What goes on in the `lazyvals` phase

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

September 22nd, 2011

Abstract

`lazy val` is one of those language constructs whose lowering involves several compiler phases: a little work in `refchecks`, more work in `lazyvals`, and a finale in `mixin`. We discuss the AST rewriting in `lazyvals` and how it fits in the larger scheme of things.

```
      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

1 Intro

The initialization semantics of lazy values (i.e., when the right-hand-side is evaluated for the first time, and when it should be re-evaluated) determine the code to emit for (a) cached value of the lazy val; and (b) guard to avoid re-evaluation.

"Re-evaluating a lazy val" has to do with the way lazy vals are implemented:

- access the cached lazy val's value is possible only via an accessor,
- the accessor checks whether certain bit in an `Int` bitmap is set.

It can therefore be indicated that a lazy val should be re-evaluated by zeroing its bit (or the whole bitmap). This should be done, for example, in each iteration of the following `while` loop:

```
  var i = 1
  while (i < 10) {
    lazy val lazyWhile = i
    Console.println(lazyWhile)
    i += 1
  }
}
```

The discussion of the AST transforms makes clear where bitmaps of the form "`@volatile var bitmap :  Int = 0`" are emitted. Additionally, Appendix B showcases corner cases reproduced from the compiler's test suite.

A quick recap of SLS § 5.2:

> *The* **`lazy`** *modifier applies to value definitions. A lazy value is initialized the first time it is accessed (which might never happen at all). Attempting to access a lazy value during its initialization might lead to looping behavior. If an exception is thrown during initialization, the value is considered uninitialized, and a later access will retry to evaluate its right hand side.*

1.1 AST shapes of interest

`refchecks` may deliver one of three kinds of ASTs shapes, given a `lazy val` occurrence at the level of surface syntax. In all cases, accesses to the lazy value take the form of callsites to the "*accessor method*". These callsites need no rewriting during `lazyvals`. Instead:

- the bodies of such accessor methods are rewritten; and
- bitmaps are added if possible:
  - it's not possible to add bitmaps just yet for a trait-level lazy val. `AddInterfaces` has run by now, and thus traits have been split into shared implementation class and abstract interface. However none of them is useful to host bitmaps: each instance of a class where the trait is mixed-in should have its own copy of those bitmaps. That's why this rewriting will have to wait till `mixin`.

The three possible AST shapes that `refchecks` may deliver are:

1. for a `lazy val` of `Unit` type, `refchecks` delivers a single "getter" whose symbol is the `lazyAccessor` and whose body is the original `rhs` of the surface-syntax construct.
2. otherwise, `refchecks` replaces a `lazy val` owned by a trait with a pair *(ValDef, getter)* where:
   (a) an un-initialized `ValDef` is returned (different from the original one but with its same symbol)
   (b) the getter's body is the `rhs` of the original `ValDef`. Its symbol is the `lazyAccessor`.
3. otherwise, `refchecks` returns a pair *(ValDef, getter)* where:
   (a) the `ValDef` is as above.
   (b) the getter's body is of the form "`{ lhs = rhs ; lhs }`" where the `lhs` points to the `ValDef`.

2 End-to-end example: Lazy vals in loops

Sec. 1 hinted at the "re-evaluation" semantics of lazy vals directly contained in a loop (per iteration re-evaluation). Reviewing first the AST transform for this AST shape allows covering helper methods that will be reused later.

> *Nota bene: although we'll keep talking about "re-evaluating" a lazy val, please keep in mind that's more of an implementation-biased description. Semantically, a lazy val is evaluated just once within its evaluation scope, and in the case of loops there's an evaluation scope for each iteration. With this caveat, we'll continue using the "re-evaluation" terminology.*

Looking at the `transform()` handler for `while` loops won't reveal the whole story about this AST rewriting, given that it accesses visitor-level caches that have been populated by the time it runs. Still, we have to start somewhere. The snippet below applies to a `while`-loop, there's a similar one for `do-while` loops:

```
case l@LabelDef(name0, params0, ifp0@If(_, _, _))
if name0.startsWith(nme.WHILE_PREFIX) =>

  val ifp1 = super.transform(ifp0)
  val If(cond0, thenp0, elsep0) = ifp1

  if (LocalLazyValFinder.find(thenp0)) {

    val newIf = treeCopy.If(ifp1,
                            cond0,
                            typed(addBitmapDefs(sym.owner, thenp0)),
                            elsep0)
    treeCopy.LabelDef(l, name0, params0, newIf)

  } else {
    l
  }
```

Another Note: the above relies on a convention to encode loops. That convention is covered in the write-up "GOTO elimination for Scala ASTs". The parser puts trees in those shapes already (see `makeDoWhile()` and `makeWhile()`).

The above shows one of the patterns for rewritings in this phase: (a) detect whether bitmap-declaration-with-initialization should be emitted in given scope; (b) if so, grab from a visitor-level map the trees for that, using as key "a symbol uniquely associated with the evaluation scope". That's a lot of hand waving, but the picture will become more clear as Sec. 3 untangles the individual steps above.

3 Transformation patterns

3.1 Should any bitmaps be zeroed in an evaluation scope? (aka "`LocalLazyValFinder`")

The answer to this question can be found by exploring the `Tree` for the evaluation scope. Upon finding any *directly* contained `lazy val`, the answer is affirmative. That's what `LocalLazyValFinder` does (Listing 1). As already mentioned, the search doesn't delve into nested evaluation scopes. For example: upon arriving at the AST node for a loop, neither condition nor body will be visited:

```
case LabelDef(name, _, _) if nme.isLoopHeaderLabel(name) => /*- see? no super.traverse(tree) here. */
```

`LocalLazyValFinder` (Listing 1) nastily mutates a flag that is checked during the main transform (also reproduced below). I can't explain at this spot what effect it has, first I'll have to understand most of the phase to understand its interplay:

```
case d@DefDef(_, _, _, _, _, _) if d.symbol.isLazy && lazyUnit(d.symbol) =>
  d.symbol.resetFlag(symtab.Flags.LAZY)
  result = true
```

It's the task of another helper method (`addBitmapDefs()`, Sec. 3.3) to actually paste the trees to zero the bitmaps.

3.2 Body of an accessor (aka "`mkLazyDef()`")

A lazy-val-accessor tests whether re-evaluation should occur (by consulting the bitmap). On exit, the bit for the accessor's `lazy val` is set to `1` (an accessor can raise bits in a bitmap but not lower them to zero). Any non-`Unit` value is returned.

That's the contract of an accessor. The builder of accessors is:

```
def mkLazyDef(methOrClass: Symbol, tree: Tree, offset: Int, lazyVal: Symbol): Tree
```

Sec. 1.1 goes into details about the AST shapes arriving at `lazyvals` in the body of accessors. The resulting pattern involves (a) doubly-checked-locking idiom; and (b) bitmap-test. We start with the latter:

Listing 1: Sec. 3.1

```
object LocalLazyValFinder extends Traverser {
  var result: Boolean = _

  def find(t: Tree) = {result = false; traverse(t); result}
  def find(ts: List[Tree]) = {result = false; traverseTrees(ts); result}

  override def traverse(t: Tree) {
    if (!result)
      t match {
        case v@ValDef(_, _, _, _) if v.symbol.isLazy =>
          result = true

        case d@DefDef(_, _, _, _, _, _) if d.symbol.isLazy && lazyUnit(d.symbol) =>
          d.symbol.resetFlag(symtab.Flags.LAZY)
          result = true

        case ClassDef(_, _, _, _) | DefDef(_, _, _, _, _, _) | ModuleDef(_, _, _) =>

        case LabelDef(name, _, _) if nme.isLoopHeaderLabel(name) =>

        case _ =>
          super.traverse(t)
      }
  }
}
```

1. when the lazy val has type `Unit`, whatever incoming body "`rhs`" is wrapped in a `Unit`-expression of the form:

```
{
  {
     <rhs>
     bitmap$n = bimap$n | MASK
  }
  ()
}
```

2. otherwise, the body must be of the form "{ `lhs = rhs ; lhs` }". It is reshuffled and expanded to become:

```
{
  {
     lhs = <rhs>
     bitmap$n = bimap$n | MASK
  }
  lhs
}
```

In both cases, the result has the form "{ `stmts ; res` }" as needed for wrapping into what becomes the accessor body that this phase delivers:

```
{
  if ( (bitmap$n & MASK) == 0 ) {
     this.synchronized {
```

```
          if ( (bitmap$n & MASK) == 0 ) {
             <stmts>
          }
       }
    }
    <res>
  }
```

It may come as a surprise, but there's a reason why not all lazy-val-accessor bodies are rewritten as shown above (hint: this rewriting requires a bitmap, and Sec. 4 details why creating a bitmap might not be possible just yet).

3.3 Prepending trees to zero bitmaps in an evaluation scope (aka "`addBitmapDefs()`")

After reading this subsection, you still won't know where bitmaps are created (see Sec. 4 for that). We assume their symbols can be found keyed under the evaluation scope's symbol (in the `bitmaps` map). The contract of

```
def addBitmapDefs(methSym: Symbol, rhs: Tree): Tree
```

can be summarized as: give me the key for an evaluation scope ("`methSym`") and its AST node ("`rhs`"), I'll give you back that same evaluation scope with zero-ing `ValDef`s prepended.

`addBitmapDefs()` seems to have started its life receiving a method body (and the source comments still say so), but nowadays it's used to expand other sorts of eval scopess (including the body of a tail-called method, i.e. a loop that follows a certain naming convention in its `LabelDef`).

4 Creating bitmaps

4.1 Block-level vs. class-level bitmap initializers

Creating a bitmap (Sec. 3.2) is tricky because doing so requires settling on an evaluation scope for the bitmap (where it will be zeroed), which can be one of "block" or "template".

1. In the former case, prepending a bitmap initializer to a block effectively completes the lowering of its lazy-val: no retouching is needed by any phase afterwards, and we can safely reset the `LAZY` flag.
2. Templates however will undergo further transformations:
   (a) Before `constructors`, templates contain executable statements and `ValDef`s with executable RHSs, whose evaluation logically belongs in the primary constructor. In order to obtain Java-like constructors, `constructors` triages the stmts of a template into early-defs, class-level, and main-constructor.
   (b) `mixin` will allocate in host classes (or objects) whatever dedicated copies they should get of mixed-in trait members.

As a rule of thumb, `lazyvals` leaves as-is those *(ValDef, getter)* pairs directly owned by a class, so that `mixin` knows what to do with them. As a sidenote: `constructors` will classify them as class-level, thus "leaving them where they are" (and the `ValDef` has an `EmptyTree` rhs which won't be moved into the primary constructor).

"Leaving as-is" above means that no bitmaps are created (nor bitmap-initializer prepended) for class-level lazy vals (because the right evaluation scope can only be determined later). This does not preclude the *body* of one such accessor from being rewritten during `lazyvals` (i.e., its `Block` may be prepended with bitmap initializers for lazy vals owned by the accessor).

This division of labor results in *two* rewritings that a `DefDef` whose `symbol.isLazy` may undergo, plus a common rewriting:

1. The body of a class-level `DefDef` (whether lazy or not) is "left as-is" (i.e., transformed as per `super.transform()`).

2. Now comes the tricky case: doing bookkeeping for a lazy accessor not directly enclosed in a class. Sec. 4.2 covers all that bookkeeping.

3. In any case, add bitmap initializers for the method's body:

```
def addBitmapDefs(methSym: Symbol, rhs: Tree): Tree
  treeCopy.DefDef(tree, mods, name, tparams, vparams, tpt,
          if (LocalLazyValFinder.find(res)) typed(addBitmapDefs(sym, res)) else res)
```

4.2 Non-class-level lazy-val-accessor

We arrived here from Sec. 4.1. To make a long story short: a bitmap symbol is created only as a side-effect of rewriting the body of a lazy-val-accessor (Sec. 3.2) and that helper method is only invoked from here.

> *Therefore, no bitmaps are allocated by `lazyvals` for class-level lazy-vals. That will also have to wait till `mixin`. As a consequence, the rewriting for `Template` nodes shouldn't try to grab any class-level bitmaps. That's another story (Sec. 4.3).*

We have to determine the (enclosing) evaluation scope of this lazy value, and grab its symbol, so as to hand it to `mkLazyDef()`:

> *To recap, `mkLazyDef()` is handed an "evaluation-scope-symbol", which besides being used as key in `bitmaps` for the new bitmap-symbol also offices as owner of the bitmap-symbol. We review next what those "evaluation-scope-symbols" may be (hint: "`enclosingClassOrDummyOrMethod`").*

See also Figure 1 (`newLocalDummy`).

```
TODO


  case class Template(parents: List[Tree], self: ValDef, body: List[Tree])
      extends SymTree {
    //
    // The symbol of a template is a local dummy (@see Symbol.newLocalDummy).
    // The owner of the local dummy is the enclosing trait or class.
```

Figure 1: Sec. 4.2

```
  // The local dummy is itself the owner of any local blocks.
  //
  // For example:
  //
  // abstract class C {
  //
  //   def foo; // owner is C
  //
  //   {
  //     def bar() {} // owner is local dummy
  //   }
  //
  //   Console.println("abc") // symbol.owner won't help us here (the symbol of an Apply is fun.symbol)
  // }
}
```

4.3 How to rewrite `Template` nodes without class-level bitmaps

As we saw in Sec. 4.2:

> *No bitmaps are allocated by `lazyvals` for class-level lazy-vals. That will also have to wait till `mixin`. As a consequence, the rewriting for `Template` nodes shouldn't try to grab any class-level bitmaps.*

```
TODO
```

4.4 Are all bitmaps created by the time they are needed?

As we saw a bitmap symbol is created only as a side-effect of rewriting the body of a lazy-val-accessor (Sec. 3.2, which is only invoked from Sec. 4.2). Does this guarantee that lookups on an eval-scope will find all relevant bitmaps?

1. As part of rewriting an accessor body. In this case, one bitmap is of interest (given by a one-based index) among those returned by the lookup.

2. To prepend zeroing `ValDef`s to an eval-scope. In this case all bitmaps (a `List[Symbol]`) are needed for prepending.

   Note: the second case also happens as at the end of "rewriting an accessor body", but we can still discuss them separately.

Regarding (1), those lazy-val-accessors that do get rewritten in `lazyvals` always succeed with lookups: otherwise `getBitmapFor()` creates a bitmap symbol and adds it to the `bitmaps` map!

The above leaves only (2) to worry about. Yes, all lazy accessors directly contained in an eval-scope will have been rewritten by the time zeroing `ValDef`s are about to be prepended. Why? Because all invocations to `addBitmapDefs()` receive an already-transformed `rhs`.

4.5 Naming convention for bitmaps

Bitmap names chosen in `lazyvals` lack suffix:

```
/** The name of bitmaps for initialized (public or protected) lazy vals. */
def bitmapName(n: Int): TermName = bitmapName(n, "")

private def bitmapName(n: Int, suffix: String): TermName =
  newTermName(BITMAP_PREFIX + suffix + n)

val BITMAP_PREFIX            = "bitmap$"
```

However we'll see a variety of suffixes in `mixin`:

- *"trans$"* for initialized transient lazy vals
- *"priv$"* for initialized private lazy vals.
- *"init$"* for checkinit values
- *"inittrans$"* for checkinit values that have transient flag

A Backgrounders

A.1 `Transformer` vs. `TypingTransformer` differences

From a distance, `lazyvals` offers no surprises (Figure 2): it emits trees using `ast.TreeDSL`, and it contains one traverser (`LocalLazyValFinder`) and one transformer (`LazyValues`).

This phase mixes trait `TypingTransformers` which means that some facilities are available to type trees once they're built. In a nutshell, a phase not mixing `TypingTransformers` (such as `CleanUp`) has to define a transformer-local `Typer` instance, obtained from `global.analyzer.typer`. For example, `CleanUp` does it like this:

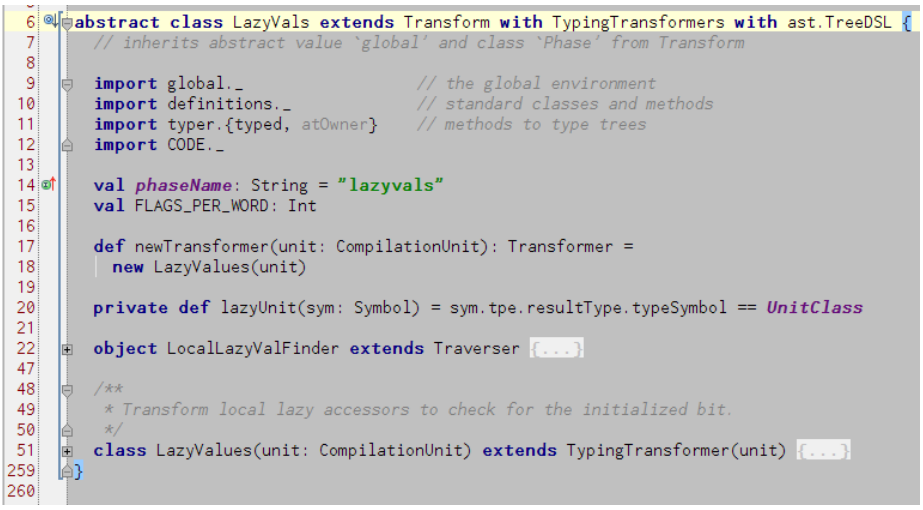


Figure 2: Sec. 1

```
private var localTyper: analyzer.Typer = null
. . .
override def transform(tree: Tree): Tree = tree match {
    . . .
    case Template(parents, self, body) =>
        localTyper = typer.atOwner(tree, currentClass)
```

This works because as shown above `CleanUpTransformer` takes care of updating the local typer. Instead, a transformer extending the abstract class `TypingTransformer` gets that for free (the typer is updated upon visiting a `Template` or a `PackageDef`), due to the interplay of the `transform` override and the `atOwner` method overloads in `TypingTransformer` (Figure 3).

If this is the first time you hear about `atOwner`, perhaps you should also know that there are many of them, some returning a `Typer` (e.g. the overloads in `Typers#Typer`), others `Unit` (e.g. in `Traverser`), and yet others whatever its by-name param evaluates to (e.g. in `Transformer`). Please fest on the details shown in Figure 4.

A.2 Tree traversing patterns

`LocalLazyValFinder` showcases how to selectively skip traversing certain children (by skipping invoking `super.traverse(t)`). A variant of this pattern:

```
class FindTreeTraverser(p: Tree => Boolean) extends Traverser {
  var result: Option[Tree] = None
  override def traverse(t: Tree) {
    if (result.isEmpty) {
      if (p(t)) result = Some(t)
      super.traverse(t)
    }
  }
}
```

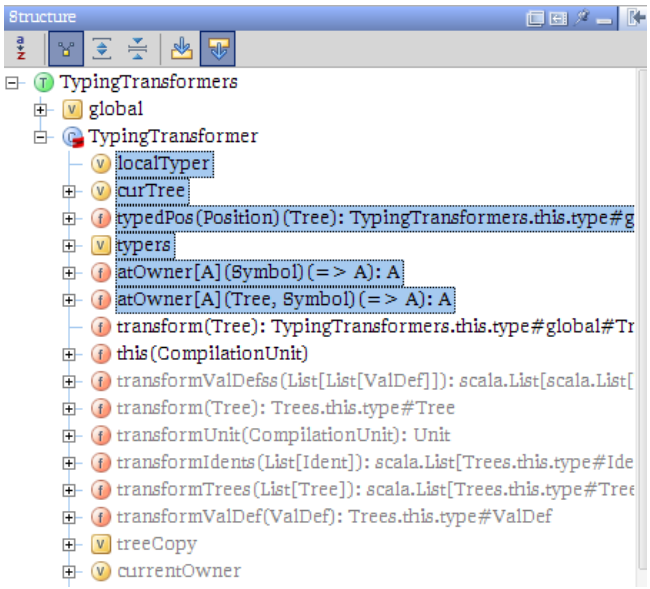


Figure 3: Sec. A.1

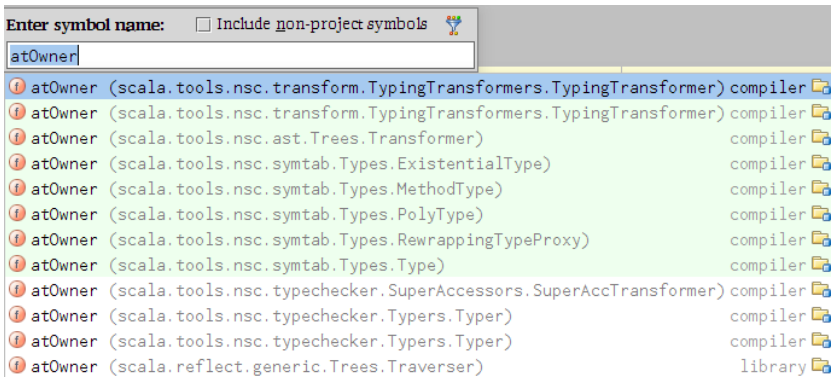
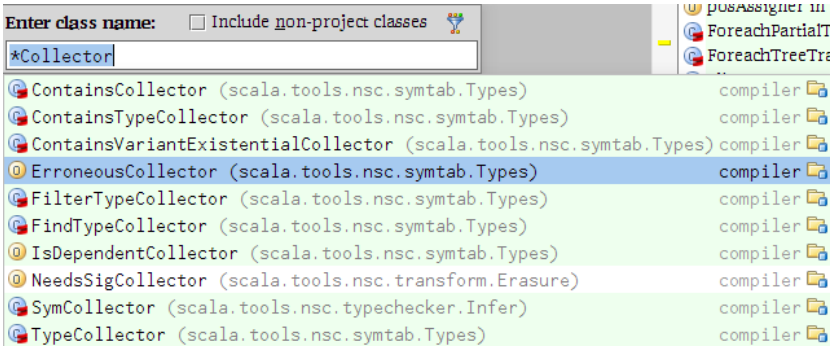


Figure 4: Sec. A.1

Yet more examples of type, symbol, tree, etc. collectors:



A.3 Beakpoints please

Maybe you too have noticed that (for some reason) IDE debuggers will sometimes ignore breakpoints inside closures. There's a workaround, if a bit cumbersome: define an auxiliary method with the closure contents (adding parameters for whatever bindings are in effect), and place the breakpoint inside the method (let's call it a "debugger-friendly" coding style). For example:

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

  tree match {
    case DefDef(mods, name, tparams, vparams, tpt, rhs) => {
      atOwner(tree.symbol) {
        debugDefDef(tree, sym, mods, name, tparams, vparams, tpt, rhs)
      }
    }
   . . .
```

B Corner cases of `lazy val`

B.1 Loops

Lazy values declared inside a loop are initialized once per iteration:

```
  for (i <- 1 to 10) {
    lazy val lazyFor = i
    Console.println(lazyFor)
  }
```

```
  var i = 1
  while (i < 10) {
    lazy val lazyWhile = i
    Console.println(lazyWhile)
    i += 1
  }
}
```

B.2 Recursion, Nesting

```
/** test recursive method with lazy vals and a single forced */
def testLazyRec(n: Int): Int = {
  lazy val t = { println("forced lazy val t at n = " + n); 42 }
  if (n > 0)
    testLazyRec(n - 1)
  else
    t
}
```

```
/** test recursive method with lazy vals and a all vals forced */
def testLazyRecMany(n: Int): Int = {
  lazy val t = { println("forced lazy val t at n = " + n); 42 }
  if (n > 0) {
    testLazyRecMany(n - 1);
    t*t
  } else
    t
}
```

```
// see #1589
object NestedLazyVals {
  lazy val x = {
    lazy val y = { println("forcing y"); 42; }
    println("forcing x")
    y
  }

  val x1 = 5 + { lazy val y = 10 ; y }

  println(x)
  println(x1)
}
```

B.3 Traits, Thunks

```
trait TNestedLazyVals {
  lazy val x = { lazy val y = 42; y }
}

object ONestedLazyVals extends TNestedLazyVals {
  println(x)
}
```

```
class Lazy(f: => Int) {
     lazy val get: Int = f
}

object Test extends App
{
 val buffer = new scala.collection.mutable.ListBuffer[Lazy]

 // This test requires 4 Mb of RAM if Lazy is discarding thunks
 // It consumes 4 Gb of RAM if Lazy is not discarding thunks
```

```
 for (val idx <- Iterator.range(0, 1024)) {
     val data = new Array[Int](1024*1024)
     val lz: Lazy = new Lazy(data.length)
     buffer += lz
     lz.get
 }
}
```