A tour of CLR Generics from a Scala vantage point

Miguel Garcia
http://lamp.epfl.ch/~magarcia

LAMP, EPFL

2011-04-26

Outline

Intro

- Goals and non-goals
- Problems and non-problems

The platform

- Statics are per-type-instantiation on CLR
- And that carries on to static methods
- Another issue: there are non-erased APIs out there

The design space

- Which syntax you like most?
- What can hide behind a C# method signature

Ongoing and Future Work

- Work not in progress (blocked by “Erasure for .NET”)
- Generics not really required, but anyway postponed
- Work in progress: Erasure for .NET (aka “generics in the backend”)

Outline

Intro
- Goals and non-goals
- Problems and non-problems

The platform
- Statics are per-type-instantiation on CLR
- And that carries on to static methods
- Another issue: there are non-erased APIs out there

The design space
- Which syntax you like most?
- What can hide behind a C# method signature

Ongoing and Future Work
- Work not in progress (blocked by “Erasure for .NET”)
- Generics not really required, but anyway postponed
- Work in progress: Erasure for .NET (aka “generics in the backend”)

Now that Scala.NET is about to get its own `erasure` phase,
I wanted to share with you some puzzles `:-)`

From what I see, "*erasure for .NET*" is quite different from "*emitting Java signatures*".

Goals and non-goals:

- Without compromising Scala semantics,
  we want Scala.NET to be "a good citizen" on .NET, i.e.:
  - not place undue barriers on using third-party assemblies
  - allow using our assemblies as components from other languages

The "Scala $\rightarrow$ .NET" direction is not problematic
(as long as we avoid some pitfalls described later).

Getting some non-problems out of the way:

- In terms of surface syntax, CLR type names appear to be "overloaded" by type-params-arity.
- However, CLR-wise, type names are unique (a generic type gets "`<backquote><arity>`" appended to its name)

Static members don't result in "multiple copies" (but see later):

- `RefCheck` eliminates modules by emitting a class.
- As far as CLR is concerned that class is monomorphic.
- BTW, a CLR interface would also do, because unlike in C# they can include methods with bodies, and fields.

Outline

Intro
- Goals and non-goals
- Problems and non-problems

The platform
- Statics are per-type-instantiation on CLR
- And that carries on to static methods
- Another issue: there are non-erased APIs out there

The design space
- Which syntax you like most?
- What can hide behind a C# method signature

Ongoing and Future Work
- Work not in progress (blocked by “Erasure for .NET”)
- Generics not really required, but anyway postponed
- Work in progress: Erasure for .NET (aka “generics in the backend”)

The C# 2.0 spec worded it concisely:

> *A static variable in a generic class declaration is shared amongst all instances of the same closed constructed type, but is not shared amongst instances of different closed constructed types . . . regardless of whether the type of the static variable involves any type parameters or not.*

For example, the following prints `0050`:

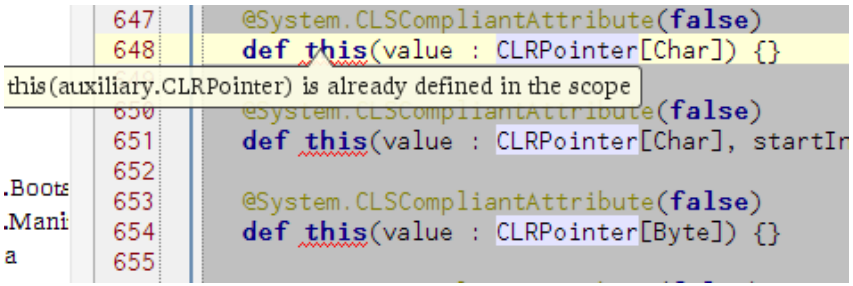
```
// C# code
class Gen<T> { public static int X = 0; }

class Test {
 static void Main() {
   Console.Write(Gen<int>.X); Console.Write(Gen<string>.X);
   Gen<int>.X = 5;
   Console.Write(Gen<int>.X); Console.Write(Gen<string>.X);
 }
}
```

And that carries on to static methods:

```
// F#

> type SomeType<'t> = static member M(a, b) = (a + b) ;;

type SomeType<'t> =
  class
    static member M : a:int * b:int -> int
  end

> SomeType.M(1, 1);;

  SomeType.M(1, 1);;
  ^^^^^^^^^^

stdin(6,1): warning FS1125: The instantiation of the generic type 'SomeType'
  is missing and can't be inferred from the arguments or return type of this member.
  Consider providing a type instantiation when accessing this type, e.g. 'SomeType<_>'.
val it : int = 2


> typedefof<SomeType<_>>.Equals(typedefof<SomeType<obj>>);; /*- testing for object identity */
val it : bool = true
```

The CLR way: class-level type-params are visible in static members.

Another issue. There are non-erased APIs out there:



That's over there, and this is over here:

```
/** The erasure transformer */
class ErasureTransformer(unit: CompilationUnit) extends Transformer {
  /** Emit an error if there is a double definition. This can happen if:
   *
   *  - A template defines two members with the same name and erased type.
   *  - A template defines and inherits two members `m` with different types,
   *    but their erased types are the same.
   *  - A template inherits two members `m` with different types,
   *    but their erased types are the same.
   */
  private def checkNoDoubleDefs(root: Symbol) {
```

Outline

Intro
- Goals and non-goals
- Problems and non-problems

The platform
- Statics are per-type-instantiation on CLR
- And that carries on to static methods
- Another issue: there are non-erased APIs out there

The design space
- Which syntax you like most?
- What can hide behind a C# method signature

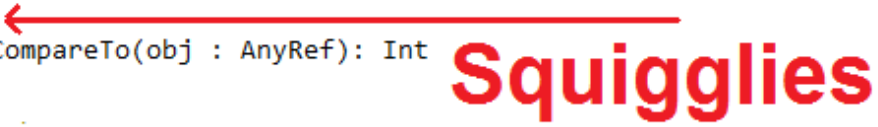
Ongoing and Future Work
- Work not in progress (blocked by “Erasure for .NET”)
- Generics not really required, but anyway postponed
- Work in progress: Erasure for .NET (aka “generics in the backend”)

Accessing static members

- don't want to adopt C#-isms that amount to invalid Scala syntax.
- Thus, no way: "`Gen[Int].x`"

More examples where new syntax would make code non-portable in the CLR → JVM direction:

```
object ArraySegment[T] {
  def ==(a : System.ArraySegment[T], b : System.ArraySegment[T]): Boolean = false

  def !=(a : System.ArraySegment[T], b : System.ArraySegment[T]): Boolean = false
}

@System.Runtime.InteropServices.ComVisibleAttribute(true)
trait IComparable {
  /* abstract */ def CompareTo(obj : AnyRef): Int
}

trait IComparable[T] {
  /* abstract */ def CompareTo(other : T): Int
}
```



- `jdk2ikvm` supports migration via source-to-source conversion.
- Even with a similar tool for the CLR → JVM direction, its focus should be API mapping, not language mapping!

What can hide behind a C# method signature (1 of 2):

```
public class Person {}
public class Employee : Person {}

public class CSharpSub : IComparable<Employee>, IComparable<Person> {
  int IComparable<Employee>.CompareTo(Employee that) { return 2; }
  int IComparable<Person>.CompareTo(Person that) { return 1; }
}
```

Transliterating into Scala: *error: trait IComparable is inherited twice*.

How to consume (if at all) types like the above?

```
trait IComparable[-T] { def CompareTo(that: T): Int }
class Person
class Employee extends Person

/*- error: trait IComparable is inherited twice */
class ScalaSub extends IComparable[Employee] with IComparable[Person]
   def CompareTo(that: Employee) = 2
   def CompareTo(that: Person) = 1
}
```

"Per type-instantiation overrides" don't add a special case to CLR method orverloading: *per convention*, the C# compiler disambiguates by (effectively) mangling method names (shown below). Callsites (which *look like* overloads in C#) follow "*C# 3.0 §7.4.3 Overload resolution*" (and the mangling convention). In terms of ILAsm, callsites make explicit the "instantiated (overridden) method signature", where no name-mangling is needed as each method signature is unique within its declaring interface (not shown).

```
.class public auto ansi beforefieldinit Sub
      extends [mscorlib]System.Object
      implements class [mscorlib]System.IComparable`1<class Employee>,
                 class [mscorlib]System.IComparable`1<class Person>
{
  .method private hidebysig newslot virtual final
        instance int32 'System.IComparable<Employee>.CompareTo'(class Employee that) cil manag
  {
    .override method instance int32
                 class [mscorlib]System.IComparable`1<class Employee>::CompareTo(!0)
    // . . .
  } // end of method Sub::'System.IComparable<Employee>.CompareTo'

  .method private hidebysig newslot virtual final
        instance int32 'System.IComparable<Person>.CompareTo'(class Person that) cil managed
  {
    .override method instance int32
                 class [mscorlib]System.IComparable`1<class Person>::CompareTo(!0)
    // . . .
  } // end of method Sub::'System.IComparable<Person>.CompareTo'
  // . . .
} // end of class Sub
```

Outline

Intro
- Goals and non-goals
- Problems and non-problems

The platform
- Statics are per-type-instantiation on CLR
- And that carries on to static methods
- Another issue: there are non-erased APIs out there

The design space
- Which syntax you like most?
- What can hide behind a C# method signature

Ongoing and Future Work
- Work not in progress (blocked by “Erasure for .NET”)
- Generics not really required, but anyway postponed
- Work in progress: Erasure for .NET (aka “generics in the backend”)

Tasks waiting for “Erasure for .NET” becoming available:

1. a standard library without IKVM dependencies
2. Visual Studio plugin
3. emitting binary assemblies (using CCI or IKVM.Reflection)

Part A, (Some) knowledge about compiler internals required:

4. REPL for Scala.NET
5. Compiler plugin loading using .NET reflection
6. Extending Scaladoc to emit API docs following .NET format.

Part B, Familiarity with IKVM is enough:

7. Porting `partest` and the test suite (mostly done by `jdk2ikvm`). BTW, a form of "behavioral equivalence testing" (cross-compiler vs. `scalacompiler.exe`) already runs[1].

Part C, For the community to tackle:

8. field testing `jdk2ikvm` on apps in the "Scala Corpus"[2]

[1] Sec. 6 in `lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/BootstrapDIY.pdf`

[2] `http://github.com/alacscala/scala-corpus`

Adapting erasure to .NET:

- Desirable (not mandatory) to erase minimally (simplifies using from other languages the assemblies emitted by Scala.NET).
- First, let’s see what most likely will change
  (next slide: what stays the same)

```
- For a typeref scala.Any or scala.AnyVal, System.Object.

- For a typeref P.C[Ts] where C refers to a class, |P|.C[|Ts|].
  (Where P is first rebound to the class that directly defines C.)

- For an empty type intersection, System.Object.

- For the class info type of System.Object,
  the same type without any parents.
```

- For a constant type, itself.
- For a type-bounds structure, the erasure of its upper bound.
- For every other singleton type, the erasure of its supertype.
- For a typeref scala.Array+[T] where T is an abstract type, AnyRef.
- For a typeref scala.Array+[T] where T is not an abstract type, scala.Array+[|T|].
- For a typeref scala.Unit, scala.runtime.BoxedUnit.
- For a typeref P.C[Ts] where C refers to an alias type, the erasure
- *For a typeref P.C[Ts] where C refers to an abstract type, the erasure of C’s* upper bound.
- For a non-empty type intersection (possibly with refinement), the erasure of its first parent.
- For a method type (Fs)scala.Unit, (|Fs|)scala#Unit.
- For any other method type (Fs)Y, (|Fs|)|T|.
- For a polymorphic type, the erasure of its result type.
- For a class info type of a value class, the same type without any parents.
- For any other class info type with parents Ps, the same type with parents |Ps|, but with duplicate references of Object removed.
- For all other types, the type itself (with any sub-components erased)