How TypeParser and GenMSIL handle properties, delegates, and events

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
http://lamp.epfl.ch/~magarcia

July 14th, 2010

Abstract

Notes for future reference about existing functionality in Scala.NET.

Contents

1 Properties

1.1 The Scala view of .NET properties

For each property with a CLS-conformant type, its `MethodInfo`s for public getter and setter are looked up in metadata (at most one of them can be null). Entering symbols for getter and setter involves basically (some steps omitted):

- a `Name` is created. For a getter, the property's name if no-args, `Name.apply` otherwise. For a setter, the property's name (plus `SETTER_SUFFIX`), or `Name.update`, resp.
- the owner `Symbol` is determined (`clazz` or `staticModule.moduleClass`) and `newMethod` invoked on it
- the thus obtained `methodSym` is entered into one of the `staticDefs` or `instanceDefs` `Scope`s.

The chosen naming scheme (first item above) enables field-syntax for no-args properties as expected, and assumes all other (parameterful) properties to be *default*, also as expected by C# developers. But assemblies emitted by VB may result in duplicate method defs under this naming scheme (Sec. 1.3 and Sec. 1.4).

For properties, method-local type-param definitions are not allowed (and thus, neither for C# indexers) because they are disallowed in the CLR. Using type-level type-params is ok[1]

1.2 Method mapping at `GenMSIL` time

The usual `Map[Symbol,MethodInfo] clrTypes.methods` that supports `GenMSIL.getMethod()` will do. No dedicated mapping is required. For example, say `Name.apply` appears in the AST, and `get_Item` is the getter as given in metadata. `getMethod` will return the `MethodInfo` for `get_Item` as for any other method:

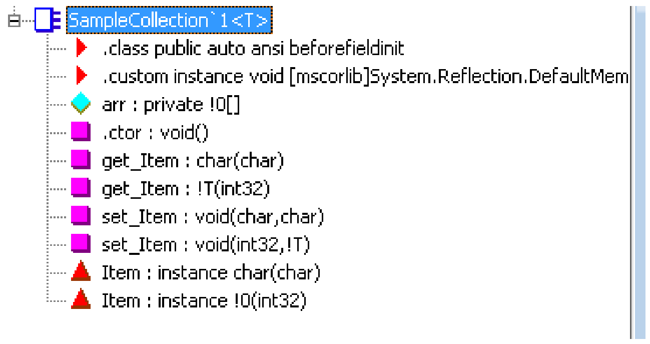
```
case CALL_METHOD(msym, style) =>
  if (msym.isClassConstructor) {
    . . .
  } else {
    . . .
    if (doEmit) {
      val methodInfo: MethodInfo = getMethod(msym)
      (style: @unchecked) match {
        case SuperCall(_) =>
          mcode.Emit(OpCodes.Call, methodInfo)
        case Dynamic =>
          mcode.Emit(if (dynToStatMapped(msym)) OpCodes.Call else OpCodes.Callvirt,
                     methodInfo)
        case Static(_) =>
          mcode.Emit(OpCodes.Call, methodInfo)
      }
    }
  }
```

1.3 Indexers, or CLR parameterful properties by another name

CLR *parameterful properties*[3, p. 256] are what C# calls *indexers* and Visual Basic *default properties.*

[1]Eric Lippert has made a series of blog posts `http://blogs.msdn.com/ericlippert/archive/tags/Covariance+and+Contravariance/default.aspx` detailing where C# does allow co- or contravariance.

When compiled with `csc`, the snippet below results in both a property `Item`, and one each of a public getter and setter (`get_Item` and `set_Item`). Additionally, a custom attribute (`DefaultMemberAttribute`) instructs `csc` to accept indexing syntax sugar when accessing setter and getter (in C#, `[]` are the visual cues for indexing, `()` in Scala).



```
SampleCollection`1<T>
    .class public auto ansi beforefieldinit
    .custom instance void [mscorlib]System.Reflection.DefaultMem
    arr : private !0[]
    .ctor : void()
    get_Item : char(char)
    get_Item : !T(int32)
    set_Item : void(char,char)
    set_Item : void(int32,!T)
    Item : instance char(char)
    Item : instance !0(int32)
```

An example from `http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx`

```
public class SampleCollection<T>
{
   private T[] arr = new T[100];

   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }

   public char this[char c]
   {
      get { return 'a'; }
      set { ; }
   }
}
```

`TypeParser` does not pay attention to the `DefaultMemberAttribute` (see Sec. 1.4). Instead, the name entered for a property depends on whether the property is parameterful or not. By definition, a C# indexer is always parameterful, and thus `Name.apply` and `Name.update` give names to getter and setter resp. In the example, where different properties have the same name but different overloads, duplicate names are not a problem. The case of different parameterless properties with the same name is not well-formed and thus shouldn't show up in metadata (if it does, the `typechecker` complains).

TODO should one check that there's at most one default property per inheritance chain? I mean, two different interfaces may each define a default property. What if a class supports both interfaces? (same goes for same-name events).

1.4 Default properties with non-default names

In C#, a type can define multiple indexers as long as parameter sets remain different (as for any method, no overloading on return type in allowed). Other compilers use the `IndexerName` attribute to distinguish between indexers with the same signature. C# won't allow this because its syntax elides precisely the indexer's name and thus complaints with `error C0111: Type 'SomeType' already defines a member called 'this' with the same parameter types`.

In case one wanted to parse property metadata using original property names instead of `Name.apply` and `Name.default`, details about `IndexerNameAttribute` appear at `http://msdn.microsoft.com/en-us/library/1k639k3z(VS.95).aspx`:

1.5 Emitting properties

TODO `GenMSIL` could emit property-metadata as shown below (for properties with at least one getter or setter), in addition to the pickled information. We want this, right? There's no `PropertyInfoBuilder` so that class should be added (BTW, there's no `EventInfoBuilder` either).

```
.property propertyType propertyName() {
 .get instance int32 DeclaringClass::whateverTheGettersNameIs(getter args if any)
                                   /*- the getter must declare `specialname' */
 .set instance void DeclaringClass::whateverTheSettersNameIs(setter args if any)
                                   /*- and the setter too */
}
```

In `ILPrinterVisitor`, this would be a great place to print property-metadata:

```
   // print each field
   val fields = `type`.fieldBuilders.iterator()
   while (fields.hasNext())
     print(fields.next().asInstanceOf[FieldBuilder])

/*- here */

   // print each constructor
   val constrs = `type`.constructorBuilders.iterator()
   while (constrs.hasNext())
     print(constrs.next().asInstanceOf[ConstructorBuilder])
```

As to `GenMSIL.BytecodeGenerator`, method `createClassMembers0` seems like a good place to populate `MSILType.propertyBuilders`:

```
def createClassMembers0(iclass: IClass) {
  val mtype = getType(iclass.symbol).asInstanceOf[TypeBuilder]
  for (ifield <- iclass.fields) {
    val sym = ifield.symbol
    if (settings.debug.value)
      log("Adding field: " + sym.fullName)

    var attributes = msilFieldFlags(sym)
    val fBuilder = mtype.DefineField(msilName(sym), msilType(sym.tpe), attributes)
    fields(sym) = fBuilder
    addAttributes(fBuilder, sym.annotations)
  }

  . . .
```

To pair setters and getters to build a `PropertyInfoBuilder`, check in `Symbol`:

```
   final def isGetter = isTerm && hasFlag(ACCESSOR) && !nme.isSetterName(name)
```

```
  final def isSetter = isTerm && hasFlag(ACCESSOR) && nme.isSetterName(name)
    //todo: make independent of name, as this can be forged.

  /** The getter of this value or setter definition in class `base', or NoSymbol if
   *  none exists.
   */
  final def getter(base: Symbol): Symbol = {
    val getterName = if (isSetter) nme.setterToGetter(name) else nme.getterName(name)
    base.info.decl(getterName) filter (_.hasFlag(ACCESSOR))
  }

  /** The setter of this value or getter definition, or NoSymbol if none exists */
  final def setter(base: Symbol): Symbol = setter(base, false)
```

2 CLR Delegates: What they are

CLR Delegates are a concoction of an abstraction that can be used only after a lot of syntactic sugar, and guess who has to provide all that syntax sugar? But I'm getting ahead of myself.

Some terminology: in the CLR literature the term *delegate* may refer to:

- a delegate type (a subclass of `System.Delegate`), or
- an instance of a delegate type. There's a public constructor,

  ```
  // delegate-type-specific constructor (for a delegate called 'Function' in the example)
  public Function(object @object, IntPtr method); /*- native int pointer, a function pointer! */
  ```

  but oftentimes the individual delegate instances thus created are aggregated for multicast purposes using runtime-managed factory methods that return new delegate instances. For example, adding a method reference to a multicast delegate instance.

A CLR delegate type equates to a subclass of `System.MulticastDelegate`, which in turn involves specifying an `Invoke` method (runtime-managed, i.e. no method body is given) whose signature is used by .NET compilers to typecheck usages of the delegate type. For example, in the surface syntax of C# a location having a delegate type can be assigned:

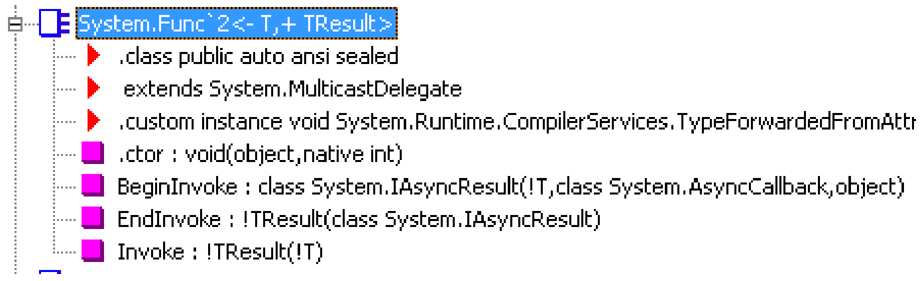
- a static method with compatible formal parameters and return type,
- an *`object.instanceMethod`* selector where the method has formals and return type compatible with the LHS delegate,
- an *anonymous function* (a.k.a. closure, e.g. `(double x) => x * 2.0`)

Other uses of delegate types (and instances) are also possible (e.g., creating out of two delegates d_1 and d_2 another delegate c, where the invocation list of c contains all elements in the invocation lists of d_1 and d_2. In C# surface syntax, that's expressed with `+` (although the invocation lists are concatenated and not set-union'ed):

```
DelegateType c = d1 + d2 // invokes the static System.Delegate.Combine with d1, d2 as args
```

The signature of `MulticastDelegate` is shown in Listing 1. Quite a few more methods are inherited from `System.Delegate`, as shown in Figure 1.

In summary, a delegate instance is to be used like a `scala.Function`*X* instance, yet its actual type is not generic, does not extend any .NET type with a strongly typed `Invoke()` method, and in fact due to nominal subtyping is not the same type as some other delegate taking the same formals to the same return type. It's a wrapper, at best. The counterparts in .NET to `scala.Function`*X* are `System.Func` and `System.Action`, which define type parameters and can be used as delegate types, because they subclass `System.MulticastDelegate` (in order to piggyback on its implementation).



```
System.Func`2<- T,+ TResult>
    .class public auto ansi sealed
     extends System.MulticastDelegate
    .custom instance void System.Runtime.CompilerServices.TypeForwardedFromAttr
    .ctor : void(object,native int)
    BeginInvoke : class System.IAsyncResult(!T,class System.AsyncCallback,object)
    EndInvoke : !TResult(class System.IAsyncResult)
    Invoke : !TResult(!T)
```

`System.Func<-T, +TResult>` can be rephrased as follows in C# syntax:

```
// Type: System.Func‘2
// Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
// Assembly location: C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\mscorlib.dll

using System.Runtime.CompilerServices;

namespace System
{
    [TypeForwardedFrom("System.Core, Version=3.5.0.0, Culture=Neutral, PublicKeyToken=b77a5c561934e089")]
    public delegate TResult Func<in T, out TResult>(T arg);
}
```

Regarding `System.Func‘2`, does its `Invoke` declaration override that of `MulticastDelegate`? Disassembling with the option

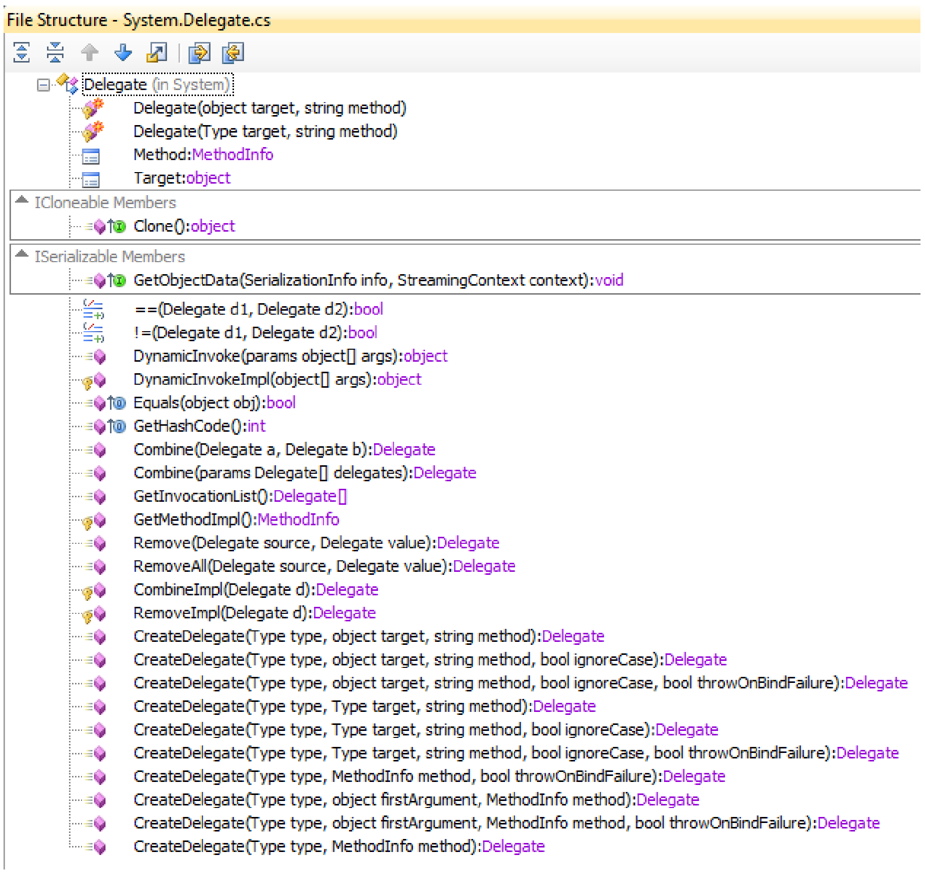
```
  /ITEM=<class>[::<method>[(<sig>)] Disassemble the specified item only
```

reveals no `.override` directive:

```
  .method public hidebysig newslot virtual
          instance !TResult Invoke(!T arg) runtime managed
  {
  } // end of method Func‘2::Invoke
```

2.1 Creating delegate instances without `ldftn` or `ldvirtftn`

This way to create delegates could be handy as it's an Scala-only way, no tricks required like expressing the IL instructions `ldftn` or `ldvirtftn`. ("no tricks required" other than a downcast, but those downcasts are needed whenever a `Delegate` static method is called, the only occassion they are not needed are



```
Delegate (in System)
    Delegate(object target, string method)
    Delegate(Type target, string method)
    Method:MethodInfo
    Target:object
ICloneable Members
    Clone():object
ISerializable Members
    GetObjectData(SerializationInfo info, StreamingContext context):void
    ==(Delegate d1, Delegate d2):bool
    !=(Delegate d1, Delegate d2):bool
    DynamicInvoke(params object[] args):object
    DynamicInvokeImpl(object[] args):object
    Equals(object obj):bool
    GetHashCode():int
    Combine(Delegate a, Delegate b):Delegate
    Combine(params Delegate[] delegates):Delegate
    GetInvocationList():Delegate[]
    GetMethodImpl():MethodInfo
    Remove(Delegate source, Delegate value):Delegate
    RemoveAll(Delegate source, Delegate value):Delegate
    CombineImpl(Delegate d):Delegate
    RemoveImpl(Delegate d):Delegate
    CreateDelegate(Type type, object target, string method):Delegate
    CreateDelegate(Type type, object target, string method, bool ignoreCase):Delegate
    CreateDelegate(Type type, object target, string method, bool ignoreCase, bool throwOnBindFailure):Delegate
    CreateDelegate(Type type, Type target, string method):Delegate
    CreateDelegate(Type type, Type target, string method, bool ignoreCase):Delegate
    CreateDelegate(Type type, Type target, string method, bool ignoreCase, bool throwOnBindFailure):Delegate
    CreateDelegate(Type type, MethodInfo method, bool throwOnBindFailure):Delegate
    CreateDelegate(Type type, object firstArgument, MethodInfo method):Delegate
    CreateDelegate(Type type, object firstArgument, MethodInfo method, bool throwOnBindFailure):Delegate
    CreateDelegate(Type type, MethodInfo method):Delegate
```

Figure 1: Signature of `System.Delegate`, Sec. 2

Listing 1: Signature of `System.MulticastDelegate`, Sec. 2

```
namespace System
{
   [ComVisible(true)]
   [Serializable]
   public abstract class MulticastDelegate : Delegate
   {
      protected MulticastDelegate(object target, string method);
      protected MulticastDelegate(Type target, string method);
      public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
      public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);

      [SecurityCritical]
      public override void GetObjectData(SerializationInfo info, StreamingContext context);

      [SecuritySafeCritical]
      public override sealed bool Equals(object obj);

      [SecuritySafeCritical]
      protected override sealed Delegate CombineImpl(Delegate follow);

      [SecuritySafeCritical]
      protected override sealed Delegate RemoveImpl(Delegate value);

      public override sealed Delegate[] GetInvocationList();
      public override sealed int GetHashCode();
      protected override MethodInfo GetMethodImpl();
   }
}
```

`.ctor` invocations on the subclass defining the custom delegate).

In summary, it's possible, just takes longer and doesn't add to type-safety. But here it goes: Listing 2 for the C# version and Listing 3 for the ILAsm translation.

```
public abstract class Delegate {
  // Construct a 'type' delegate wrapping the specified static method.
  public static Delegate CreateDelegate(Type type, MethodInfo method);
  public static Delegate CreateDelegate(Type type, MethodInfo method, Boolean throwOnBindFailure);

  // Construct a 'type' delegate wrapping the specified instance method.
  public static Delegate CreateDelegate(Type type, Object firstArgument, MethodInfo method);
  // firstArgument means 'this'
  public static Delegate CreateDelegate(Type type, Object firstArgument, MethodInfo method,
                                        Boolean throwOnBindFailure);
}
```

Quoting from [3]:

> *Important: The `System.Delegate` class has many more overloads of the `CreateDelegate` method that I do not show here. You should never call any of these other methods. As a matter of fact, Microsoft regrets even defining them in the first place. The reason is because these other methods identify the method to bind to by using a String instead of a `MethodInfo`. This means that an ambiguous bind*

Listing 2: Example using `CreateDelegate`, Sec. 2.1

```
using System;
using System.Reflection;
using System.Collections.Generic;

namespace CD
{
   delegate void ComputeDelegate(Employee emp, double percent);

   public class Employee
   {
      public double Salary;
      public Employee(double salary) { this.Salary = salary; }
      public void ApplyRaiseOf(double percent) { Salary *= (1 + percent); }
   }

   public class MainClass
   {
      static void Main3()
      {
         MethodInfo mi = typeof(Employee).GetMethod("ApplyRaiseOf",
                                     BindingFlags.Public | BindingFlags.Instance);
         ComputeDelegate applyRaise =
           (ComputeDelegate)Delegate.CreateDelegate(typeof(ComputeDelegate), mi);

         Employee e = new Employee(20);
         applyRaise(e, 0.10);
         Console.WriteLine(e.Salary);
      }
   }
}
```

is possible causing your application to behave unpredictably.

2.2 Creating delegates the usual way (with `ldftn` or `ldvirtftn`)

Quoting from [2, p. 293]:

> *When a delegate is being instantiated – its constructor takes a function pointer as the last argument – the `newobj` instruction must be immediately preceded by the `ldftn` or `ldvirtftn` instruction, which loads the function pointer. If anything appears between these two instructions, the code becomes unverifiable.*

Let's say we have (C# code):

```
namespace Dele
{
   delegate double Function(double x);

   public class Test
   {
      static double Square(double x) { return x * x; }

      static void Main2()
```

Listing 3: Creating delegate instances without using `ldftn` or `ldvirtftn`, Sec. 2.1

```
.method private hidebysig static void Main3() cil managed
{
  // Code size       91 (0x5b)
  .maxstack 3
  .locals init ([0] class [mscorlib]System.Reflection.MethodInfo mi,
           [1] class CD.ComputeDelegate applyRaise,
           [2] class CD.Employee e)
  IL_0000:  nop
  IL_0001:  ldtoken    CD.Employee /*- idiom: from TypeDef token to System.Type instance */
  IL_0006:  call       class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_000b:  ldstr      "ApplyRaiseOf"
  IL_0010:  ldc.i4.s   20 /*- i.e. the "binding flags" argument */
  IL_0012:  call       instance class [mscorlib]System.Reflection.MethodInfo [mscorlib]System.Type::GetMethod(string,
                                                                                  valuetype [mscorlib]System.Reflection.BindingFlags)
  IL_0017:  stloc.0
  IL_0018:  ldtoken    CD.ComputeDelegate
  IL_001d:  call       class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0022:  ldloc.0
  IL_0023:  call       class [mscorlib]System.Delegate [mscorlib]System.Delegate::CreateDelegate(class [mscorlib]System.Type,
                                                                                 class [mscorlib]System.Reflection.MethodInfo)
  IL_0028:  castclass  CD.ComputeDelegate /*- the cast is necessary, both in C# and here */
  IL_002d:  stloc.1
  IL_002e:  ldc.r8     20.
  IL_0037:  newobj     instance void CD.Employee::.ctor(float64)
  IL_003c:  stloc.2
  IL_003d:  ldloc.1
  IL_003e:  ldloc.2
  IL_003f:  ldc.r8     0.10000000000000001
  IL_0048:  callvirt   instance void CD.ComputeDelegate::Invoke(class CD.Employee,
                                                           float64) /*- the 'this' instance can be made explicit in
 the parameters list of the custom delegate type, in that case the object target has to be mentioned in each Invoke */
  IL_004d:  nop
  IL_004e:  ldloc.2
  IL_004f:  ldfld      float64 CD.Employee::Salary
  IL_0054:  call       void [mscorlib]System.Console::WriteLine(float64)
  IL_0059:  nop
  IL_005a:  ret
} // end of method MainClass::Main3
```

```
        {
            Function delStatic = Square; /*- a delegate to static method */

            Multiplier m = new Multiplier(2.0);
            Function delInstance = m.Multiply; /*- a delegate to (target, method) */
            Function delAnonymous = (double x) => x * 3.0; /*- a delegate to a closure */

            Function delAggr = delStatic; /*- aggregating all those delegates one at time */
            delAggr += delInstance;
            delAggr += delAnonymous;

            delAggr(6); /*- and invoking all in one go */
        }
    }
}
```

2.3 With delegates available, aggregating them is easy

Using two static factory methods from `System.Delegate` (see below), as exemplified in Listing 4.

```
.method public hidebysig static class System.Delegate
Combine(class System.Delegate a, class System.Delegate b)
{ ... } /*- this is what clrTypes.DELEGATE_COMBINE refers to */

.method public hidebysig static class System.Delegate
Remove(class System.Delegate source, class System.Delegate 'value')
{ ... } /*- this is what clrTypes.DELEGATE_REMOVE refers to */
```

Listing 4: Aggregating is easy

```
.method private hidebysig  static  void  Main2() cil  managed
{
  // Code size        125 (0x7d)
  .maxstack  3
  .locals  init  ([0]  class  Dele.Function  delStatic ,
                  [1]  class  Dele. Multiplier  m,
                  [2]  class  Dele.Function  delInstance ,
                  [3]  class  Dele.Function delAnonymous,
                  [4]  class  Dele.Function delAggr)
  IL_0000:  nop
  IL_0001:  ldnull
  IL_0002:  ldftn      float64  Dele.Test :: Square( float64 )
  IL_0008:  newobj     instance  void  Dele.Function ::. ctor ( object ,
                                                              native  int ) /*— first delegate  created  */
  IL_000d:  stloc .0
  IL_000e:  ldc . r8   2.
  IL_0017:  newobj     instance  void  Dele. Multiplier  ::. ctor ( float64 )
  IL_001c:  stloc .1
  IL_001d:  ldloc .1
  IL_001e:  ldftn      instance  float64  Dele. Multiplier :: Multiply ( float64 )
  IL_0024:  newobj     instance  void  Dele.Function ::. ctor ( object ,
                                                              native  int ) /*— 2nd delegate created (the same way as above, in  fact )  */
  IL_0029:  stloc .2
  IL_002a:  ldsfld     class  Dele.Function  Dele.Test :: 'CS$<>9__CachedAnonymousMethodDelegate1'
  IL_002f:  brtrue . s IL_0044
  IL_0031:  ldnull
  IL_0032:  ldftn      float64  Dele.Test :: '<Main2>b__0'(float64)
  IL_0038:  newobj     instance  void  Dele.Function ::. ctor ( object ,
                                                              native  int )
  IL_003d:  stsfld     class  Dele.Function  Dele.Test :: 'CS$<>9__CachedAnonymousMethodDelegate1'
  IL_0042:  br . s     IL_0044
  IL_0044:  ldsfld     class  Dele.Function  Dele.Test :: 'CS$<>9__CachedAnonymousMethodDelegate1'
  IL_0049:  stloc .3
  IL_004a:  ldloc .0
  IL_004b:  stloc . s  delAggr
  IL_004d:  ldloc . s  delAggr
  IL_004f:  ldloc .2   /*— aggregating is easy */
  IL_0050:  call       class  [mscorlib]System.Delegate [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
                                                                                      class  [mscorlib]System.Delegate)
  IL_0055:  castclass  Dele.Function
  IL_005a:  stloc . s  delAggr
  IL_005c:  ldloc . s  delAggr
  IL_005e:  ldloc .3
  IL_005f:  call       class  [mscorlib]System.Delegate [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
                                                                                      class  [mscorlib]System.Delegate)
  IL_0064:  castclass  Dele.Function
  IL_0069:  stloc . s  delAggr
  IL_006b:  ldloc . s  delAggr
  IL_006d:  ldc . r8   6.
  IL_0076:  callvirt   instance  float64  Dele.Function :: Invoke( float64 )
  IL_007b:  pop
  IL_007c:  ret
} // end of method Test::Main2
```

Quoting from [2]:

> *If one of the arguments of these methods is a null reference, the methods simply return the non-null argument. If both arguments are null references, the methods return a null reference. If the arguments are incompatible – that is, if the delegated methods have different signatures –* `Combine`*, which internally calls* `CombineImpl`*, throws an* `Argument` *exception, and* `Remove`*, which internally calls* `RemoveImpl`*, simply returns the aggregated delegate unchanged.*

3 What `TypeParser` does for delegates

3.1 Entering syntax sugar for delegates

Syntax sugar consists of operators (`+=`, `-=`, `+`, `-`, `==`, `!=`), delegate invocation, and delegate creation:

- the mapping from operators to invocations of static methods of `System.Delegate` is discussed below, alongside with semantics.

- an `apply` method is entered to facilitate delegate invocation,
- two implicits are entered (as static `defs` in the custom delegate type) to convert in both directions between a delegate instance and `scala.Function`*X* (Sec. 3.1.3).

3.1.1 Operators

Quoting from the C# language spec [1]:

> *Delegates are combined using the binary* `+` *(§7.7.4) and* `+=` *operators (§7.16.2). A delegate can be removed from a combination of delegates, using the binary* `-` *(§7.7.5) and* `-=` *operators (§7.16.2). Delegates can be compared for equality (§7.9.8). ...*
>
> *Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.*
>
> *... a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. ... when that delegate is removed, the last occurrence in the invocation list is the one actually removed. ... Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from a non-empty list) is not an error.*

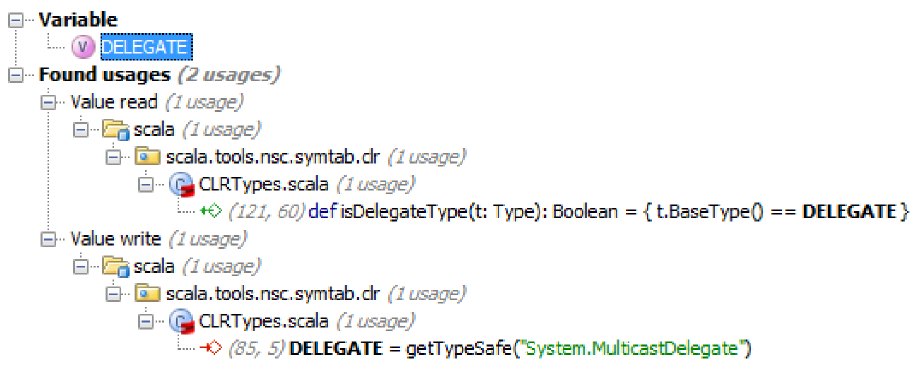
The mapping from operators on delegates to invocations of static methods of `System.Delegate` is as follows:

- The `Delegate.Equality` static operator should be invoked to evaluate d_1 `==` d_2 (there's also a `Delegate.Inequality` version). Javadoc-style remarks at `http://msdn.microsoft.com/en-us/library/system.delegate.op_equality.aspx`, but more complete is the description in §7.9.8 of the C# spec [1].
- for the mapping of `+=` and `-=`, consider §6.12.4 *Assignment Operators* of the SLS. C# developers will interpret the syntax in terms of §7.16.2 *Compound assignment* of the C# spec [1]. Fortunately, those semantics agree!

TODO If `+=` and `-=` are *not* generated for a custom delegate, then `+` (or `-`) followed by assignment will be emitted anyway. That's the semantics we're after, right? What's the purpose of entering method symbols for `+=` and `-=` ???

3.1.2 Implementation

For the purposes of `TypeParser`, a delegate type is any direct subclass of `System.MulticastDelegate` as shown below:



```
Variable
  DELEGATE
Found usages (2 usages)
  Value read (1 usage)
    scala (1 usage)
      scala.tools.nsc.symtab.clr (1 usage)
        CLRTypes.scala (1 usage)
          (121, 60) def isDelegateType(t: Type): Boolean = { t.BaseType() == DELEGATE }
  Value write (1 usage)
    scala (1 usage)
      scala.tools.nsc.symtab.clr (1 usage)
        CLRTypes.scala (1 usage)
          (85, 5) DELEGATE = getTypeSafe("System.MulticastDelegate")
```

PETypes representing delegates are handled as follows:

```
if (clrTypes.isDelegateType(typ)) {
  createDelegateView(typ)
  createDelegateChainers(typ)
}
```

Taking `System.Func‘2<-T, +TResult>` as example, by the end of `createDelegateView` and `createDelegateChainers` the following has been entered:

```
// instanceDefs
/*- <method> <java> as well as <param> <synthetic> elided for readability */
Scope{
  /*- was called .ctor in metadata */
  def this(x$1: scala.this.Any,
           x$2: System.this.IntPtr): System.this.Func‘2[T,TResult];

  /*- renamed (was Invoke), but usages of type params duplicate those in metadata */
  def apply(x$1: T): TResult;

  def EndInvoke(x$1: System.this.IAsyncResult): TResult;
  def BeginInvoke(x$1: T,
                  x$2: System.this.AsyncCallback,
                  x$3: scala.this.Any): System.this.IAsyncResult;

  /*- yes but how can mutators on this instance be invoked
      we didn’t write its definition and can’t call
      protected CombineImpl and RemoveImpl */
  final def +=(x$1: System.this.Func‘2[T,TResult]): scala.this.Unit;
  final def -=(x$1: System.this.Func‘2[T,TResult]): scala.this.Unit;

  final def +(x$1: System.this.Func‘2[T,TResult]): System.this.Func‘2[T,TResult];
  final def -(x$1: System.this.Func‘2[T,TResult]): System.this.Func‘2[T,TResult]
}
```

```
// staticDefs
Scope{
  implicit def view(x$1: System.this.Func‘2[T,TResult]): scala.this.Function1[T,TResult];
  implicit def view(x$1: scala.this.Function1[T,TResult]): System.this.Func‘2[T,TResult]
}
```

3.1.3 Interplay with `GenMSIL`

- first part (codegen), Listing 6
- second part (support definitions), Listing 7
- third part (`object definitions`), Listing 5

TODO Check if it works (specially `+=` and `-=`)

4 Events

An event declaration can appear as a member of a value type (but not an enum), interface, or object type. An instance of the declaring type plays the role of *Subject* for the named event in question, (un)registration of *Observer*s amounts to passing an argument (of the delegate type for the event) to the *adder* resp. *remover* methods. At the metadata level, there's no connection between the (usually private) field of delegate type (a `MulticastDelegate`) and the event declaration (thus allowing events to be declared in interfaces, where fields can't but properties may be declared).

The "semantics" of syntax sugar like:

```
subject.namedevent += delegateObserver
```

is `subject.add_namedevent(delegateObserver)` (whether that method actually adds the argument to the invocation list is at the mercy of the eventing type's author).

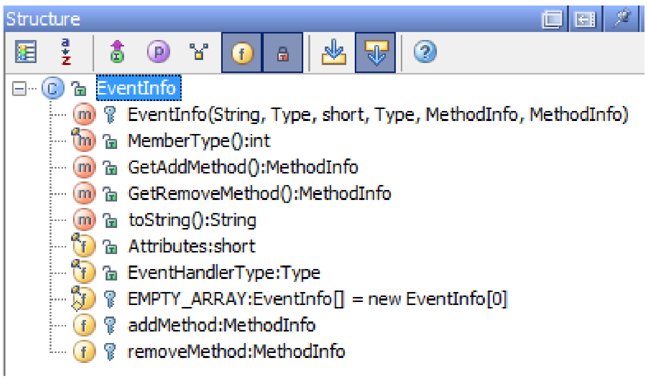
4.1 Consuming events

The C#-favored notation ("`subject.namedevent += delegateObserver`") does not fit the getter/setter idiom (because `subject.namedevent` does not follow the eventing protocol, in fact observers shouldn't access the invocation list). A possible workaround (not followed) consists in having two different setters, to invoke the `.addon` and `.removeon` methods behind the scenes.

As workaround, the long-form notation can be used, because the underlying methods are as accessible as the event itself (at least, for CLS-conformant assemblies): *CLS Rule 30: The accessibility of an event and of its accessors shall be identical.* These accessors don't include the `.fireon` method, which is to be invoked by the eventing instance only (usually by calling `Invoke(..)` on the `MulticastDelegate`).

- A more faithful alternative consists in defining a nested class, to be returned by `subject.namedevent`, which implements the adder and remover.
- Yet another alternative consists in making "`+=`" and "`-=`" trail the `namedevent`. Sample usage: "`subject namedevent+= delegateObserver`" TODO what about operator precedence?

`Type.GetEvents()` returns `EventInfo[]`, where



```
EventInfo
    EventInfo(String, Type, short, Type, MethodInfo, MethodInfo)
    MemberType():int
    GetAddMethod():MethodInfo
    GetRemoveMethod():MethodInfo
    toString():String
    Attributes:short
    EventHandlerType:Type
    EMPTY_ARRAY:EventInfo[] = new EventInfo[0]
    addMethod:MethodInfo
    removeMethod:MethodInfo
```

4.2 Emitting events

TODO So far, no annotation exists to trigger the compiler to emit event-metadata. There is no workaround for this.

References

[1] Microsoft Corporation. C# version 3.0 language specification, 2007. `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

[2] Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress, Berkely, CA, USA, 2006.

[3] Jeffrey Richter. *CLR Via C#, Third Edition*. Microsoft Press, Redmond, 2009.

Listing 5: Delegates-related code in `object definitions`

```
  //
  // .NET backend
  //

  lazy val ComparatorClass = getClass("scala.runtime.Comparator")
  // System.ValueType
  lazy val ValueTypeClass: Symbol = getClass(sn.ValueType)
  // System.MulticastDelegate
  lazy val DelegateClass: Symbol = getClass(sn.Delegate)
  var Delegate_scalaCallers: List[Symbol] = List()
  // Symbol -> (Symbol, Type): scalaCaller -> (scalaMethodSym, DelegateType)
  // var Delegate_scalaCallerInfos: HashMap[Symbol, (Symbol, Type)] = _
  lazy val Delegate_scalaCallerTargets: HashMap[Symbol, Symbol] = new HashMap()

  def isCorrespondingDelegate(delegateType: Type, functionType: Type): Boolean = {
    isSubType(delegateType, DelegateClass.tpe) &&
    (delegateType.member(nme.apply).tpe match {
      case MethodType(delegateParams, delegateReturn) =>
        isFunctionType(functionType) &&
        (functionType.normalize match {
          case TypeRef(_, _, args) =>
            (delegateParams.map(pt => {
                    if (pt == AnyClass.tpe) definitions.ObjectClass.tpe else pt})
             ::: List(delegateReturn)) == args
          case _ => false
        })
      case _ => false
    })
  }


var nbScalaCallers: Int = 0
def newScalaCaller(delegateType: Type): Symbol = {
  assert(forMSIL, "scalaCallers can only be created if target is .NET")
  // object: reference to object on which to call (scala-)method
  val paramTypes: List[Type] = List(ObjectClass.tpe)
  val name: String = "$scalaCaller$$" + nbScalaCallers
  // tparam => resultType, which is the resultType of PolyType, i.e. the result type after applying the
  // type parameter =-> a MethodType in this case
  // TODO: set type bounds manually (-> MulticastDelegate), see newTypeParam
  val newCaller = newMethod(DelegateClass, name, paramTypes, delegateType) setFlag (FINAL | STATIC)
  // val newCaller = newPolyMethod(DelegateClass, name,
  // tparam => MethodType(paramTypes, tparam.typeConstructor)) setFlag (FINAL | STATIC)
  Delegate_scalaCallers = Delegate_scalaCallers ::: List(newCaller)
  nbScalaCallers += 1
  newCaller
}

// def addScalaCallerInfo(scalaCaller: Symbol, methSym: Symbol, delType: Type) {
// assert(Delegate_scalaCallers contains scalaCaller)
// Delegate_scalaCallerInfos += (scalaCaller -> (methSym, delType))
// }

def addScalaCallerInfo(scalaCaller: Symbol, methSym: Symbol) {
  assert(Delegate_scalaCallers contains scalaCaller)
  Delegate_scalaCallerTargets += (scalaCaller -> methSym)
}
```

Listing 6: Delegates-related code in `GenMSIL` (code gen part)

```
case CALL_METHOD(msym, style) =>
  if (msym.isClassConstructor) {
    . . .
  } else {

    . . .

          // method: implicit view(FunctionX[PType0, PType1, ...,PTypeN, ResType]):DelegateType
          val (isDelegateView, paramType, resType) = atPhase(currentRun.typerPhase) {
            msym.tpe match {
              case MethodType(params, resultType)
              if (params.length == 1 && msym.name == nme.view_) =>
                val paramType = params(0).tpe
                val isDel = definitions.isCorrespondingDelegate(resultType, paramType)
                (isDel, paramType, resultType)
              case _ => (false, null, null)
            }
          }
          if (doEmit && isDelegateView) {
            doEmit = false
            createDelegateCaller(paramType, resType)
          }

          if (doEmit &&
              (msym.name == nme.PLUS || msym.name == nme.MINUS)
              && clrTypes.isDelegateType(msilType(msym.owner.tpe)))
            {
            doEmit = false
            val methodInfo: MethodInfo = getMethod(msym)
            // call it as a static method, even if the compiler (symbol) thinks it's virtual
            mcode.Emit(OpCodes.Call, methodInfo)
            mcode.Emit(OpCodes.Castclass, msilType(msym.owner.tpe))
          }

          if (doEmit && definitions.Delegate_scalaCallers.contains(msym)) {
            doEmit = false
            val methodSym: Symbol = definitions.Delegate_scalaCallerTargets(msym)
            val delegateType: Type = msym.tpe match {
              case MethodType(_, retType) => retType
              case _ => abort("not a method type: " + msym.tpe)
            }
            val method: MethodInfo = getMethod(methodSym)
            val delegCtor = msilType(delegateType).GetConstructor(Array(MOBJECT, INT_PTR))
            if (methodSym.isStatic) {
              mcode.Emit(OpCodes.Ldftn, method)
            } else {
              mcode.Emit(OpCodes.Dup)
              mcode.Emit(OpCodes.Ldvirtftn, method)
            }
            mcode.Emit(OpCodes.Newobj, delegCtor)
          }

          . . .
  }
```

Listing 7: Delegates-related code in GenMSIL (supporting methods)

```
// delegate callers

var delegateCallers: TypeBuilder = _
var nbDelegateCallers: Int = 0

private def initDelegateCallers() = {
  delegateCallers = mmodule.DefineType("$DelegateCallers", TypeAttributes.Public |
                                    TypeAttributes.Sealed)
}

private def createDelegateCaller(functionType: Type, delegateType: Type) = {
  if (delegateCallers == null)
    initDelegateCallers()
  // create a field an store the function-object
  val mFunctionType: MsilType = msilType(functionType)
  val anonfunField: FieldBuilder = delegateCallers.DefineField(
    "$anonfunField$$" + nbDelegateCallers, mFunctionType,
    (FieldAttributes.InitOnly | FieldAttributes.Public | FieldAttributes.Static).toShort)
  mcode.Emit(OpCodes.Stsfld, anonfunField)


  // create the static caller method and the delegate object
  val (params, returnType) = delegateType.member(nme.apply).tpe match {
    case MethodType(delParams, delReturn) => (delParams, delReturn)
    case _ => abort("not a delegate type: " + delegateType)
  }
  val caller: MethodBuilder = delegateCallers.DefineMethod(
    "$delegateCaller$$" + nbDelegateCallers,
    (MethodAttributes.Final | MethodAttributes.Public | MethodAttributes.Static).toShort,
    msilType(returnType), (params map (_.tpe)).map(msilType).toArray)
  for (i <- 0 until params.length)
    caller.DefineParameter(i, ParameterAttributes.None, "arg" + i) // FIXME: use name of parameter symbol
  val delegCtor = msilType(delegateType).GetConstructor(Array(MOBJECT, INT_PTR))
  mcode.Emit(OpCodes.Ldnull)
  mcode.Emit(OpCodes.Ldftn, caller)
  mcode.Emit(OpCodes.Newobj, delegCtor)


  // create the static caller method body
  val functionApply: MethodInfo = getMethod(functionType.member(nme.apply))
  val dcode: ILGenerator = caller.GetILGenerator()
  dcode.Emit(OpCodes.Ldsfld, anonfunField)
  for (i <- 0 until params.length) {
    loadArg(dcode)(i)
    emitBox(dcode, toTypeKind(params(i).tpe))
  }
  dcode.Emit(OpCodes.Callvirt, functionApply)
  emitUnbox(dcode, toTypeKind(returnType))
  dcode.Emit(OpCodes.Ret)

  nbDelegateCallers = nbDelegateCallers + 1

} //def createDelegateCaller
```