Lambda Lifting

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

August 28th, 2011

Abstract

The reasoning behind the `lambdalift` transformation can be found in Philippe Altherr's PhD report [1] (which was completed in 2006). In contrast, these notes contain a code walkthrough of the `lambdalift` implementation as of `r24886`. This phase turns local methods and classes into class-level definitions ("lifting"). A method or class is local iff its definition is lexically enclosed by a block expression or by a method definition. Lifting involves adding extra value params and value arguments, and rephrasing accesses to free variables.

Just to clarify: inner classes are not "lifted" by `lambdalift`. Instead, `explicitouter` turns inner classes into nested classes, `lambdalift` turns local classes into nested classes, and `flatten` (a `forJVM` phase) un-nests them into top-level classes.

```
          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

1.1 Terminology

Throughout this write-up, the following terminology is used:

- **local value**: a function-local value, i.e. a value which normally exists for as long as an activation record is available on the call stack. Examples: method-level and block-level `val` and `var`; and value parameters (but not a `val` or `var` value param to a primary constructor). Before `lambdalift`, the symbol of a local value `isLocal`:

```
/** Is this symbol locally defined? I.e. not accessed from outside 'this' instance */
final def isLocal: Boolean = owner.isTerm
```

- **local def**: A method or class defined in a method, constructor, or block. In this case, its symbol `isLocal`. For example, `LocalA` and `LocalB` are local classes:

```
object Test {

  { class LocalA }

  def m() = { class LocalB }

}
```

- **liftable method/class**: a local method/class (*before* being lifted)
- **lifted method/class**: the counterpart to a liftable, *after* the transformation has run. It has been renamed, to avoid name collision.
- **free var**: a reference to a non-locally defined local value. We say the local value has been *captured* by the reference (it's only local values that can be captured). Renaming also done here.
- **extra set of a (local) method**: "*The set of variables for which a function needs an extra parameter is called its extra set ... The extra set of a function depends on the extra sets of the functions it calls*" [1, §3.2.1] For example, a local function may contain no free vars, yet call (local) functions which do.
- **extra set of a (local) class**: A class may capture locals (Sec. 4.1) thus we may speak of its extra set. That class can be local, or nested to some depth in a local class (details under "*A comment about inner classes*" in Sec. 1.2).
- **conveyor param**: a value param added to a lifted method or constructor in order to provide one of:
  - For captured read-only locations:
    - a primitive value (for primitive free vals), or
    - an object-reference (for non-primitive free vals)

  - For captured read-write locations:
    * a `Ref` or `VolatileRef` wrapper (for both `AnyVal` and `AnyRef` free vars).

- **`Ref` wrapper**: (a.k.a. "proxy"). An instance of a Java class, used to emulate "passing by reference" a data location (emulation works provided that both definition site and all access sites are rewritten to follow the `Ref` protocol). For example:

```
package scala.runtime;

public class VolatileBooleanRef implements java.io.Serializable {
    private static final long serialVersionUID = -5730524563015615974L;

    volatile public boolean elem;
    public VolatileBooleanRef(boolean elem) { this.elem = elem; }
    public String toString() { return String.valueOf(elem); }
}
```

- **conveyor argument**: a value param for a conveyor param (in a method or constructor).

1.2 Recipe

What follows is very condensed description of lambda lifting. For details, please check the 17-page Ch. 3 (Lambda Lift) in Altherr's PhD report [1], especially as it discusses in §3.5 issues around lifting of methods and classes that refer to type variables defined in an enclosing context. Currently `lambdalift` runs after `erasure` but from a Scala.NET perspective that discussion is certainly relevant. Going deeper, [1, §5.6] formalizes the transformation in terms of *ScalaCore*, a formal Scala subset.

Methods and classes are lifted differently:

- Because a local method M is not visible outside its owner, all its callsites are known. This allows changing the signature of M as part of lifting (by adding conveyor params). No callsite can be missed and thus all are rewritten accordingly.

- Instances of a local class C may escape the activation record of the method owning C, therefore the public API of C (except constructors) should be preserved ("public API" means: API inherited from non-local classes). This is achieved by leaving methods signatures as-is, and by rewriting constructors (and their callsites) to account for conveyor params:

  > *local classes are augmented with fields containing the values of the accessed variables of the enclosing functions. If some instances of a local class outlive the invocation of their enclosing function, they can still refer the variables of that function through ... the additional fields. [1, §3.1].*

So far we've focused on the extra sets of local methods. Local classes have extra sets too, which amount to extra fields where the values of conveyor params are made available (across methods and nested classes). Local capture, when

lexically enclosed in a constructor, does not contribute to the extra set of a class, but only to that of the constructor. In contrast, local capture in methods of the class will result in extra fields [1, §3.2.2]:

> *References to free variables within constructors [of a liftable class] do not contribute to the extra fields of their class. They contribute only to the extra parameters of the constructor in which they occur. In fact, constructors are treated as if they were functions defined in the same scope as their class*

A comment about inner classes. There are no more inner classes by the time `lambdalift` runs. Instead, an AST arriving at `lambdalift` exhibits a class `N` nested in a local class `L` (Sec. 1.5) for what used to be an outer-inner configuration. In this case:

- After lifting `L`, any nested class it contains will remain in place.
- What to do about usages of captured locals in `N`? Its enclosing class plays a role: the extra set of a class includes those of its nested classes and, because of this, `L` already gets extra fields *for all captured locals in `N`*. Therefore, `N` can access those captured locals via outer-paths (however, `N`'s constructors need conveyor params). Details in Sec. 1.5.

1.3 Example 1: Lifting a local method (`loop` in `fold`)

The example in [1, Ch. 3] and reproduced below,

```
object Example {
  def fold(z: Int, ls: List[Int], f: (Int, Int) => Int): Int = {
    def loop(x: Int, ys: List[Int]): Int = { if (ys.isEmpty) x else loop(f(x, ys.head), ys.tail) }
    loop(z, ls)
  }
}
```

is rewritten as shown in Listing 1 on p. 6 (using `-no-specialization -g:notailcalls -Xprint:lambdalift`)

1.4 Example 2: Lifting a local class (anonymous closure)

After and before `lambdalift` in Figure 1 on p. 7.

```
  def main(args: Array[String]) {
    args find { s => s.length == args.length }
  }
```

1.5 Example 3: Interplay with outer instances

A liftable def may contain usages of two kinds of non-locally defined terms:

1. values or methods that are `isLocal`; and
2. values or methods belonging to an outer instance.

Listing 1: Sec. 1.3

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

  final object Example extends java.lang.Object with ScalaObject {

    def this(): object Example = {
      Example.super.this();
      ()
    };

    def fold(z: Int, ls: List, f$1: Function2): Int = Example.this.loop$1(z, ls, f$1);

    final private[this] def loop$1(x: Int, ys: List, f$1: Function2): Int = {
      if (ys.isEmpty())
        x
      else
        Example.this.loop$1(
          scala.Int.unbox(
            f$1.apply(scala.Int.box(x), ys.head())
          ),
          ys.tail().$asInstanceOf[List](),
          f$1
        )
    } // end of def loop$1

  } // end of object Example

}
```

As we'll see, only the former are "captured" and require rewriting by `lambdalift`, the latter are accessed over an outer-accessor path as any other outer-reference (and it's `explicitouter` that performs the required rewriting).

Example:

```
def f(farg: Int) = {

  class O {
    var ofield = "abc"

    class I {
      val i1 = farg
      val i2 = ofield
    }

  }
}
```

Only `farg` `isLocal`, classes `O` and `I` form an outer-inner pair, and of the two accesses in `I` only `farg` is free (as per the terminology in Sec. 1.1). `-Xprint:explicitouter` shows that `I` contains:

```
private[this] val i1: Int = farg;
<stable> <accessor> def i1(): Int = I.this.i1;

private[this] val i2: java.lang.String = I.this.Test$O$I$$$outer().ofield();
```

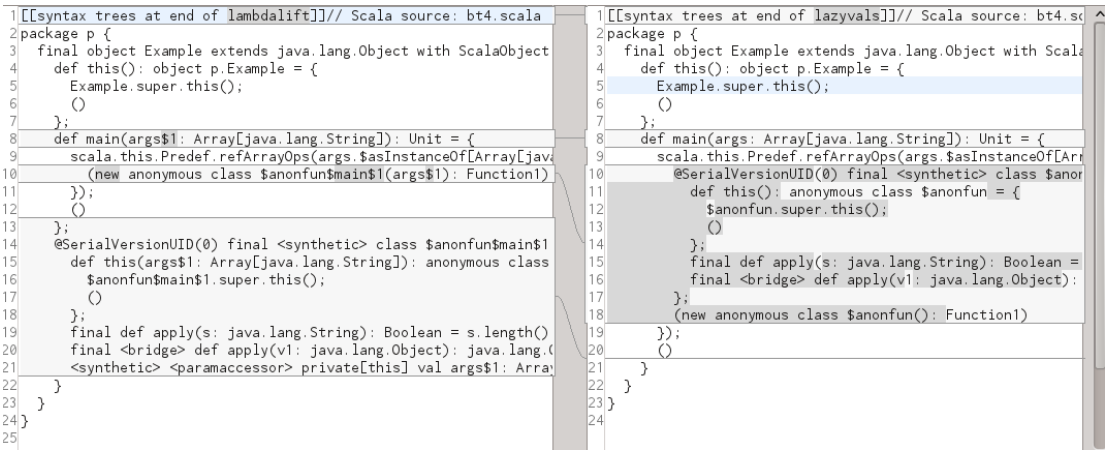


Figure 1: Sec. 1.4

```
<stable> <accessor> def i2(): java.lang.String = I.this.i2;
```

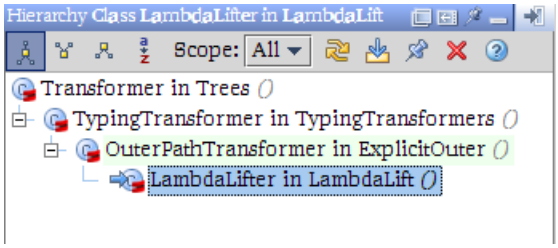
Looking ahead, `-Xprint:lambdalift` shows that `farg` was passed via a conveyor param to `O`'s constructor but not to `I`, being accessed instead from there over an outer-path (to build the AST for that path `LambdaLifter` inherits from `OuterPathTransformer`, Sec. 2.2):

```
class O$1 extends java.lang.Object with ScalaObject {

  def this(farg$1: Int): Test#O$1 = {
    O$1.super.this();
    ()
  };

  <synthetic> <paramaccessor> val farg$1: Int = _

  . . .

      class I extends java.lang.Object with ScalaObject {
        def this($outer: Test#O$1): Test#O$1#I = {
          I.super.this();
          ()
        };

        private[this] val i1: Int = I.this.Test$O$I$$$outer().farg$1;
        <stable> <accessor> def i1(): Int = I.this.i1;

        . . .
      };

  . . .
```

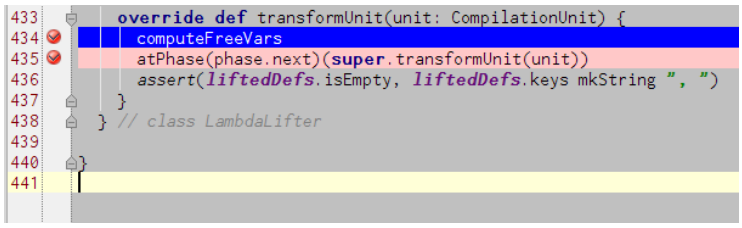
2 A 10'000-ft view of the code

2.1 A brief choreography of lifting

Just to be clear about what the `super` calls of `transformUnit(unit)`, `transformStats(stats, exprOwner)`, and `transform(tree)` refer to, here are the base classes of `LambdaLifter`:



It all starts here:



Because it runs in `transformUnit`, `computeFreeVars` is run just once before any node is transformed. Nodes are transformed under the auspices of `atPhase(phase.next)` in a `LambdaLifter` override:

```
override def transform(tree: Tree): Tree =
  postTransform(super.transform(tree) setType lifted(tree.tpe))
```

The operation of the `lifted` type-map does not depend on traversal-order but just on its `Type` argument. Other operations do depend (via `liftedDefs`) on traversal order, in particular:

```
/** Transform statements and add lifted definitions to them. */
override def transformStats(stats: List[Tree], exprOwner: Symbol): List[Tree] = {
  . . .
```

On the way back to the tree root, `postTransform(tree: Tree)` runs (Sec. 5.1). Among other rewritings, it adds params to constructors of lifted classes and to lifted methods, and replaces captured variables.

2.2 Need to know before delving: `OuterPathTransformer`

Figure 2 depicts a bird's eye view of the `LambdaLift InfoTransform`, with two big fragments collapsed: the `LambdaLifter` transformer and the `lifted` type map.

But first, why does the transformer extend `explicitOuter.OuterPathTransformer(unit)`? Looks like that class is not used anywhere in `ExplicitOuter` itself (which instantiates an `ExplicitOuterTransformer` in `newTransformer`) but only in `lambdalift` (Figure 3). What is `explicitOuter.OuterPathTransformer(unit)` good for?

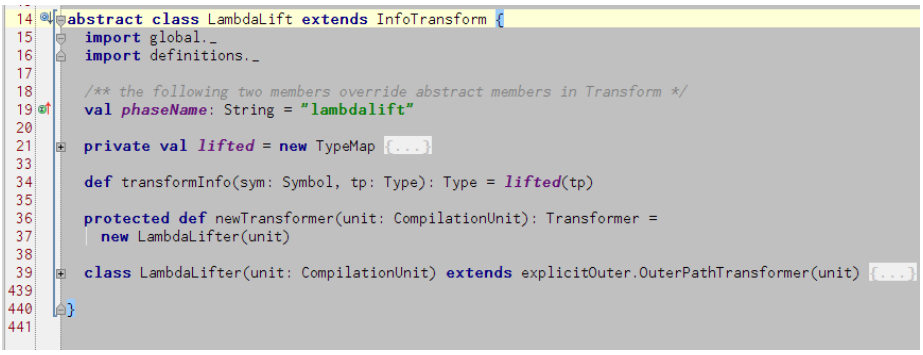


Figure 2: Sec. 2.2

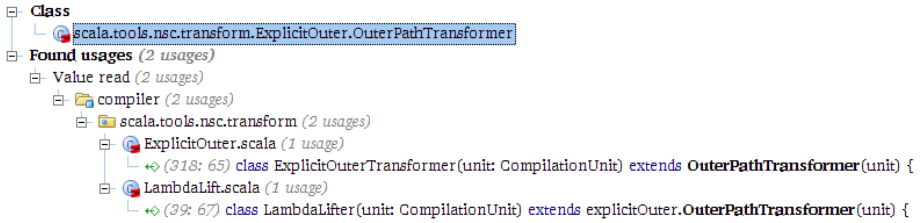


Figure 3: Sec. 2.2

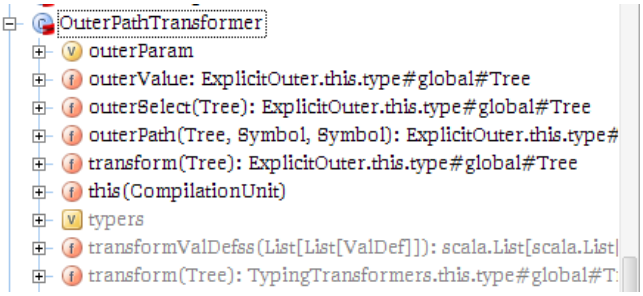
Recap: any transformation running after `explicitouter` cannot add inner classes and expect them to be lowered to nested classes. Instead, a nested class can be added and the lowering performed "manually": any outer-instance references have to be expressed as (chains of) outer-accessor invocations. BTW, in `forJVM` compilation mode, nested classes can be added as late as `constructors` because then `flatten` runs. In `forMSIL` compilation mode, nested classes can be added as late as `cleanup`: no `flatten` takes place because class-nesting can be represented at the bytecode level.

A phase which adds a nested class with some "manual" building of outer-path is `constructors`, as part of processing `extends DelayedInit` (a nested class is added for the delayed-init closure).

Back to `lambdalift`. This phase preserves the nesting of any classes nested in a local class, and thus it may be necessary to build an AST for an outer-path (to the extra field of the enclosing class, as illustrated in Sec. 1.5). For that, `OuterPathTransformer` is the tool of choice.

The essential pieces of information about `OuterPathTransformer` are (a) header source comment; and (b) structure. Both are straightforward enough to cut&paste, so here they go:

```
/** A base class for transformers that maintain outerParam
 *  values for outer parameters of constructors.
 *  The class provides methods for referencing via outer.
 */
abstract class OuterPathTransformer(unit: CompilationUnit) extends TypingTransformer(unit) {
```



Besides keeping the `OuterPathTransformer`'s `outerParam` up-to-date, three utility tree builders are provided, summarized below:

```
/** The first outer selection from currently transformed tree.
 *  The result is typed but not positioned.
 */
protected def outerValue: Tree =
  if (outerParam != NoSymbol) ID(outerParam)
  else outerSelect(THIS(currentClass))

/** Select and apply outer accessor from 'base'
 *  The result is typed but not positioned.
 *  If the outer access is from current class and current class is final
 *  take outer field instead of accessor
 */
private def outerSelect(base: Tree): Tree = {
  . . .
}
/** The path
 *  <blockquote><pre>'base'.$outer$$C1 ... .$outer$$Cn</pre></blockquote>
 *  which refers to the outer instance of class to of
 *  value base. The result is typed but not positioned.
 */
protected def outerPath(base: Tree, from: Symbol, to: Symbol): Tree = {
  . . .
}
```

3 Gathering information for later: `freeVarTraverser`

Unsurprisingly, `freeVarTraverser` (Listing 2) can find (a) local definitions and (b) usages of locals as it traverses a unit.

1. All of the former are marked "`renamable`". Renaming is needed due to the possibility that a definition already exists with that name in the target `ClassDef` (i.e., the `ClassDef` that becomes the new home for the lifted def). Renaming adds `$` as suffix (for details see Sec. 4.4) to rule out name clashes.

2. Regarding usages of locals, they fall into two syntactic variations:

   (a) `Ident(name)` matches two things. First, invocations of local methods are never path-qualified and thus always match `Ident(name)`. Second, usages of value params (always `islocal`), and usages of method-level or block-level vals or vars (similarly, always `islocal`); also match `Ident(name)`.

Listing 2: Sec. 3

```
val sym = tree.symbol;
tree match {

  case ClassDef(_, _, _, _) =>
    liftedDefs(tree.symbol) = Nil
    if (sym.isLocal) renamable addEntry sym /*- (1.a) definition of local class. */

  case DefDef(_, _, _, _, _, _) =>
    if (sym.isLocal) {                  /*- (1.b) definition of local method. */
      renamable addEntry sym
      sym setFlag (PRIVATE | LOCAL | FINAL)
    } else if (sym.isPrimaryConstructor) { /*- (2.b) definition of (an always non-local) primary constructor
                                             (not necessarily of a local-class). */
      symSet(called, sym) addEntry sym.owner
    }

  case Ident(name) =>
    if (sym == NoSymbol) {
      assert(name == nme.WILDCARD)
    } else if (sym.isLocal) { /*- (2.a) access to a local method or value. */
      val owner = currentOwner.logicallyEnclosingMember
      if (sym.isTerm && !sym.isMethod) markFree(sym, owner)
      else if (sym.isMethod) markCalled(sym, owner)
        //symSet(called, owner) addEntry sym
    }

  case Select(_, _) => /*- (2.b) invocation of the constructor of a local class. */
    if (sym.isConstructor && sym.owner.isLocal)
      markCalled(sym, currentOwner.logicallyEnclosingMember)

  case _ =>
}
```

i. For callsites invoking local methods, we'll want to track the lexical context where the callsite occurs (that's what `markCalled()` does, as part of building the call-graph, Sec. 3.1)

ii. For usages of value terms, a decision has to be made whether the usage implies capture (i.e., whether the local is free in the context it occurs). That decision is done inside `markFree()` (which thus may not mark the access as free, after all). For `markFree()` to make that decision, the lexical context where the access occurs is passed as second argument. Details in Sec. 4.2.

(b) instantiations of local classes pick a constructor name. They are matched by the pattern:

```
case Select(_, _)
if (sym.isConstructor && sym.owner.isLocal)
=>
```

The snippet above participates (along with others) in building the call-graph (Sec. 3.1).

3.1 Call graph

The call graph is used just to compute extra sets. Other than that, it's used nowhere else.

For the purposes of `lambdalift`, the call-graph is a one-to-many symbol-map ("`called`") that associates each key (a symbol representing the caller) with a set of symbols for its callees. Each entry in this map contains (in general several) `(caller, callee)` pairs. Each such pair can be classified into:

1. `callee` is method or constructor, Sec. 3.1.1
2. `callee` is class, Sec. 3.1.2

A few mnemonics:

```
                    markCalled(callee, caller)
                                ≡
              symSet(called, caller) addEntry callee
                                ≡
 called.getOrElseUpdate(caller, new TreeSet[Symbol]) addEntry callee
```

3.1.1 `callee` is method or constructor

Given a method or constructor tracked as `callee`, all that can be said about its corresponding `caller` is that it's the `logicallyEnclosingMember` for some callsite targeting the callee. More can be said about the callee:

1. `(callee.isMethod && callee.isLocal)`, or
2. `(callee.isConstructor && callee.owner.isLocal)`

To recap, `somesym.logicallyEnclosingMember` is the method or class *logically* enclosing `somesym`'s definition. The snippets in `freeVarTraverser` that contribute pairs as above are resp.:

1. For the method-`callee` case:

```
val owner = currentOwner.logicallyEnclosingMember
if (sym.isTerm && !sym.isMethod) markFree(sym, owner)
else if (sym.isMethod) markCalled(sym, owner)
```

2. For the constructor-`callee` case:

```
case Select(_, _) =>
  if (sym.isConstructor && sym.owner.isLocal)
    markCalled(sym, currentOwner.logicallyEnclosingMember)
```

3.1.2 `callee` is class

A pair can assume another shape: a `caller.isPrimaryConstructor` is tracked as "invoking" its declaring class (not the other way around). Irrespective of whether the declaring class is local or not. The purpose of this arrangement is described in Sec. 4.1.

Summing up,

1. `caller.isPrimaryConstructor`
   implies both
   `called(caller).size == 1` and `callee.isClass`

The snippet in `freeVarTraverser` that adds pairs as above is:

```
case DefDef(_, _, _, _, _, _) =>
  if (sym.isLocal) {
    renamable addEntry sym
    sym setFlag (PRIVATE | LOCAL | FINAL)
  } else if (sym.isPrimaryConstructor) {
    symSet(called, sym) addEntry sym.owner
  }
```

3.2 Postconditions after `freeVarTraverser` has run

After `freeVarTraverser` has run on a compilation unit:

- `liftedDefs` contains an entry for each `ClassDef` C in the unit. The entry targets no trees so far (later, the key will address the liftable `DefDef`s and `ClassDef`s that will become members of C).
- A call-graph has been built as described in Sec. 3.1.

Useful postconditions for free variables haven't been established yet, that will have to wait till Sec. 4.2.

By now `renamable` contains the symbol of each liftable `ClassDef` and `DefDef`, but we're not done adding symbols. Those for captured local values are missing, to be added while computing extra sets (Sec. 4.1). In particular, it's the `markFree` invocation there that effects this.

4 Before nodes are transformed: `computeFreeVars`

Given that `computeFreeVars` is invoked in `transformUnit`, it can prepare data before any node is transformed by the `transform` override. After gathering information about locals via `freeVarTraverser` (Sec. 3) `computeFreeVars` goes on to:

- compute extra sets (Sec. 4.1 and Sec. 4.2),
- rename some symbols (Sec. 4.4), and
- create others ("proxies", Sec. 4.5).

The bulk of tree-rewriting and re-attribution however takes places during `transform()` (Sec. 5).

4.1 Computing extra sets (i.e., populating the `free` map)

A method M_1 (liftable or not) may call a liftable method M_2 that captures some value v defined outside M_1. In this case, v should be marked as free in the invoker too (i.e., in M_1) irrespective of whether it contains a reference to v or not (this is necessary to later add a conveyor param to M_1 that will convey

the captured value when invoking M_2). This iterative, upwards, percolation is achieved by:

```
do {
  changedFreeVars = false
  for (caller <- called.keys ; callee <- called(caller) ; fvs <- free get callee ; fv <- fvs)
    markFree(fv, caller)
} while (changedFreeVars)
```

Back in Sec. 3.1.2 a "`(callee, caller)`" arrangement is described where the `caller` (a primary constructor) is tracked as "invoking" its declaring class. Why? A class may have a non-empty extra set, in which case its constructor should include those symbols in its own extra set (so as to assign from conveyor params to extra fields). Summing up, the arrangement in question allows free vars to "percolate upwards" from the class to its constructor.

Quoting from Altherr's PhD report:

> ***3.3.5 Generalized Algorithm***
>
> *When classes are taken into account, the lambda lifting algorithm stays the same except that extra sets are also computed for local classes and local classes are augmented with an extra field for each variable in their extra set. Like local functions, local classes are attributed a call set and the extra sets of local classes are computed the same way as the extra sets of local functions.*
>
> *Constructors of local classes are treated like local functions defined in the same scope as their class. While computing extra sets, it is important to remember that a constructor is implicitly called in each instance creation. Furthermore, the extra set of each primary constructor has to be augmented with its class. Indeed, in the step "parameter and argument addition", the bodies of primary constructors have to be augmented with code to initialize the extra fields of their class.*
>
> *Methods and inner classes are not treated like local functions and local classes but are considered as a part of their enclosing class. They remain in place and contribute to the free variables and the call set of their enclosing class. So the free variables of a local class are all the free variables that occur in its methods and in methods of its inner classes.*

4.2 Invariant for the `free` map (extra sets)

After extra sets have been computed (Sec. 4.1) the following holds:

```
free toList forall {
  (enclosure, accessedVars)
  =>
  (enclosure.isMethod || enclosure.isClass) &&
  (accessedVars forall {accessed => accessed.isTerm && !accessed.isMethod && accessed.isLocal } )
}
```

```
TODO assert()
```

How was that established? An invocation of the form `markFree(accessed, enclosure)` gets to decide whether `accessed` occurs free in `enclosure`, recording that fact in the `free` map (a one-to-many map). Three conditions must be met (simplifying somewhat):

1. the access should occur lexically within `enclosure`
2. `accessed` hasn't been defined in `enclosure`
3. there is no class between `enclosure`'s owner `accessed`'s owner. The need for this condition can be seen at the end of Sec. 1.5.

If met, `free` is updated (more mnemonics):

```
symSet(free, enclosure) addEntry accessed
                    ≡
free.getOrElseUpdate(enclosure, new TreeSet[Symbol]) addEntry accessed
```

4.3 Scala.NET and `markFree()`

A lot of processing goes on in `markFree`, for example the following which has to do with "proxying".

```
TODO Scala.NET alarm: what now is an ObjectRefClass should be an Ref[T].
     .NET valuetypes should have their their own ValueRef[T] (with T upperbounded by System.Value)
```

```
TODO Scala.NET alarm: what now is an ObjectRefClass should be an Ref[T].
     .NET valuetypes should have their their own ValueRef[T] (with T upperbounded by System.Value)

  if (sym.isVariable && !sym.hasFlag(CAPTURED)) {
    sym setFlag CAPTURED
    val symClass = sym.tpe.typeSymbol
    atPhase(phase.next) {
      sym updateInfo (
        if (sym.hasAnnotation(VolatileAttr))
          if (isValueClass(symClass))
            volatileRefClass(symClass).tpe
          else
            VolatileObjectRefClass.tpe
        else
          if (isValueClass(symClass))
            refClass(symClass).tpe
          else
            ObjectRefClass.tpe
      )
    }
  }
```

4.4 Renaming of liftables and captured locals

Renaming applies uniformly to all liftables and captured locals, including value params. For example, `fargA` is renamed to `fargA$1`, to avoid a name clash in case `ff()` declared an `fargA` value param.

```
def f(fargA: Int) = {
  def ff() = fargA
}
```

In the example below, renaming the value param is necessary because `L` gets an extra field for it. Also in this case, `L` could (conceivably) have declared a field with that name.

```
def f(fargA: Int) = {
  class L { def ff() = fargA }
}
```

Please notice the following allows experimenting with alternative naming schemes from the command line:

```
object NameTransformer {
  // XXX Short term: providing a way to alter these without having to recompile
  // the compiler before recompiling the compiler.
  val MODULE_SUFFIX_STRING = sys.props.getOrElse("SCALA_MODULE_SUFFIX_STRING", "$")
  val NAME_JOIN_STRING   = sys.props.getOrElse("SCALA_NAME_JOIN_STRING", "$")
  val MODULE_INSTANCE_NAME = "MODULE$"
```

4.5 Creating symbols for conveyor params and extra fields

By now the extra sets for different contexts (classes, methods, constructors) have been determined, and captured locals have been renamed. Given those extra sets, conveyor params (in methods and constructors) and extra fields (in classes) will be needed to enable access to the captured locals.

Just before returning, `computeFreeVars` creates symbols for extra fields and conveyor params, tracking them in the `proxies` map:

```
atPhase(phase.next) {
  for ((owner, freeValues) <- free.toList) {
    debuglog("free var proxy: %s, %s".format(owner.fullLocationString, freeValues.toList.mkString(", ")))

      proxies(owner) =
        for (fv <- freeValues.toList) yield {
          val proxy = owner.newValue(owner.pos, fv.name)
            .setFlag(if (owner.isClass) PARAMACCESSOR | PRIVATE | LOCAL else PARAM)
            .setFlag(SYNTHETIC)
            .setInfo(fv.info);
          if (owner.isClass) owner.info.decls enter proxy;
          proxy
        }
  }
}
```

5 Term rewriting

Conveyor params have to be added, callsites updated to pass the values of captured locals or their proxies, constructors extended to initialize extra fields, and some accesses (for symbols in an extra set) rewritten to access a conveyor param or extra field instead (in the latter case, possibly over an outer-path).

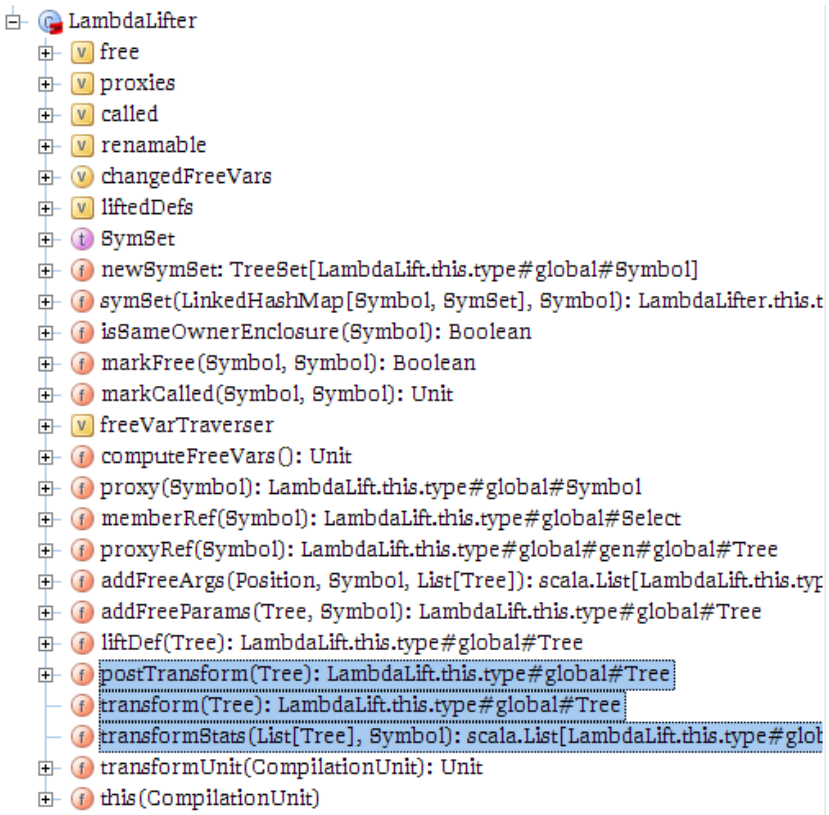


Figure 4: Sec. 5

Basically, it's `postTransform` (Sec. 5.1) that takes charge of "adding" `Trees` (as listed above) except that it also removes (by transforming into `EmptyTree`) liftable defs. But don't worry, before `LambdaLifter` is done (Figure 4), those liftable defs will happily live thereafter as lifted defs in their new home: `transformStats` is responsible for "pasting" the `Tree` of a liftable definition in its `enclClass`. The "cut" part is performed in `postTransform` (more specifically it's `liftDef(tree1)` that does it).

5.1 `postTransform`

5.1.1 Adding conveyor params and extra fields

Symbols for conveyor params and for extra fields were created back in `computeFreeVars` (Sec. 4.5) and now it's `addFreeParams()` chance to add `Trees` for them:

```
case ClassDef(_, _, _, _) =>
  val tree1 = addFreeParams(tree, sym)
  if (sym.isLocal) liftDef(tree1) else tree1
case DefDef(_, _, _, _, _, _) =>
  val tree1 = addFreeParams(tree, sym)
  if (sym.isLocal) liftDef(tree1) else tree1
```

5.1.2 Rewrite the definition site of a proxied `ValDef`

```
case ValDef(mods, name, tpt, rhs) =>
  if (sym.isCapturedVariable) {
    val tpt1 = TypeTree(sym.tpe) setPos tpt.pos
    /* Creating a constructor argument if one isn't present. */
    val constructorArg = rhs match {
      case EmptyTree =>
        sym.primaryConstructor.info.paramTypes match {
          case List(tp) => gen.mkZero(tp)
          case _       =>
            log("Couldn't determine how to properly construct " + sym)
            rhs
        }
      case arg => arg
    }
    treeCopy.ValDef(tree, mods, name, tpt1, typer.typedPos(rhs.pos) {
      Apply(Select(New(TypeTree(sym.tpe)), nme.CONSTRUCTOR), List(constructorArg))
    })
  } else tree
```

5.1.3 Adding conveyor arguments (for methods and constructors)

```
case Apply(fn, args) =>
  treeCopy.Apply(tree, fn, addFreeArgs(tree.pos, sym, args))
```

5.1.4 Accesses to captured locals and invocations to local methods

```
case Ident(name) =>
  val tree1 =
    if (sym != NoSymbol && sym.isTerm && !sym.isLabel)
      if (sym.isMethod)
        atPos(tree.pos)(memberRef(sym))
      else if (sym.isLocal && !isSameOwnerEnclosure(sym))
        atPos(tree.pos)(proxyRef(sym))
      else tree
    else tree
  if (sym.isCapturedVariable)
    atPos(tree.pos) {
      val tp = tree.tpe
      val elemTree = typer typed Select(tree1 setType sym.tpe, nme.elem)
      if (elemTree.tpe.typeSymbol != tp.typeSymbol) gen.mkAttributedCast(elemTree, tp) else elemTree
    }
  else tree1
```

5.1.5 Bring initializers for lazy vals and modules to `Block` start

```
case Block(stats, expr0) =>
  val (lzyVals, rest) = stats.partition {
             case stat@ValDef(_, _, _, _) if stat.symbol.isLazy => true
             case stat@ValDef(_, _, _, _) if stat.symbol.hasFlag(MODULEVAR) => true
             case _                                          => false
         }
  treeCopy.Block(tree, lzyVals:::rest, expr0)
```

Listing 3: Sec. 5.2

```
/** Transform statements and add lifted definitions to them. */
override def transformStats(stats: List[Tree], exprOwner: Symbol): List[Tree] = {

  def addLifted(stat: Tree): Tree = stat match {

    /*- this class becomes host to lifted definitions. */
    case ClassDef(mods, name, tparams, impl @ Template(parents, self, body)) =>
      /*- recursively */
      val lifted = liftedDefs(stat.symbol) reverseMap addLifted
      val result = treeCopy.ClassDef(
        stat, mods, name, tparams, treeCopy.Template(impl, parents, self, body ::: lifted))
      liftedDefs -= stat.symbol
      result

    /*- minor clean up of expression-valued method. */
    case DefDef(mods, name, tp, vp, tpt, Block(Nil, expr)) if !stat.symbol.isConstructor =>
      treeCopy.DefDef(stat, mods, name, tp, vp, tpt, expr)

    /*- do nothing. */
    case _ => stat
  }

  super.transformStats(stats, exprOwner) map addLifted
}
```

5.1.6 Misc

```
case Return(Block(stats, value)) =>
  Block(stats, treeCopy.Return(tree, value)) setType tree.tpe setPos tree.pos

case Return(expr) =>
  assert(sym == currentMethod, sym)
  tree

case Assign(Apply(TypeApply(sel @ Select(qual, _), _), List()), rhs) =>
  // eliminate casts introduced by selecting a captured variable field
  // on the lhs of an assignment.
  assert(sel.symbol == Object_asInstanceOf)
  treeCopy.Assign(tree, qual, rhs)

case _ =>
  tree
```

5.2 Pasting lifted trees: `transformStats` (including `addLifted`)

Listing 3 has been slightly edited to elide non-essential stuff. It shows that `transformStats` does a main thing and an auxiliary one:

1. it returns a `ClassDef` with an augmented template (augmented with the lifted `Trees` that were waiting in the `liftedDefs` map).

2. it cleans up the `rhs` of expression-valued methods.

Listing 4: Sec. 6

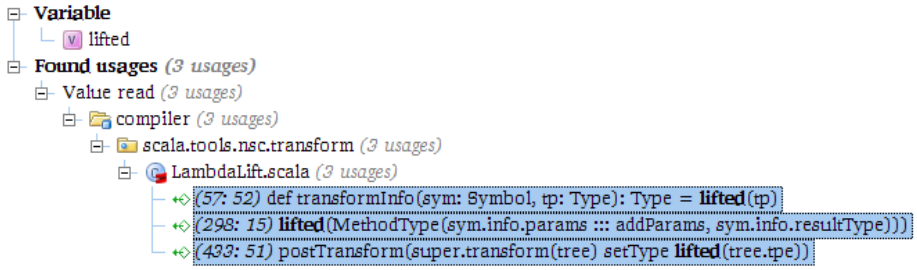
```
private val lifted = new TypeMap {

  def apply(tp: Type): Type = {
    tp match {

      case TypeRef(NoPrefix, sym, Nil) if sym.isClass && !sym.isPackageClass =>
        val newPrefix = apply(sym.owner.enclClass.thisType)
        typeRef(newPrefix, sym, Nil)

      case ClassInfoType(parents, decls, clazz) =>
        val parents1 = parents mapConserve this
        if (parents1 eq parents) tp
        else ClassInfoType(parents1, decls, clazz)

      case _ => mapOver(tp)
    }
  }

}
```

6 Type rewriting

The `lifted TypeMap` (shown in Listing 4) is used

- in `transformInfo()`,
- to `updateInfo` of a method or constructor's symbol (after lifting or after adding conveyor params, resp.)
- as the very last mutation of a tree node during term rewriting.

Details:



References

[1] Philippe Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings*. PhD thesis, EPFL Lausanne, Switzerland, 2006. `http://library.epfl.ch/theses/?nr=3509`.