After the meat and potatoes, before the pickles: `superaccessors` (Part 1)

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

September 16$^{\text{th}}$, 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

This phase does no type rewriting (and `atPhase` isn't mentioned at all). Instead, it performs a number of checks and term rewritings.

1. Checks on ASTs:

   (a) Targets of `super`-refs guaranteed concrete at runtime, Sec. 7.1.

```
TODO

  - checkCompanionNameClashes(sym)

  - checkPackedConforms(tree: Tree, pt: Type): Tree
```

2. Transformations:

   (a) An overriding constructor param doesn't require a dedicated field. Instead, its accessors can delegate to those in the superclass, provided that accesses to the overriding constructor param are also rewritten (Sec. 2).

   (b) `superaccessors` gets its name from two transformations that add synthetic methods (again, for delegation) in two cases:

   - due to super-refs appearing in a trait (Sec. 4.1)
   - due to super-refs targeting members of the super-class of an outer class (Sec. 4.2).

   Background for the above can be found in Sec. 3, with implementation aspects of these transformations covered in Sec. 5.

   (c) `needsProtectedAccessor` (Sec. 6).

```
TODO

  - mangles the names of class-members which are private up to
    an enclosing non-package class, in order to avoid overriding conflicts.
```

2 Use the superclass-based alias of a param-accessor if available

In the snippet below, `B.ap` is an alias for `A.ap`:

```
class A(val ap: String) { def am() = ap }
class B(override val ap: String) extends A(ap) { def bm() = ap }
```

- no field is added for `B.ap`, just an accessor (which both overrides and invokes the aliased getter in the superclass).

- an access in `B` to the alias is rewritten, and the resulting callsite targets an overriding-accessor (getter or setter).

```
[[syntax trees at end of superaccessors]]

class B extends A with ScalaObject {

  <paramaccessor> private[this] val ap: String = _; /*- this will eventually go away */

  override <stable> <accessor> <paramaccessor> def ap: String = /*- override and delegate */
    B.super.ap.asInstanceOf[String];

  def bm(): String = B.this.ap

  def this(ap: String): B = { B.super.this(ap); () };

}
```

```
TODO which phase elides
        <paramaccessor> private[this] val ap: String = _;
    in B
```

The transformation itself relies on `Symbol.alias`, which is set during typing:

```
def computeParamAliases(clazz: Symbol,
                        vparamss: List[List[ValDef]],
                        rhs: Tree): List[ErrorTree] = {
```

3 Background for super-ref rewritings

Background for all the rewritings in Sec. 4:

3.1 Actual supertype, Least proper supertype

Assume D defines some aspect of an instance x of type C (i.e. D is a base class of C) (SLS § 5.3.3).

1. If D is a trait:

   > *Then the actual supertype of D in x is the compound type consisting of all the base classes in $L(C)$ that succeed D. The actual supertype gives the context for resolving a super reference in a trait (§6.5). Note that the actual supertype depends on the type to which the trait is added in a mixin composition; it is not statically known at the time the trait is defined.*

2. If D is not a trait:

   > *Then its actual supertype is simply its least proper supertype (which is statically known).*

Summing up: *actual super-type* is a runtime concept, *least proper supertype* a compile-time one.

3.2 Runtime semantics

A reference `super.m` (occurring in a template) stands for `C.super.m` where C stands for the class or object definition that immediately encloses the template. The following determines runtime semantics (SLS § 6.5):

> *A reference* `C.super.m`
>
> 1. *refers statically to a method or type* `m` *in the least proper supertype of the innermost enclosing class or object definition named* `C` *which encloses the reference.*
> 2. *evaluates to the member* `m'` *in the actual supertype of that class or object which is equal to* `m` *or which overrides* `m`*.*

3.3 Useful facts about `Super` nodes

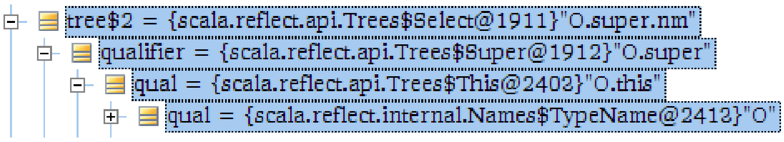
- "`qual` denotes the corresponding `this` reference"

  ```
  case class Super(qual: Tree, mix: TypeName) extends TermTree { . . .
  ```

- "The symbol of a `Super` node is the class *from which* the super reference is made. For instance in `C.super.x`, it would be `C`."

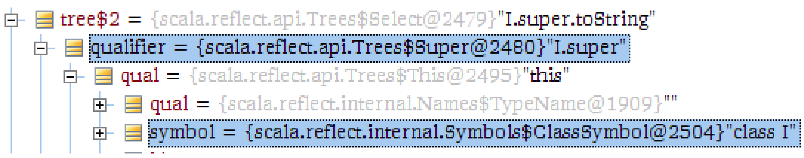
  ```
  override def symbol: Symbol = qual.symbol
  override def symbol_=(sym: Symbol) { qual.symbol = sym }
  ```

Expressions of the form `C.super.m` are represented internally as shown below (for the example in Sec. 4.2).



In more detail:

- the symbol of the `This` node above is a `ClassSymbol` (denoting the class whose super has `nm` as member)
- the symbol of the `Super` node above is also that symbol. Although (in the example) the innermost enclosing class where the super-expression occurs is `I`. This in contrast to non-qualified `super`-refs (say, `super.toString`) which, as shown below, consist of a `Super` node whose `This` qual has in turn an `tpnme.EMPTY` type name, and the symbol refers to the `I` class:



- `super.toString` is an example of a *static super reference* (the other kind of static super refs are those specifing a super trait, Sec. 4.3). Static super references are not rewritten. The applicability condition for rewriting is summarized in Sec. 4.3.

4 super-refs: some are rewritten, others aren't

4.1 Appearing in a trait

The runtime semantics described in Sec. 3.2 are pre-requisite reading.

A super-call that appears in a trait is rewritten into a call on a synthetic super-accessor `sa`, with `sa` added to the trait (at this point, `sa` is abstract, with flags "`private <method> <superaccessor>`"). Example:

```
trait MyTrait {
   def m() = super.toString
}
```

gets lowered to:

```
[[syntax trees at end of superaccessors]]

  abstract <defaultparam/trait> trait MyTrait extends lang.this.Object with scala.this.ScalaObject {

    private <method> <superaccessor> def super$toString(): lang.this.String; /*- added */

    <method> def m(): lang.this.String =
      MyTrait.this.super$toString() /*- rewritten */

    <method> def /*MyTrait*/$init$(): scala.this.Unit = { () };
  };
```

The name of the resulting super-accessor is prefixed as follows:

```
  val SUPER_PREFIX_STRING        = "super$"
  . . .
  def superName(name: Name): TermName     = newTermName(SUPER_PREFIX_STRING + name)
```

A concrete override can be inserted by the compiler only after the trait in question has been mixed-in into a `class`-class (whether `abstract` or concrete) or an `object` (Sec. 3.2).

After `superaccessors`, the next phase touching these synthetic methods is `erasure` (actually, `AddInterfaces`) where another prefix is prefixed:

```
[[syntax trees at end of erasure]]
package <empty> {

  abstract trait MyTrait extends java.lang.Object with ScalaObject {
    final <superaccessor> def MyTrait$$super$toString(): java.lang.String;
    def m(): java.lang.String
  };

  abstract trait MyTrait$class extends java.lang.Object with ScalaObject with MyTrait {
    def /*MyTrait$class*/$init$(): Unit = { () };
    def m(): java.lang.String = MyTrait$class.this.MyTrait$$super$toString()
  }
```

Listing 1: Sec. 4.1

```
[[syntax trees at end of mixin]]
package <empty> {

  abstract trait MyTrait extends java.lang.Object with ScalaObject {
    final <superaccessor> def MyTrait$$super$toString(): java.lang.String;
    def m(): java.lang.String
  };

  abstract class AC extends java.lang.Object with MyTrait with ScalaObject {

    /*- got a body */
    final <superaccessor> def MyTrait$$super$toString(): java.lang.String =
      AC.super.toString();

    def m(): java.lang.String = MyTrait$class.m(AC.this);
    def this(): AC = {
      AC.super.this();
      MyTrait$class./*MyTrait$class*/$init$(AC.this);
      ()
    }
  };

  abstract trait MyTrait$class extends {
    def m($this: MyTrait): java.lang.String = $this.MyTrait$$super$toString();
    def /*MyTrait$class*/$init$($this: MyTrait): Unit = {
      ()
    }
  }

}
```

```
}
```

To illustrate a super-accessor getting a body, we can compile the following, and then take a look at the ASTs after `mixin` (Listing 1).

```
abstract class AC extends MyTrait
```

4.2 Targeting a member of the superclass of an outer-class

A super-call that:

1. appears in an inner class `I`, and
2. targets a member of the superclass `S` of some outer-class `O`;

is rewritten into a call to a super-accessor '`sa`' added to the outer-class `O`. (for the time being, '`sa`' is abstract, with flags "`private <method> <superaccessor>`"). Example (with type params added for greater effect):

```
class N[T] { def nm(t: T) = t }

class O[U] extends N[U] {
```

```
  class I {

    def oim(u: U) = O.super.nm(u)

  }
}
```

After `superaccessors`, the AST looks as follows:

```
  class O[U >: Nothing <: Any] extends N[U] with ScalaObject {

    private <superaccessor> def super$nm(t: U): U; /*- synthesized super-accessor */

    def this(): O[U] = {
      O.super.this();
      ()
    };

    class I extends java.lang.Object with ScalaObject {
      def this(): O.this.I = {
        I.super.this();
        ()
      };
      def oim(u: U): U = O.this.super$nm(u) /*- rewritten callsite */
    }

  }
```

```
TODO Explain when the synthesized super-of-outer-accessor gets a body.
```

4.3 When super-refs aren't rewritten

Quoting from SLS § 6.5:

> *The super prefix may be followed by a trait qualifier [T], as in C.super[T].x. This is called a static super reference. In this case, the reference is to the type or method named 'x' in the parent trait of C whose simple name is T. That member must be uniquely defined. If it is a method, it must be concrete.*

Examples:

```
trait MyTrait {
    def m() = super.toString
}

class D extends MyTrait { def dm() = D.super[MyTrait].m() }

// alternatively
class E extends MyTrait { def em() = super[MyTrait].m() }
```

```
TODO Explain where the above is lowered.
```

The other kind of static super-refs are those where the superclass of interest is that of the innermost class where the super-ref occurs (e.g., `super.toString`).

```
private def transformSuperSelect(tree: Tree): Tree = tree match {
  case Select(sup @ Super(_, mix), name)  =>
    val sym = tree.symbol
    val clazz = sup.symbol

    if (sym.isDeferred) {
      val member = sym.overridingSymbol(clazz);
      if (mix != tpnme.EMPTY || member == NoSymbol ||
          !((member hasFlag ABSOVERRIDE) && member.isIncompleteIn(clazz)))
        unit.error(tree.pos, ""+sym+sym.locationString+" is accessed from super. It m
                             "unless it is overridden by a member declared `abstra
    }
    if (tree.isTerm && mix == tpnme.EMPTY &&
        (clazz.isTrait || clazz != currentOwner.enclClass || !validCurrentOwner)) {
      val supername = nme.superName(sym.name)
      var superAcc = clazz.info.decl(supername).suchThat(_.alias == sym)
      if (superAcc == NoSymbol) {
        if (settings.debug.value) log("add super acc " + sym + sym.locationString + "
        superAcc =
          clazz.newMethod(tree.pos, supername)
            .setFlag(SUPERACCESSOR | PRIVATE)
```

Figure 1: Sec. 5

Bringing together the discussion in Sec. 3.3, the applicability condition for the `superaccessors` rewriting of a super-ref is as follows:

```
case Select(sup @ Super(_, mix), name) =>

  val clazz = sup.symbol

  if (    tree.isTerm
      &&  mix == tpnme.EMPTY
      &&
          (     clazz.isTrait
            ||  clazz != currentOwner.enclClass
            ||  !validCurrentOwner
          )
     ) {

     /*- rewrite non-static super-ref, ie. replace Select node with another Select node
        that selects the name of the synthesized method. */
```

5 Mechanics of super-ref rewriting

It all starts in `SuperAccTransformer.transform`:

```
override def transform(tree: Tree): Tree = {

    case Select(Super(_, mix), name) =>
      if (sym.isValue && !sym.isMethod || sym.hasAccessorFlag) {
        unit.error(tree.pos, "super may be not be used on "+
                      (if (sym.hasAccessorFlag) sym.accessed else sym))
      }
      else if (isDisallowed(sym)) {
        unit.error(tree.pos, "super not allowed here: use this." + name.decode +
      }
      transformSuperSelect(tree)
```

and then the `super`-ref rewriting proper is done (or not) in `transformSuperSelect`, as the excerpt in Figure 1 sketches.

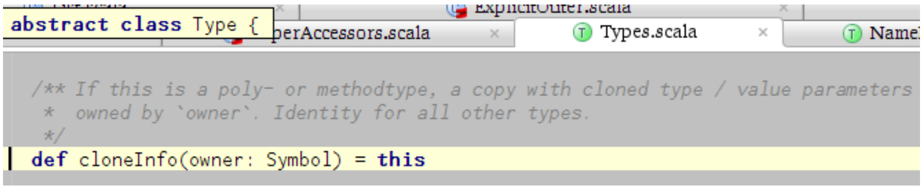
What was a "select-on-super" becomes a "select-on-this" which in turn is part of a callsite to the newly added synthetic (not shown):

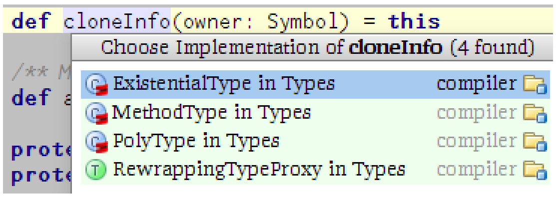
```
atPos(sup.pos) {
  Select(gen.mkAttributedThis(clazz), superAcc) setType tree.tpe;
}
```

The new method's symbol gets an `info` that has to be cloned from that of the selected member as seen from `thisType`.

```
superAcc.setInfo(superAccTpe.cloneInfo(superAcc))
```

via:



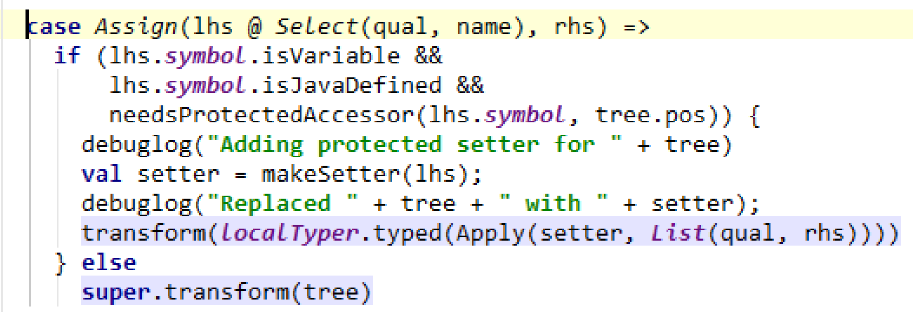


Naturally, the `tpe` of the resulting "select-on-this" has to be a `MethodType` (or a `NullaryMethodType`):

6 needsProtectedAccessor

There are two entry points:

1. `Assign`



2. Several shapes of accesses, wrapped by calls to:

```
mayNeedProtectedAccessor(sel: Select, args: List[Tree], goToSuper: Boolean)
```

Quoting from the source comment for `needsProtectedAccessor`:

> *Does `sym` need an accessor when accessed from `currentOwner`?*
>
> *A special case arises for classes with explicit self-types. If the self type is a Java class, and a protected accessor is needed, we issue an error. If the self type is a Scala class, we don't add an accessor. An accessor is not needed if the access boundary is larger than the enclosing package, since that translates to '`public`' on the host sys (as Java has no real package nesting).*
>
> *If the access happens inside a trait, access is more problematic since the implementation code is moved to an `$class` class which does not inherit anything. Since we can't (yet) add accessors for 'required' classes, this has to be signaled as error.*

TODO

7 Checks on ASTs (what and how)

7.1 Targets of super-refs guaranteed concrete at runtime

Quoting from SLS § 6.5:

> *[In a reference C`super.m`,] the statically referenced member `m` must be a type or a method.*
>
> 1. *If the statically referenced member `m` is a method, it must be concrete, or the innermost enclosing class or object definition named `C` must have a member `m'` which overrides `m` and which is labeled `abstract override`.*

And now the mechanics (good to know: Sec. 3.3)

```
private def transformSuperSelect(tree: Tree): Tree = tree match {
  case Select(sup @ Super(_, mix), name) =>
    val sym = tree.symbol
    val clazz = sup.symbol

    if (sym.isDeferred) {
      val member = sym.overridingSymbol(clazz);
      if (mix != tpnme.EMPTY || member == NoSymbol ||
          !((member hasFlag ABSOVERRIDE) && member.isIncompleteIn(clazz)))
        unit.error(tree.pos, ""+sym+sym.locationString+" is accessed from super. It may not be abstract "+
                             "unless it is overridden by a member declared `abstract' and `override'");
    }

    . . .
```