Zero-Overhead Composable Aspects for .NET

Rasmus Johansen, Peter Sestoft, and Stephan Spangenberg

IT University of Copenhagen, Denmark
{johansen,sestoft,spangenberg}@itu.dk

Abstract. We present a new static aspect weaver for C#. The weaver, which is called Yiihaw, works by transforming a program's bytecode and types, stored in so-called assemblies, and performs extensive checks at weave-time to ensure correctness of the resulting woven assembly. The design makes four contributions: (a) Application of generic advice is typesafe; (b) application of "around" advice incurs no runtime overhead; (c) woven assemblies can be further woven; and (d) advice can itself be woven before being applied to target code – in effect advice can be composed. These contributions are achieved by minimal means, basing much of the type checking on the bytecode's generic type system. Yiihaw's aspects are less expressive than those of AspectJ: an aspect does not have an identity of its own; only static join points are supported; and the pointcut language does not allow logical combinations of join points. However, Yiihaw is sufficiently expressive for many purposes, and for these purposes it provides statically typesafe weaving and highly efficient woven code.

1 Introduction

This paper presents a new static aspect weaver for C# and other programming languages based on Microsoft .NET, also known as the Common Language Infrastructure (CLI). The weaver works by transforming CLI/.NET assemblies in the form of `.dll` and `.exe` files and performs extensive checks at weave-time to ensure the correctness, including static type correctness, of the resulting woven assemblies. This aspect weaver, which is called Yiihaw, is intended to address those applications of Aspect Oriented Programming (AOP) in which static type safety and efficiency of the woven code is of paramount importance, and for which some reduction in aspect expressiveness is acceptable.

The design and implementation of Yiihaw makes four contributions, all giving significant practical advantages: (a) Application of generic advice is typesafe; (b) application of "around" advice incurs no runtime overhead; (c) woven assemblies can be further woven; and (d) advice can itself be woven before being applied to target code – in effect advice can be composed.

These contributions are achieved by rather minimal means, where Microsoft's C# compiler and the type system of the CLI take care of most of the type checking. In particular, contributions (a) and (b) rely on the generic methods of the CLI bytecode. Hence it is unlikely that the same advantages can be as easily achieved in aspect weavers for Java, because the Java Virtual Machine bytecode does not include generic types.

E. Börger and A. Cisternino (Eds.): Software Engineering, LNCS 5316, pp. 185–215, 2008.

In return for these advantages, Yiihaw imposes a number of restrictions: an aspect does not have an identity or instance of its own; only join points that can be evaluated statically are supported and hence no "cflow" advice; only execution join points [22] are supported; and the pointcut language does not offer logical combinations of join points. Hence Yiihaw's aspects are considerably less expressive than those of, say, AspectJ [22]. Also, advice supported by Yiihaw is currently generic in the specific sense of parametric polymorphism in the target method's return type, not the more general sense investigated by Kniesel and Rho [23]. However, Yiihaw is sufficiently expressive for many purposes, and for these purposes provides statically typesafe weaving and highly efficient woven code.

For instance, in one application we decompose a collection library into a simple core and a set of features (in the sense of additional functionality), and then use Yiihaw to subsequently, and optionally, add those features again by weaving, without loss of efficiency; see section 7.3. This permits strong modularization of the collection library.

Another potential application is customization of a layered enterprise system, where the ability to further-weave an already woven assembly (section 5), and the ability to compose advice with advice (section 6), are useful.

Many current implementations of AOP for C# and Java use reflection, proxies or similar auxiliary constructs to implement interceptions, which may cause considerable runtime overhead compared to what could be achieved by "manual weaving" of aspects. By contrast, AOP implementations for C and C++, such as AspectC++ [35], generally favour efficiency over flexibility of the aspects. Modern implementations of AspectJ avoid much of the runtime overhead too, though not all [7,17].

Our aspect weaver for CLI/.NET uses inlining of bytecode instructions when applying advice to a target assembly, and hence its goals and its way of working are closer to those of weavers for C and C++ than to those of many weavers for C# and Java. Bytecode inlining restricts the expressiveness of aspects, but avoids many of the auxiliary constructs that other aspect weavers add to the woven assemblies, such as assembly references, transformed copies of advice and target methods, and "around" closures [17]. This means that the program structure defined in the target assemblies is preserved by weaving.

Our current prototype supports various AOP constructs, such as introductions and typestructure modifications. The weaver performs static typechecking on all constructs and guarantees that only valid assemblies are generated, that is, assemblies that are verifiable by the CLI/.NET bytecode loader. Empirical tests show that the weaver prototype does not introduce any runtime overhead in the generated assemblies, making it suitable for applying aspects in performance-critical applications.

Yiihaw source code and its usage guide [21] can be downloaded at

http://yiihaw.tigris.org/

2 Introduction to Yiihaw

Yiihaw is an aspect weaver for CLI/.NET that statically applies aspects to CLI-compatible assemblies.

2.1 An Example Weaving

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

```
public class Invoice {
  public virtual double GrandTotal() {
    double 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 double DoDiscountAspect() {
    double total = JoinPointContext.Proceed<double>();
    return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

and let the pointcut file be this:

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

Next we use the C# compiler `csc` to compile the target class and the advice class, and then invoke `yiihaw` to weave the advice into the target:

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

Note that the C# compiler separately typechecks the target and advice assemblies, and subsequently the weaver ensures that the advice method is applicable around the target method. The resulting woven assembly `LowerLayer.exe` is entirely equivalent to that which would be obtained by compiling this source code:

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

In particular, there is no runtime overhead in the woven method relative to the above hand-written method. The actual CLI/.NET bytecode of the `GrandTotal` method, the `DoDiscountAspect` method and the woven method are shown below.

The Target Method GrandTotal
The target method code computes a `double`, stores it in local variable `total` at offset 0 (using `stloc.0`), loads it again (using `ldloc.0`) and returns it:

```
.locals init ([0] float64 total)
... computation ...
stloc.0
ldloc.0
ret
```

The Advice Method DoDiscountAspect
The advice method `DoDiscountAspect` first calls `Proceed` and stores the result in the local variable `total`. Then `total` is loaded twice onto the stack by `ldloc.0`; first for use in the multiplication and then for the comparison with 10 000 at the conditional branch instruction `blt.s`. The `ldc.r8` instruction pushes a `double` constant, and the `mul` instruction multiplies:

```
.locals init ([0] float64 total)
call ... JoinPointContext::Proceed<float64>()
stloc.0
ldloc.0
ldloc.0
ldc.r8           10000.
blt.s            label
ldc.r8           0.95
br.s             label2
label:   ldc.r8  1.
label2:  mul
ret
```

The Woven Method
In the woven method, instructions from the target and the advice method have been merged, replacing the call to `Proceed` by the target method's instructions. Hence the woven method is completely self-contained; it does not use reflection or proxies and does not call auxiliary methods:

```
.locals init ([0] float64 total,
         [1] float64 V_1)
... computation ...
stloc.1
ldloc.1
stloc.0
ldloc.0
ldloc.0
ldc.r8           10000.
```

```
blt.s            label
ldc.r8           0.95
br.s             label2
label:   ldc.r8  1.
label2:  mul
ret
```

Also note that the `ret` instruction found in the original `GrandTotal` method has been removed, so execution falls through to the first instruction of the advice method, namely `stloc.0`. Similarily, local variable offsets have been updated where necessary, so the bytecode instructions continue to refer to the correct variables.

The woven method has the exact same name, signature, return type and accessibility as the original target method. This ensures that callers of the target method need not be modified or instrumented, and permits further weaving of advice into the woven method; see section 5.

The main drawback of not using a proxy for the behaviour represented by `Proceed` is that one must allow at most one occurrence of `Proceed` in the advice method, or else risk a serious increase in code size. The main advantage of having no proxies is that the structure of the code remains unchanged; further weaving does not need to take proxy methods into account.

2.2 Interceptions

Yiihaw supports interception of all kinds of methods, regardless of their return type, arguments, scope, and so on. Only "around" interception is supported, as it generalizes "before" and "after" interception. Some aspect weavers support "before" and "after" interception because they implement these with less runtime overhead than "around" interception. Since Yiihaw implements "around" interception without any runtime overhead, there is no performance reason to support "before" and "after" interception. Yiihaw does not support "cflow" and other dynamic forms of advice as supported by e.g. AspectJ.

2.3 Introductions

Yiihaw supports introduction of methods, properties, classes, struct types, fields, enum types, delegate types and events into the target assembly. The body of an advice method can refer to constructs that are being introduced into the target assembly by the same weaving; in this case, Yiihaw will automatically update these references so they refer to the corresponding constructs introduced within the target assembly. For details, see section 4.7.

2.4 Modifications

Yiihaw supports modification of any class or struct type defined within the target assembly, either by changing its basetype or by making it implement one or more additional interfaces. In either case, Yiihaw will verify that all required abstract methods, properties and events are implemented by the target class.

3 Generic Types in "Around" Advice

As can be seen from the preceding section, one goal of Yiihaw is to minimize the runtime overhead introduced when applying advice to an assembly, and the bytecode inlining approach guarantees that no auxiliary instructions, references or other constructs are introduced by weaving.

Another goal of Yiihaw is to provide a familiar and efficient programming model for advice code. Many existing aspect weavers provide a primitive *advice language* that leads to wrapping and unwrapping overhead when implementing even simple advice methods. In Yiihaw we want to avoid this.

3.1 Why Wrapping/Unwrapping Overhead?

To see why wrapping/unwrapping overhead occurs, consider the following example written in the syntax of the AspectDNG [5] aspect weaver for CLI/.NET:

```
Object Advice(JoinPointContext jpc) {
  double result = (double)jpc.Proceed();
  return result + 2.0;
}
```

In AspectJ for Java, and in AspectDNG and most other CLI/.NET weavers, the `Proceed` method has return type `Object`. The reason is that different target methods may have different return types, and obviously the advice language should support interception of all types of target methods. Type `Object` is used by AspectDNG as a placeholder for all types. If the user wishes to alter or use the returned value in the advice method, he must typecast the result from `Proceed`, which incurs runtime overhead. Furthermore, when returning a CLI/.NET value type such as `double` in the example above, boxing occurs: the value must be wrapped as a reference type and allocated in the runtime heap. These problems (or similar ones) exist in almost all current aspect weavers for CLI/.NET. However, the AspectC++ [35] weaver for C++ avoids much overhead and in general is closer to our goals for Yiihaw; see section 7.4.

3.2 The Proceed Method

We propose a simple solution to these problems using the generic types of CLI/.NET, which eliminates the need for boxing, typecasting and unboxing. The signature of Yiihaw's `Proceed` method for use in advice methods is this:

```
public T Proceed<T>();
```

The `Proceed` method takes a type parameter `T`. Advice code specifies the target method's return type by instantiating this type parameter, and hence avoids typecasts on the return value:

```
public double Advice() {
  return JoinPointContext.Proceed<double>();
}
```

Moreover, the Yiihaw weaver will check that the return type specified as an argument to `Proceed` equals the target method's return type. Two advantages are obtained. First (a) advice is strongly typed, because the C# compiler will check the advice method's use of the value returned by the `Proceed` method, before applying the advice, and Yiihaw verifies that the specified return type equals the return type of the intercepted methods. Secondly (b) after these compile-time and weave-time checks, it is unnecessary to insert any run-time boxing, typecast or unboxing operations, so no runtime overhead is incurred. This would be impossible to achieve if a too general return type were used on the advice method, such as `Object` in AspectDNG.

Using or Modifying the Value Returned. Yiihaw allows the advice code to use and modify the value returned from the `Proceed` method in any conceivable way that agrees with its stated type:

```
public double Advice() {
  return JoinPointContext.Proceed<double>() + 2.0;
}
```

The advice code can even replace the target method completely with a new implementation by not invoking `Proceed` at all:

```
public double Advice() {
  ...
  return 3.0;
}
```

In this case, Yiihaw will make sure that no instruction or variable defined in the original target method is retained in the generated assembly. The "call" to `Proceed` can also appear in a conditional, a loop, a `try-catch` block and so on, but there can be at most one call to `Proceed` in the advice method source code. This restriction is imposed only to rule out the explosion in code size that could otherwise result from bytecode inlining.

3.3 Generic Advice Methods

Now one might think that the type argument `T` in `Proceed<T>()` means that a given advice method `Advice()` can be applied only to a single type of target method. It turns out that we can again use generic types to overcome this apparent limitation.

Namely, we can make the *advice method itself* generic by giving it a type parameter `T`, to obtain `T Advice<T>(...)` where its return type equals its type parameter `T`. In this case Yiihaw will allow it to be applied to a target method with any return type, and indeed to any number of target methods with any number of different return types.

Consider the following generic advice:

```
public static T Advice<T>() {
  T result = JoinPointContext.Proceed<T>();
  ...
  return result;
}
```

We can think of the type parameter `T` of `Advice<T>` as representing "any type". At weave-time, Yiihaw will replace `T` with the actual return type of the target method being intercepted. This means that the woven method has the exact same return type as the original target method, which helps support further weaving of woven assemblies as well as composition of advice; see sections 5 and 6. For details about replacement of generic variables, see section 4.3.

3.4 Bounded Generic Advice

If the return type of an advice method `T Advice<T>(...)` is the same as the generic type parameter `T` of the method, it is essentially completely abstract, and there is very little the advice method can do with the returned value. More precisely, the *effective base class* [13] of the return type is `Object`, so it is known only to implement methods such as `Equals(Object)`, `GetHashCode()` and `ToString()` that are supported by all CLI/.NET types.

In CLI/.NET and its languages, such as C#, a type parameter can be constrained to implement particular interfaces. This can be used in connection with generic advice methods to (i) tell the advice method what can be done with the return value, and (ii) limit the application of the advice to only such target methods whose return type implements the same interfaces.

3.5 Using the Receiver Object

Like most aspect weavers, Yiihaw supports getting and using the *receiver object*, that is, the object enclosing the method being intercepted (for non-static target methods). This is done using the `GetTarget` method of the Yiihaw API, which has the following signature:

```
public T GetTarget<T>();
```

This method uses the same principle as `Proceed`: The user is forced to specify the actual type `T` of the value he expects `GetTarget<T>` to return. At weave-time Yiihaw will verify that this type corresponds to the actual type being intercepted.

Consider the following example:

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

The `GetTarget` method is invoked with type parameter `TargetClass`, assumed to exist within the target assembly. As `GetTarget` returns a value of this type, no typecasting or boxing is needed. When applying this advice, Yiihaw will verify at weave-time that (1) the target method is non-static and (2a) that the receiver is actually of type `TargetClass` or (2b) `TargetClass` is `Object` and the receiver has reference type, and hence can be cast to `Object` without boxing.

3.6 Example: Universal and Statically Typesafe Synchronization

Using `GetTarget<Object>` and the generic `Proceed<T>` method (section 3.3), one can write a completely generic, yet statically typesafe, aspect for synchronization or locking. For instance, consider a class `Out` with instance methods for writing output, for counting the number of bytes written, and the like:

```
class Out {
  void WriteByte(byte b) { ... }
  void WriteInt(int i) { ... }
  void WriteChar(char c) { ... }
  int BytesWritten() { ... }
}
```

It seems sensible to add synchronization as an aspect, by wrapping the C# statement `lock(this){...}` around the body of each method. With Yiihaw, universal synchronization advice can be expressed like this:

```
class AspectConstructs {
  T SyncAspect<T>() {
    lock (JoinPointContext.GetTarget<Object>()) {
      return JoinPointContext.Proceed<T>();
    }
  }
}
```

This advice can be applied, in a statically typesafe way, to any instance method on any reference type. In AspectJ for Java 5.0, such universal locking advice apparently cannot be written in a statically typesafe way according to Jagadeesan *et al.* [18]. Also, note that the restriction to receivers of reference types is natural and essential. In C#, receiver-based locking is meaningless for value types, because the receiver would be boxed anew in every execution of `lock(this)`, so locking would always succeed.

The C# compiler expands the `lock` statement to a `try-finally` block with calls to entry and exit methods from a monitor library, and the Yiihaw weaver then inlines each target method into such a `try-finally` clause.

3.7 The Applicability of Advice

The rules for applying an advice method to target methods are as follows:

1. A non-static advice method can only be used for intercepting non-static target methods, because it can refer to the target method's receiver reference

`this`. A static advice method can be used for intercepting both static and non-static advice methods, because it cannot refer to the target's `this`.
2. The sequence of parameter types of the advice method must be a prefix of the sequence of parameter types of the target method. This implies that the target method must take at least the same number of parameters as the advice method, and all parameter types must match those of the advice method.
3. The return type of the advice method must equal the return type of the target method. Alternatively, the advice method may use a generic return type, see section 3.3, or `Object` in case the target method's return type is a reference type.

Yiihaw will enforce these rules at weave-time.

4 Yiihaw Implementation

Yiihaw is implemented using the Cecil bytecode manipulation library [9]. Cecil was chosen as it is simple and efficient and supports the low-level operations needed for merging bytecode instructions.

4.1 Assembly Rewriting

Invoking Yiihaw requires that the user specifies (i) a valid pointcut file as a text file, (ii) an existing target assembly to which the aspects should be applied, and (iii) an existing aspect assembly containing advice and other constructs that should be introduced.

The resulting woven assembly is of the same kind — *exe*, *winexe*, *library* or *module* — as the target assembly. The woven assembly will be completely self-contained; it does not depend on the aspect assembly.

4.2 Handling Interceptions

The weaver applies the advice to one target method at a time. Multiple advice may be applied to the same target method, if specified by the pointcut file; the advice will be applied in the order specified. Hence the application of advice by Yiihaw can be seen as a transformation of the (bytecode of) target methods and target types. This transformation is static, performed after compilation but before loading the compiled bytecode. The strengths (type safety, efficiency) and limitations (aspects do not have identity, no dynamic join points) of Yiihaw derive from this staticness. Moreover, such transformations are composable as we shall see in section 6.

The rest of this section describes the approach used for applying advice to a single target method. This approach is repeated for each interception statement in the pointcut file and for each target method.

Merging the Target and Advice. Prior to performing any merging of the advice and the target method, a copy of all instructions of the target method is created. For the sake of discussion, we refer to this copy as the *original body* throughout this section. If the advice contains no call to the `Proceed` method, then the original target method will be ignored, as explained in section 3.2.

The weaver therefore cannot assume that the original implementation should always be kept available. Creating a copy of the body of the target method allows subsequent deletion of some or all instructions in the target method. This way, all instructions of the advice method can just be copied one by one to the woven method without considering whether they fit into any existing method body. Whenever a call to `Proceed` is encountered in the advice, the weaver simply copies all instructions from the *original body* into the woven method. This will be elaborated upon later in this section.

Local Variable Renumbering. Before the *original body* is inserted into the target method, all references to local variables are updated to make sure that they refer to the correct local variable. This is necessary because local variables of the advice method are prepended to the local variables of the original target method.

Handling Return Instructions. When inserting the *original body* the weaver also replaces all `ret` (return) instructions with `br` (unconditional branch) instructions that jump to a fresh label. This is necessary to maintain the correct control flow; a `ret` instruction would prematurely terminate the woven method.

Consider the following target method bytecode:

```
ldarg.1
ldc.i4.5
ble.s     label
ldarg.1
ldc.i4.2
mul
ret
label:  ldarg.1
ret
```

which corresponds to this C# source method:

```
int M(int x) { if (x > 5) return x * 2; else return x; }
```

Further, suppose we want to apply the following advice to that method:

```
call      int YIIHAW.API.JoinPointContext::Proceed<int>()
stloc.0
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

This advice bytecode calls `Proceed` and then prints the string `"advice"` and returns the original return value. It might be written like this in C#:

```
int Advice() {
  int res = JoinPointContext.Proceed<int>();
  Console.Write("advice");
  return res;
}
```

During weaving, the call to `Proceed` in the latter bytecode fragment must be replaced by all instructions from the target method, from the former bytecode fragment. Doing this naively would produce this wrong woven result:

```
ldarg.1                      // From target
ldc.i4.5                     // From target
ble.s     label              // From target
ldarg.1                      // From target
ldc.i4.2                     // From target
mul                          // From target
ret                          // From target
label:  ldarg.1              // From target
ret                          // From target
stloc.0                      // From advice henceforth
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

When executing this method, the advice starting with the `stloc.0` instruction would never be reached, because the method would return as soon as it reached either of the `ret` instructions from the target method (instructions number 7 and 9).

Yiihaw therefore replaces any `ret` instruction with an unconditional branch to a fresh label, just after the last instruction of the target method:

```
ldarg.1
ldc.i4.5
ble.s     label
ldarg.1
ldc.i4.2
mul
br.s      label2             // <-- replaces ret instruction
label:  ldarg.1              // <-- fallthrough instead of ret
label2: stloc.0
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

This maintains the expected control flow. As a small optimization, if the last instruction of the target method is `ret`, Yiihaw will just delete it so that control

falls through to the advice method's code. This explains why the second `ret` instruction from the target method is not replaced in the woven method shown above.

Verifiability of the Generated Bytecode. The procedure described above will produce verifiable bytecode. To see why, consider a given non-`void` target method `R M(...)`, with concrete return type `R`. Whenever execution of the target method reaches an exit point, represented by a `ret` instruction, the stack contains a value of type `R` and nothing else [14, I.12.4]; the net effect of executing the target method's body is to push its return value on the stack before reaching `ret`. Since each `ret` is replaced with a jump `br` to the first instruction P_{resume} following the call to `Proceed`, this means that when P_{resume} is reached in the woven method, the stack top holds a value of type `R` on top of any contents that was already there before executing the code inserted from the target method instead of `Proceed<R>()`. This relies on Yiihaw's weave-time check, prior to applying any advice, that the type `R` expected by the advice code is compatible with the return type of the target method. This applies to generic advice methods and target methods of type `void` as well, as we shall see in the next section.

4.3 Replacing Generic Variables during Weaving

Recall from section 3.3 that when applying generic advice, Yiihaw will change the type of a variable that stores the result of `Proceed`. Consider again this generic advice method from section 3.3:

```
public static T Advice<T>() {
  T result = JoinPointContext.Proceed<T>();
  ...
  return result;
}
```

At weave-time, Yiihaw will replace `T` with the actual return type of the method being intercepted. Consider a target method with return type `int`:

```
public int Target() {
  ...
}
```

Yiihaw will modify the variable `result` from type `T` to `int`. Hence, the type parameter `T` will only exist in the advice, not in the woven methods.

When intercepting methods that return `void` one should not attempt to modify the type of the variable storing the result from `Proceed`, as the variable will contain no value and `void` is not a legal CLI/.NET type for local variables. In this case, Yiihaw will instead remove the variable altogether along with any instructions that refer to it (such as `ldloc` and `stloc` instructions).

Verifiability of the Generated Bytecode. Replacing the generic type parameter with the actual return type of the target method produces CLS-compliant

bytecode. Consider any non-`void` target method `R M(...)`, which returns concrete type `R`: When reaching the point P_{resume} (as defined in section 4.2), we know that a value of type `R` is on top of the stack. Since the `stloc` instruction following the call to `Proceed` stores this value in the local variable, it is safe to modify that variable to have type `R`, because that is the type of the value on top of the stack.

For a `void` target method `void M(...)`, Yiihaw removes the local variable along with any `ldloc` or `stloc` instructions that refer to it. Since the target method obviously does not return anything, the net effect of the *original body* is to *not* place any return value on top of the stack, and there will be no value to load and store. Removing the `ldloc` and `stloc` instructions means that no value will exist on the stack at the time a `ret` instruction is reached, which is just what is intended when intercepting methods of type `void`.

4.4 Updating Code and Variable References

When all instructions have been transferred to the woven method, the weaver scans all of these instructions, looking for dangling code addresses and unoptimized instructions. A dangling code address might occur if an instruction refers to another instruction that has been removed. For instance, instructions that load or store the return value are either modified or removed by the weaver, as described above. If a reference exists to such an instruction it will be invalid at this point. The weaver updates all such references using a mapping table that is built and maintained as instructions get replaced or removed during the weaving. Also, for optimization purposes the weaver checks each instruction to see whether modifying it to a short-form instruction is possible, for instance to modify `ldloc` to the shorter `ldloc.s`.

4.5 Handling GetTarget during Weaving

The `GetTarget` method can be used to get the target method's receiver object, as described in section 3.5. Consider again this example from section 3.5:

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

At weave-time, Yiihaw will verify that the type argument `TargetClass` is compatible with the target method's receiver type. If so, the call to the `GetTarget` method will simply be replaced by a `ldarg.0` instruction which loads the target method's `this` reference. This is possible because the instructions from the advice method and target method are merged, which implies that `SomeMethod` can now be invoked directly on the receiver. Hence, no typecasts, proxies or reflexive calls are introduced for this purpose.

4.6 The Join Point API

Besides the `Proceed` and `GetTarget` methods, which we have already described, the Yiihaw API contains several properties that can be invoked from an advice method. These are summarized in table 1.

Table 1. The Yiihaw API's methods and properties. The type Type below is System.Type from the CLI/.NET Framework Class library.

Property/method	Type	Value
AccessSpecifier	string	Access specifier(s) of the intercepted method
DeclaringType	Type	Declaring type of the intercepted method
DeclaringTypeAsString	string	Name of the declaring type of the intercepted method
GetTarget⟨T⟩	T	Target method's receiver: its `this` reference
IsStatic	bool	True if the target method is static, else false
Name	string	Name of the target method
ParameterNames	string[]	Parameter names of the intercepted method
ParameterTypes	Type[]	Parameter types of the intercepted method
Proceed⟨T⟩	T	Execute the intercepted method and get its value
ReturnType	Type	Return type of the intercepted method
ReturnTypeAsString	string	Name of the return type of the intercepted method
Signature	string	Signature of the intercepted method

All calls to these methods or properties are determined and replaced at weave-time. Consider the following advice, which prints the signature of the target method:

```
public static T Advice<T>() {
  Console.WriteLine(JoinPointContext.Signature);
  return JoinPointContext.Proceed<T>();
}
```

Yiihaw will replace the call to the Signature property with a `ldstr` instruction, such as this:

```
ldstr "Foo(int x, double y, string z)"
```

Similar transformations are performed for all other properties. Hence, using the Yiihaw API does not introduce any runtime overhead in the woven assembly. In particular, the API is not linked in and does not contribute to the size or runtime footprint of the woven code.

4.7 Weave-Time Checks

The following checks are performed at weave-time by Yiihaw:

1. If any target construct (such as a method that should be introduced) cannot be found, the weaving is aborted.

2. If an advice method contains more than one call to `Proceed`, the weaving is aborted.
3. In a call to `Proceed<TA>` where the type argument `TA` is not a generic parameter of the enclosing advice method, `TA` must equal the target method's return type.
4. When implementing interfaces or changing the basetype, Yiihaw will verify that (i) all required abstract methods, properties and events are already implemented by the target class or (ii) that implementations of these methods are being introduced in the same weaving.
5. When an advice method is generic, its type parameter `T` can be used only as the type argument of `Proceed`, as the type of local variables, and in the expressions `default(T)` and `typeof(T)`. Any other use of `T` will be rejected by Yiihaw, as `T` is only used as a substitute for the actual return type and only exists in the advice method.
6. When introducing types, Yiihaw will verify that any assemblies referenced by the aspect assembly are also referenced by the generated assembly, if needed. However, Yiihaw will never make the generated assembly refer to the aspect assembly, only to other assemblies referred by the aspect assembly.
7. If advice is applied to a target method and that advice refers to another construct in the aspect assembly (such as a field), then that construct must be inserted into the target assembly as well. Yiihaw will require that the construct is inserted. Furthermore, Yiihaw updates that reference so that it refers to the "new" copy inserted into the generated assembly, not to the "old" construct in the aspect assembly. For instance, when introducing this class into the woven assembly:

```
namespace Aspects {
  public class Foo {
    ...
  }
}
```

   Yiihaw will update the CLI/.NET reference from `Aspects.Foo` to `Foo` in the target namespace.

Some of these checks, such as rule (3) on `Proceed<TA>`, could be relaxed to admit certain subtypes of `TA` without compromising correctness of the woven assembly. See section 10 on future work.

4.8 Properties of the Woven Result

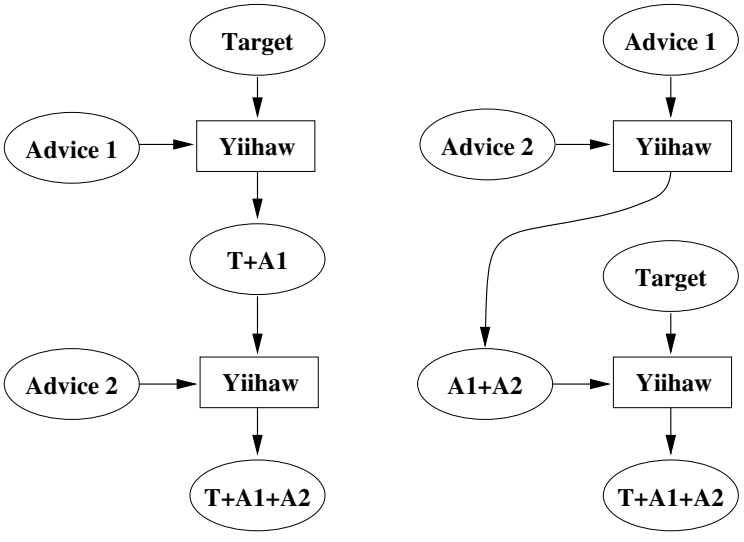
The assembly resulting from the weaving process has several noteworthy properties:

- The woven method that results from weaving advice into a target method has the exact same signature — name, argument types, and result type

— as the original target method. In particular, no name mangling occurs, no wrapper methods are generated, and the return type does not change (section 3). This property enables further weaving of a woven assembly, and further weaving can be done in the exact same way as any other weaving. Moreover, it enables weaving of advice into an advice method before it in turn is woven into a target method; see section 6.
- Inserted fields have the same type and name they had in the advice assembly. Again, there is no name mangling of fields, and no need to represent fields as properties or similar.
- Applying "around" advice to a target does not introduce any runtime casts or any overhead in order to wrap value type results as objects.

5 Further Weaving: Advising Woven Code

Since the result of weaving is an ordinary assembly, an already-woven assembly can be further woven, as shown in figure 1 (a). For a concrete example, consider the woven invoice assembly from section 2.1 and assume that we want to advise it so that when the customer is a charity it returns a grand total of 0 Euros, but adds the grand total to a running sum of tax-deductible gifts. The advice class might look like this, in file `Advice3.cs`:



(a) Further weaving **(b) Advice composition**

Fig. 1. **(a)** Further weaving, where Advice 1 is first woven into Target, giving the assembly T+A1, then Advice 2 is woven into that assembly. **(b)** Advice composition, where Advice 2 is first woven into Advice 1, giving assembly A1+A2, then that assembly is woven into Target. Given appropriate pointcut files, the final woven result is the same in both cases.

```
public class MyNewInvoiceAspect {
  private bool noncharity;
  private double deductible;

  public double CharityAspect() {
    double total = JoinPointContext.Proceed<double>();
    if (!noncharity) {
      deductible += total;
      Console.WriteLine("Deducing {0:F2}", total);
      total = 0;
    }
    return total;
  }
}
```

and the pointcut file then says to insert the `noncharity` and `deductible` fields into the target assembly (which is the already-woven assembly from section 2.1):

```
insert field private instance bool
  MyNewInvoiceAspect:noncharity into Invoice;
insert field private instance double
  MyNewInvoiceAspect:deductible into Invoice;
around * * double Invoice:GrandTotal()
  do MyNewInvoiceAspect:CharityAspect;
```

The necessary compilation and weaving commands are:

```
csc /r:YIIHAW.API.dll /t:library Advice3.cs
yiihaw pointcut2.txt LowerLayer.exe Advice3.dll
```

The result is a new assembly `LowerLayer.exe`, which is equivalent to one that would be obtained by compiling this source code:

```
public class Invoice {
  private bool noncharity;
  private double deductible;
  public virtual double GrandTotal() {
    double total = ... computation ...;
    total = total * (total < 10000 ? 1.0 : 0.95);
    if (!noncharity) {
      deductible += total;
      Console.WriteLine("Deducing {0:F2}", total);
      total = 0;
    }
    return total;
  }
}
```

6 Advice Composition: Advising Advice

Since advice is represented by an assembly, and Yiihaw works by weaving assemblies, Yiihaw can apply advice to advice; see figure 1 (b). With an appropriate

pointcut file, this amounts to composition of advice, in a disciplined manner as also proposed, but apparently not implemented for aspects, by Lopez-Herrejon, Batory and Lengauer [28].

The double weaving

$$t \xrightarrow{a} a(t) \xrightarrow{b} b(a(t))$$

can instead be realized by advising the advice

$$a \xrightarrow{b} b(a)$$

and then applying the composed advice to the target:

$$t \xrightarrow{b(a)} (b(a))(t)$$

The latter approach may have several advantages: It gives early checking of the advice composition, it speeds up advice application when advice a and b must be applied to many target assemblies, it is easier to distribute and apply one piece of advice than multiple pieces of advice, and it makes for conceptual neatness and closure.

For a concrete example, we now show that the two-step weaving of the `Invoice` class shown in sections 2.1 and 5 can be achieved in a different way. First we weave the two advice assemblies together, then we apply the composite advice to the target `Invoice` class.

In the first step we now use this pointcut file:

```
insert field private instance bool
  MyNewInvoiceAspect:noncharity into MyInvoiceAspect;
insert field private instance decimal
  MyNewInvoiceAspect:deductible into MyInvoiceAspect;
around * * decimal MyInvoiceAspect:DoDiscountAspect()
  do MyNewInvoiceAspect:CharityAspect;
```

and these compilation and weaving steps, which advise the target `Advice1.dll` with advice `Advice3.dll`:

```
csc /r:YIIHAW.API.dll /t:library Advice1.cs
csc /r:YIIHAW.API.dll /t:library Advice3.cs
yiihaw pointcut3.txt Advice1.dll Advice3.dll
```

The result is a woven version of `Advice1.dll` which represents the composition of the two advice classes. In the second step we can now apply this composite advice to the `Invoice` target class (section 2.1), using this pointcut file:

```
insert field private instance bool
  MyInvoiceAspect:noncharity into Invoice;
insert field private instance decimal
  MyInvoiceAspect:deductible into Invoice;
around * * decimal Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

and these compilation and weaving commands:

```
csc LowerLayer.cs
yiihaw pointcut4.txt LowerLayer.exe Advice1.dll
```

The resulting woven assembly `LowerLayer.exe` is identical to that obtained by further-weaving in section 5.

7 Evaluation and Applications

Here we discuss two potential applications of Yiihaw, show that it introduces no runtime overhead for generic "around" advice, and compare its capabilities with other aspect weavers for .NET.

7.1 Generating Customized Collection Libraries

The C5 collection library provides generic collection classes for C# and other CLI languages [24]. The library includes the core functionality usually found in a collection library (lists, sets, bags, and so on), but it also provides many extra features, such as support for update events and fail-early enumerations, slidable updateable list views, hash indexes on arraylists and linked lists, and much more. While these extra features will be useful in some scenarios, in other scenarios they will just waste space and time.

By implementing a generator that can build specialized versions of the library, it would be possible to create versions of the library containing only the features actually needed in a given context [19]. The idea is to make a base library that contains only the core functionality, and then generate customized versions by adding features to the base library.

Having defined the base library (with the core functionality), and the extra features represented as advice and aspects, the generator can build a pointcut file based on the selections made by the user. Yiihaw can then weave the desired features into the base library. As Yiihaw uses inlining, the generated library will correspond, in structure and runtime efficiency, to hand-specialized versions of the library.

This vision has been implemented [20] for a small subset of the C5 collection library. The base library implements linked lists and array lists *without* update events and fail-early enumerations. Here is an outline of the linked list class, where the methods `Add`, `Remove`, `RemoveAt` and the `this` set accessor perform updates:

```
public class LinkedList : IList {
  internal int size;
  public Node first, last;
  public class Node { ... }
  public int Count { get { ... } }
  public Object this[int index] { set { ... } }
  public bool Add(int i, Object item) { ... }
```

```
  public Object Remove() { ... }
  public object RemoveAt(int i) { ... }
  public bool Contains(Object item) { ... }
}
```

The implementation of the event aspect consists of an event handler field, a generic advice method `AddCallToOnChanged<T>` to apply around all update methods, and an auxiliary method to raise the event, if any event handler has been added:

```
class EventConstructs {
  public event EventHandler changed;
  public T AddCallToOnChanged<T>() {
    OnChanged(System.EventArgs.Empty);
    return JoinPointContext.Proceed<T>();
  }
  public void OnChanged(System.EventArgs e) {
    if (changed != null)
      changed (this, e);
  }
}
```

The event field and the auxiliary method are inserted into the linked list class and the array list class using these pointcut statements:

```
insert event public * EventHandler EventConstructs:changed
  into Collections.LinkedList;
insert event public * EventHandler EventConstructs:changed
  into Collections.ArrayList;
insert method public instance void EventConstructs:OnChanged(EventArgs)
  into Collections.LinkedList;
insert method public instance void EventConstructs:OnChanged(EventArgs)
  into Collections.ArrayList;
```

The event advice method `AddCallToOnChanged<T>` is wrapped around the four update methods of the two list classes using these pointcut statements:

```
around public * * Collections.*:Add(int,object)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:Remove(object)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:RemoveAt(int)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:set_Item(*)
  do EventConstructs:AddCallToOnChanged;
```

The other feature, fail-early enumerations, can be added in the form of an aspect that inserts an update stamp instance field into each collection class, wraps an update stamp increment around each update method, and inserts a method that returns an enumerator. (The stamp is required for the enumerator to throw an

exception if an update method is called while the enumerator is being used). This aspect can be applied either before or after the above-mentioned update event aspect.

Inspection shows that Yiihaw produces the same bytecode as one would have expected by adding these features to the corresponding collection classes by hand. Also, measurements confirm that Yiihaw introduces no runtime overhead; see section 7.3.

7.2 Customization of a Layered ERP System

In a companion paper [33] in this volume, we consider the use of static aspects for customization of enterprise systems, such as Microsoft Dynamics AX [11].

7.3 Performance of Woven Code

A prominent design goal for Yiihaw was that aspects should incur no runtime overhead in woven code. This goal has been achieved as evidenced both by microbenchmarks and by the case study discussed in section 7.1.

For one microbenchmark, consider a target class with a simple method that takes two `double` arguments and returns a `double`:

```
class Target {
  public double Linear(double x, double y) {
    return x + 0.01 * y;
  }
}
```

Now let us advise it with a generic advice method that simply counts the number of calls:

```
public class CountAdvice {
  private int count;
  public R CountCall<R>() {
    count++;
    return JoinPointContext.Proceed<R>();
  }
}
```

Using a pointcut file that inserts the `count` field into class `Target`, and intercepts method `Linear` by wrapping `CountCall` around it, one obtains a woven class equivalent to this handwoven class:

```
class Handwoven {
  private int count;
  public double Linear(double x, double y) {
    count++;
    return x + 0.01 * y;
  }
}
```

Table 2. Performing 2 billion calls to method `Linear`

	Runtime (s)	Per call (ns)
`Target` before weaving	10.56	5.28
`Target` after weaving	10.77	5.38
`Handwoven`	10.84	5.42

Table 2 shows the execution time for 2 billion calls to the original method, to the method woven by Yiihaw, and to the handwoven method shown above. Each time measure is the average of 7 runs on a 1.6 GHz Pentium M processor and Microsoft .NET 3.5 beta.

Clearly no runtime overhead at all is incurred by weaving the `count++` statement into method `Linear`. In fact, inspection of the bytecode generated by weaving shows it to be identical to that compiled from `Handwoven`.

For a more substantial benchmark, consider the generation of customized collection libraries discussed in section 7.1. We studied the performance of a library obtained by adding update events and fail-early enumerations as aspects to a core implementation of array lists and linked lists, and compared the results to a handwritten library with those features [20, chapter 11]; see table 2. As can be seen, also in this more substantial benchmark, the weaving by Yiihaw introduces no overhead at all.

Table 3. Performance of three implementations of a collection library with update events and fail-early enumerations. Execution time in milliseconds.

How implemented	Events	Enumeration
Handwritten	8547	602
Woven by Yiihaw	8545	600
Woven by AspectDNG	13941	30247

7.4 Related Work: Other Aspect Weavers

Yiihaw is far from the only aspect weaver to use bytecode rewriting to implement interception. In particular, the AspectJ implementations `ajc` [17] and `abc` [7] use bytecode rewriting to implement "around" advice in many cases. They still seem to incur boxing and unboxing overhead when calling `Proceed` on target methods with primitive return type, although the exact circumstances are not so clear [7, §3.3].

What we believe is particular to Yiihaw is that it achieves type safety of generic advice application and efficiency of the woven code by rather simple means, relying on the generically typed bytecode of CLI/.NET. The resulting predictably of the results and the implementational and conceptual simplicity come at a cost, which is limited expressiveness relative to many other aspect weavers. Nevertheless, Yiihaw fits an interesting and non-empty niche of applications.

Table 4. Comparison of available weavers for .NET. The Around column indicates whether "around" advice incurs extra method calls, argument marshalling, or reflection overhead. The Proceed column indicates whether the use of `Proceed` in generic "around" advice incurs overhead such as boxing, casting and unboxing for primitive values. Notes: Aspect.NET has no `Proceed` but a `RetValue` property of type `Object`. Wicca Phx.Morph binary weaving does not support "around", only "before" and "after" advice; the only overhead incurred by `before` seems to be a method call and parameter passing. It is plausible that Wicca Phx.Morph can support aspect composition but we have found no explicit mention or evidence of this.

Name	Pointcuts	Around	Proceed	Advice weavable	Further weavable	Ref.
AspectDNG	Static	Overhead	Overhead	No	Yes	[5]
Aspect.NET	Static	Overhead	Overhead	No	Yes	[6,32]
Aspect#	Dynamic	Overhead	Overhead	No	No	[4]
DotSpect	Static	Overhead	(No Proceed)	No	No	[10]
EOS	Dynamic	Overhead	Overhead	No	No	[15]
NKalore	Static	Overhead	Overhead	No	No	[29]
PostSharp LAOS	Static	Overhead	Overhead	No	Yes	[30]
Rapier LOOM	Dynamic	Overhead	Overhead	No	Yes	[27]
Wicca Phx.Morph	Static	(No around)	(No Proceed)	No?	Yes	[38]
Yiihaw	Static	No overhead	No overhead	Yes	Yes	[20,39]

Table 4 compares several features of known weavers for CLI/.NET and shows that only Yiihaw offers generic "around" advice without boxing, casting and unboxing overhead. Also, Yiihaw is the only one known to support both advice composition (that is, pre-weaving of advice) and further-weaving of already woven assemblies.

In addition to the .NET weavers listed, we are aware of AOP.NET [1], Gripper-LOOM.NET [27], Setpoint [34] and Weave.NET [37], but these seem to be either unavailable for experimentation or no longer maintained, which makes it difficult or unfair to assess their capabilities.

Yiihaw admits only static pointcuts, not "cflow" and similar, and its design goals and achievements appear more closely related to those of AspectC++ [35] than to most Java aspect weavers such as AspectJ. In fact, for static pointcuts Yiihaw seems to incur even less overhead than AspectC++ because some shortcomings in current C++ compilers slightly impair the performance of AspectC++ [26].

AspectC++ seems to support typesafe application of generic "around" advice and to avoid overhead when primitive type values are returned [25]. To our knowledge no Java aspect weaver has this property, and no C# aspect weaver has it except for the Yiihaw weaver presented in this paper.

Unlike Yiihaw, AspectC++ does not seem able to perform advice composition by weaving, because AspectC++ weaving implies a relatively radical transformation of the target program.

Ways to control *dynamic* advice composition, in which advice may advise itself, have been studied and implemented in the AspectJ* weaver [8]. Apel and

others [2] investigate the relation between aspect refinement, mixins and feature-oriented programming.

8 Current Limitations of the Yiihaw Weaver

There are some limitations in the current version of Yiihaw that we may want to lift in a future version.

8.1 Yiihaw Does Not Support Aspect Instances

In Yiihaw, an aspect does not have its own state, neither as a singleton (per aspect declaration) nor per target, unlike in AspectJ and related systems.

8.2 No Dynamic Join Points

Yiihaw does not support join points, such as "cflow", where advice is applied only if a particular method has been called and has not yet returned. Dynamic join points are clearly more expressive, and useful in some applications, but we have not yet encountered a need for them in our motivating application: collection library specialization. Also, they pose interesting implementation and optimization challenges that we would rather avoid; we would prefer to statically ensure that no runtime overhead (in time or space) is imposed, even at the cost of limited expressiveness. The purpose of a collection library is to achieve high performance, and we want to avoid any too-general mechanism that imposes runtime overhead.

Although Yiihaw does adhere to the motto "*aspect-oriented programming is quantification and obliviousness*" [16], the quantification permitted by Yiihaw pointcuts is rather limited. Hence one might question whether Yiihaw can be considered an aspect weaver at all, or whether it should be seen as a tool for bytecode-level composition of mixins or roles; for a conceptual clarification see Apel et al. [3].

8.3 Only Method Execution Pointcuts

Yiihaw supports interception by method execution pointcuts and constructor execution pointcuts, but not advice around read access or write access to fields. The latter could be implemented by bytecode weaving of the assemblies in which the accesses occur, but would require access to all assemblies that *use* the advised fields, which is undesirable.

Note that C# properties `P` and indexers `this[]` *can* be advised, by targeting the methods `get_P`, `set_P`, `get_Item` and `set_Item` to which they are compiled.

8.4 Limited Pointcut Language

The pointcut language currently can express only pointcut literals, not logical combinations such as intersection ("and"), union ("or") or complement ("not") of pointcuts.

8.5 No Instances of Generic Advice Classes

While Yiihaw can weave generic advice methods into a target as shown in section 3, it cannot weave particular *type instances* of generic advice classes and generic advice methods. To some extent this is due to a temporary limitation in the pointcut file syntax, which in turn is related to the C# compiler's renaming of a generic source class `C<T,U> { ... T ... }` to the generic bytecode class `C‘2[T,U] { ... !1 ... }`. This renaming is standardized by Common Language Subset rule 43 in the Ecma CLI standard [14, I.10.7.2].

For an example where advising with an instance of a generic advice class would be extremely useful, consider the (hypothetical) caching aspect below, which ought to be applicable to any one-argument method with static type checks and no runtime overhead, even when the argument or return type is a primitive type:

```
public class CacheAdvice<A,R> {
  static Dictionary<A,R> cache = ...;

  static R MethodAdvice(A x) {
    if (cache.ContainsKey(x))
      return cache[x];
    else
      return cache[x] = JoinPointContext.Proceed<R>();
  }
}
```

8.6 No Generic Target Classes

Yiihaw *does* support the weaving of generic target methods, both with non-generic and generic advice methods, but currently the pointcut file must specify the names of the targeted methods in the CLI/.NET bytecode format `Method‘2` instead of the source format `Method<T,U>`, using the renaming performed by the Microsoft and Mono [31] implementations when compiling generic methods (similar to the CLI generic class renaming mentioned above). This ability also implies that generic advice can be composed (section 6) .

The pointcut language syntax does not allow a target class to be a generic class such as `List<T>` or a type instance such as `List<int>` of a generic class.

8.7 The `Proceed<T>` Method Can Be Called Only Once in Advice

Since Yiihaw "around" advice is implemented by inlining the target method at every occurrence of `Proceed<T>` in the advice method, multiple occurrences would lead to code duplication. To avoid this, Yiihaw allows at most one occurrence of `Proceed<T>` in an advice method. The restriction could be lifted at modest extra work (renaming of local variables) in the weaver, but the restriction has not been onerous in the applications we have considered.

8.8 No Special Debugging Support

A woven assembly will consist of a mixture of the target assembly and any number of copies of fragments from the advice assembly. A standard debugging environment cannot track each type, member and bytecode instruction back to the original source file without some assistance.

Currently Yiihaw does not provide any such assistance, but it might be extended to weave also debugging information in parallel with the assembly weaving, for instance by manipulating `.pdb` ("program database") files associated with the .NET assemblies. Work in this direction can build on existing research on debuggable aspect weaving [12].

8.9 Cannot Weave into Signed Assemblies

Yiihaw (naturally) cannot weave advice into signed assemblies, and therefore cannot add aspects to the .NET Framework Library classes, for instance.

9 The Expression Problem

One touchstone for a program composition technique is whether it offers a plausible solution to the *expression problem*. This is the well-known challenge of how to organize expression syntax definition and expression processors so that the set of data (syntax) variants and the set of processors can be extended independently and in a typesafe manner. See Torgersen [36] or Zenger and Odersky [40] for an introduction and references.

It would seem that one could use normal object-oriented structure for adding new data variants, and use aspects for adding new processors. But the latter does not quite work with the current Yiihaw weaver, because it does not take into account that the base type of some other type has changed, and that new operations have become available.

To see this, consider the following base target assembly where we have data variants `Num` and `Plus`, and one operation `Eval`:

```
public interface IEval {
  int Eval();
}
public interface IExpr : IEval { }
public class Num : IExpr {
  int value;
  public Num(int value) {
    this.value = value;
  }
  public int Eval() {
    return value;
  }
}
class Plus : IExpr {
```

```
  IExpr left, right;
  public Plus(IExpr left, IExpr right) {
    this.left = left;
    this.right = right;
  }
  public int Eval() {
    return left.Eval() + right.Eval();
  }
}
```

Adding a new data variant, say `Neg`, can be done in one place in standard object-oriented style. We will now try to add a new operation `Show()` as an aspect. We can define a new interface `IShow` to describe the show method:

```
public interface IShow {
  String Show();
}
```

and then either modify interface `IExpr` to extend that interface, or insert method `Show()` into interface `IExpr`.

Then we can add `Show()` to the `Num` class, thus ensuring it implements the modified `IExpr` interface, by defining an advice method and inserting it into the existing Num class:

```
public class NumShow {
  public String Show() {
    return value.ToString();
  }
}
```

However, this will be rejected by the C# compiler because there is no field called `value`. One solution to this problem is to make `NumShow` a subclass of `Num`, provided `Num`'s `value` field were not private.

Another solution is to add a "preliminary" field to the `NumShow` class, like this:

```
public class NumShow {
  int value;
  public String Show() {
    return value.ToString();
  }
}
```

Then this advice file would compile. Moreover, the Yiihaw aspect weaver would allow it to be applied to the `Num` target class, because that class has a field of the same name and type, and the references to `value` in the advice method `Show` will be adjusted to refer to the target class's `value` field instead. Hence the weaving should succeed from the point of view of the `value` field.

But even more is needed. Consider how to add `Show()` to the `Plus` class. Using some foresight and the idea of "preliminary" fields from above, we add fields of type `IShow` to the advice class:

```
public class PlusShow {
  IShow left, right;
  public String Show() {
    return left.Show() + "+" + right.Show();
  }
}
```

Again this advice file would be accepted by the C# compiler because the required fields exist and their type has the `Show()` method. However, the weaver will now have to realize not only that the target class (`Plus`) has fields called `left` and `right`. It will also have to realize that while the declared type of those fields is `IExpr`, that type has been extended to be a subtype of `IShow`, or at least declare `Show()`, thanks to the ongoing weaving.

Since the woven classes `Num` and `Plus` implement `IShow` only thanks to the same weaving, the weaver's checks must find a maximal fixpoint (checking everything under the assumption that all is well until proven otherwise) rather than a minimal fixpoint (checking everything under the assumption that everything is ill until proven otherwise). This is the subject of future work.

10 Future Work

In future work, we want to remove the type-related limitations listed in section 8 above, in particular to allow weaving with type instances of generic advice classes (section 8.5) and weaving into generic target classes (section 8.6). Moreover, more sophisticated weave-time checks (section 4.7) on required fields and methods should permit a solution to the expression problem (section 9) while ensuring that Yiihaw will produce only well-formed and verifiable CLI/.NET assemblies.

Other future work involves better pointcut file syntax for describing generic advice and target classes, and in general for describing composite types.

It is not an immediate goal for Yiihaw to support more join points, such as "cflow", or to support aspect instances.

11 Conclusion

We have presented Yiihaw, a new static aspect weaver for C#, VB.NET and other languages for the Common Language Infrastructure (CLI) [14], also known as the Microsoft .NET platform. The design makes several practical advances, in part by leveraging the CLI/.NET platform's existing features well. We have shown that for this reason the implementation is relatively simple and non-redundant, and we have given a few examples of the application of the weaver. Finally we have compared Yiihaw with other known weavers for CLI/.NET, and we have listed Yiihaw's limitations and some desirable improvements and avenues for future work.

Acknowledgements. Thanks to Microsoft Development Center Copenhagen and IFIP Working Group 2.11 on Program Generation for comments and feedback,

to Don Batory for comments, feedback and encouragement at the Lípari 2007 summer school, to Sven Apel for pointers to the literature, and to the anonymous referees of AOSD'08 and the Lipari volume for their constructive comments.

Yiihaw is a recursive acronym for *Yiihaw is an intelligent and high performing aspect weaver.*

References

1. AOP .NET.: Home page, `http://sourceforge.net/projects/aopnet/`
2. Apel, S., Kästner, C., Leich, T., Saake, G.: Aspect refinement. unifying AOP and stepwise refinement. Journal of Object Technology 6(9), 13–33 (2007)
3. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering 34(2), 162–180 (2008)
4. Aspect#. Home page., `http://www.castleproject.org/AspectSharp/`
5. AspectDNG. Home page, `http://aspectdng.tigris.org/`
6. Aspect.NET. Home page, `http://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6801`
7. Avgustinov, P.: Optimising AspectJ. In: Programming language design and implementation (PLDI 2005), pp. 117–128. ACM, New York (2005)
8. Bodden, E., Forster, F., Steimann, F.: Avoiding infinite recursion with stratified aspects. In: Hirschfeld, R., Polze, A., Kowalczyk, R. (eds.) NODe 2006 GSEM 2006, GI-Edition edn., September 2006. Lecture Notes in Informatics, vol. P-88, pp. 49–64. Gesellschaft für Informatik (2006)
9. Cecil. Home page., `http://www.mono-project.com/Cecil/`
10. DotSpect. Home page., `http://dotspect.tigris.org/`
11. Microsoft Dynamics. Home page, `http://www.microsoft.com/dynamics/`
12. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging aspect-enabled programs. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 200–215. Springer, Heidelberg (2007)
13. Ecma International TC39 TG2. C# Language Specification. Standard ECMA-334, 3rd edition. Geneva, Switzerland (June 2005), `http://www.ecma-international.org/publications/standards/Ecma-334.htm`
14. Ecma International TC39 TG3. Common Language Infrastructure (CLI). Standard ECMA-335, 3rd edition. Geneva, Switzerland (June 2005), `http://www.ecma-international.org/publications/standards/Ecma-335.htm`
15. EOS. Home page, `http://www.cs.iastate.edu/`
16. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis (October 2000)
17. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: Third international conference on Aspect-oriented software development (AOSD 2004), pp. 26–35. ACM, New York (2004)
18. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming 63(3), 267–296 (2006)
19. Johansen, R., Spangenberg, S.: Generation of specialized collection libraries. Fourweek project, IT University of Copenhagen (2006)
20. Johansenand, R., Spangenberg, S.: Yiihaw. An aspect weaver for. NET. Master's thesis, IT University of Copenhagen, Denmark (February 2007)

21. Johansen, R., Spangenberg, S., Sestoft, P.: Yiihaw .NET aspect weaver usage guide. Technical report, IT University of Copenhagen, Denmark (September 2007)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Kniesel, G., Rho, T.: A definition, overview and taxonomy of generic aspect languages. L'Objet, Special Issue on Aspect-Oriented Software Development 11(2–3), 9–39 (2006)
24. Kokholm, N., Sestoft, P.: The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 254 pages (January 2006)
25. Lohmann, D., Blaschke, G., Spinczyk, O.: Generic advice: On the combination of AOP with generative programming in aspectC++. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 55–74. Springer, Heidelberg (2004)
26. Lohmann, D., et al.: A quantitative analysis of aspects in the eCos kernel. In: Berbers, Y., Zwaenepoel, W. (eds.) EuroSys 2006, Leuven, Belgium, April 2006, pp. 191–204. ACM, New York (2006)
27. Rapier LOOM. Home page, `http://www.dcl.hpi.uni-potsdam.de/research/loom/`
28. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. In: PEPM 2006: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 68–77. ACM, New York (2006)
29. NKalore. Home page, `http://aspectsharpcomp.sourceforge.net/`
30. PostSharp. Home page, `http://www.postsharp.org/`
31. Mono Project. Home page, `http://www.mono-project.com/`
32. Safonov, V.: Aspect.net: Concepts and architecture. NET Developer's Journal (October 2004), `http://dotnet.sys-con.com/read/46616.htm`
33. Sestoft, P., Vaucouleur, S.: Technologies for evolvable software products. In: Börger, E., Cisternino, A. (eds.) Software Engineering. LNCS, vol. 5316, pp. 216–253. Springer, Heidelberg (2008)
34. Setpoint. Home page, `http://setpoint.codehaus.org/`
35. Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: An AOP extension for C++. Software Developer's Journal 5(68-76) (2005)
36. Torgersen, M.: The expression problem revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
37. Weave.NET. Home page, `http://www.dsg.cs.tcd.ie/dynamic/?category`
38. Wicca. Home page, `http://www1.cs.columbia.edu/`
39. Yiihaw. Home page, `http://yiihaw.tigris.org/`
40. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: Workshop on Foundations of Object-Oriented Languages, Long Beach, USA (January 2005)