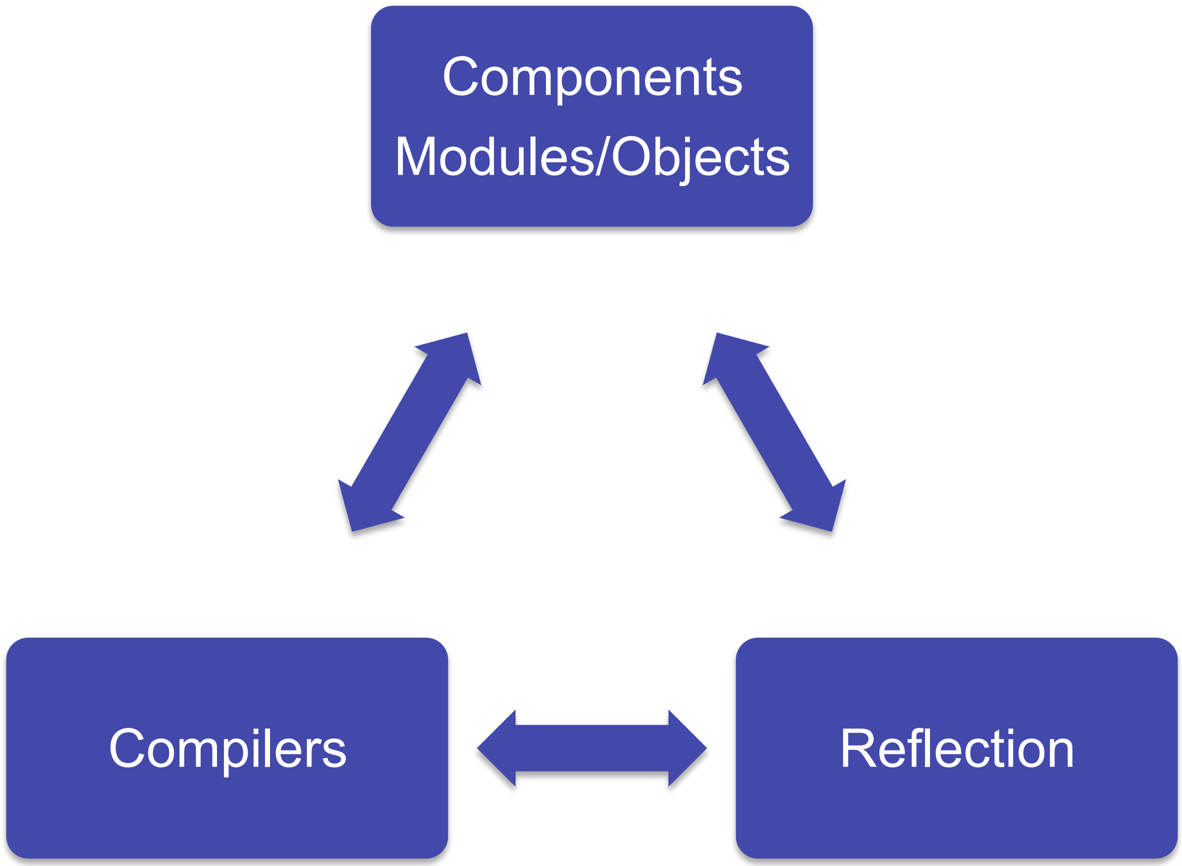
Objects and Modules – Two sides of the same coin?

Martin Odersky

Typesafe and EPFL

Milner Symposium,
16 April 2012


Components
Modules/Objects
Compilers
Reflection

Modules *vs* Objects

- Modules and Objects have the same purpose: containers to put things into.
- Differences in traditional OO languages:

Objects:

- dynamic values
- contain terms only
- (mutable)

Modules:

- static values
- contain terms and types
- immutable

In Scala:

- dynamic values
- contain terms and types
- encouraged to be immutable

Component Basics

- A *component* is a program part, to be combined with other parts in larger applications.
- Requirement: Components should be *reusable*.
- To be reusable in new contexts, a component needs *interfaces* describing its *provided* as well as its *required* services.
- Most current components are not very reusable.
- Most current languages can specify only provided services, not required services.

- Note: Component ≠ API!

No Statics!

- A component should refer to other components not by hard links, but only through its required interfaces.
- Another way of expressing this is:

  All references of a component to others should be via its members or parameters.

- In particular, there should be no global static data or methods that are directly accessed by other components.
- This principle is not new.
- But it is surprisingly difficult to achieve, in particular when we extend it to type references.

Functors

One established language abstraction for components are SML functors. Here,

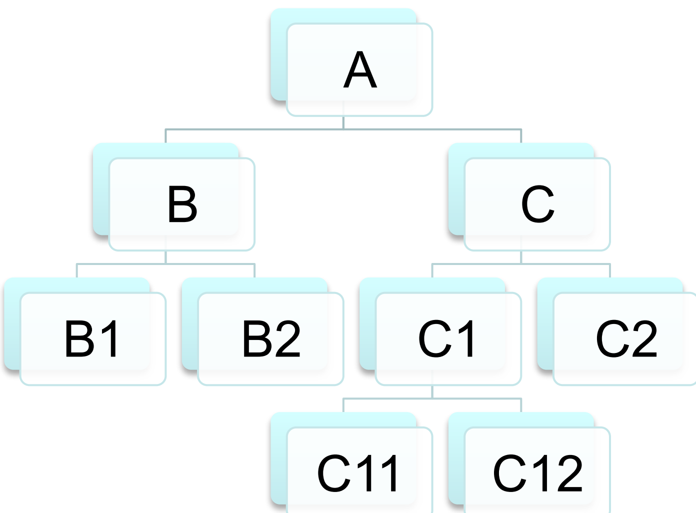
Component	≅	*Functor or Structure*
Interface	≅	*Signature*
Required Component	≅	*Functor Parameter*
Composition	≅	*Functor Application*

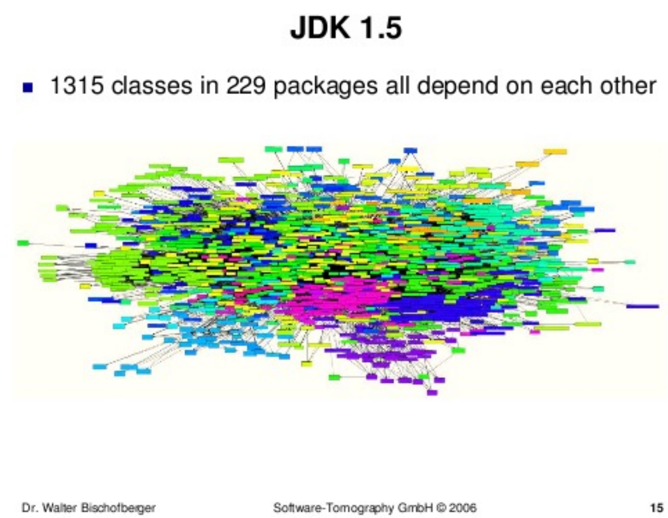
Sub-components are identified via sharing constraints or where clauses.

Restrictions (of the original version):

- No recursive references between components.
- No ad-hoc reuse with overriding
- Structures are not first class.

Functors work well for this: But the reality is often like this:





Component Abstraction

- Two principal forms of abstraction in programming languages:
  - parameterization (functional)
  - abstract members (object-oriented)
- ML uses parameterization for composition and abstract members for encapsulation.
- Scala uses abstract members for both composition and encapsulation.

  (In fact, Scala works with the functional/OO duality in that parameterization can be expressed by abstract members).

Mixin Composition

- Scala can express functors, but more often a different composition structure is used (e.g. scalac, Foursquare, lift):

Component	≅	*Trait*
Interface	≅	*Fully Abstract Trait*
Required Component	≅	*Abstract Member*
Composition	≅	*Mix in*

- Advantages:
  - Components instantiate to objects, which are first-class values.
  - Recursive references between components are supported.
  - Inheritance with overriding is supported.
  - Sub-components are identified by name; no explicit "wiring" is needed.

Abstract types

- Here is a type of "cells" using object-oriented abstraction.

```
trait AbsCell {
  type T
  val init: T
  private var value : T = init
  def get: T = value
  def set(x: T) = { value = x }
}
```

- The `AbsCell` trait has an abstract type member `T` and an abstract value member `init`.
- Instances of the trait can be created by implementing these abstract members with concrete definitions.

```
val cell = new AbsCell { type T = Int; val init = 1 }
cell.set(cell.get * 2)
```

- The type of `cell` is `AbsCell { type T = Int }`.

Path-Dependent Types

- It is also possible to access `AbsCell` without knowing the binding of its type member.
- For instance:

```
def reset(c : AbsCell): unit = c.set(c.init);
```

- Why does this work?
  - `c.init` has type `c.T`.
  - The method `c.set` has type `(c.T)Unit`.
  - So the formal parameter type and the argument type coincide.
- `c.T` is an instance of a path-dependent type.

Example: Symbol Tables

- Compilers need to model symbols and types.
- Each aspect depends on the other.
- Both aspects require substantial pieces of code.
- Encapsulation is essential (for instance, for hash-consing types).
- The first attempt of writing a Scala compiler in Scala defined two global objects, one for each aspect:

First Attempt: Global Data

```
object Symbols {
  trait Symbol {
    def tpe : Types.Type
  }
  ... // static data for symbols
}
```

```
object Types {
  trait Type {
    def sym : Symbols.Symbol
  }
  ... // static data for types
}
```

Problems:

- Symbols and Types contain hard references to each other.
- Hence, impossible to adapt one while keeping the other.
- Symbols and Types contain static data.
- Hence the compiler is not reentrant, multiple copies of it cannot run in the same OS process.
  (This is a problem for the Scala Eclipse plug-in, for instance).

Second Attempt: Nesting

- Static data can be avoided by nesting the Symbols and Types objects in a common enclosing trait:

```
trait SymbolTable {
  object Symbols {
    trait Symbol { def tpe : Types.Type; ... }
  }
  object Types {
    trait Type { def sym : Symbols.Symbol; ... }
  }
}
```

- This solves the re-entrancy problem.
- But it does not solve the component reuse problem
  - Symbols and Types still contain hard references to each other.
  - Worse, they can no longer be written and compiled separately.

Third attempt: Abstract members

Question: How can one express the required services of a component?

Answer: By abstracting over them!

Two forms of abstraction: parameterization and abstract members.

Only abstract members can express recursive dependencies, so we will use them.

```
trait Symbols {
  type Type
  trait Symbol { def tpe: Type }
}
```

```
trait Types {
  type Symbol
  trait Type { def sym: Symbol }
}
```

Symbols and Types are now traits that each abstract over the identity of the “other type”.

How can they be combined?

Modular Mixin Composition

```
trait SymbolTable extends Symbols with Types
```

- Instances of the `SymbolTable` trait contain all members of `Symbols` as well as all members of `Types`.
- Concrete definitions in either base trait override abstract definitions in the other.

Fourth Attempt: Mixins + Self-types (the cake pattern)

- The last solution modeled required types by abstract types.
- In practice this can become cumbersome, because we have to supply (possibly large) interfaces for the required operations on these types.
- A more concise approach makes use of self-types:

```
trait Symbols { this: Types with Symbols =>
  trait Symbol { def tpe: Type }
}
trait Types { this: Symbols with Types =>
  trait Type { def symbol }
}
```

- Here, every component has a self-type that contains all required components (in reality there are not 2 but ~20 slices to the cake).

Self Types

In a trait declaration

```
trait C { this: T => ... }
```

`T` is called a self-type of trait `C`.

If a self-type is given, it is taken as the type of `this` inside the trait.

Without an explicit type annotation, the self-type is taken to be the type of the trait itself.

Safety Requirement:

- The self-type of a trait must be a subtype of the self-types of all its base traites.
- When instantiating a trait in a new expression, it is checked that the self-type of the trait is a supertype of the type of the object being created.


Part 2: Compilers for Reflection
(its all about cakes)

Compilers and Reflection do largely the same thing ...

- Both deal with types, symbols, names, trees, annotations, ...
- Both answer similar questions, e.g:
  - what are the members of a type?
  - what are the types of the members of a basis type?
  - are two types compatible with each other?
  - is a method applicable to some arguments?
- In a rich type system, answering these questions requires some deep algorithms.

... But there are also differences

Compilers

read source and class-files
generate code
produce error messages
are typically single-threaded
types depends on phases

Reflection

relies on underlying VM info
invokes pre-generated code
throw exceptions
needs to be thread-safe
types are constant

Reflection in Scala 2.10

Previously: Needed to use Java reflection,
no runtime info available on Scala's types.

Now you can do:

```
import scala.reflect.mirror._
val clazz = symbolForName("scala.Function1") // get a Scala class
val obj = Vector(1, 2, 3)                    // create an object
val objType = typeOfInstance(obj)            // get a Scala type
val superType = objType.baseType(clazz)      // get a base type
val ms = superType.members                   // get its members
val app = superType member newTermName("apply")
                                             // get a specific member
val sig = app typeSignatureIn objType        // get its instantiated type
```

Reflection is Mirror Based

- A mirror: An object that can return reflective information about runtime values.

- In Scala, a mirror contains everything needed to describe reflective information as nested traites:
  Symbols, Types, Names, Annotations, Trees...

- What’s more, we enforce that the types of members of different mirrors are incompatible.

```
          reflect.api.Universe # Symbol
              ↗                    ↖
reflect.mirror.Symbol   ≠   remote.mirror.Symbol
```

Reflection Implementation

- Full reflection of a statically typed language covers a large ground.
- For Scala:
  - ~ 40 tree classes
  - ~ 5 symbol classes
  - ~ 10 Type classes
  - ~ 2 Name classes

  including all essential methods that decompose these classes, explore relationships between them, etc.
- This is roughly equivalent to a language spec
- ... and also to a compiler.

(Bare-Bones) Reflection in Java

java.lang.reflect

Interface Type

All Known Subinterfaces:
GenericArrayType, ParameterizedType, TypeVariable<D>, WildcardType

All Known Implementing Classes:
Class

`public interface Type`

Type is the common superinterface for all types in the Java programming language. These include raw types, parameterized types, array types, type variables and primitive types.

Since:
1.5

Overview Package Class Use Tree Deprecated Index Help *Java™ 2 Platform Standard Ed. 5.0*

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Why not add some meaningful operations?

Need to write essential parts of a compiler (hard).

Need to ensure that both compilers agree (almost impossible).

Want to know whether type A conforms to B?

Write your own Java compiler!

Towards Better Reflection

Can we unify the core parts of the compiler and reflection?

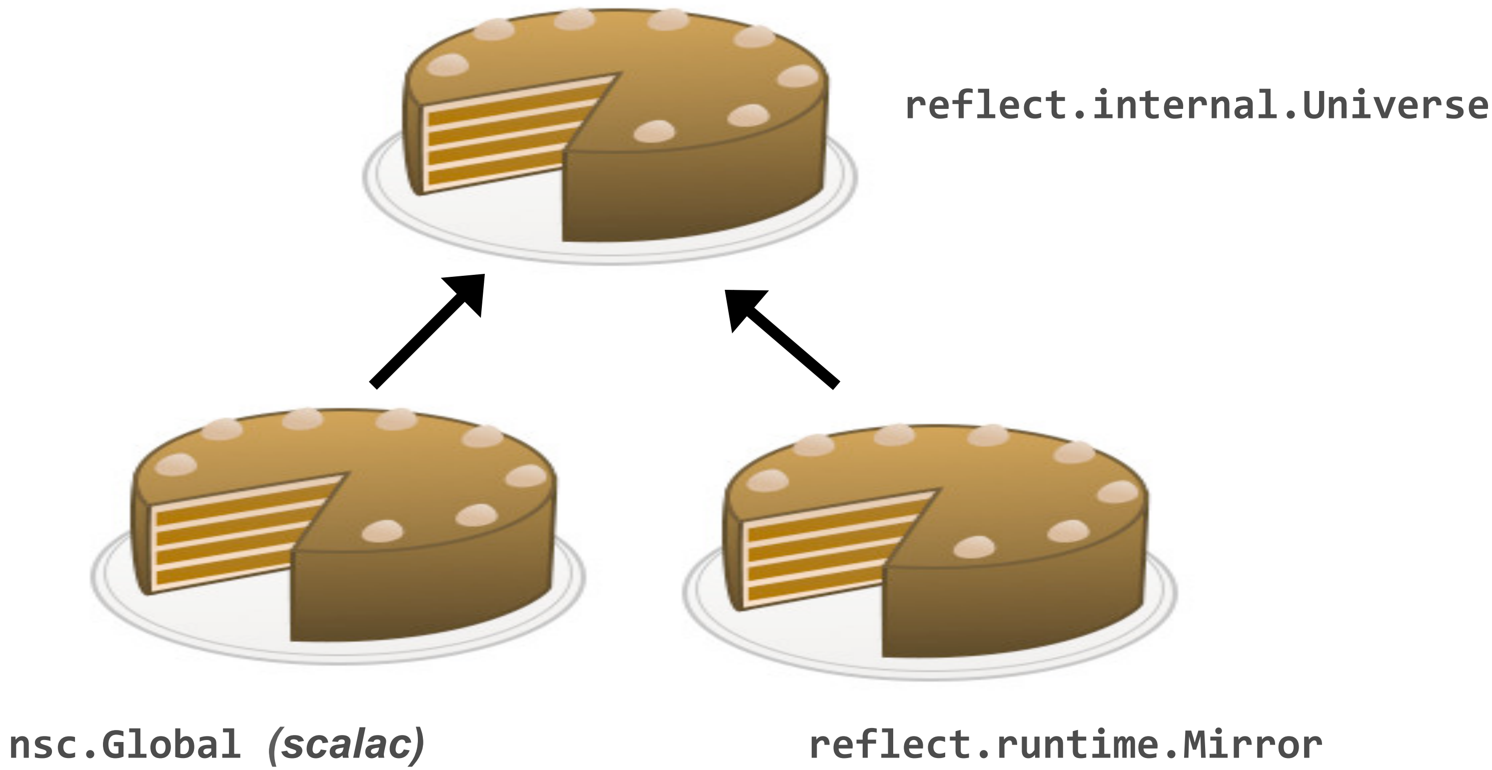
Compiler  Reflection

Different requirements: Error diagnostics, file access, classpath handling - but we are close!

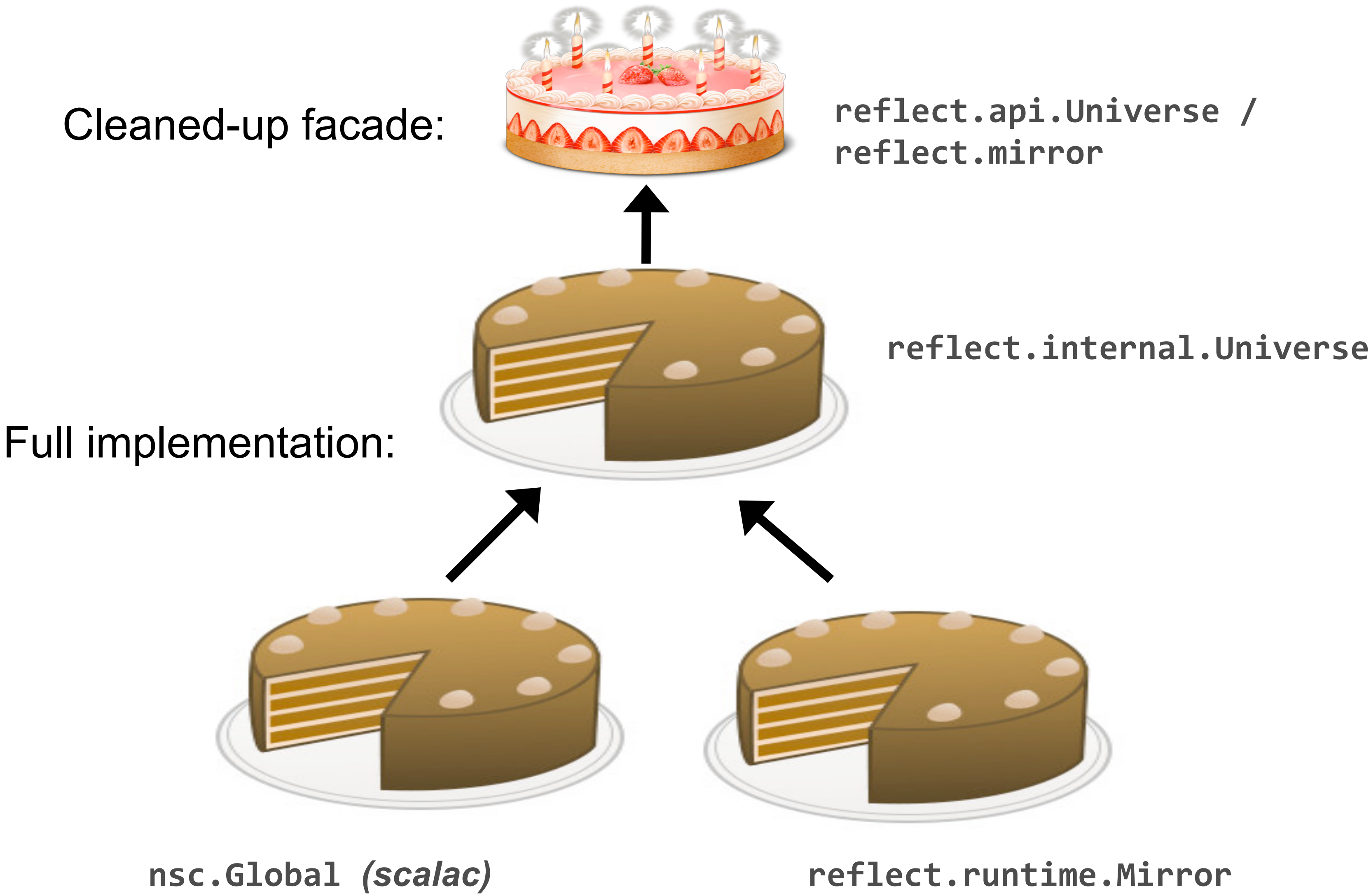
Compiler Architecture

Idea: Make compiler cake and reflection cake inherit from a common super-cake, which captures the common information.

Problem: This exposes too much detail!



Complete Reflection Architecture



How to Make a Facade

```scala
package scala.reflect.api

trait Types { self: Universe =>

  abstract class AbsType {
    def typeSymbol: Symbol
    def declaration(name: Name): Symbol
    def member(name: Name): Symbol
    def allMembers: Iterable[Symbol]
    def <:< (that: Type): Boolean
    def baseType(clazz: Symbol): Type
    ...
  }

  /** The type of Scala types, and also Scala type signatures.
   *  (No difference is internally made between the two).
   */
  type Type <: AbsType

  ...
}
```

The Facade

Interfaces are not enough!

```scala
package scala.reflect.internal

trait Types extends reflect.api.Types { self: SymbolTable =>

  class Type extends AbsType {

    def <:< (that: Type) = ...
  }
}
```

The Implementation

Summary Part 1

Scala is a pretty regular language when it comes to composition:

1. Everything can be nested:
   – classes, methods, objects, types
2. Everything can be abstract:
   – methods, values, types
3. The type of `this` can be declared freely, can thus express dependencies

This lets us express *cake hierarchies* as a new pattern for software design in the large.

Part 3: Reflection for Compilers

Macros

- What happens when a compiler makes use of reflection?
- It can call user-defined methods during the compilation (e.g. during type-checking)
- These methods can consume trees and types and produce a tree.
- This leads to a simple macro system.

Defining Macros

Here is a prototypical macro definition:

```
def m(x: T): R = macro impl.mi
```

The macro signature is a normal method signature.

Its body consists of macro, followed by a reference to the macro implementation. E.g.:

```
object impl {
  def mi(x: Expr[T]): Expr[R] = ...
}
```

`Expr[T]` represents an AST trees that describes an expression of type `T`

Expanding Macros

Say the compiler encounters during type checking an application of a macro method

```
m(expr)
```

It will expand that application by invoking the corresponding macro implementation `impl.mi` with two arguments:

- A context which contains info about the call-site of the macro
- the AST of `expr`.

The AST returned by the macro implementation replaces the macro application and is type-checked in turn.

A Simple Example

- The following code snippet declares a macro definition `assert` that references a macro implementation `Asserts.assertImpl`.

```
def assert(cond: Boolean, msg: Any) =
  macro Asserts.assertImpl
```

- A call `assert(x < 10, “limit exceeded”)` would then lead at compile time to an invocation

```
assertImpl(ctx)(<[ x < 10 ]>, <[ “limit exceeded” ]>)
```

Expressing Syntax Trees

- In reality, syntax trees written here

```
<[ x < 10 ]>
<[ “limit exceeded” ]>
```

would be expressed like this:

```
Apply(
  Select(Ident(newTermName(“x”)), newTermName(“$less”),
  List(Literal(Constant(10))))

Literal(Constant(“limit exceeded”))
```

Implementation of Assert

Here's a possible implementation of `assertImpl`:

```
import scala.reflect.makro.Context

object Asserts {
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
                (cond: c.Expr[Boolean],
                 msg: c.Expr[Any]) : c.Expr[Unit] =
    if (assertionsEnabled)
      <[ if (!cond) raise(msg) ]>
    else
      <[ () ]>
}
```

Generic Macros

Macros can also have type parameters. Example:

```
class Queryable[T] {
  def map[U](p: T => U): Queryable[U] = macro QImpl.map[T, U]
}

object QImpl {
  def map[T: c.TypeTag, U: c.TypeTag]
         (c: Context)
         (p: c.Expr[T => U]): c.Expr[Queryable[U]] = …
}
```

Generic Macro Expansion

Consider a value `q` of type `Queryable[String]` and a macro call

```
q.map[Int](s => s.length)
```

The call is expanded to:

```
QImpl.map(ctx)(<[ s => s.length ]>)
     (implicitly[TypeTag[String]], implicitly[TypeTag[Int]])
```

`implictly` realizes implicit values:

```
def implicitly[T](implicit x: T) = x
```

Contexts

- A macro context contains a mirror that anchors the trees, types, etc which are passed in and out of the macro.

```
trait Context {
  /** The mirror that represents the compile-time universe */
  val mirror: api.Universe
```

- It also defines some important data about the context of the macro call, in particular the receiver tree of the macro invocation and its type.

```
type PrefixType
val prefix: Expr[PrefixType]
```

Tagged Trees and Types

- Two other types in a context wrap compiler trees and types with reflect types:

```
case class Expr[T](tree: Tree) { def eval: T }
case class TypeTag[T](tpe: Type)
```

- An `Expr[T]` wraps a `reflect.mirror.Tree` of type `T`
- A `TypeTag[T]` wraps a `reflect.mirror.Type` that represents `T`.
- Implicit TypeTags can be synthesized by the compiler – this is Scala’s mechanism to get reified types.

Hygiene Problems

Consider again a fr

```
<[ i
```

`raise` gets bound at macro-expansion time.

Will either not be found or be resolved to something else.

To actually produce the AST for th one might try:

```
import c.mirror._
c.Expr(
  If(Select(cond, newTermName(“u ary_$bang”)),
    Apply(Ident(newTermName(“raise”)), List(msg)),
    Literal(Constant(()))))
```

This is ugly, but also wrong. Why?

The Reify Macro

- Reify is a key macro. It's definition as a member of context is:

```
def reify[T](expr: T): Expr[T] = macro ...
```

That is, reify

- takes a tree representing an expression of type `T` as argument,
- returns a tree representing an expression of type `Expr[T]`, which contains a tree that represents the original expression tree.

Reify is like *time-travel:* It builds the given tree one stage later

So reify expresses a core idea of LINQ:
Make ASTs available at runtime

Splicing

Reify and eval are inverses of each other.

```
reify:  T => Expr[T]
 eval: Expr[T] => T
```

```
val expr = reify(tree); expr.eval   →  tree
                  reify(expr.eval)   →  expr
```

So we have gained a *splicing* operation in the macro system.

Hygiene through Reify

Here's an implementation of the assert macro with reify:

```
import scala.reflect.makro.Context
object Asserts {
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)(cond: c.Expr[Boolean],
                             msg: c.Expr[Any]) : c.Expr[Unit] =
   if (assertionsEnabled)
    c.reify(if (!cond.eval) raise(msg.eval))
   else
    c.reify(())
}
```

raise is now type-checked at macro-expansion type, hence hygienic.

Types prevent "silly mistakes" that come from confusing staging times

Summary Part 3

A classical bootstrap operation

Start with a minimalistic macro system

- cumbersome to express syntax trees
- no hygiene

Express reification as a macro in that system

Use compile-time staging to regain

- source-level expression of syntax trees
- hygiene

The relationship of hygienic macros and staging has been known since Macro ML (Ganz et al, ICFP 01).

The ability to express staging through a reify macro seems to be new.