTemplStat.symbol)
  else
    EmptyTree
```

The following helpers are used to prune the incoming trait's template:

```
private def ifaceTemplate(traitTempl: Template): Template

   private def ifaceMemberDef(traitTemplStat: Tree): Tree
```

3.1.2 Implementation facet

There's no `ClassDef` for an implementation class among the incoming ASTs, so one is built from the ground up using the following helpers (indentation shows calling-called relationship):

```
def implClassDefs(stats: List[Tree]): List[Tree]

   private def implTemplate(implClazz: Symbol, traitTempl: Template): Template // changes owner

      private def implMemberDef(traitTemplStat: Tree): Tree

         private def implMethodDef(traitTemplStat: Tree, ifaceMethod: Symbol): Tree

      private def addMixinConstructorDef(implClazz: Symbol, implMembers: List[Tree]): List[Tree]
      // not to be confused with addMixinConstructorCalls
```

As for the shape of the resulting AST, an implementation template receives:

1. Initially `parents` match the original `parents` of the incoming trait, but that's only transient: the output of `implClassDefs()` will go through `transformStats()`, and from there to the fix-up in Sec. 3.3.

2. Given that `constructors` hasn't run yet, the incoming trait's template may contain executable statements. They go unmodified to the implementation facet. Any accesses they contain to members of the trait remain well-typed because any such access refers to:

   (a) a public member, and the impl extends the iface;

   (b) a super-accessor, which also ended up in the iface although super-accessors are private;

   (c) otherwise the access refers to a member that goes to the implementation facet.

3. In general, all non-method trees go to the implementation facet.

4. All non-interface methods of the original trait's template, carrying their symbols. In general, a selection of anyting but an interface member or a super-accessor now refers to a member of the implementation class.

5. private methods.

6. the `$init$` constructor.

For example, the `private[this]` field backing a trait-level "public" `var` goes to the implementation class. Getter and setter show up in both iface and impl, as signature in the iface *along with original symbols* and with the original bodies in the impl (*along with new symbols*).

Right after `AddInterfaces`, none of the added impl-methods (always with new symbols) is referred from any other `Tree`.

3.2 Calling trait initializers in the primary constructor of a non-trait class

The real purpose of this rewriting is to put in place enough information ("`implClazz.primaryConstructor`") for `mixin` later to fix-up the trait-init-calls added here. A trait-init-call invokes the `$init$` constructor of an implementation class.

Did you know that `isClassConstructor` returns `false` for `$init$` constructors, although they are `isPrimaryConstructor`?

Summing up: only a non-trait class is candidate for this rewriting, and if it has no non-interface trait among its `mixinClasses`, it doesn't get rewritten either.

After fulfilling those conditions, our `clazz` of interest will contain (after the super-init-call in its primary constructor) trait-init-calls for its `mixinClasses`, added by:

```
def mixinConstructorCall(implClazz: Symbol): Tree = atPos(primaryConstrBody.pos) {
  Apply(Select(This(clazz), implClazz.primaryConstructor), List())
}
```

Now we see the need for a fix-up later: How come our `clazz` inherits from (several) `implClazz`, as implied by the above? In fact, it extends interface facets, but thanks to `asInstanceOf[]` the trait-init-calls type-check:

```
final object
cid
extends IntCell with Incrementing
with Doubling with ScalaObject {

  def this(): object cid = {
    cid.super.this();




    ()
  }
};
```

```
final object
cid
extends IntCell with Incrementing
with Doubling with ScalaObject {

  def this(): object cid = {
    cid.super.this();

    cid.this.$asInstanceOf[
      Incrementing$class]()./*Incrementing$class*/$init$();
    cid.this.$asInstanceOf[
      Doubling$class]()./*Doubling$class*/$init$();
    ()
  }
};
```

3.3 `Trees` for updated parents

```
case Template(parents, self, body) =>
  val parents1 = sym.owner.info.parents map (t => TypeTree(t) setPos tree.pos)
  treeCopy.Template(tree, parents1, emptyValDef, body)
```

3.4 Rebinding self-ref still referring to splitted trait

All expressions (method bodies, template statements) were moved from the incoming-trait to the implementation facet, and they may contain `This(traitClazz)` nodes. If the self-reference in question is enclosed at some depth by `implClass(traitClazz)`, then the self-reference should be made to point to it instead (otherwise remains as is).

```
TODO Details.
```

Mechanics: "*The symbol of a This is the class to which the this refers. For instance in `C.this`, it would be `C`.*"

```
case This(_) =>
  if (sym.needsImplClass) {
    val implClazz = implClass(sym)
    var owner = currentOwner
    while (owner != sym && owner != implClazz) owner = owner.owner;
    if (owner == implClazz) This(implClazz) setPos tree.pos
    else tree
  } else tree
```

```
TODO "All expressions were moved from the incoming-trait to the impl-class".
     Example where that can be seen for default params.
```

4 Type rewriting

As mentioned in the Abstract, `AddInterfaces` runs after trees have been pre-transformed by `erasure` but before they have been re-typed:

```
class ErasureTransformer(unit: CompilationUnit) extends Transformer {

  /** The main transform function: Pretransfom the tree, and then
   *  re-type it at phase erasure.next.
   */
  override def transform(tree: Tree): Tree = {
    val tree1 = preTransformer.transform(tree)
    atPhase(phase.next) {
      val tree2 = mixinTransformer.transform(tree1)
      if (settings.debug.value)
        log("tree after addinterfaces: \n" + tree2)

      newTyper(rootContext(unit, tree, true)).typed(tree2)
    }
  }
}
```

4.1 Interface facet

The `info` of an interface facet (a `ClassInfoType`) is updated by

```
def transformMixinInfo(tp: Type): Type
```

Changes affect parents and scope (the class-symbol also gets `lateINTERFACE` set):

1. An interface can't possibly have a super-class. Whatever super-class there was, it's made to be `Object` from now on. This won't break any super-refs, because `superaccessors` got rid of them in favor of invocations to interface-owned synthetic methods (Sec. 2.1). The other `Types` in `parents` necessarily denote traits, and in fact from now on denote interface-only traits.

2. The computed `decls` agree with term rewriting: from those in the incoming non-interface trait, only super-accessors and `isInterfaceMember()` symbols are kept. Off-topic: here's where type aliases and abstract member types are elided.

Summing up: the interface-facet is almost ready for VM-consumption (pending erasure). By now it's a VM-ready interface that may extend other VM-ready interfaces.

4.2 Implementation facet

The `info` of an implementation facet (a `ClassInfoType`) is updated by `LazyImplClassType`. Changes affect parents and scope (the class-symbol got `IMPLCLASS` set back when the symbol was created):

1. The mandatory parents an implementation class have to do with the contents of its template, discussed in Sec. 3.1.2. In particular:

   > *Given that `constructors` hasn't run yet, the incoming trait's template may contain executable statements. They go unmodified to the implementation facet. (Any accesses they contain to members of the trait remain well-typed because any such access refers to:*

(a) *a public member, and the impl extends the iface;*
(b) *a super-accessor, which also ended up in the iface although super-accessors are private;*
(c) *otherwise the access refers to a member that goes to the implementation facet.*

What parents are mandatory for the implementation class? Clearly, the original super-class is not mandatory, while the interface facet is. In detail:

```
ObjectClass.tpe +:
(parents.tail map mixinToImplClass filter (_.typeSymbol != ObjectClass)) :+
traitClazz.tpe
```

2. Regarding `decls`,

```
TODO
```

Mechanics: the `Type` taken as starting point is that of the incoming trait *before erasure has touched it in any way*:

```
implClazz setInfo implType(
                              atPhase(currentRun.erasurePhase)(
                                                                  traitClazz.info
                                                              )
                          )
```

During `setInfo` however, `currentRun.erasurePhase.next` is in effect (to recap from the Abstract, "*`AddInterfaces` transforms them further assuming erased types (because it runs under `atPhase(erasure.next)`) although `erasure` isn't over yet*")

Example:

```
// before AddInterfaces
IntCell#7577 with ScalaObject#450{
  def $init$#9721(): Unit#447;
  override def setCell#9722(i#19004: Int#375): Unit#447;
  final def Doubling$$super$setCell#17599(i#19009: Int#375): Unit#447
}

// after AddInterfaces
java.lang.Object#2337 with ScalaObject#450 with Doubling#7578{
  def $init$#9721(): Unit#447;
  override def setCell#19018(i#19019: Int#375): Unit#447
}
```

4.3 Splitted traits "don't go wrong" after all

```
TODO Bring together a grand summary of all sufficient conditions
    given piecemeal in previous sections.
```

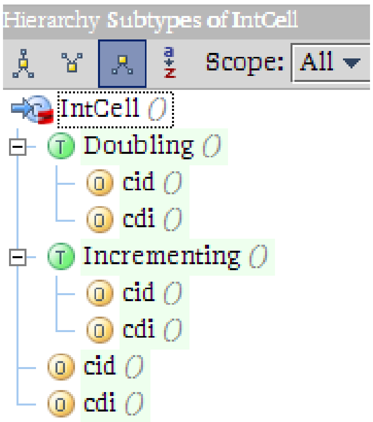


Figure 2: Type hierarchy of the running example, Listing 3 on p. 14

5 Example

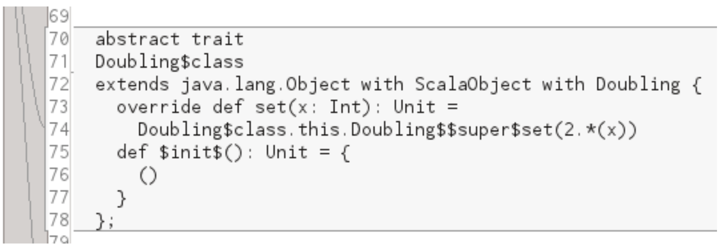
A running example (Listing 3 on p. 14) is used to visually depict the workings of `AddInterfaces`:

Figure 2 depicts the type hierarchy of the running example.

1. Adding implementation classes (Sec. 5.1)
2. Turning non-interface traits into interfaces (Sec. 5.2)
3. Adding trait initialization calls (Sec. 5.3)
4. The rest (Sec. 5.4)

5.1 Adding implementation classes

- Example:



```
abstract trait
Doubling$class
extends java.lang.Object with ScalaObject with Doubling {
  override def set(x: Int): Unit =
    Doubling$class.this.Doubling$$super$set(2.*(x))
  def $init$(): Unit = {
    ()
  }
};
```

- How:

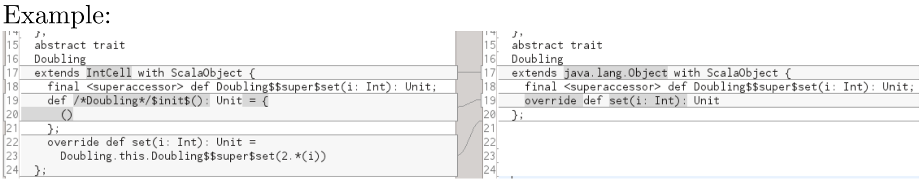
```
override def transformStats(stats: List[Tree], exprOwner: Symbol): List[Tree] =
  (super.transformStats(stats, exprOwner) :::
   super.transformStats(implClassDefs(stats), exprOwner))
```

Listing 3: Running example, reproduced from `TODO`

```
class IntCell {
  var x = 0
  def get() = this.x
  def set(i: Int) { this.x = i }
}

trait Doubling extends IntCell {
  override def set(i: Int) { super.set(2*i) }
}

trait Incrementing extends IntCell {
  override def set(i: Int) { super.set(i+1) }
}

object cid extends IntCell with Incrementing with Doubling;
  /*- cid.x is always odd (or zero) */

object cdi extends IntCell with Doubling with Incrementing;
  /*- cdi.x is always even (or zero) */
```

5.2 Turning traits into interfaces

- Example:



- How:

```
override def transform(tree: Tree): Tree = {
  val sym = tree.symbol
  val tree1 = tree match {

    case ClassDef(mods, name, tparams, impl) if (sym.needsImplClass) =>
      implClass(sym).initialize // to force lateDEFERRED flags
      treeCopy.ClassDef(tree, mods | INTERFACE, name, tparams, ifaceTemplate(impl))
```

5.3 Adding trait initialization calls

- Example:

```
final object
cid
extends IntCell with Incrementing
with Doubling with ScalaObject {

  def this(): object cid = {
    cid.super.this();




    ()
  }
};
```

```
final object
cid
extends IntCell with Incrementing
with Doubling with ScalaObject {

  def this(): object cid = {
    cid.super.this();

    cid.this.$asInstanceOf[
      Incrementing$class]()./*Incrementing$class*/$init$();
    cid.this.$asInstanceOf[
      Doubling$class]()./*Doubling$class*/$init$();
    ()
  }
};
```

- How:

```
case DefDef(mods, name, tparams, vparamss, tpt, rhs)
if (sym.isClassConstructor && sym.isPrimaryConstructor && sym.owner != ArrayClass) =>
  treeCopy.DefDef(tree, mods, name, tparams, vparamss, tpt,
              addMixinConstructorCalls(rhs, sym.owner)) // (3)
```

5.4 The rest

- How:

```
case Template(parents, self, body) =>
  val parents1 = sym.owner.info.parents map (t => TypeTree(t) setPos tree.pos)
  treeCopy.Template(tree, parents1, emptyValDef, body) // TODO Note: self goes away.

case This(_) =>
  if (sym.needsImplClass) {
    val implClazz = implClass(sym)
    var owner = currentOwner
    while (owner != sym && owner != implClazz) owner = owner.owner;
    if (owner == implClazz) This(implClazz) setPos tree.pos
    else tree
  } else tree
```