How the `constructors` phase works

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

April 30th, 2011

Abstract

These notes cover `constructors`, the phase in charge of rephrasing template initialization in terms of VM-level fields and constructors. The phase also lowers `scala.DelayedInit` and needs to be aware about `@specialized` in some cases. These notes don't delve into details of the latter, other than including pointers to related material and samples of before-after ASTs.

```
     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
        inliner 21  optimization: do inlining
       closelim 22  optimization: eliminate uncalled closures
            dce 23  optimization: eliminate dead code
            jvm 24  generate JVM bytecode
       terminal 25  The last phase in the compiler chain
```

Contents

```
14  /** This phase converts classes with parameters into Java-like classes with
15   *  fields, which are assigned to from constructors.
16   */
17  abstract class Constructors extends Transform with ast.TreeDSL {
18    import global._
19    import definitions._
20
21    /** the following two members override abstract members in Transform */
22    val phaseName: String = "constructors"
23
24    protected def newTransformer(unit: CompilationUnit): Transformer =
25      new ConstructorTransformer(unit)
26
27    private val guardedCtorStats: mutable.Map[Symbol, List[Tree]] = new mutable.Hash
28    private val ctorParams: mutable.Map[Symbol, List[Symbol]] = new mutable.HashMap[S
29
30    class ConstructorTransformer(unit: CompilationUnit) extends Transformer {
31
32      def transformClassTemplate(impl: Template): Template = {...} // transformClassTem
586
587     override def transform(tree: Tree): Tree =
588       tree match {
589         case ClassDef(mods, name, tparams, impl)
590         if !tree.symbol.isInterface && !isValueClass(tree.symbol) =>
591           treeCopy.ClassDef(tree, mods, name, tparams, transformClassTemplate(impl))
592         case _ =>
593           super.transform(tree)
594       }
595   } // ConstructorTransformer
596 }
```

Figure 1: Sec. 1

1 Intro

`scala.tools.nsc.transform.Constructors` contains just one transformer, with the structure shown in Figure 1. Just to be clear: a transformer extends the abstract class `Transformer` in `scala.tools.nsc.ast.Trees`.

With method `transformClassTemplate` collapsed, the main aspects of this phase become clear, so it only remains to see what goes on between lines 32 and 585. Besides turning templates (in tandem with primary constructors) into "Java-like constructors", `transformClassTemplate` handles two additional concerns: (a) rewriting for `DelayedInit`; and (b) handling the interplay with `@specialized`. We postpone their treatment to Sec. 5.1 and Sec. 5.2. Till then, we discuss the operation of `constructors` without those features.

2 Before and after

2.1 The parser part of the deal

In order to have an intuition for the shapes arriving at `constructors`, we start with `parser`, that in its machinations inserts a `DefDef` for a primary constructor:

```
// Input program

class A(a: Int)

class B(b: Int) extends A(b)
```

```
// After parser
```

```
  class A extends scala.ScalaObject {
    <paramaccessor> private[this] val a: Int = _;
    def <init>(a: Int) = {
      super.<init>();
      ()
    }
  };
  class B extends A with scala.ScalaObject {
    <paramaccessor> private[this] val b: Int = _;
    def <init>(b: Int) = {
      super.<init>(b);
      ()
    }
  };
```

2.2 Example: Early defs

Although this subsection contains before&after snippets obtained with `-Xprint`, a better way to see changes across phases involves `-Yshow-syms` (using `SymbolTrackers`). Sample output:

```
[[symbol layout at end of constructors]]
  . . .
!!! 1 symbols vanished:
( 1) value a -> class ShowMe -> package <empty> -> ...
    ValDef: <paramaccessor> private[this] val a: Int = _
```

The SLS gives the following input program when discussing *Early defs* in §5.1.6 (this example will come handy in Sec. 3.1):

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}

class C extends {
  val name = "Bob"
} with Greeting {
 println(msg)
}
```

After `constructors`:

```
[[syntax trees at end of constructors]]// Scala source: bt4.scala
package <empty> {
  abstract trait Greeting extends java.lang.Object with ScalaObject {
    <stable> <accessor> def name(): java.lang.String;
    <stable> <accessor> def msg(): java.lang.String
  };
  class C extends java.lang.Object with Greeting with ScalaObject {
    private[this] val name: java.lang.String = _;
    <stable> <accessor> def name(): java.lang.String = C.this.name;
    def this(): C = {
      val name: java.lang.String("Bob") = "Bob";
      C.this.name = "Bob";
      C.super.this();
      C.this.$asInstanceOf[Greeting$class]()./*Greeting$class*/$init$();
```

```
      scala.this.Predef.println(C.this.msg());
      ()
    }
  };
  abstract trait Greeting$class extends java.lang.Object with ScalaObject with Greeting {
    private[this] val msg: java.lang.String = _;
    <stable> <accessor> def msg(): java.lang.String = Greeting$class.this.msg;
    def /*Greeting$class*/$init$(): Unit = {
      Greeting$class.this.msg = "How are you, ".+(Greeting$class.this.name());
      ()
    }
  }
}
```

3 `transformClassTemplate`: setting the scene

Informally, phrases like "the body of the template", "the statements in the primary constructor" may refer to either the before-`constructors` state or the after-transform state. To avoid confusion, that distinction can be kept in mind, for example:

- `val stats = impl.body`, the before-contents of the template as a `List[Tree]`
- `constrParams`, the before-xform list of param-symbols for the (primary) constructor. N.B.: that's the constructor being meant whenever we talk of "the constructor".
- `constrBody`, before-contents of the primary constructor as a `Block(stats: List[Tree], expr: Tree)`.

Those are examples of before-state. The buffers for post-state all contain tree lists, conveniently separated into their position post-transform:

- template level, subdivided into auxiliary constructors and the rest (`auxConstructorBuf` and `defBuf` resp.)
- primary-constructor level, subdivided into before super-call (if any), on the one hand; and at-or-after the super call, on the other: `constrPrefixBuf` and `constrStatBuf` resp.

The buffers mentioned above are:

```
// The list of definitions that go into class
val defBuf = new ListBuffer[Tree]

// The auxiliary constructors, separate from the defBuf since they should
// follow the primary constructor
val auxConstructorBuf = new ListBuffer[Tree]

// The list of statements that go into constructor after and including the superclass constructor call
val constrStatBuf = new ListBuffer[Tree]

// The list of early initializer statements that go into constructor before the superclass constructor call
val constrPrefixBuf = new ListBuffer[Tree]
```

The last two items may give the impression that their concatenation constitutes the post-transform primary constructor. Not quite so. Some pieces haven't been discussed yet but their meaning is hinted at by the following snippet:

```
  // Assemble final constructor
  defBuf += treeCopy.DefDef(
    constr, constr.mods, constr.name, constr.tparams, constr.vparamss, constr.tpt,
    treeCopy.Block(
      constrBody,
      paramInits ::: constrPrefixBuf.toList ::: uptoSuperStats ::: /*- so far we've heard only about constrPrefi
        guardSpecializedInitializer(remainingConstrStats),
      constrBody.expr));
```

3.1 Letting the instance know about presuper values

Mutation action starts by filling the statements at or after the super-call (`constrStatBuf`), only that (contrary to its name) first of all assignments are added for early defs, which by their very nature are executed before invoking a super constructor and trait initializers (all will be fine after `splitAtSuper`, Sec. 4.3).

In terms of the example from Sec. 2.2, there's the following "presuper"

```
class C extends {
  val name = "Bob" /*- presuper */
} with Greeting {
 println(msg)
}
```

to put it into perspective, the primary constructor after `constructors` looks like:

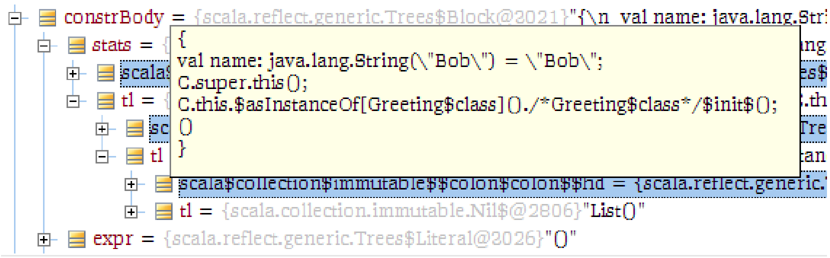
```
def this(): C = {
  val name: java.lang.String("Bob") = "Bob"; /*- this subsection covers how this line */
  C.this.name = "Bob";                        /*- and this line are added to constrStatBuf */
  C.super.this();
  C.this.$asInstanceOf[Greeting$class]()./*Greeting$class*/$init$();
  scala.this.Predef.println(C.this.msg());
  ()
}
```

The snippet below informs us that (among the statements in the before-xform constructor) "constructor-local `ValDefs` for pre-supers" can be found. Each such `ValDef` goes unchanged into the after-xform constructor *and is immediately followed* by an assignment to the class-level field for that presuper. Thus the title of this section. BTW, no clue what the `rhs` of the constructor-local `ValDef`, if any, was.

```
  // generate code to copy pre-initialized fields
  for (stat <- constrBody.stats) {
    constrStatBuf += stat
    stat match {
      case ValDef(mods, name, _, _) if (mods hasFlag PRESUPER) =>
        // stat is the constructor-local definition of the field value
        val fields = presupers filter (
          vdef => nme.localToGetter(vdef.name) == name)
        assert(fields.length == 1)
        val to = fields.head.symbol
        if (!to.tpe.isInstanceOf[ConstantType])
```

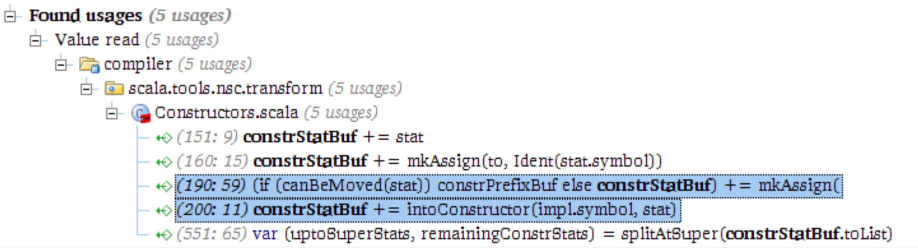
```
        constrStatBuf += mkAssign(to, Ident(stat.symbol))
      case _ =>
    }
  }
```

For the "C and Greeting" example, `constrBody.stats` contains presuper, super call, and trait init:



This is one of those transforms where the resulting ASTs can be mapped to bytecode but not to Java or C#. (Unless "*IL Inlining in High-Level Languages*" is used, as implemented by `InlineIL`[1]. See Sec. 5.3 for details).

We're not yet done with mutating `constrStatBuf`: highlighted below are two more cases, to be discussed later.



3.2 The first big spill-over

Having started to fill one of the "receptacles" what about the other three? Each of them (`defBuf`, `auxConstructorBuf`, `constrPrefixBuf`) as well as `constrStatBuf` itself potentially gets something during the triaging performed from lines 166 to 201.

Why is such triaging needed? Before `constructors`, templates contain executable statements and `ValDefs` with executable RHSs, whose evaluation logically belongs in the primary constructor.

Or put in yet another way :-) in Sec. 3.1, we didn't iterate over template stmts but over those in the constructor, now we need to add what logically belongs in the constructor but so far is owned by the template

The before-xform template statements are classified into "definitions" and "others":

1. Three kinds of definitions (Figure 2) are distributed here:

   (a) `ClassDefs` remain always template-level

[1] `http://blogs.msdn.com/jmstall/archive/2005/02/21/377806.aspx`

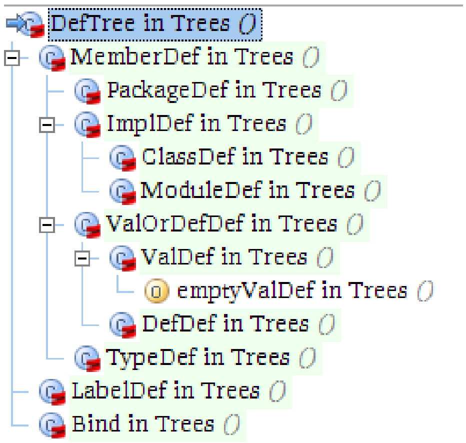


Figure 2: Sec. 3.2

(b) Regarding constructors:
   i. the primary constructor is skipped as it will be added later
   ii. auxiliary constructors go to `auxConstructorBuf`

(c) All (non-constructor) methods go to the post-xform template (i.e., to `defBuf`). However the body of methods with constant result type is rewritten into a literal.

(d) (mutable) value definitions go to `defBuf` (unless constant), and one of `constrPrefixBuf` or `constrStatBuf` (unless lazy).

```
// val defs with constant right-hand sides are eliminated.
// for all other val defs, an empty valdef goes into the template and
// the initializer goes as an assignment into the constructor
// if the val def is an early initialized or a parameter accessor, it goes
// before the superclass constructor call, otherwise it goes after.
// Lazy vals don't get the assignment in the constructor.
```

2. all other statements of the before-xform template go into the post-xform constructor (by first going to `constrStatBuf`).

3.3 intoConstructorTransformer

The list in Sec. 3.2 shows that cases (1.d) and (2) result in expressions being moved from template-level to constructor-level. There's a transformer (`intoConstructorTransformer`, Listing 3) to help with that. Moving one such `Tree` involves:

1. First, its owner is changed from `impl.symbol` to `constr.symbol`.

2. Second, it goes through `intoConstructorTransformer` which has to be aware of `@specialized` (details omitted!). Instead, we look at the most common rewritings it performs (in all cases, the rewritings are conditional, details in code shown below):

   (a) references to parameter accessor methods of own class become references to parameters

```scala
// A transformer for expressions that go into the constructor
val intoConstructorTransformer = new Transformer {
  def isParamRef(sym: Symbol) =
    sym.isParamAccessor &&
    sym.owner == clazz &&
    !(sym.isGetter && sym.accessed.isVariable) &&
    !sym.isSetter
  private def possiblySpecialized(s: Symbol) = specializeTypes.specializedTypeVars(s).nonEmpty
  override def transform(tree: Tree): Tree = tree match {
    case Apply(Select(This(_), _), List()) =>
      // references to parameter accessor methods of own class become references to parameters
      // outer accessors become references to $outer parameter
      if (isParamRef(tree.symbol) && !possiblySpecialized(tree.symbol))
        gen.mkAttributedIdent(parameter(tree.symbol.accessed)) setPos tree.pos
      else if (tree.symbol.outerSource == clazz && !clazz.isImplClass)
        gen.mkAttributedIdent(parameterNamed(nme.OUTER)) setPos tree.pos
      else
        super.transform(tree)
    case Select(This(_), _) if (isParamRef(tree.symbol) && !possiblySpecialized(tree.symbol)) =>
      // references to parameter accessor field of own class become references to parameters
      gen.mkAttributedIdent(parameter(tree.symbol)) setPos tree.pos
    case Select(_, _) =>
      if (specializeTypes.specializedTypeVars(tree.symbol).nonEmpty)
        usesSpecializedField = true
      super.transform(tree)
    case _ =>
      super.transform(tree)
  }
}
```

Figure 3: Sec. 3.3

(b) outer accessors become references to `$outer` parameter

(c) references to parameter accessor field of own class become references to parameters

Example:

```scala
class C (p: Int, var v: Char) {
  Console.println(p)
  Console.println(v)
  v = 10
  Console.println(v)
}
```

Before-and-after ASTs:

```
[[syntax trees at end of lambdalift]]// Scala source: bt4.scala
package <empty> {
  class C extends java.lang.Object with ScalaObject {
    <paramaccessor> private[this] val p: Int = _;
    <paramaccessor> private[this] var v: Char = _;
    <accessor> <paramaccessor> def v(): Char = C.this.v;
    <accessor> <paramaccessor> def v_=(x$1: Char): Unit = C.this.v = x$1;
    def this(p: Int, v: Char): C = {
      C.super.this();
      ()
    };
    scala.Console.println(scala.Int.box(C.this.p));
    scala.Console.println(scala.Char.box(C.this.v()));
    C.this.v_=(\'\\n\');
    scala.Console.println(scala.Char.box(C.this.v()))
  }
}
```

```
[[syntax trees at end of constructors]]// Scala source: bt4.scala
package <empty> {
  class C extends java.lang.Object with ScalaObject {
    <paramaccessor> private[this] var v: Char = _;
    <accessor> <paramaccessor> def v(): Char = C.this.v;
    <accessor> <paramaccessor> def v_=(x$1: Char): Unit = C.this.v = x$1;
    def this(p: Int, v: Char): C = {
      C.this.v = v;
      C.super.this();
      scala.Console.println(scala.Int.box(p));
      scala.Console.println(scala.Char.box(C.this.v()));
      C.this.v_=(\'\\n\');
      scala.Console.println(scala.Char.box(C.this.v()));
      ()
    }
  }
}
```

Code in charge of carrying out the transformation:

```scala
  case Apply(Select(This(_), _), List()) =>
    // references to parameter accessor methods of own class become references to parameters
    // outer accessors become references to $outer parameter
```

```
    if (isParamRef(tree.symbol) && !possiblySpecialized(tree.symbol))
      gen.mkAttributedIdent(parameter(tree.symbol.accessed)) setPos tree.pos
    else if (tree.symbol.outerSource == clazz && !clazz.isImplClass)
      gen.mkAttributedIdent(parameterNamed(nme.OUTER)) setPos tree.pos
    else
      super.transform(tree)
  case Select(This(_), _) if (isParamRef(tree.symbol) && !possiblySpecialized(tree.symbol)) =>
    // references to parameter accessor field of own class become references to parameters
    gen.mkAttributedIdent(parameter(tree.symbol)) setPos tree.pos
```

4 Second part of `transformClassTemplate`

4.1 Collecting symbols accessed outside the primary constructor

Simplifying somewhat, on entry to `constructors` there is a field for each constructor param (and possibly accessors depending on whether the param was marked `val` or `var`). Sometimes, constructor params are used as they would be in Java, i.e. only within the primary constructor. If so, no dedicated field for that param is necessary.

The machinery required to realize the intuition above permeates the rest of `transformClassTemplate`. In this subsection we cover the first steps of that process, by first rephrasing in more detail the idea just sketched.

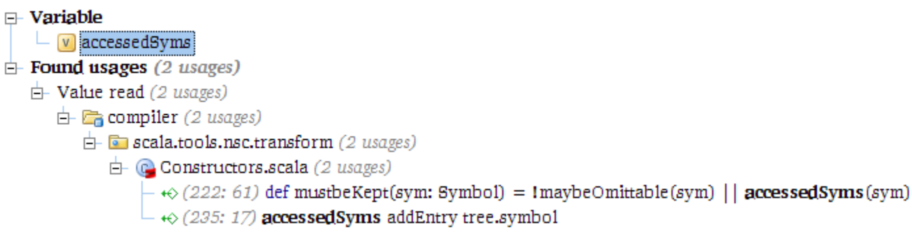
`transformClassTemplate` relies a number of times on `mustbeKept(sym: Symbol)` to leave out of the post-xform template some definition. For example:

```
// Eliminate all field definitions that can be dropped from template
treeCopy.Template(impl, impl.parents, impl.self,
  defBuf.toList filter (stat => mustbeKept(stat.symbol)))
```

In turn, `mustbeKept` bases its decision (among others) on:

```
// A sorted set of symbols that are known to be accessed outside the primary constructor.
val accessedSyms = new TreeSet[Symbol]((x, y) => x isLess y)
```

As part of collecting `accessedSyms`, collecting `outerAccessors` is also necessary (but `outerAccessors` aren't used for anything else afterwards). And once `accessedSyms` have been collected, `outerAccessors` won't be queried directly, but through `mustbeKept(sym)`, as the following screencapture shows:



Accessed-symbols are collected by running `accessTraverser` (Figure 4) twice:

- first on all members of the after-xform template (better said, what so far is considered to be the after-xform template), except the primary constructor

```
// A traverser to set accessedSyms and outerAccessors
val accessTraverser = new Traverser {
  override def traverse(tree: Tree) = {
    tree match {
      case DefDef(_, _, _, _, _, body)
      if (tree.symbol.isOuterAccessor && tree.symbol.owner == clazz && clazz.isFinal) =>
        log("outerAccessors += " + tree.symbol.fullName)
        outerAccessors ::= (tree.symbol, body)
      case Select(_, _) =>
        if (!mustbeKept(tree.symbol)) {
          log("accessedSyms += " + tree.symbol.fullName)
          accessedSyms addEntry tree.symbol
        }
        super.traverse(tree)
      case _ =>
        super.traverse(tree)
    }
  }
}
```

Figure 4: Sec. 4.1

and outer accessors

- afterwards, on those outer accessors which were detected as accessed by the previous stage.

It's high time for a code snippet!

```
// first traverse all definitions except outeraccesors
// (outeraccessors are avoided in accessTraverser)
for (stat <- defBuf.iterator ++ auxConstructorBuf.iterator)
  accessTraverser.traverse(stat)

// then traverse all bodies of outeraccessors which are accessed themselves
// note: this relies on the fact that an outer accessor never calls another
// outer accessor in the same class.
for ((accSym, accBody) <- outerAccessors)
  if (mustbeKept(accSym)) accessTraverser.traverse(accBody)
```

4.2 Initialize field from param, or omit field altogether

Once all symbols accessed outside the constructor are known (be they for params, fields, or otherwise) it is possible to emit code to:

1. initialize (with param values) those paramaccessor fields that will remain template-level. The code to perform such assignments is emitted by invoking:

```
// Create code to copy parameter to parameter accessor field.
// If parameter is $outer, check that it is not null so that we NPE
// here instead of at some unknown future $outer access.
def copyParam(to: Symbol, from: Symbol): Tree = {
  import CODE._
  val result = mkAssign(to, Ident(from))

  if (from.name != nme.OUTER) result
  else localTyper.typedPos(to.pos) {
    IF (from OBJ_EQ NULL) THEN THROW(NullPointerExceptionClass) ELSE result
  }
```

Listing 1: Sec. 4.2

```
// Conflicting symbol list from parents: see bug #1960.
// It would be better to mangle the constructor parameter name since
// it can only be used internally, but I think we need more robust name
// mangling before we introduce more of it.
val parentSymbols = Map((for {
  p <- impl.parents
  if p.symbol.isTrait
  sym <- p.symbol.info.nonPrivateMembers
  if sym.isGetter && !sym.isOuterField
} yield sym.name -> p): _*)

// Initialize all parameters fields that must be kept.
val paramInits =
  for (acc <- paramAccessors if mustbeKept(acc)) yield {
    if (parentSymbols contains acc.name)
      unit.error(acc.pos, "parameter '%s' requires field but conflicts with %s in '%s'".format(
        acc.name, acc.name, parentSymbols(acc.name)))

    copyParam(acc, parameter(acc))
  }
```

```
  }
```

2. later in `transformClassTemplate`, omit those paramaccessor fields that are accessed only within the constructor.

`Trees` for the assignments prepared as per item 1. above are kept in `paramInits`. The for-comprehension computing them also has to cater for an obscure error situation (inheriting from a trait a getter with the same name as a parameter being initialized) and is thus a bit more involved (Listing 1).

4.3 "Splitting at super"

From here (Sec. 4.3) till Sec. 4.5 the last part of method `transformClassTemplate` is shown.

```
/** Return a pair consisting of (all statements up to and including superclass and trait constr calls, rest) */
def splitAtSuper(stats: List[Tree]) = {
  def isConstr(tree: Tree) = (tree.symbol ne null) && tree.symbol.isConstructor
  val (pre, rest0) = stats span (!isConstr(_))
  val (supercalls, rest) = rest0 span (isConstr(_))
  (pre ::: supercalls, rest)
}

var (uptoSuperStats, remainingConstrStats) = splitAtSuper(constrStatBuf.toList)
```

4.4 Some words about `DelayedInit`

Details in Sec. 5.1.

```
val needsDelayedInit =
```

```
  (clazz isSubClass DelayedInitClass) /*&& !(defBuf exists isInitDef)*/ && remainingConstrStats.nonEmpty

if (needsDelayedInit) {
  val dicl = new ConstructorTransformer(unit) transform delayedInitClosure(remainingConstrStats)
  defBuf += dicl
  remainingConstrStats = List(delayedInitCall(dicl))
}
```

4.5 One more thing

```
// Assemble final constructor
defBuf += treeCopy.DefDef(
  constr, constr.mods, constr.name, constr.tparams, constr.vparamss, constr.tpt,
  treeCopy.Block(
    constrBody,
    paramInits ::: constrPrefixBuf.toList ::: uptoSuperStats :::
      guardSpecializedInitializer(remainingConstrStats),
    constrBody.expr));

// Followed by any auxiliary constructors
defBuf ++= auxConstructorBuf

// Unlink all fields that can be dropped from class scope
for (sym <- clazz.info.decls.toList)
  if (!mustbeKept(sym)) clazz.info.decls unlink sym

// Eliminate all field definitions that can be dropped from template
treeCopy.Template(impl, impl.parents, impl.self,
  defBuf.toList filter (stat => mustbeKept(stat.symbol)))
```

5 Bonus

5.1 `DelayedInit`

The entry point to the rewriting for `DelayedInit` was shown in Sec. 4.4, but the actual rewriting was skipped (i.e., `delayedInitClosure(remainingConstrStats)` and `delayedInitCall(dicl)`). Let's recap the SLS description:

> ***Delayed Initializaton***. *The initialization code of an object or class (but not a trait) that follows the superclass constructor invocation and the mixin-evaluation of the template's base classes is passed to a special hook, which is inaccessible from user code. Normally, that hook simply executes the code that is passed to it. But templates inheriting the `scala.DelayedInit` trait can override the hook by re-implementing the delayedInit method, which is defined as follows:*
>
> `def delayedInit(body: => Unit)`

The input program below is converted into that in Listing 2 (page 14).

```
object Main extends App {
  Console.println(args mkString)
}
```

Listing 2: Sec. 5.1

```
[[syntax trees at end of constructors]]// Scala source: bt4.scala
package <empty> {

  final object Main extends java.lang.Object with App with ScalaObject {

    final <synthetic> class delayedInit$body extends scala.runtime.AbstractFunction0 with ScalaObject {
      <paramaccessor> private[this] val $outer: object Main = _;
      final def apply(): java.lang.Object = {
        scala.Console.println(
            scala.this.Predef.refArrayOps(
                delayedInit$body.this.$outer.args().$asInstanceOf[Array[java.lang.Object]]()
            ).mkString()
        );
        scala.runtime.BoxedUnit.UNIT
      };
      def this($outer: object Main): Main#delayedInit$body = {
        if ($outer.eq(null))
          throw new java.lang.NullPointerException()
        else
          delayedInit$body.this.$outer = $outer;
        delayedInit$body.super.this();
        ()
      }
    }; // end of class delayedInit$body

    def this(): object Main = {
      Main.super.this();
      Main.this.$asInstanceOf[App$class]()./*App$class*/$init$();
      Main.this.delayedInit(new Main#delayedInit$body(Main.this));
      ()
    } // end of this(): object Main

  } // end of object Main

}
```

5.2 `@specialized`

Covered in §4.4.3 (Specialized instance initialization) of Iulian Dragos' PhD report [1].

For the following input program, the AST after `constructors` is shown in Listing 3 on p. 16.

```
abstract class Stack[@specialized(Int) T : ClassManifest](size: Int) {
  val data = new Array[T](size)
  println("created array of size " + data.length)

  def push(x: T)
  def pop: T
}
```

5.3 Inlining ILAsm bytecode in C# programs

This section follows up the discussion in Sec. 3.1. Example:

```
using System;
class Program
{
    static void Main()
    {
        int x = 3;
        int y = 4;
        int z = 5;

        // Here's some inline IL; x=x+y+z
#if IL
        ldloc x
        ldloc y
        add
        ldloc z
        add
        stloc x
#endif
        Console.WriteLine(x);
    }
}
```

The webpage for `InlineIL`[2] mentions limitations of the tool:

1. *The compiler (e.g.* `csc.exe`*) is completely ignorant to the IL snippets. This greatly simplifies the model but also introduces some issues:*

   (a) *The compiler doesn't know about any locals declared in the IL snippets.*

   (b) *The compiler can't do any analysis on the IL snippets. This can be critical in dead-code detection. If the only reference to C# code is via the IL snippet,* `csc.exe` *will think it's dead code and remove it, and then the code will be unavailable for the inliner. This is why the C# filter example above puts the throw in its own function.*

2. *There are limitations to stitching the high-level source code and the IL together. For example, you can't share labels across the boundary. Also, the compiler doesn't know about declarations from the IL snippets.*

3. *The inliner only supports IL statements. It doesn't support IL expressions, members, or types. Supporting expressions would require real integration with the compiler, and also provide little value since they can trivially be converted into statements. Supporting members would also require real integration with the compiler so that the rest of the compiler could see the newly declared member. Supporting types don't make sense since the type could just be in its own IL file.*

References

[1] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, Lausanne, 2010. http://lamp.epfl.ch/~dragos/files/dragos-thesis.pdf.

[2] http://blogs.msdn.com/jmstall/archive/2005/02/21/377806.aspx

Listing 3: Sec. 5.2

```
[[syntax trees at end of constructors]]// Scala source: bt4.scala
package <empty> {
  abstract class Stack extends java.lang.Object with ScalaObject {
    protected[this] val data: java.lang.Object = _;
    <stable> <accessor> def data(): java.lang.Object = Stack.this.data;
    def push(x: java.lang.Object): Unit;
    def pop(): java.lang.Object;
    <stable> <specialized> def data$mcI$sp(): Array[Int] = Stack.this.data().$asInstanceOf[Array[Int]]();
    <specialized> def push$mcI$sp(x: Int): Unit = Stack.this.push(scala.Int.box(x).$asInstanceOf[java.lang.Objec
    <specialized> def pop$mcI$sp(): Int = scala.Int.unbox(Stack.this.pop());
    def specInstance$(): Boolean = false;
    def this(size: Int, evidence$1: scala.reflect.ClassManifest): Stack = {
      Stack.super.this();
      if (Stack.this.specInstance$().unary_!())
        {
          Stack.this.data = evidence$1.newArray(size);
          scala.this.Predef.println("created array of size ".+(scala.Int.box(runtime.this.ScalaRunTime.array_leng
          ()
        };
      ()
    }
  };
  abstract <specialized> class Stack$mcI$sp extends Stack {
    private <paramaccessor> val size: Int = _;
    implicit private <paramaccessor> val evidence$1: scala.reflect.ClassManifest = _;
    <specialized> protected[this] val data$mcI$sp: Array[Int] = _;
    <stable> <accessor> <specialized> def data$mcI$sp(): Array[Int] = Stack$mcI$sp.this.data$mcI$sp;
    override <stable> <accessor> <specialized> def data(): Array[Int] = Stack$mcI$sp.this.data$mcI$sp();
    <specialized> def push(x: Int): Unit = Stack$mcI$sp.this.push$mcI$sp(x);
    <specialized> def pop(): Int = Stack$mcI$sp.this.pop$mcI$sp();
    def specInstance$(): Boolean = true;
    <bridge> <specialized> def pop(): java.lang.Object = scala.Int.box(Stack$mcI$sp.this.pop());
    <bridge> <specialized> def push(x: java.lang.Object): Unit = Stack$mcI$sp.this.push(scala.Int.unbox(x));
    override <stable> <bridge> <specialized> def data(): java.lang.Object = Stack$mcI$sp.this.data();
    <specialized> def this(size: Int, evidence$1: scala.reflect.ClassManifest): Stack$mcI$sp = {
      Stack$mcI$sp.this.size = size;
      Stack$mcI$sp.this.evidence$1 = evidence$1;
      Stack$mcI$sp.super.this(size, evidence$1);
      Stack$mcI$sp.this.data$mcI$sp = evidence$1.newArray(size).$asInstanceOf[Array[Int]]();
      scala.this.Predef.println("created array of size ".+(scala.Int.box(runtime.this.ScalaRunTime.array_length(
      ()
    }
  }
}
```