The `refchecks` phase (Part 1)

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

September 20th, 2011

Abstract

TODO

```
      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 `refchecks` phase is an `InfoTransform` whose transformer (`RefCheckTransformer`) is tasked with most of the transform's bookkeeping (Figure 1). In terms of line count, AST checking takes the lion share, followed by term rewriting and a very focused type rewriting (Sec. 1.1).

1. Term rewriting comprises:
   (a) module lowering, Sec. 2
   (b) first part of lazy val lowering, Sec. 3
   (c) `Import` nodes are sent into oblivion. Just an example why refactorings can't be made much later than `typer`.
2. Checks comprise:

TODO

1.1 Type rewriting

Type rewriting is fairly simple:

```
// in reflect.internal.transform.RefChecks
  def transformInfo(sym: Symbol, tp: Type): Type =
    if (sym.isModule && !sym.isStatic) NullaryMethodType(tp)
    else tp


// in nsc.typechecker.RefChecks
  override def transformInfo(sym: Symbol, tp: Type): Type = {
    if (sym.isModule && !sym.isStatic) sym setFlag (lateMETHOD | STABLE)
    super.transformInfo(sym, tp)
  }
```

The "`!sym.isStatic`" guard is a consequence of the way that top-level (i.e., static) modules are lowered as compared to non-static ones (i.e, those having an outer instance). Internally, the class of non-static modules will be an inner class. If this is all new to you, here's a good tutorial[1].

At the risk of stepping ahead of ourselves (details in Sec. 2): the main difference between lowering top-level vs. non-static modules is that the module's symbol becomes associated to a static field (for a top-level module) or to an instance getter (for a non-static module). Thus the `NullaryMethodType` for the latter.

The following excerpt from the SLS helps in seeing why those lowerings have to be different ("Object definitions", SLS §5.4).

> *[An object definition] is roughly equivalent to the following definition of a lazy value:*
>
> `lazy val m = new sc with mt1 with . . . with mtn { this: m.type => stats }`

[1] `http://weblogs.java.net/blog/cayhorstmann/archive/2011/08/05/inner-classes-scala-and-java`

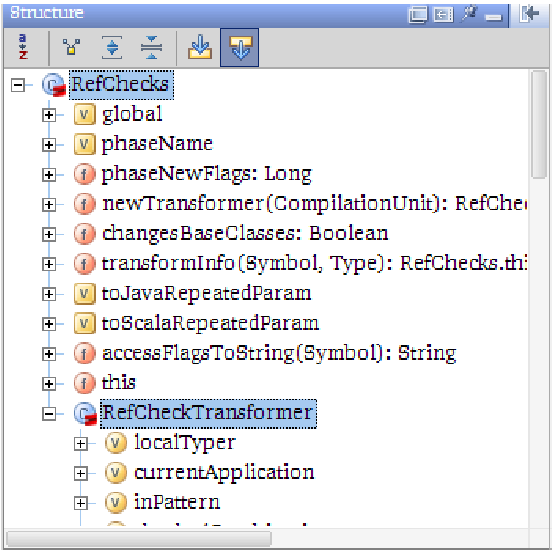

Figure 1: Sec. 1

> *. . . The expansion given above is not accurate for top-level objects. It cannot be because variable and method definitions cannot appear on the top-level outside of a package object (§9.3). Instead, top-level objects are translated to static fields.*

2 Lowering of modules

This lowering is self-contained, so we review it first:

```
private def eliminateModuleDefs(tree: Tree): List[Tree]
```

Before looking at the translation recipe, it's useful to keep in mind that the module's symbol is mentioned in general in other trees. Given that `eliminateModuleDefs()` doesn't return the original `ModuleDef`, among the trees it returns one should fulfill the role of "module getter" (and carry the `ModuleDef`'s `symbol`).

Common to all the code generation alternatives below (Sec. 2.1 to Sec. 2.3) a tree is built for the "invisible" *module class* of the module (Listing 1). This tree ("`cdef`" for short) carries the existing *module class* symbol:

```
val classSym = sym.moduleClass

val cdef    = ClassDef(mods | MODULE, name.toTypeName, Nil, impl)
              setSymbol classSym setType NoType
```

2.1 Top-level module

In this case, no other tree is returned. Where does the module symbol go, then? It's still reachable via `sourceModule` in `cdef`'s `ModuleClassSymbol` (Figure 2). Please notice this navigation path is not shown in Listing 1.

Listing 1: Sec. 2

```
The internal representation of classes and objects:

  class Foo is "the class" or sometimes "the plain class"
  object Foo is "the module"
  class Foo$ is "the module class" (invisible to the user: it implements object Foo)

  class Foo <
   ^  ^ (2)   \
   |  |  |     \
   | (5) |     (3)
   |  |  |       \
  (1) v  v        \
  object Foo (4)-> > class Foo$

  (1) companionClass
  (2) companionModule
  (3) linkedClassOfClass
  (4) moduleClass
  (5) companionSymbol
```

In order to see that no other phase further processes `cdef`, let's `-Xprint-icode -uniqid` the following:

```
object p
class C { def m = p }
```

Say the module symbol has id `#7576`. Throughout all phases, an access to `p` points to that symbol, and what is returned is an instance of its module class (whose id is `#7577`):

```
def m(): object p#7577 { /*- 'object' is displayed but in fact it's not a singleton type,
                            we're in ICode after all. */
  locals:
  startBlock: 1
  blocks: [1]

  1:
    4   LOAD_MODULE object p#7576
    4   RETURN(REF(object p#7577))

  }
```

Further AST processing *does* come into play, at the very last minute. `GenJVM` and in `GenMSIL` add a static field (and its initializer) to the class emitted for `cdef` (look for `addModuleInstanceField()`):

```
val MODULE_INSTANCE_FIELD: NameType = NameTransformer.MODULE_INSTANCE_NAME // "MODULE$"
```

Additionally, `LOAD_MODULE` is translated as `getstatic` (JVM) or `ldsfld` (MSIL) of that field.

```
TODO Scala.Net: if cdef is polymorphic, then a naive translation
    would result in several instances for the same module.
```

```
/** A class for module class symbols
 *  Note: Not all module classes are of this type; when unpickled, we get
 *  plain class symbols!
 */
class ModuleClassSymbol(owner: Symbol, pos: Position, name: TypeName)
extends ClassSymbol(owner, pos, name) {
  private var module: Symbol = null
  def this(module: TermSymbol) = {
    this(module.owner, module.pos, module.name.toTypeName)
    setFlag(module.getFlag(ModuleToClassFlags) | MODULE)
    sourceModule = module
  }
  override def sourceModule = module
  private var implicitMembersCacheValue: List[Symbol] = List()
  private var implicitMembersCacheKey: Type = NoType
  def implicitMembers: List[Symbol] = {
    val tp = info
    if (implicitMembersCacheKey ne tp) {
      implicitMembersCacheKey = tp
      implicitMembersCacheValue = tp.implicitMembers
    }
    implicitMembersCacheValue
  }
  override def sourceModule_=(module: Symbol) { this.module = module }
}
```

Figure 2: Sec. 2.1

2.2 A non-overridding module that has no outer instance

If there's an outer class, then it's possible to rephrase *(module def, module access)* in terms of *(field, getter)* in addition to the module class `cdef` that's already built. That's a good approximation, however the field is sometimes a variable (dubbed "mod-var") because the *(mod-var, getter)* may also end up being added to a method, for example:

```
class MethodModuleExample { def m() { object p } }
```

or to a block:

```
class BlockModuleExample {
  def m(b: Boolean) {
    if(b) {
      object p
    }
  }
}
```

Don't worry too much about this because anyway you won't learn the full story by looking at `refchecks` alone. The lowering of modules is completed in `mixin`. For now, we can see how `refchecks` adds mod-var + getter for all the shapes above ("module in class", "module in method", "module in statement block").

A piece of information that will be useful later: the mode-var is always annotated `@volatile` whether it's a field or local (the method in Listing 2 is invoked with `accessor` bound to `ModuleDef.symbol`). On related note, `@volatile` is cru-

Listing 2: Sec. 2.2

```
// TreeGen.scala

  def mkModuleVarDef(accessor: Symbol) = {
    val mval = (
      accessor.owner.newVariable(accessor.pos.focus, nme.moduleVarName(accessor.name))
      setInfo accessor.tpe.finalResultType
      setFlag (MODULEVAR)
    )

    mval.addAnnotation(AnnotationInfo(VolatileAttr.tpe, Nil, Nil))
    if (mval.owner.isClass) {
      mval setFlag (PRIVATE | LOCAL | SYNTHETIC)
      mval.owner.info.decls.enter(mval)
    }
    ValDef(mval)
  }
```

cial in connection with lazy vals (the condition of `mkDoubleCheckedLocking()` has to access a volatile, the `Int` holding a bitmap). But that's another story.

```
TODO Summarize where:
       - cdef,
       - the mod-var with type ModuleDef.symbol.info.finalResultType, and
       - the getter
     end up being added to.
     Also: what the body of the getter is, which phases further process it.

Cheat-sheet:
  - when the getter ends up in a trait,
    it just consists of New of cdef, the module class.
  - otherwise the getter consists of "{ lhs = rhs ; lhs }"
    where 'rhs' is as above and lhs is mod-var's symbol.
```

2.3 An overridding module that has outer instance

```
TODO

  /**
   * -Y "Private" settings
   */
  val overrideObjects = BooleanSetting ("-Yoverride-objects", "Allow member objects to be overridden.")
```

3 First lowering of lazy vals

`refchecks` does some rewriting for this construct, and `lazyvals` and `mixin` do the rest.

Some background: the symbol of a `lazy val` has something called "`lazyAccessor`" (a `MethodSymbol`, actually, while the surface-syntax construct has a `TermSymbol`):

```
    /** For a lazy value, its lazy accessor. NoSymbol for all others. */
```

```
def lazyAccessor: Symbol = NoSymbol
```

Basically, `makeLazyAccessor(tree, rhs): List[Tree]` replaces the original `ValDef` whose `symbol.isLazy` (whether owned by a class, object, method, or block) by one of the shapes below. Similarly to the lowering of objects, the original tree is not returned:

1. a `lazy val` of `Unit` type is replaced by a single "getter" whose symbol is the `lazyAccessor` and whose body is the original `rhs` of the surface-syntax construct.

   `TODO reachability of the symbol of the original ValDef afterwards`

2. otherwise, a `lazy val` owned by a trait is replaced by 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`.

   So far, a reshuffling.

3. otherwise, a pair *(ValDef, getter)* is returned 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`.

   Also in this case, pretty much a reshuffling.

Common to all three codegen alternatives above is the emitted `DefDef` with a `lazyAccessor` symbol (thus explaining the name of this tree builder, `makeLazyAccessor()`). From now on, other trees will interact only with that *getter* to side-effect the `ValDef` variable (if any).

4 Checks

`TODO`