Yiihaw .NET aspect weaver usage guide

Rasmus Johansen Stephan Spangenberg Peter Sestoft
johansen@itu.dk spangenberg@itu.dk sestoft@itu.dk
IT University of Copenhagen, Denmark

2007-09-30

Abstract Yiihaw is a static aspect weaver for the Common Language Infrastructure (CLI), also known as Microsoft .NET version 2.0 and later. The Yiihaw weaver works by transforming CLI assemblies in the form of `.dll` and `.exe` files and performs extensive checks at weave-time to ensure the correctness of the resulting woven assemblies.

1 Getting Yiihaw

The Yiihaw weaver is available from http://yiihaw.tigris.org/ under the GNU public license. Yiihaw works with Microsoft .NET 2.0 or later. To use Yiihaw you need the following libraries, all included with the distribution:

- `Mono.Cecil.dll`
- `YIIHAW.API.dll`
- `YIIHAW.Exceptions.dll`
- `yiihaw.exe`
- `YIIHAW.Output.dll`
- `YIIHAW.Pointcut.dll`
- `YIIHAW.Weaver.dll`

The design and implementation of the Yiihaw weaver is described in Johansen and Spangenberg: *Yiihaw. An aspect weaver for .NET*, IT University of Copenhagen, Denmark, March 2007, available at http://www.itu.dk/people/sestoft/itu/JohansenSpangenberg-Aspects-2007.pdf

2 Invoking Yiihaw

Yiihaw is a command-line program that takes as input a pointcut file, a target assembly and an advice assembly, and produces as output a woven assembly:

```
yiihaw <pointcut_file> <target_assy> <advice_assy> [woven_assy] [-v]
```

Arguments written in angle brackets are mandatory, and those written in square brackets are optional. If the name of the woven assembly is left out, Yiihaw will overwrite the target assembly. The optional argument `-v` can be used to request verbose output on the console concerning the weaving.

3 A complete example

For a sample use of the Yiihaw weaver, consider a target class Invoice declared in file `LowerLayer.cs`, with a method `GrandTotal()` that returns the invoice grand total:

```
public class Invoice {
  public virtual decimal GrandTotal() {
    decimal total = ... computation ...;
    return total;
  }
}
```

Now we want to apply advice to this method so that it provides a 5 percent discount if the grand total exceeds 10 000 Euros. Let this advice class be declared in file `Advice1.cs`:

```
public class MyInvoiceAspect {
  public decimal DoDiscountAspect() {
    decimal total = JoinPointContext.Proceed<decimal>();
    return total * (total < 10000 ? 1.0M : 0.95M);
  }
}
```

and let the pointcut file be this:

```
around * * decimal Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

To compile the target assembly and the advice assembly, and then weave the advice into the target, we perform these commands in the Command Prompt:

```
csc LowerLayer.cs
csc /r:Yiihaw.API.dll /t:library Advice1.cs
yiihaw pointcut1.txt LowerLayer.exe Advice1.dll
```

Compilation of the target and advice assemblies will typecheck those separately. During the subsequent weaving, Yiihaw checks that the advice method is applicable around the target method. Yiihaw will produce the following output on the console:

```
Methods intercepted: 1
Methods targeted, but not intercepted: 0
0 method(s) were introduced at 0 location(s).
0 properties were introduced at 0 location(s).
0 field(s) were introduced at 0 location(s).
0 event(s) were introduced at 0 location(s).
0 type(s) were introduced at 0 location(s).
---------------------------------------
0 warning(s)
```

In addition, Yiihaw writes the resulting woven assembly to disk. Since we did not specify a name for it when invoking Yiihaw, it will overwrite the existing `LowerLayer.exe`. The woven assembly is equivalent to one obtained by compiling the following source code:

```
public class Invoice {
  public virtual decimal GrandTotal() {
    decimal total = ... computation ...;
    return total * (total < 10000 ? 1.0M : 0.95M);
  }
}
```

4 Introductions

Yiihaw currently supports the introduction of a range of constructs into a target class, namely fields, events, methods, properties, classes, struct types, interfaces, enum types and delegate types. There are no restrictions on how these constructs are defined in the advice assembly; Yiihaw inserts the construct exactly as it is defined. For instance, if a method is defined as `public static void` in the advice assembly, it will remain so in the target assembly. It is not possible to instruct the weaver to take a `private` method from the advice assembly and make it `public` in the target assembly.

For details about how to instruct Yiihaw to introduce constructs, see the pointcut language description in section 8.2.

5 Typestructure modification

Using Yiihaw you can make two types of typestructure modifications:

- change the basetype of one or more classes
- implement one or more interfaces

You can instruct a class to implement as many interfaces as you want. Yiihaw will check that the target classes implement all the methods, properties and events of the interfaces. If some of these constructs are not located in the target class already, you need to instruct Yiihaw to insert them first (using the pointcut language).

For details about how to instruct Yiihaw to make typestructure modifications, see section 8.3.

6 Intercepting methods

Yiihaw can intercept both static and instance methods, and methods with any return type, also `void` methods. A typical advice method might look like this:

```
public class Aspects {
  public int Advice(string s) {
    Console.WriteLine("value of s is: " + s);
    return Yiihaw.API.JoinPointContext.Proceed<int>();
  }
}
```

This advice method can be used for intercepting any method in the target assembly that returns `int` and takes a `string` as the first argument. Thus, the following target methods can be intercepted using this advice method:

```
public int TargetMethodA(string x) {
  ...
}
public int TargetMethodB(string s, int i, float f) {
  ...
}
```

The parameter names of the target method need not be the same as those of the advice method, but the parameter types must match in the following sense: The target method must have at least as many parameters as the advice method, and the sequence of parameter types of the advice method must be a prefix of the sequence of parameter types of target method. In the above example, the advice method's parameter list (`string`) is a prefix of the target method's (`string`,`int`,`float`).

During weaving, each parameter of the advice method is replaced by the corresponding parameter of the target method. Hence the advice method's body can use (and update) the target method's parameters simply by using (and updating) the corresponding parameters of the advice method itself.

6.1 Invoking the original target method

The original target method can be invoked from the advice method using the `Proceed<T>()` method, which is defined as a static method on the YIIHAW.API.JoinPointContext class. The value of the call `Proceed<T>()` has type `T` and is the value returned by the target method's body, if any; see section 6.3 below for the case where the target method has return type `void`. There can be at most one call to `Proceed<T>()` in an advice method.

To use the `Proceed<T>()` method you must include a reference to the YIIHAW.API library when compiling the advice assembly. That library is found in a file named `YIIHAW.API.dll`, as shown in section 3.

The type argument `T` of the `Proceed<T>()` method specifies the return type of the target method, which must be the same as that of the advice method. In the above example the type argument `T` was `int`. By specifying the target method's return type as part of the call to `Proceed<T>()`, we obtain two advantages: the compiler can statically check the consistency of the advice source code, and runtime boxing, casting and unboxing of value type results is avoided.

6.2 Generic advice methods

The advice method shown above calls `Proceed<int>()` and can therefore only intercept target methods with return type `int`. If an advice method needs to intercept any kind of method, regardless of its return type, one can make the advice method itself generic with a some type parameter `T`, and let it invoke `Proceed<T>()`:

```
public class Aspects {
  public static T Advice<T>() {
    Console.WriteLine("advice method here...");
    return Yiihaw.API.JoinPointContext.Proceed<T>();
  }
}
```

This advice method can be used for intercepting any method, regardless of its return type. The type T is used as a substitute for the actual return type of the target method being intercepted. The name of the generic parameter does not matter; any name will do. Yiihaw automatically replaces `T` with the target method's actual return type when applying the advice. Like other advice methods, generic advice method can take ordinary value parameters. Consider the following advice method:

```
public class Aspects {
  public static T Advice<T>(int i, string s) {
    ...
  }
}
```

This advice method can intercept any method that takes an `int` and a `string` as the first two arguments. Figure 1 shows some example advice methods and the methods that they can target.

6.3 Targeting `void` methods

As mentioned above, the type argument `T` of the `Proceed<T>()` method call is the target method's return type. As a special case, the target method may have return type `void`. Since `void` is not a valid .NET type, this special case is addressed by using the special `Void` class included in the Yiihaw API:

Advice method	Targetable methods

`void Advice1() {`
`  ... Proceed<Void>() ...`
`}`
	`void M() { ... }`
	`void M(bool x) { ... }`
	`void M(bool x, int y) { ... }`

`void Advice2(bool x) {`
`  ... Proceed<Void>() ...`
`}`
	`void M(bool x) { ... }`
	`void M(bool x, int y) { ... }`

`int Advice3() {`
`  ... Proceed<int>() ...`
`}`
	`int M() { ... return ... }`
	`int M(bool x) { ... return ... }`
	`int M(bool x, int y) { ... return ... }`

`R Advice4<R>() {`
`  ... Proceed<R>() ...`
`}`
	`void M() { ... }`
	`int M() { ... return ... }`
	`string M() { ... return ... }`
	`string[] M() { ... return ... }`

Figure 1: Advice methods and some of their targetable methods.

```
using YIIHAW.API;
using Void = YIIHAW.API.Void;

public class Aspects {
  public static void Advice() {
    JoinPointContext.Proceed<Void>();
  }
}
```

Note the `using` declarations for introducing type abbreviations. As shown in section 6.2, generic advice methods can also intercept target methods with return type `void`.

6.4 Static and non-static (instance) advice methods

A static advice method can be used to intercept both static and non-static (instance) target methods. A non-static advice method can only be used to intercept non-static (instance) target methods; in this case weaving will replace occurrences of the `this` reference in the advice method body with references to the target method's receiver, or `this` reference. Thus, if you need to be able to intercept any kind of target method, you must make the advice method static, in which case you cannot use the `this` reference.

6.5 Storing and using the result of `Proceed<T>()`

The advice method does not have to return the result of `Proceed<T>()` immediately, but may store it in a variable:

```
public class Aspects {
  public static double Advice<double>() {
```

```
    double result = YIIHAW.API.JoinPointContext.Proceed<double>();
    ...
    if (...)
      return result;
    else
      return Math.PI;
  }
}
```

In fact, the result can be used in any way that is compatible with its declared type, which is the type argument of `Proceed<T>()`:

```
public class Aspects {
  public static int Advice() {
    int result = YIIHAW.API.JoinPointContext.Proceed<int>();
    ...
    return result * 7;
  }
}
```

This advice method invokes the original target methods, takes its return value, multiplies it by 7 and returns the new value (effectively replacing the original return value).

7 The Yiihaw API

You have already seen how to use the `Proceed<T>()` method. All static properties and methods defined in the Yiihaw API are shown in figure 2.

Property/method	Type	Value
`AccessSpecifier`	`string`	The access specifier(s) of the method being intercepted
`DeclaringType`	`System.Type`	The declaring type of the method being intercepted
`DeclaringTypeAsString`	`string`	The name of the declaring type of the method being intercepted
`GetTarget<T>()`	`T`	The target method's receiver: its `this` reference
`IsStatic`	`bool`	True if the target method is static, else false
`Name`	`string`	The name of the target method
`ParameterNames`	`string[]`	An array of the target method's parameter names
`ParameterTypes`	`System.Type[]`	An array of the target method's parameter types
`Proceed<T>()`	`T`	Execute the target method's body and get its value, unless `T` is `YIIHAW.API.Void` in which case there is no value
`ReturnType`	`System.Type`	The return type of the method being intercepted
`ReturnTypeAsString`	`string`	The name of the return type of the method being intercepted
`Signature`	`string`	The signature (name, parameter names and types) of the method being intercepted

Figure 2: Static methods and properties in class YIIHAW.API.JoinPointContext.

7.1 Example use of the Yiihaw API

This advice method demonstrates the use of all the Yiihaw API members, except for `Proceed<T>()`, which is described in section 6.1 and `GetTarget<T>()`, which is described in section 7.2.

```
using YIIHAW.API;

public class AdviceClass {
  public static T AdviceApi<T>() {
    Console.WriteLine("Access spec:     " + JoinPointContext.AccessSpecifier);
    Console.WriteLine("Declaring type:  " + JoinPointContext.DeclaringType.Name);
    Console.WriteLine("Declaring type:  " + JoinPointContext.DeclaringTypeAsString);
    Console.WriteLine("Method name:     " + JoinPointContext.Name);
    Console.WriteLine("Is static:       " + JoinPointContext.IsStatic);
    Console.WriteLine("Parameter count: " + JoinPointContext.ParameterTypes.Length);
    Console.WriteLine("1st param type:  " + JoinPointContext.ParameterTypes[0]);
    Console.WriteLine("1st param name:  " + JoinPointContext.ParameterNames[0]);
    Console.WriteLine("Signature:       " + JoinPointContext.Signature);
    Console.WriteLine("Return type:     " + JoinPointContext.ReturnTypeAsString);
    Console.WriteLine("Return type:     " + JoinPointContext.ReturnType.Name);
    return JoinPointContext.Proceed<T>();
  }
}
```

The above generic advice method `AdviceApi<T>()` can be woven `around` any method. Let us use it on the two methods in this class:

```
class Target {
  public static void M1(int x) { ... }
  internal string M2(double y, bool b) { return ...; }
}
```

The output from calling the two methods after weaving is:

```
Access spec:     public
Declaring type:  Target
Declaring type:  Target
Method name:     M1
Is static:       True
Parameter count: 1
1st param type:  System.Int32
1st param name:  x
Signature:       M1(System.Int32 x)
Return type:     System.Void
Return type:     Void

Access spec:     internal
Declaring type:  Target
Declaring type:  Target
Method name:     M2
Is static:       False
Parameter count: 2
1st param type:  System.Double
1st param name:  y
Signature:       M2(System.Double y, System.Boolean b)
Return type:     System.String
Return type:     String
```

Calls to the above-mentioned properties are determined statically and are replaced with appropriate bytecode instructions in the woven assembly. Hence their use imposes no runtime overhead.

7.2 Referring to the intercepted object

The Yiihaw API `GetTarget` method returns the intercepted object, that is, the intercepted method's receiver object:

```
using YIIHAW.API;

public class Aspects {
  public static T Advice<T>() {
    TargetClass targetObject = JoinPointContext.GetTarget<TargetClass>();
    targetObject.SomeMethod();
    return JoinPointContext.Proceed<T>();
  }
}
```

This example obtains the intercepted object (of type TargetClass), binds it to variable `targetObject` and invokes a method on it, where we assume that TargetClass has a method called `SomeMethod`. For this to be type checked, the advice class must know about the target assembly and hence compilation of the advice must include a reference `/r:Target.dll` to the target assembly. The `GetTarget<T>()` method is only valid when intercepting instance methods; a static method has no receiver object or `this` reference. Yiihaw checks this for you.

8 The pointcut language

All pointcuts are defined in a separate text file. There are no restrictions on the name or extension of this file - you can name it whatever you like. Yiihaw defines three types of pointcut statements:

- interception, see section 8.1;
- introduction, see section 8.2;
- typestructure modification, see section 8.3.

8.1 Interception

The general format of the `around` pointcut statement is this:

```
around <access> <invocation kind> <return type> <type>:<method(arguments)>
  [inherits <type>] do <advice_type>:<advice method>;
```

Arguments in angle brackets are mandatory; those in square brackets are optional. All statements must be terminated with a semicolon. A `type` and `advice_type` argument has the form `NamespaceName.TypeName`; if the namespace prefix is left out, the assembly's default namespace is used. Note that period (`.`) is used to separate namespace from type, and colon (`:`) is used to separate type from non-type members, such as methods.

An interception statement starts with the keyword `around`. All arguments between `around` and `do` describe the methods in the target assembly that should be intercepted. You can use wildcards (`*`) for all of these properties. The arguments following the keyword `do` describe the advice method(s) to use. Some examples:

```
(a) around public static void TargetNamespace.TargetClass:Foo(int,string)
    do AdviceNamespace.AdviceClass:AdviceMethod;

(b) around * * * *.*:*(*) inherits System.Collections.Hashtable
    do AdviceNamespace.AdviceClass:AdviceMethod;
```

The first statement (a) matches any method that:

- is public
- is static

- has return type `void`
- is defined on the type TargetNamespace.TargetClass
- is named `Foo`, whether non-generic `Foo` or generic `Foo<T>`, `Foo<T,U>`, ...,
- whose first two arguments have type `int` and `string`

The method named `AdviceMethod` defined on the type AdviceNamespace.AdviceClass is used as advice when intercepting the target methods.

The second statement (b) matches any target method that:

- has any access specifier, because of the asterisk (`*`)
- has any invocation kind, static or instance
- has any return type
- is defined on any type in any namespace, because of (`*.*`)
- has any name
- takes any number of arguments of any type
- whose declaring type inherits from System.Collections.Hashtable

Thus, statement (b) matches any method whose declaring type inherits from System.Collections.Hashtable. If the `inherits` specification were left out, the pointcut would match any method.

8.1.1 Specifying the advice method

The `around` pointcut specifies an advice method by its name, not by its complete signature. Hence there can be multiple advice methods with the same name, if only they have different signatures. For each target method that is being intercepted, Yiihaw will pick the advice method whose signature is the best match for that target method. For example, suppose you define the following two advice methods:

```
public class Aspects {
  public static T Advice<T>() {
    Console.WriteLine("catch-all advice method here...");
    return JoinPointContext.Proceed<T>();
  }

  public static int Advice() {
    Console.WriteLine("int advice method here...");
    return JoinPointContext.Proceed<int>();
  }
}
```

The advice method that has return type `int` is the closer match for all target methods that have return type `int` and will be used for those. The generic advice method will be used for all other methods.

An advice method must match a target method in two ways: By return type and by parameter types, or method signature. Yiihaw will always pick the advice method that has the same return type as the target method (if possible) and pick the advice method that has the longest prefix of parameter types in common with the target method. If given the choice of picking an advice method that matches all parameters or an advice method that matches the return type, Yiihaw

will pick the latter.

Yiihaw only supports `around` interception on methods. Using Yiihaw, `before` and `after` interceptions can be written as `around` interceptions without any performance penalty.

8.1.2 Intercepting generic methods

To intercept a generic method `MyMethod<T>` in the target assembly, the pointcut file must give the method's name in the internal CLI format `MyMethod‘1`, that is, using a backquote and an integer literal that specifies the number of type parameters.

8.1.3 Intercepting properties and indexers

To intercept a property `X` of type `int`, say, in the target assembly, the pointcut file must intercept it as one or both of the two methods corresponding to its `get` and `set` accessors. Consider this class `Target` with an instance property and a static property, each having both `get` and `set` accessors:

```
class Target {
  private int x;
  private static int y;
  public int X {
    get { return x/2; }
    set { x = value * 2; }
  }
  public static int Y {
    get { return y-1; }
    set { y = value + 1; }
  }
}
```

and this advice class:

```
public class AdviceClass {
  public static int AdviceGet() { ... }
  public static void AdviceSet(int value) { ... }
}
```

Then the methods `AdviceGet` and `AdviceSet` can be woven into the target `get` and `set` accessors using this pointcut file:

```
around * instance int Target:get_X() do AdviceClass:AdviceGet;
around * instance void Target:set_X(int) do AdviceClass:AdviceSet;
around * static int Target:get_Y() do AdviceClass:AdviceGet;
around * static void Target:set_Y(int) do AdviceClass:AdviceSet;
```

Now consider a target class with an indexer such as `double this[int]`:

```
class Target {
  double scale = 0;
  public double this[int x] {
    get { return x * scale; }
    set { scale = value/x; }
  }
}
```

Just like a property, an indexer must be intercepted as the two methods corresponding to its `get` and `set` accessors:

```
around * instance double Target:get_Item(int) do AdviceClass:AdviceGet;
around * instance void Target:set_Item(int, double) do AdviceClass:AdviceSet;
```

Although C# provides no static indexers, only instance indexers, other .NET languages such as VB.NET do. Hence the `around` pointcut statement for indexers still requires a specification of `instance` or `static`.

8.1.4 Intercepting constructors

Both static and instance constructors can be intercepted. A constructor is described as a method that has return type `void` and whose name is the same as that of the class. For instance, consider class `Target` with a static constructor and two instance constructors:

```
class Target {
  static Target() { Console.WriteLine("static constructor"); }
  public Target() { Console.WriteLine("instance constructor"); }
  public Target(int x) { Console.WriteLine("instance (int x) constructor"); }
}
```

Pointcut statements to intercept each of these constructors may look like this:

```
around * static void Target:Target() do AdviceClass:AdviceMethod;
around * instance void Target:Target() do AdviceClass:AdviceMethod;
around * instance void Target:Target(int) do AdviceClass:AdviceMethod;
```

8.1.5 Specifying arrays

Arrays can be specified using ordinary C# syntax. The pointcut language only allows specifying single-dimensional, non-jagged arrays, such as `int[]`, `string[]`, etc. However, such an array-specification will match *any* array of the given type. Hence, specifying `int[]` will match `int[]`, `int[,]`, `int[][]` and so on.

8.2 Introduction or insertion

The most elaborate format of the `insert` pointcut statement is this:

```
insert <construct> <access> <invocation kind> <return type>
  <advice type>:<construct_name[(arguments)]> into <type>;
```

An introduction starts with the keyword `insert` followed by a `<construct>` argument that specifies the kind of construct to insert:

- `event`
- `field`
- `indexer`
- `method`
- `property`
- `type` — for inserting classes, structs, interfaces, enum types and delegate types; the construct specifiers `class`, `struct`, `interface` and `delegate` can be used instead.

All arguments preceding the keyword `into` describe the advice assembly constructs that should be inserted into the target class. The arguments `<access>`, `<invocation kind>` and `<return type>` are mandatory for non-`class` constructs, but can be given as wildcards (`*`).

An `insert type` pointcut statement, which can insert a class, struct, interface, enum type or delegate type, has this simplified form:

```
insert type <advice type> into <type>;
```

In any case, the argument following the keyword `into` describes the target type (namespace and class) into which the construct should be inserted. Some examples:

```
(c) insert method public * int Namespace.AspectClass:Foo(string,System.Object)
      into TargetNamespace.Class;

(d) insert type Namespace.AspectClass into TargetNamespace;
```

In a successful weaving, the `insert` statement must match exactly one construct from the advice assembly. That construct is then inserted into the target type TargetNamespace.Class.

The first `insert` statement (c) matches any advice class member that:

- is a method
- is public
- is of any invocation kind (static and instance)
- has return type `int`
- is defined on the type Namespace.AspectClass
- is named `Foo`
- whose first two arguments have type `string` and `object`

The second `insert` statement (d) matches any class, struct, interface or enum type that has name Namespace.AspectClass.

For a more elaborate artificial example, consider this advice class:

```
public class AdviceClass {
  private static int sfi;
  private double fi;
  public event System.EventHandler changed;
  public double P {
    get { return fi; }
    set { fi = value; }
  }
  private class NestedC {
    public int f;
  }
  private struct NestedS : INested {
    public int f;
    public int F { get { return f; } }
  }
  private interface INested {
    int F { get; }
  }
  private enum Months {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
  }
  private double this[int x] { get { ... } }
  private static void PrintMessage(int y) { ... }
}
```

A pointcut file to insert most of the members from AdviceClass into a target class may look like this:

```
insert field * static int AdviceClass:sfi into TargetNamespace.Target;
insert field * instance double AdviceClass:fi into TargetNamespace.Target;
insert event public instance System.EventHandler AdviceClass:changed
  into TargetNamespace.Target;
insert method private static void AdviceClass:PrintMessage(int)
  into TargetNamespace.Target;
insert property * instance double AdviceClass:P into TargetNamespace.Target;
insert indexer * instance double AdviceClass:this(int) into TargetNamespace.Target;
insert type AdviceClass.NestedC into TargetNamespace.Target;
insert type AdviceClass.NestedS into TargetNamespace.Target;
insert type AdviceClass.INested into TargetNamespace.Target;
insert type AdviceClass.Months into TargetNamespace.Target;
```

8.3 Typestructure modification

The general format of the `modify` pointcut statement is this:

```
modify <type> <action> <advice type>;
```

A typestructure modification starts with the keyword `modify` followed by the target type, the one to be modified. The `<action>` argument must be either `inherit` or `implement` and tells the weaver how to modify the target type. If `inherit` is specified, the weaver will make the target type inherit from the advice type. If `implement` is specified, the weaver will make the target type implement the advice type, which must be an interface. Yiihaw will check that all methods of the base class or interface are implemented by the target class, and report an error if not. Some examples:

```
(f) modify TargetNamespace.Class inherit Namespace.AspectClass;

(g) modify TargetNamespace.Class implement Namespace.AspectInterface;
```

8.4 Short form notation for types

All types used in the pointcut statements, such as the return type of a method, must be fully specified with namespace and type name. However, the short-form names for the standard C# types `object`, `string`, `int`, `float` and so on shown in figure 3, may be used instead of the lesser-known full .NET names.

9 Limitations of Yiihaw

The current version (September 2007) of Yiihaw has some limitations:

- An advice class must be an ordinary non-generic class, not a type instance of a generic class.
- A target class must be a non-generic class or a generic class, but not a type instance of a generic class. Generic target classes `C<T>`, `D<T,U>` and so on must be targeted as `C‘1`, `D‘2` and so on in the pointcut file, that is, using the CIL-internal notation for generic types.
- An `around` pointcut statement currently does not distinguish between generic and non-generic target methods. That is, `around * * * Target:M(*) do ...` will apply advice to methods `M`, `M<T>`, `M<T,U>` regardless of type parameter arity.
- The pointcut file parser does not understand type instances of generic classes in field types, return types, parameter types, and so on.

Short name	Full .NET name
`bool`	System.Boolean
`byte`	System.Byte
`char`	System.Char
`decimal`	System.Decimal
`double`	System.Double
`float`	System.Single
`int`	System.Int32
`long`	System.Int64
`object`	System.Object
`sbyte`	System.SByte
`short`	System.Int16
`string`	System.String
`uint`	System.UInt32
`ulong`	System.UInt64
`ushort`	System.UInt16

Figure 3: Short-form type names recognized in pointcut files.

- An advice method cannot be generic beyond the generic parameter used to make it applicable to target methods with different return types.
- When inserting an initialized field `public int x = 42 + 8;` from an advice class into a target class, the initialization code is not automatically inserted into the target class. Doing so in a reliable way is nearly impossible, because at the CLI bytecode level all initialization code appears in constructors, not on the field declarations. Hence this limitation is unlikely to be overcome.