Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

A. A. 2009-2010

Master's thesis

Manipulating Code Annotations

Candidate

Nicola Giordani

Supervisors

Antonio Cisternino

Reviewer

Quei che mi corrono dietro
presto presto mi annoiano,
la nobiltà non fa per me,
la ricchezza la stimo e non la stimo.

Tutto il mio piacere consiste
nel vedermi servita, vagheggiata, adorata.

Questa è la mia debolezza
e questa è la debolezza
di quasi tutte le donne.

Carlo Goldoni, *La locandiera*

Acknowledgment

I was sitting on a sofa at the Computer Science department of my university, when I suddenly remembered that I should better had to write some decent sentences to thank many people. Soon after I started to type some words on the keyboard, I realized that seldom is too easy to reward with a simple "thank you" all the efforts of the people who helped us. To show our real gratitude to the people who deserved our acknowledgment is much more difficult. Indeed, this time someone could feel just like he has been forgotten. This will not happen, if anybody who offered me his help will not be disappointed by this thesis, because he can find his contribution, in a way or another, in making this work possible. Of course history, progress and evolution seem to follow a non linear growing, since only the great endeavors and great discoveries seem to leave a durable trace. On the other hand, even physics showed that even the whole universe is defined, in the end, only by the apparently insignificant value of a very few constants. However, whatever is the effort's entity, there is no doubt that it is originated by the human desire of knowledge and truth. So, although science could never be able to discover the answers to the fundamental questions, we know that -in spite of any fate- where science may not arrive, Hope will.

My deepest gratitude goes to my supervisors, Vincenzo Gervasi and Antonio Cisternino, for their invaluable help and constant trust, which never failed.

A very affectionate thank you to my parents, for everything and their patience.

Last but not least, my sweetest thanks to that wonderful beauty who does not fear me anymore.

Ero seduto su uno dei divanetti del dipartimento di informatica della mia università, quando improvvisamente mi sono ricordato che avrei dovuto scrivere qualche bella frase per ringraziare molte persone. Subito dopo aver battuto qualche parola sulla tastiera, mi sono accorto che spesso è troppo facile ricompensare con un semplice grazie, tutti gli sforzi delle persone che ci hanno aiutato. Ancora più difficile è mostrare la nostra gratitudine più vera alle persone che la meritano. Infatti, in questa occasione qualcuno potrebbe pensare di essere stato dimenticato. Ciò non succederà se chiunque mi ha offerto il suo aiuto, non rimarrà deluso da questa mia tesi, perché potrà trovare il suo contributo nell'averla resa possibile. Certamente la storia, il progresso e l'evoluzione sembrare seguire una crescita non lineare, visto che solo le grandi imprese e le grandi scoperte sembrano lasciare una traccia duratura. D'altra parte, anche la fisica dimostra che perfino l'intero universo è definito, in ultima analisi, dal valore apparentemente insignificante di pochissime costanti. Comunque sia, qualunque sia l'entità dello sforzo, non c'è dubbio che esso sia generato dall'umano desiderio di conoscenza e verità. Così, anche se la scienza non dovesse mai riuscire a trovare le risposte alle domande fondamentali, sappiamo che, nonostante qualunque fato, dove la scienza non potrà arrivare, giungerà la Speranza.

Il mio più profondo ringraziamento ai miei relatori, Vincenzo Gervasi e Antonio Cisternino, per il loro inestimabile aiuto e la loro fiducia costate, che non è mai mancata.

Ai miei genitori con tutto il mio affetto, per ogni cosa e la loro pazienza.

Per ultimo perché più importante, il mio dolce grazie a quella meravigliosa bellezza, che ora non ha più paura di me.

Apologue

When I was a boy I used to build quite extravagant starships with composable bricks, not to launch a mighty attack to an ostile planet, but rather to undertake dreamy journeys throughout the deepest space.

While I was attending school, I was used to read -often secretly- anthologies or epic books, not because I was afraid of the upcoming test, rather to simply taste the pleasure to know and to feel emotions, to feed my curiosity, to appreciate the difference -given it actually exists- between reality and fiction, ideas crowding our mind and actions constrained by physic restraints or the trenches of our conscience.

Still now, when I look at one of the countless nature's manifestations, I find out that I do that not because I want to think over the scientific reasons of them; rather, I do that only for it is a depiction of Beauty.

After reading all the pages that follows, someone might be excepting, being not afraid to be wrong, that this work is not very useful.

Such a criticism could be faced peacefully -with an harmless dose of proud- using the same words contained in the famous preface to Oscar Wilde's "The Picture of Dorian Gray", which states that "all art is quite useless".

Quando ero un bambino ero solito costruire stravaganti astronavi con i mattoncini, ma non allo scopo di sferrare poderosi attacchi contro pianeti ostili, quanto piuttosto per compiere immaginifici viaggi attraverso i più remoti spazi siderali.

Quando frequentavo la scuola, leggevo, spesso di nascosto, il libro di antologia o quello di epica, non perchè temessi l'esito della prossima interrogazione, bensì semplicemente per assaporare il piacere di conoscere e provare emozioni, di sfamare la mia curiosità, di apprezzare la differenza, se poi davvero esiste, fra la realtà e la finzione, fra le idee che affollano la nostra mente e le azioni costrette dai limiti della fisica o trattenute dalle trincee della coscienza.

Anche ora, quando mi capita di osservare una delle innumerevoli manifestazioni della natura, mi accorgo di farlo non tanto per riflettere sulla sua spiegazione scientifica, quanto piuttosto perchè è una rappresentazione della Bellezza.

Dopo aver letto le prossime pagine, qualcuno potrà muovere l'obiezione, senza timore di poter essere smentito, di come questa tesi non sia, dopotutto, molto utile.

Ad una tale critica, potrò serenamente rispondere con una innocua dose d'orgoglio e con le stesse parole contenute nella famosa famosa prefazione al racconto di Oscar Wilde "The picture of Dorian Gray", le quali recitano: "all art is quite useless".

Contents

Listings

List of Figures

Chapter 1

Introduction

"The world is a stage, but the play is badly cast"
Oscar Wilde

This thesis concerns language theory and metaprogramming, more specifically, a kind of metaprogramming performed during program execution.

Metaprogramming is constantly increasing its importance and usefulness in many areas of modern software design and development. This is due to a rich variety of reasons which can be easily summarized by the catch phrase "code specialization": there can be (and usually there are often) many different parts of a program that undergo changes, less or more complex, according to different needs or events, such as execution environment constraints or specific configurations, user-specific customization, system integration, performance tuning, and so on; as we will see later, metaprogramming can be very useful even to modify a program's behaviour in response to user actions or, more generally, to environment modifications. The need for specialization arises mainly from two important concerns, which we have to compose in order to get an optimal compromise:

- *code reuse* because we must be able to take advantage of already written functions, classes and libraries
- *code optimization* in terms of both performance and memory requirements

Anyway, both reuse and optimization tends to make the software development process longer and more complex. Indeed, reuse requires at least an accurate software design. Besides, optimization is usually a difficult and bug-prone work. This thesis proposes a technique that help simplify and partially automate many tasks involved on these two aspects of software design and development.

1.1 Run-Time metaprogramming

Most of currently available metaprogramming languages and tools carry out their effects at compilation time, therefore they allow programmers only to use static metaprogramming techniques, making it impossible to apply many other interesting paradigms.

It will be shown how it has been made possible to get a powerful and smart metaprogramming mechanism, which works at runtime on virtual machine level, and which is applicable to any language supported by the virtual machine itself. The mechanism uses only simple source code annotations.

1.2 Goal

This thesis' target is to provide runtime support to the manipulation of annotated code fragments.

More precisely, this thesis shows a technique to perform an efficient metaprogramming activity on a program while the program itself is running, that is at program's runtime; from now on we will refer to this feature as "runtime metaprogramming".

Applications of this technique varies from code specialization to code reuse and deployment.

1.3 Outline

This thesis is organized in 3 parts.

The first part is composed by Chapters 2 and 3. Chapter 2 introduces all relevant matters on which our technique is based as well as technologies and tools that have been used to develop a working implementation of the technique. On Chapter 3 the technique is discussed in full details.

The second part, which is composed by chapters 4 and 5 shows the implementation work, discussing our choices; in Chapter 5 are presented some examples where the developed technique is applied to some realistic scenarios.

The third part contains our conclusions together with some proposals for improvements and future works.

Finally, the appendix contains all source code that has been written from scratch or modified to realize this thesis.

Chapter 2

Background and Tools

"He that is proud eats up himself; pride is his own glass, his own trumpet, his own chronicle."
William Shakespeare, *Troilus and Cressida*

2.1 Reflection

Reflection is the ability of a program to observe and possibly modify its own structure and behavior. Usually the term refers to runtime reflection, though some programming languages support compile time (static) reflection. Reflection is common to almost all modern high-level virtual machine programming languages. More generally, reflection make it possible for a program to reason about itself and its execution. The programming paradigm exploiting reflection is called reflective programming.

Normally, when a program's source code is compiled, information about the structure and semantic of the program, which is called metadata[1] is discarded because it is no longer necessary; on the other hand if a system supports reflection, metadata is preserved and typically is stored in the resulting executable.

A language supporting reflection provides many features available at runtime such as:

- Discovery and, possibly, modification of program elements, such as classes and methods, as first-class objects.

- Istantiation of types known only by their name.

[1]Metadata (from the ancient Greek $\mu\varepsilon\tau\acute{\alpha}$ which meant "after", "beside" and later interpreted as "beyond") means "data about data" that is a set of information about another set of information.

- Indirect invocation of a method or function by its name.
- Generation and execution of new code (code that was not included in original source code).

There are many ways to implement these features but we have to point out that these tasks are easier to implement for interpreted programming languages such as *Python* [?] and *PHP* [?] or virtual machine based languages such as *Java* [?] or *C#* [?] since metadata can be retained and exposed in a natural way by their runtime system.

The following C# piece of code exemplifies some basic reflection's features:

Listing 2.1: Reflection example

```
using System;
using System.Reflection;

public class Test {

  private String msg;

  public Test(String msg) {
    this.msg = msg;
  }

  public void SayHello() {
    Console.WriteLine("Hello World from " + msg);
  }

  public static void Main(string[] args) {
    Type aclass = typeof(Test);
    ConstructorInfo[] ctor = aclass.GetConstructors();
    Console.WriteLine("Instantiating class " + aclass.Name);
    object test = ctor[0].Invoke(new object[]{"Reflection !"})
        as Test;
    MethodInfo m = aclass.GetMethods()[0];
    Console.WriteLine("Invoking method " + m.Name);
    m.Invoke(test, null);
  }

}
```

On line 17 the program obtains an instance of a Type class containing information about the type “Test”; calling “GetConstructors” on this instance the program obtains the constructor’s list of “Test”, then the first list element is used to create an instance of the class. On line 20, constructor’s arguments are passed by an object array.

In a similar way, on line 21, “GetMethods” returns the class’ methods list and, finally, the program invokes Test’s “SayHello” method on the previously created instance of “Test”.

2.2 Metaprogramming

Metaprogramming is the writing of programs that write or manipulate other programs, including themselves, as their data.

It is worth noting that anything related to the word “meta” often lead to some kind of paradox or, in the best case, to something poorly concrete when it is not completely virtual.

Nevertheless, metaprogramming is a very powerful and elegant technique to solve many different problems.

Anyway, it is possible to give some formal definitions:

Meta-programs are programs which represent and manipulate other programs including, potentially, themselves.

Meta-language is the language used to write meta-programs.

Object-programs are those programs that are manipulated by meta-programs.

Generally speaking, it is possible to denote at least three characteristics of meta-programming systems regarding:

- compilation and execution:
  - *static*: the meta-program is executed as part of program compilation or, anyway, before object program execution (e.g.; install-time).
  - *runtime*: the meta-language is executed during program runtime.
- the distinction between meta-program and program:
  - *manually annotated*: the programmer may explicitly specify which parts of the code are meta-program and which are object-program.
  - *automatically annotated*: an external system, like a preprocessor or a source-to-source compiler, is able to distinguish object-program from meta-program.

- language and meta-language:
  - *homogeneous*: meta-language is the object-program language itself.
  - *heterogeneous*: meta-language and object-program language are distinct.

The most famous (and historically first) example of metaprogramming-capable language is *LISP* [?] [?] which permits to process functions exactly like data. Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

Nowadays *MetaML* [?] and *Ruby* [?] present more powerful metaprogramming support at language level enabling them to be used also as *multistage programming*[2] languages.

Multistage programming [?], also called *staged programming*, is a method of producing specialized code by using a program generator. A stage is an object-program evaluation step. Multistage refers to the chance left to the programmer to delay the evaluation of any expression for as many stages as wanted.

A multistage programming language offers (at least) some special operators to

- delay computation of an expression producing a value whose type is code (or an expression), that is a piece of code that has to be evaluated again later.

- *splice* a code value inside a delayed expression. An operator permit to denote sub-expressions inside a delayed expression that are to be replaced by code value (or an expression) in a successive stage.

- *lift* that is to transform a constant expression in a piece of code. Differently from the first operator, lift evaluates the expression before returning a code value representing the expression value itself.

- *run* (computing the value of) a delayed expression. Computation is no longer deferred and the resulting value is a pure value.

A good example of multistage programming is the following specialized implementation of the power function. The power function can be represented

[2]A metaprogramming system can be used to support multistage programming. Another possible tool to realize multistaging is a partial evaluator.

by the following λ-calculus[3] expression

$$power = (\lambda x\ n. \quad n = 1 \quad ? \quad x \quad : \quad x \cdot power(n-1))$$

Using a standard functional language like ML we can translate the expression above in the following function:

Listing 2.2: ML example

```
fun power n = fn x = if n=0 then 1 else x * power(n-1) x;
```

As mentioned above, MetaML is a programming language (a conservative extension of Standard SML) which is oriented to metaprogramming purposes, specifically to multistage programming.

In MetaML, a staged code is described by annotations that are called *meta-brackets* ("⟨" and "⟩").

The splice operator is a $\sim$ preceding an expression.

The lift and run operators are, respectively, the keywords *lift* and *run*.

The previous function can be written as follows:

Listing 2.3: MetaML example

```
fun power n =
  <fn x => ~(if n=0 then <1> else <x*(~(power (n-1)) x)>) >;
```

`power` generates the encoding of an unary function (the outermost angle brackets).

Applying an integer to `power` yields a specialized version of the function which no longer contains conditional nor recursive calls.

Indeed, the body of this function depends upon the value of n. Since n is static we can escape the conditional and it can be delayed at program generation time. Both branches of the `if` construct a piece of code which will be spliced into the body of the function being generated. In the `then` branch this piece of code is the constant $< 1 >$. In the `else` branch, a piece of code is built, containing the multiplication of x by an expression which is spliced in. This spliced expression is built by a recursive call to `power`.

We claim that the operations on code annotations described by this thesis can be used as a part of both a metaprogramming and multistage programming runtime system.

[3]λ-calculus is a formal system designed to investigate function definition, function application, and recursion. It was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s.

2.3 Aspect Oriented Programming

An emerging programming paradigm is the so called “Aspect Oriented Programming”[4] [?], which attempts to focus programmer’s attention towards the separation of concerns avoiding the presence, as much as possible, of different program’s parts overlapping in functionality; this goal is achieved mainly through encapsulation of “cross-cutting” concerns which are recurrent parts of the program that are hard to be encapsulated into single separated entities like methods or classes.

To face this problem AOP introduces a particular construct called an “aspect” that is a fragment of code altering the behaviour of the base code according to some “pointcut” which, in turn, is a definition of a set of “joint points”; whenever program execution reaches each of these join points, a piece of code called an “advice” is executed too, this way changing the default program’s behaviour. Summarizing these concepts:

- An *aspect* is a definition of pointcuts and associated advices.
- An *advice* is an additional behaviour expressed as a code fragment.
- A *join point* is any location in the object-program wherever is possible to insert an advice.
- A *pointcut* is a definition of a set of join points. A pointcut can be a query, a pattern or some other linguistic device to identify one or more join points.

The example listed below should be useful to get an appreciation of AOP:

Listing 2.4: AspectJ example

```
aspect FaultHandler {

  private boolean Server.disabled = false;

   private void reportFault() {
    System.out.println("Failure! Please fix it.");
   }

  public static void fixServer(Server s) {
    s.disabled = false;
```

[4]AOP has been promoted by Gregor Kiczales and his team at Xerox Parc [?]; they developed AspectJ [?] [?], the most popular AOP language

```
11    }
12
13    pointcut services(Server s): target(s) && call(public * *(..))
          ;
14
15     before(Server s): services(s) {
16      if (s.disabled) throw new DisabledException();
17    }
18
19    after(Server s) throwing (FaultException e): services(s) {
20      s.disabled = true;
21      reportFault();
22    }
23
24  }
```

As we can see, an *aspect* definition is made of different parts:

- line 13: a pointcut that will cause insertion of advices at each invocation of any *Server* class' public methods
- line 15: an advice that will be executed *before* any *Server* class' public methods call
- line 19: an advice that will be executed *after* any *Server* class' public methods call but only if the method itself raised a *FaultException*

The operation of injecting advices or, using a more technical terminology, *weaving* aspects into the object-program's code, therefore merging advices' code at the specified join points, can be carried out by a preprocessor, a compiler or even at runtime. It is worth noting that the last option could be easily implemented using annotations together with the runtime code manipulation proposed by this thesis; join points would be indicated by annotations while the "inclusion" (see Section **??**) operation would be used as weaving mechanism. We will discuss again this subject on Section **??**.

2.4 Annotations and Attributes

Recently, code annotation is turned out to be an effective programming technique, applied to almost all non-trivial source code writing activities, because of its undoubtful usefulness.

Indeed, many modern languages permit to "annotate" a program's source code through means of "annotations".

Both Java and C#, two among the most famous general-purpose programming languages, have a special syntax to annotate source code elements with attributes.

An annotation is a specification of one or more attributes which is related to a certain program entities.

Attributes are defined through the declaration of attribute classes; anyway both .NET and Java platforms provides a full set of predefined attribute classes for the most common cases of use. Attributes are attached to entities in a program using attribute specifications and can be retrieved at run-time as attribute instances.

Generally, an attribute specification consists of an attribute name and an optional list of named arguments.

Let us show a small example of annotations usage; the same example is presented in Java and C# to highlight some relevant differences.

Listing 2.5: Attribute usage example (Java)

```java
import java.lang.annotation.*;

@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Test {

  String severity default "normal";

}

public class Foo {

  public static float divide(int a, int b) {
    return  a / b;
  }

  @Test(severity="critical") public static void t1() {
    divide(1, 0);
  }

  @Test public static void t2() {
    divide(0, 1);
  }

}
```

In Java, an attribute is a special interface type (the `@` symbol preceding the `interface` keyword means that the type is an attribute) which can have an arbitrary number of fields with an optional default value. Anyway an attribute's field type is restricted to be `String`, `Class`, `enums`, attribute or an array of these types.

The annotation is a special kind of modifier (see lines 16 and 20) and can be used anywhere other modifiers can be used. Anyhow, it possible for the programmer to permit use of an attribute to certain program's elements only; this is done with an annotation to the attribute class, as we can see on line 3.

Here is the equivalent C# version:

Listing 2.6: Attribute usage example (C#)

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Field, AllowMultiple=true)]
public class TestAttribute : Attribute {

  public string severity;

  public Test() {
    this.mode = "normal";
  }

  public Test(string severity) {
    this.severity = severity;
  }

  public string Severity {
    get { return severity; }
    set { severity = value; }
  }

}

public class Foo {

  public static float divide(int a, int b) {
    return a / b;
  }

  [Test("critical")]
  public static void t1() {
    divide(1, 0);
31   }
32
33   [Test(Severity = "normal")]
34   public static void t2() {
35     divide(0, 1);
36   }
37
38 }
```

In the C# language, an attribute is a normal class type inheriting from the `Attribute` class. As in Java, an annotation is a special modifier (the expressions enclosed by square brackets) and it is still possible to limit the usage of the attribute itself using special attributes (see lines 1 and 2). Despite being the attribute type a normal class type, there are some limitations on attribute's fields type, though they are less restrictive with respect to Java. The C# language also permit both named and positional arguments; positional arguments, if any, precede the named arguments. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class. The order of named arguments is not significant. Furthermore it is a mandatory C# convention to discard the "Attribute" part of the custom attribute's class name when used in an annotation (lines 27 and 32).

In both examples we have defined a new custom attribute named "Test" that simply contains a field whose type is *string*.

Annotations are used on lines 16 and 20 of the Java example and on lines 27 and 32 of the C# example.

Of course, we must point out that in actual compiled code (Java bytecode or .NET IL) there will not be any calls to annotation's constructors; instead their data will be stored in the assembly's metadata section.

An external tool or the program itself will be able to perform introspection to retrieve all the attributes that have been defined inside the program; the C# method below is an example of how to get this kind of information (anyway what follows is nearly the same in Java too):

Listing 2.7: Attribute retrieval example (C#)

```
public static void Main() {
  Attribute[] attrs = Attribute.GetCustomAttributes(typeof(Foo))
      ;
  foreach(MethodInfo m in typeof(Foo).GetMethods())
    foreach(Attritute attr in m.GetCustomAttributes(false))
      if (attr is TesyAttribute) {
            TestAttribute ta = attr as TestAttribute;
            Console.WriteLine(m.Name + " is a test method of " + ta.
                Severity + " severity".);
        }
}
```

Annotation retrieval is obviously obtained through the usage of reflection again.

As expected, the output of the program will be:

```
t1 is a test method of critical severity.
t2 is a test method of normal severity.
```

The example above could be easily extented to invoke any methods that is annotated with the “Test” attribute to obtain a simple automatic testing tool, just like the most famous JUnit [?] or NUnit [?] do.

A very interesting application of annotations to code parallelization has been discussed on [?].

For a more complete technical reference about this topic see [?].

2.5 Annotated C#

Annotated C# (*[a]C#* [?]) is an extension to the C# language to enhance and to promote annotations usage.

[a]C# that has been designed to extend the use of annotations within method's body in attempt to give users a further mean to enrich their code with more information about the code's semantic.

As said above, there can be many interesting ways to reuse these metadata, but we think the most important is those experimented in this thesis which is a chance for the programmer to specify portions, or fragments, of code that can be manipulated to perform various metaprogramming activities; see the examples (Section **??**) discussed later.

Despite standard C# allows annotations to be placed on almost any program's element like classes, methods, fields, properties and so on (see [?]), it doesn't allow the programmer to insert annotations inside methods' body. [a]C# removes this limitation enabling the programmer not only to put custom attributes within a method but to annotate one or more fragments of method's code too. Furthermore, these annotated fragments can be nested. As a result, annotation's expressivity is greatly improved.

The next listing contains a custom attribute declaration and a class with a method exploiting [a]C# annotations:

Listing 2.8: [a]C# example

```
 1 [AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
 2 public class DebugAttribute : CodeAttribute {
 3
 4   public string message;
 5
 6   public DebugAttribute(String msg) {
 7     message = msg;
 8   }
 9
10 }
11
12 public class TestDebugAttribute {
13
14   public void VeryBuggyMethod() {
15     int z;
16     x = DoSomeComputation();
17     [Debug("x can be zero!")] {
18       z = C / x;
19       Console.WriteLine(z);
20     }
21     x = x * z;
22     [Debug("x can still be zero!")] {
23       z = D / x;
24       Console.WriteLine(z);
25     }
26   }
27
28 }
```

It is interesting to note line 1: we have decorated "DebugAttribute" with a special custom attribute (`AttributeUsage`) to declare that is possible for a "DebugAttribute" to be declared on every program's element (`AttributeTargets.All`) and more than once within the same method or class (`AllowMultiple=true`).

An [a]C# program can rely on the runtime of the language for retrieving the annotations stored inside its assembly's[5] metadata.

The most remarkable feature of [a]C# runtime support is its ability to

[5]A configured set of loadable code modules and other resources that together implement a unit of functionality therefore an assembly file is a collection of an arbitrary number of type definitions which binary format is a variant of the standard COFF [?] executable format.

return the complete method's annotation tree; furthermore the tree can be also comprising annotation's trees of any other methods that are called inside the main one.

Consider the following [a]C# class:

Listing 2.9: Nested annotations

```
public class HilbertMatrixTest {

  public int HilbertMatrix(int a, int b) {
    int h;
    [Debug("computation")] {
      h = i + j - 1;
    }
    [Debug("inverse")] {
     return 1 / h;
    }
  }

  public void buildHilbertMatrix() {
    [Debug("rows loop")] {
      for (int x=0; x<N; x++)
      [Debug("columns loop")] {
        for (int y=0; y<N; y++)
          matrix[x][y] = HilbertMatrix(x, y);
      }
    }
  }

}
```

Let's see what will be the output of the following code:

Listing 2.10: Retrieving annotations tree

```
public class AnnotationsVisit {

  private static int depth = 0;

  public static void Visit(AnnotationTree[] at) {
    depth = 0;
    foreach (AnnotationTree t in at) {
      VisitTree(t);
    }
  }

  private static void VisitTree(AnnotationTree node) {
    for (int i=0; i<depth; i++)
      Console.Write("\t");
      Console.WriteLine("[" + ((DebugAttribute) node.Node[0]).
          message + "] {");
      foreach (AnnotationTree tree in node.Children) {
        depth++;
        VisitTree(tree);
        depth--;
      }
    for (int i=0; i<depth; i++)
      Console.Write("\t");
    Console.WriteLine("}");
  }

  public static void Main() {
    MethodInfo m = typeof(HilbertMatrixTest).GetMethod("
        buildHilbertMatrix");
    AnnotationTree[] tree = Annotation.GetCustomAttributes(m,
        true);
    AnnotationsVisit.Visit(tree);
  }

}
```

This listing performs a simple visit of the annotations' tree retrieved by analyzing "buildHilbertMatrix"; as we can see the tree itself contains attribute defined inside "HilbertMatrix" too.

Retrieving annotations' tree

```
[rows loop] {
        [columns loop] {
                [computation] {
                }
                [inverse] {
                }
        }
}
```

Besides that, [a]C# is no further useful, unless it is used in conjunction with other metaprogramming-oriented tools. Indeed, it doesn't provide any further support to manipulation of annotated code fragments (see Section **??**) which is this thesis' primary goal.

As a result, [a]C# grammar has been further extended and its runtime support has been modified accordingly, aiming to the integration with our enhanced version of CodeBricks [**?**], which will be discussed later in Section **??**.

All of this was necessary to reach the final goal of our work, which is to provide a suitable support to perform, at runtime, a set of operations on methods whose body have been annotated like described above.

Finally, we will show an [a]C# bug (see Section **??**) discovered and fixed meanwhile writing and debugging the source code developed in this thesis.

2.6 Common Intermediate Language

Both *.NET* and *Mono* frameworks are different implementors of the same international standard endorsed by *ECMA* [**?**] that is known as *ECMA-335* [**?**]. The Common Language Infrastructure (CLI [**?**]) provides a specification for executable code and the execution environment in which it runs.

At the center of the CLI is a unified type system, the Common Type System that is shared by compilers, tools, and the CLI itself. CTS is the model that defines the rules the CLI follows when declaring, using, and managing types.

The CTS establishes a framework that enables cross-language integration, type safety, and high performance code execution.

The CLI form a unifying infrastructure for designing, developing, deploying, and executing distributed components and applications.

CLI is composed by the following parts:

- The Common Type System (CTS)

  The CTS provides a rich type system that supports the types and operations found in many programming languages. The CTS is intended to support the complete implementation of a wide range of programming languages.

- Metadata

  The CLI uses metadata to describe and reference the types defined by the CTS.

  Metadata is stored (that is, persisted) in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools (such as compilers and debuggers) that manipulate programs, as well as between these tools and the VES.

- The Common Language Specification (CLS)

  The CLS is an agreement between language designers and framework (that is, class library) designers. It specifies a subset of the CTS and a set of usage conventions. Languages provide their users the greatest ability to access frameworks by implementing at least those parts of the CTS that are part of the CLS. Similarly, frameworks will be most widely used if their publicly exposed aspects (e.g., classes, interfaces, methods, and fields) use only types that are part of the CLS and that adhere to the CLS conventions.

- The Virtual Execution System (VES)

  The VES implements and enforces the CTS model.

  The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code, using the metadata to connect separately generated modules together at runtime (late binding).

An appropriate subset of the CTS is available from each programming language that targets the CLI.

Language-based tools communicate with each other and with the VES using metadata to define and reference the types used to construct the application.

The VES uses the metadata to create instances of the types as needed and to provide data type information to other parts of the infrastructure (such as remoting services, assembly downloading, and security).

In summary, the Common Language Infrastructure is a specification describing how applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments.

As core of CLI we have to introduce the Common Intermediate Language (CIL [**?**]) which is an assembler-like language to be interpreted by a multi-threaded stack based virtual machine where each operations read values from a stack and push its result value on the stack.

IL instructions can be subdivided in two parts which we can consider to be “basic operations” and “object model”.

The former constitues a Turing complete set of basic operations and they are independent of the object model that might be employed; these instructions correspond closely to what would be found on a real CPU.

The latter are a set of instructions that are less built-in than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system. These object model instructions provide a common, efficient implementation of a set of services used by many (but by no means all) higher-level languages.

While the CTS defines a rich type system and the CLS specifies a subset that can be used for language interoperability, the CLI itself deals with a much simpler set of types. These types include user-defined value types and a subset of the built-in types. The subset, collectively known as the "basic CLI types", contains the following types:

- a subset of the full numeric types (32 and 64 bits integers, float, etc.).
- object references without distinction between the type of object referenced.
- pointer types without distinction as to the type pointed to.

Note that object references and pointer types can be assigned the value null. This is defined throughout the CLI to be zero (a bit pattern of all-bits-zero).

To get the general feel of *ILAsm* or *IL* (a nickname for the CIL instruction set), consider the following simple example, which prints the well known "Hello world!" salutation.

The salutation is written by calling WriteLine, a static method found in the class *System.Console* that is part of the standard assembly *mscorlib*.

IL code for ''Hello World''
```
.assembly extern mscorlib {}
.assembly hello {}
.method static public void main() cil managed
{ .entrypoint
   .maxstack 1
   ldstr "Hello world!"
   call void [mscorlib]System.Console::WriteLine(
                              class System.String)
   ret
}
```

The `.assembly` extern declaration references an external assembly, mscorlib, which contains the definition of *System.Console*.

The `.assembly` declaration in the second line declares the name of the assembly for this program. (Assemblies are the deployment unit for executable content for the CLI.)

The `.method` declaration defines the global method main, the body of which follows, enclosed in braces.

The first line in the body indicates that this method is the entry point for the assembly (`.entrypoint`), and the second line in the body specifies that it requires at most one stack slot (.maxstack).

Method main contains only three instructions: `ldstr`, `call`, and `ret`.

The ldstr instruction pushes the string constant “Hello world!” onto the stack and the call instruction invokes *System.Console::WriteLine*, passing the string as its only argument.

Note that string literals in CIL are instances of the standard class *System.String*.

As shown, call instructions shall include the full signature of the called method.

Finally, the last instruction, `ret`, returns from main. Every method specifies a maximum number of items that can be pushed onto the CIL evaluation stack. The value is stored in the structure that precedes the CIL body of each method. A method that specifies a maximum number of items less than the amount required by a static analysis of the method, using a traditional control flow graph without analysis of the data, is invalid hence also unverifiable, and need not be supported by a conforming implementation of the CLI.

Maxstack is related to analysis of the program, not to the size of the stack at runtime. It does not specify the maximum size in bytes of a stack frame, but rather the number of items that shall be tracked by an analysis tool.

By analyzing the CIL stream for any method, it is easy to determine how many items will be pushed on the CIL evaluation stack.

However, specifying that maximum number ahead of time helps a CIL-to- native-code compiler (especially a simple one that does only a single pass through the CIL stream) in allocating internal data structures that model the stack and/or verification algorithm.

2.7 CodeBricks

CodeBricks [?] is a framework (an API[6]) designed to allow generating code at runtime by composing methods.

[6]*A*pplication *P*rogramming *I*nterface, a set of functions or, more recently, classes dedicated to one ore more specific tasks which are generally related.

It has been written in C# on the .NET platform and, as a further result of this thesis' work, is now available on the Mono platform too. It uses code generation facilities provided by the standard *System.Reflection* and *System.Reflection.Emit* packages together with functionalities provided by *CLIFileRW* [?] (see Section **??**) which is an assembly's binary format access library.

CodeBricks has been chosen as the base infrastructure to implement our idea on, because of its most interesting property of exposing code as a new data type. Indeed, CodeBricks introduces the notion of *code value*: a value which represents a well-defined piece of code.

The library allows a *Code* object (that is, a code value) to be created from any method, enabling the programmer to handle it as a normal object but, more interestingly, to compose it with other *Code* objects; this is done mainly through the application of two transformations:

- all free variables present in a code value are lifted into bound variables of the resulting code value function.
- partial application of the code value function to the supplied arguments, provided they are compatible with the function's signature types.

Moreover it provides the key feature of carrying out code transformations on the runtime (IL) level whereas the programmer defines these trasformations on the language level in a completely transparent way.

A full description of CodeBricks would out of scope here; see [?], [?] for full details.

We prefer to go on giving another small example that will be complex enough to exploit CodeBricks' power; it is an exponentiation's optimal implementation which relies on the following equivalences:

$$x^{2n} = (x^2)^n$$

$$x^{2n+1} = x^{2n} \cdot x$$

Listing 2.11: CodeBricks example

```
public class PowerGen {

  static Code one = new Code(typeof(PowerGen).GetMethod("One"));
  static Code id  = new Code(typeof(PowerGen).GetMethod("Id"));
  static Code mul = new Code(typeof(PowerGen).GetMethod("Mul"));
  static Code sqr = new Code(typeof(PowerGen).GetMethod("Sqr"));

  public static int One(int i) {
    return 1;
  }

  public static int Id(int i) {
    return i;
  }

  public static int Sqr(int x) {
    return x * x;
  }

  public static int Mul(int x, int y) {
    return x * y;
  }

  public static Code Power(Code x, int n) {
    if (n == 0)
      return one.Bind(x);
    else if (n == 1)
      return x;
    else if (n % 2 == 0)
      return sqr.Bind(Power(x, n/2));
    else
      return mul.Bind(x, Power(x, n-1));
  }

  public static Code Power(int n) {
    Free x = new Free();
    return Power(id.Bind(x), n);
  }

  delegate int power(int x);

  public static void Main() {
    Code p = Power(7);
    power p7 = p.MakeDelegate(typeof(power)) as power;
    Console.WriteLine(p7(2));
  }

}
```

Invoking *Power(x, n)* generates a code object that computes x^n as follows: if n is 0 then the code object for the constant function $\lambda x.1$ is produced (from method One). If the exponent is 1, a code object that computes the value of the second argument is produced, that is the second argument itself. If the exponent is even, the code object for *Power(n/2, x)* is generated and passed to the squaring code object $\lambda x.x^2$ (sqr); otherwise the code object for *Power(n-1, x)* is generated and passed to the multiplier code object $\lambda xy.x \cdot y$ (`mul`).

Method *Power(n)* corresponds to the partial application of *Power(x, n)* to x, obtained by binding a Free object (a Free object represent the concept of free variable) to its first argument.

Looking again at the example we notice that a Code object can be executed through the delegate[7] mechanism; on line 40 it's the definition of a delegate whose signature will match that of the new "Code" object we are going to create.

The *MakeDelegate()* method generates on the fly (that is, at runtime) an appropriate method whose semantic will be the same of that defined by any previous call to *Bind()*.

Below it is the actual IL code generated by invoking *MakeDelegate()* on the Code instance "p7" (line 44); there are not any branch instructions and only 4 `mul` as expected.

IL code generated for ''p7''

```
ldarg.0
stloc.0
ldarg.0
stloc.1
ldarg.0
stloc.2
ldloc.2
ldloc.2
mul
stloc.3
ldloc.1
ldloc.3
mul
stloc.s
```

$\boxed{(x \cdot x = x^2)}$ $\boxed{(x^2 \cdot x = x^3)}$

[7]A delegate is a reference type such that an instance of it can encapsulate one or more methods in an invocation list. Given a delegate instance and an appropriate set of arguments, one can invoke all of the methods in a delegate's invocation list with that set of arguments.

```
ldloc.s
ldloc.s
mul $\boxed{(x^3 \cdot x^3 = x^6)}$
stloc.s
ldloc.0
ldloc.s
mul $\boxed{(x^6 \cdot x = x^7)}$
ret
```

In the next Chapter we will show how CodeBricks has been used as a base support to IL manipulation and runtime code generation as well as how we extended CodeBricks' capabilities. Indeed, we want to make CodeBricks aware of method's body annotations in order to enable the user to treat annotated code fragments like normal methods. Besides, we want also to allow annotations composition (see Section **??**).

Anyway, CodeBricks is not indispensable, it is rather an implementation choice, because we could have directly managed annotations and methods' IL code.

Therefore, the fundamental concept of this thesis, which is manipulation of annotated code fragments, is completely indipendent by CodeBricks. Indeed, we could have also used another IL library such as Cecil[8].

However, this choice would have requested a very hard and bug-prone work without giving any significant advantage on performances. Furthermore, CodeBricks has the remarkable property of exposing a piece of code as a normal data type (*Code* class).

2.8 Lambda Expressions

The 3.0 version of the C# language specification introduced a new feature known as "Lambda expressions" which allows to exploit functional programming [**?**] style and techniques.

Lambda expressions can be regarded as a more concise and elegant syntax for anonymous methods, though they are somewhat more expressive.

We recall that an anonymous method is a delegate declared and defined "in line". Let us show a simple example:

Listing 2.12: Anonymous method example

[8]Cecil (`http://www.mono-project.com/Cecil` is a library to read and write the ECMA CIL format

```
delegate double Compute(int x);

Compute successor = delegate(int x) { return x + 1; };

Console.WriteLine(''The successor of 1 is {0}'', successor(1));
```

The same result can be obtained using a lambda expression, with a syntax that looks like the following:

Listing 2.13: Lambda expression example 1

```
delegate double Compute(int x);

Compute successor = x => x + 1;

Console.WriteLine(''The successor of 1 is {0}'', successor(1));
```

Notice that, in this case, the type of x is inferred by the compiler; whenever any ambiguity arises, it is always possible to declare the type of the arguments.

More complicated statements are also allowed:

Listing 2.14: Lambda expression example 2

```
Func<int, int, int> cantorPair = (int a, int b) => {
  int w = a + b;
  return (w * (w+1)) / 2) + b;
}

// print 18
Console.WriteLine(''The Cantor's pair (2,3) is {0}'',
  cantorPair(2, 3));
```

Note that Func is a framework-defined template delegate:

`public delegate TR Func<T0, T1, TR> (T0 a0, T1 a1)`

While it is quite natural to use lambda expressions to define pure functions, nevertheless they can access variables that have been declared outside the lambda expression itself, thus it is possible to define functions having an internal state:

Listing 2.15: Lambda expression example 3

```
public class TestLambdaExpression
{
  public static void Main (string[] args)
  {
    int x = 0;

    Func<int,int> accumulator = y =>  x += y;

     // print 1
    Console.WriteLine("x is now {0}", accumulator(1));

    // print 2
    Console.WriteLine("x is now {0}", accumulator(1));
      Console.WriteLine(x);
  }
}
```

Taking a look to the IL code generated from the compilation of the source code above, we observe that the compiler has silently produced an hidden class containing, among other things, a method whose body has actually the semantic of the lambda expression defined in the Main method.

For the sake of readability we have removed all non-essentials parts:

IL code of lambda expression

```
.method public static  hidebysig default void Main (string[] args)  cil managed
{
.entrypoint
.maxstack 9
.locals init (
        class System.Func`2<int32, int32>       V_0,
        class TestLambdaExpressionTree/'<Main>c__AnonStorey0'       V_1
        )
newobj instance void class TestLambdaExpressionTree/'<Main>c__AnonStorey0'::'.ctor'()
stloc.1
ldloc.1
ldc.i4.0
stfld int32 TestLambdaExpressionTree/'<Main>c__AnonStorey0'::x
ldloc.1
ldftn instance int32 class TestLambdaExpressionTree/'<Main>c__AnonStorey0'::'<>m__0'(int32)
newobj instance void class System.Func`2<int32, int32>::'.ctor'(object, native int)
stloc.0
ldstr "x is now {0}"
ldloc.0
ldc.i4.1
callvirt instance !1 class System.Func`2<int32, int32>::Invoke(!0)
box System.Int32
call void class System.Console::WriteLine(string, object)
ldstr "x is now {0}"
ldloc.0
ldc.i4.1
callvirt instance !1 class System.Func`2<int32, int32>::Invoke(!0)
box System.Int32
call void class System.Console::WriteLine(string, object)
ldloc.1
ldfld int32 TestLambdaExpressionTree/'<Main>c__AnonStorey0'::x
call void class System.Console::WriteLine(int32)
ret
} // end of method TestLambdaExpressionTree::Main

.class nested private auto ansi sealed beforefieldinit '<Main>c__AnonStorey0'
 extends System.Object
  {
    .custom instance void class
    System.Runtime.CompilerServices.CompilerGeneratedAttribute::'.ctor'() =  (01 00 00 00 )

    .field  assembly  int32 x

    .method public hidebysig  specialname  rtspecialname
           instance default void '.ctor' ()  cil managed
    {
        .maxstack 8
        ldarg.0
        call instance void object::'.ctor'()
        ret
    } // end of method <Main>c__AnonStorey0::.ctor

    .method assembly hidebysig
           instance default int32 '<>m__0' (int32 y)  cil managed
    {
        .maxstack 4
        .locals init (
                int32       V_0)
        ldarg.0
        ldarg.0
        ldfld int32 TestLambdaExpressionTree/'<Main>c__AnonStorey0'::x
        ldarg.1
        add
        dup
        stloc.0
        stfld int32 TestLambdaExpressionTree/'<Main>c__AnonStorey0'::x
        ldloc.0
        ret
    } // end of method <Main>c__AnonStorey0::<>m__0

  } // end of class <Main>c__AnonStorey0
```

To execute the lambda expression, a new instance of a compiler-generated class (<Main>c__AnonStorey0) is created and then the related method invoked.

It is worth noting that the variable x of the Main method has been actually allocated as a field of the compiler-generated class instead of as local variable of the method; in this way, both the Main method and the lambda expression method can access the variable. We will get back on this issue later in **??**.

Lambda expressions have been introduced in C# only in version 3.0, long after the [a]C# compiler was completed; hence, we will not support the lambda expression syntax. Nevertheless, later in **??**, we will discuss briefly about how to allow annotations to be declared also inside lambda expressions.

2.8.1 Expression trees

Another interesting feature of C# 3.0 involving lambda expressions consists of the compiler's ability to automatically generate the code to construct a representation of any lambda expression as an expression tree. An expression tree is a tree-like data structure holding the lambda expression's structure and data which can be accessed by the program itself. Each node of an expression tree represents a (sub) expression of the lambda expression from which the tree is derived.

In summary, an expression tree represents a lambda expression as data instead of as code. In addition, Both the .NET and Mono implementation provides a standard API to programmatically build an expression tree and actually execute its related code.

For what concerns [a]C#, it would be easy to modify the compiler to produce an annotation node representing the annotation's type and attributes for each annotation declared inside a lambda expression.

Here is an example where an expression tree is defined:

Listing 2.16: Expression tree example

```
using System;
using System.Linq.Expressions;

public class TestLambdaExpressionTree
{
  public static void Main (string[] args)
  {
    Expression<Func<int, int, double>> mean = (x,y) => (x+y) /
        2.0;
  }
}
```

Encountering a variable declaration of type "System.Linq.Expressions.-Expression" assigned to a lambda expression, the compiler will automatically produce the IL code to instantiate and populate accordingly an expression tree object representing that lambda expression.

Here follows the key portion of IL code produced by the compiler for the example above:

IL code of expression tree

```
.locals init (
        class Expressions.Expression`1
         <class System.Func`3<int32, int32, float64>> V_0,
        class Expressions.ParameterExpression V_1,
        class Expressions.ParameterExpression V_2)
```

```
ldtoken System.Int32
call class System.Type
 class System.Type::GetTypeFromHandle(valuetype System.RuntimeTypeHandle)
ldstr "x"
call class Expressions.ParameterExpression
 class Expressions.Expression::Parameter(class System.Type, string)
stloc.1
ldtoken System.Int32
call class System.Type
 class System.Type::GetTypeFromHandle(valuetype System.RuntimeTypeHandle)
ldstr "y"
call class Expressions.ParameterExpression
 class Expressions.Expression::Parameter(class System.Type, string)
stloc.2
ldloc.1
ldloc.2
call class Expressions.BinaryExpression
 class Expressions.Expression::Add(class Expressions.Expression,
  class Expressions.Expression)
ldtoken System.Double
call class System.Type
 class System.Type::GetTypeFromHandle(valuetype System.RuntimeTypeHandle)
call class Expressions.UnaryExpression
 class Expressions.Expression::Convert(class Expressions.Expression, class System.Type)
ldc.r8 2.
box System.Double
ldtoken System.Double
call class System.Type
 class System.Type::GetTypeFromHandle(valuetype System.RuntimeTypeHandle)
call class Expressions.ConstantExpression
 class Expressions.Expression::Constant(object, class System.Type)
call class Expressions.BinaryExpression
 class Expressions.Expression::Divide(class Expressions.Expression,
  class Expressions.Expression)
ldc.i4.2
newarr Expressions.ParameterExpression
dup
ldc.i4.0
ldloc.1
stelem.ref
dup
ldc.i4.1
ldloc.2
stelem.ref
call class Expressions.Expression‘1<!!0>
 class Expressions.Expression::Lambda<class System.Func‘3<int32,int32,float64>>
  (class Expressions.Expression, class Expressions.ParameterExpression[])
stloc.0
ret
```

As we can see, the compiler simply translates the lambda expression in terms of a tree-shaped composition of proper classes and methods that are defined in the standard class library accompanying the compiler itself.

2.9 Mono

Mono [?] is the UNIX version of the Microsoft .NET platform.

Mono is an open source development initiative aiming to enable UNIX developers build and deploy cross-platform .NET applications.

The Mono runtime implements the *ECMA* [?] Common Language Infrastructure.

Mono is essentially made up on the following components:

- A Common Language Infrastructure (CLI) virtual machine that contains a class loader, Just-in-time compiler, and a garbage collecting runtime.

- A class library that can work with any language which works on the CLR. Both .NET compatible class libraries as well as Mono-provided class libraries are included.

- A compiler for the C# language. In the future it is supposed to work on other compilers that target the CLR.

Its runtime engine provides a Just-in-Time compiler (*JIT*), an Ahead-of-Time compiler (*AOT*), a library loader, the garbage collector (*Boehm* [?] conservative garbage collector), a threading system and interoperability functionality.

Mono has support for both 32 and 64 bit systems on a number of architectures as well as a number of operating systems.

Mono has both an optimizing just-in-time (JIT) runtime and a interpreter runtime. The interpreter runtime is far less complex and is primarily used in the early stages before a JIT version for that architecture is constructed. The interpreter is not supported on architectures where the JIT has been ported.

On the beginning of 2010, Mono supports the platforms and architectures listed on the table **??**.

Mono aims to support as many programming languages as possible; as of 2010 the situation is summarized in table **??**.

The Mono open source project started in 2001, but the first stable release, 1.0, was issued in 2004, as an effort to implement the Microsoft's .NET Framework [?] to Unix, to bring both the new programming model based on the Common Language Infrastructure (CLI [?]) and C# as well as helping people migrate their existing knowledge and applications to Unix. In November

Supported Architectures	**Runtime**	**Operating system**
s390, s390x (32 and 64 bits)	JIT	Linux
SPARC (32 and 64 bits)	JIT	Solaris, Linux
PowerPC	JIT	Linux, Mac OSX, Nintendo Wii, Sony PlayStation 3
x86	JIT	Linux, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, Solaris, OSX
x86-64: AMD64 and EM64T (64 bit)	JIT	Linux
IA64 Itanium2 (64 bit)	JIT	Linux
ARM: little and big endian	JIT	Linux
HP-PA	JIT	HP-UX
Alpha	JIT	Linux

Table 2.1: Mono's supported architectures

Supported languages	**Under development**
C#, Java, Boo, Nemerle, Visual Basic.NET, Python, JScript, Oberon, Object Pascal, LUA, PHP, Ruby, Cobra	C, Ruby, ADA, Kylyx, Tachy

Table 2.2: Mono's supported languages

2009, the 2.6 version was released which supports almost all the C# 3.0 language features. The work was initiated by *Ximian*, which was further bought by *Novell*[9], and based upon the Microsoft's Rotor (SSCLI [?]).

The Mono Project has also sparked a lot of interest in developing C#-based components, libraries and frameworks. The most important ones, some of which were developed by the Mono team, are:

- Gtk#[10]: Bindings for the popular Gtk+ GUI toolkit for UNIX and Windows systems. Other bindings are available: Diacanvas-Sharp and MrProject.
- #ZipLib (`http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx`): A library to manipulate various kinds of compressed files and archives (Zip and tar).
- Tao Framework: bindings for OpenGL
- Mono.Directory.LDAP / Novell.Directory.LDAP: LDAP access for .NET apps.
- Mono.Data: support for PostgreSQL, MySql, Sybase, DB2, SqlLite, Tds (SQL server protocol) and Oracle databases.
- Mono.Cairo: Bindings for the Cairo (`http://www.cairographics.org/`) rendering engine.
- Mono.Posix / Mono.UNIX: Bindings for building POSIX applications using C#.
- Mono.Remoting.Channels.Unix: Unix socket based remoting
- Mono.Security: Enhanced security and crypto framework
- Mono.Math: BigInteger and Prime number generation
- Mono.Http: Support for creating custom, embedded HTTP servers and common HTTP handlers for applications.
- Mono.XML: Extended support for XML

[9]`http://www.novell.com/`

[10]Gtk# (`http://gtk-sharp.sf.net/`) is a .Net language binding for the **Gtk+** (`http://www.gtk.org/`) toolkit and other assorted GNOME libraries.
Gtk+ is a multi-platform toolkit for creating graphical user interfaces.
It was initially developed for the GIMP, the GNU Image Manipulation Program.
Today GTK+ is used by GNOME and a large number of applications.

- Managed.Windows.Forms (aka System.Windows.Forms): A complete and cross platform, System.Drawing based WinForms implementation.
- Remoting.CORBA (`http://remoting-corba.sourceforge.net/`): A CORBA implementation for Mono.
- Ginzu: An implementation on top of Remoting for the ICE (`http://www.zeroc.com/`) stack

Finally, it worth considering that Mono, after almost 10 years of development, with millions of lines of code and still growing, seems to be a promising software plaftorm well suited both for academic and commercial use.

Chapter 3

Manipulating annotations

"While farmers generally allow one rooster for ten hens, ten men are scarcely sufficient to service one woman."
Giovanni Boccaccio, *Decameron.*

We will show a technique for manipulating at runtime a method's IL code, based on annotations of the same method, in order to obtain a new method. The semantic of this new method will depend on a set of operations that are applied to a set of original method's annotated code fragments. Allowed operations together with their meanings will be described on Section **??**.

The fundamental reasons beneath this effort is that such a feature, for a programming language, is sufficiently general to:

- Give an additional way to hide implementaion details without avoiding code reuse. Some proving examples will be provided on Chapter **??**.
- Promote and simplify code specialization.
- Enable the user to explicit a code fragments' semantic, thus simplifying both writing and reading of source code.

Moreover, this thesis represents a new proof of how runtime metaprogramming can be exploited to allow creation of new programs without actually writing their source code.

3.1 Annotated code fragment

In Section **??** we have seen how to delimit a portion of a method's source code using [a]C# annotations. The boundaries of the delimited portion can be thought to be the opening and the closing brackets; from a technical point of view they actually are, as it will be shown later on.

When no confusion can arise, in the following we will use the term "code fragment" to mean a portion of a method source code or the corresponding IL code, depending on the context.

Not surprisingly, we will refer to the portion of a method's code enclosed by an annotation as an "annotated code fragment".

Now we have to give some definitions that will be used in the following. We call:

- *annotation's free variables* all those variables that are visible in the annotated code fragment, according to scoping rules of the language (see [?]), and are not declared inside the fragment. Annotation's free variables are also any method's arguments that are referred in the annnotated fragment.

- *annotation's local variables* all those variables that are declared inside the annotated code fragment.

Of course, the [a]C# runtime will have to be capable of inferring what are both the *free* and *locals* variables of any given annotation. Full details of this really important matter are presented on Section ??.

3.2 Extending [a]C#

Despite [a]C# annotations seeming to be naturally suited to our purpose, their use is not very natural from a programmer's point of view. Indeed, they lack a way to explicitly declare what we can consider to be the annotation's signature, as to say the annotation's free variables bindings with respect to the external environment.

In fact, though it is possible to automatically infer which are the annotation's free variables through source code parsing or IL code analysis, we are interested on the *order* of the binding because we are going to provide an operation whose semantic will depend on it.

Therefore, we have to extend annotation's syntax to optionally include these binding declarations.

Adding such a feature to [a]C# required to extend language's grammar, source-to-source compiler and runtime support.

3.3 Transforming a fragment in a brick

So far we introduced all those elements needed to perform the "real job" this thesis is focused on.

We can start to go further inside the matter.

3.3.1 Annotated fragment's signature

We need to add a syntactical construct enabling the user to specify annotation's signature.

The best way is to exploit the standard C# attribute specification.

We have chosen to give the user a chance to specify annotation's signature (or bindings in case we are considering the inclusion operation) as normal attribute's positional arguments except for some special characters to distinguish among free variables and return variable.

On Section **??** all relevant details are shown.

Of course, the new [a]C# grammar will have to allow the user to use this feature only on an attribute declaration that is part of an [a]C# annotated block, not just in any attribute declaration.

However, the annotated fragment's signature specification in the attribute declaration will be always optional. Rather an exception will be thrown, at runtime, whenever a signature declaration was needed to perform a particular operation but none was specified.

Whereas the user is allowed to specify whatever signature for an annotated fragment, a signature must considered *invalid* when any of the following conditions holds

- declared signature parameters count is not equal to actual annotated fragment's free variables count.
- declared signature parameters type is not compatible with actual annotation's free variables types.

Both conditions could be checked by a static analysis of the source code that is while parsing the [a]C# source.

Anyway, not to increase parser's complexity we preferred to check them at runtime that is whenever an operation is requested on the annotation itself.

As said, when performing the manipulation of an annotation whose declared signature has been found to be invalid, a runtime error will be signaled by simply throwing an exception.

Instead, it will be the C# compiler to stop compilation in case the user specifies an unknown variable or a variable that is out of scope.

3.4 Operations on annotations

We have defined four different operations, for a total of six variants, on annotated code fragments; their informal description follows:

- *Removal*: annotated code fragment is removed from main method. Although useful, this is the most simple operation, at least for what concerns its implementation.
- *Extrusion*: annotated code fragment becomes a new method, more precisely a brick, whose signature depends on fragment's free variables (see Section **??**).
- *Pre-Inclusion*: a code fragment is injected inside another (or the same) method's code before one of its annotations.
- *Post-Inclusion*: a code fragment is injected inside another (or the same) method's code after one of its annotations.
- *Pre-Copy*: annotated code fragment is copied before another annotation of the same method.
- *Post-Copy*: annotated code fragment is copied after another annotation of the same method.

More formally:

$$Annot_M = \bigcup annotations \in M$$

$$\forall A \in Annot_M, B_A(M) = Pre_A \oplus Begin_A \oplus In_A \oplus End_A \oplus Post_A \tag{3.1}$$

$$Remove_A(M) = Pre_A \oplus Post_A \tag{3.2}$$

$$Extrude_A(M) = In_A \tag{3.3}$$

$$PreInclude_A(M, C) = Pre_A \oplus C \oplus Begin_A \oplus In_A \oplus End_A \oplus Post_A \tag{3.4}$$

$$PostInclude_A(M, C) = Pre_A \oplus Begin_A \oplus In_A \oplus End_A \oplus C \oplus Post_A \tag{3.5}$$

$$PreCopy_A(M, A') = In_{A'} \oplus Pre_A \oplus Begin_A \oplus In_A \oplus End_A \oplus Post_A \quad (3.6)$$

$$PostCopy_A(M, A') = Pre_A \oplus Begin_A \oplus In_A \oplus End_A \oplus Post_A \oplus In_{A'} \quad (3.7)$$

Generated code correctness

To guarantee the correctness of the code generated by the application of the operations above, we have to deal with, at least, the following issues:

- Jump offsets relocation: both inclusion and removal operations requires for any jump instruction's offset, belonging to the input method, to be adjusted accordingly.
- Max stack (see Section **??**): the stack behaviour of the method resulting from an inclusion operation can be different from that of the input method's one. Since the IL code generation framework requires to specify the method maximum stack height we should infer the new value of this parameter. For simplicity, our implementation will just calculate this new value as sum of the input method's max stack height and the included code fragment's max stack height. Both these stack heights are already known.

Other issues related to generated code correctness will be discussed on next Sections.

3.4.1 Removal

The *removal* operation allows to generate a method whose body will be composed of all the instructions contained in the original method, except for all the instructions contained inside the specified annotated fragment.

What we have to do is discarding IL code belonging to a specified annotation, that is all the IL instructions enclosed between the two calls to the annotation boundaries placeholders.

We are just going to make a cut-and-sew-operation, so its implementation is straightforward. Besides, this implementation could be optimized as explained on Section **??**.

In order to grant resulting method validity, that is to guarantee generated IL code correctness, the following conditions must be fulfilled:

- correct jump addresses: in the case that we remove an annotated fragment targeted by a branch instruction placed outside the fragment itself, then the resulting method will contain a jump to an invalid location.

  Fortunately C# semantics assures that this condition never occurs. Indeed the C# language forbids the usage of a `goto` instruction whose target label is contained inside an inner statement (see [?]) or, more formally, the label referenced by a `goto` instruction must be declared in the same scope of that instruction.

- return value: if the input method returns a value (the method's return type is not `void`) and we remove an annotated fragment containing a `return` instruction, then we could possibly[1] generate an invalid method.

  The condition mentioned above could be verified by analyzing the stack behaviour of the generated method. To simplify, our implementation will not perform the check, leaving it to the user.

The next figure shows the main steps of the removal process.

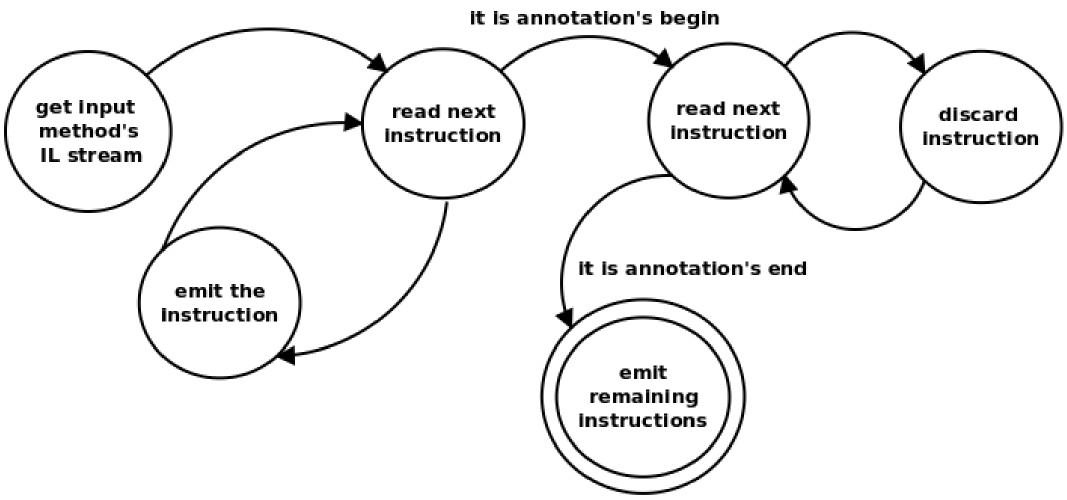


Figure 3.1: Removal flow chart

[1]In fact, the removed fragment could contain all original method's `return` instructions, or for the resulting method there could be some execution flows lacking a `return` instruction.

3.4.2 Extrusion

Extrusion is the operation allowing to transform an annotated fragment in a new method, that is to create, on the fly, a method whose signature and body will be the same of the annotated fragment's ones.

Once we have obtained the annotation's signature and local variables, the extrusion can be carried out by emitting all necessary preliminary IL instructions and by declaring local variables. The *System.Reflection.Emit* namespace contains classes and methods suited to this purpose.

Furthermore, we have to insert instructions performing the storage of the method's arguments value that will be present on the stack, on method invocation, to the respective annotation's free variables. In other words, for each annotation's free variable will be emitted two IL instructions: the first will load an argument's value from the stack and the second will store that value on the the respective method's local variable.

Then all the instructions belonging to the annotated code fragment will be emitted to form the method's body.

Finally, a `ret` instruction must be emitted to conclude the method's body.

$$head_N = \emptyset$$

$$\forall v_i \in free, \quad head_N = head_N \oplus load(arg_i) \oplus store(v_i)$$

$$B(N) = head_N \oplus In_A \oplus ret$$

As for the removal operation, we must be sure to generate a valid method when applying the extrusion operation too. For this reason, the next conditions must be respected:

- consistent variables indexes: the method resulting from the application of the extrusion operation could have a number of local variables that is different from that of the original one. In this case, it would be necessary to adjust variable indexes (to reindex them).

  Our implementation will avoid this problem simply by retaining all the original method's local variables (see also Section**??**).

- correct jump addresses: if we would extrude an annotated fragment containing a branch instruction whose target address is outside the annotated fragment, then we will generate an invalid method.

  Our implementation checks for the existence of this condition and signals it to the user by throwing a runtime exception.

The main steps of the extrusion algorithm are presented on the next figure.

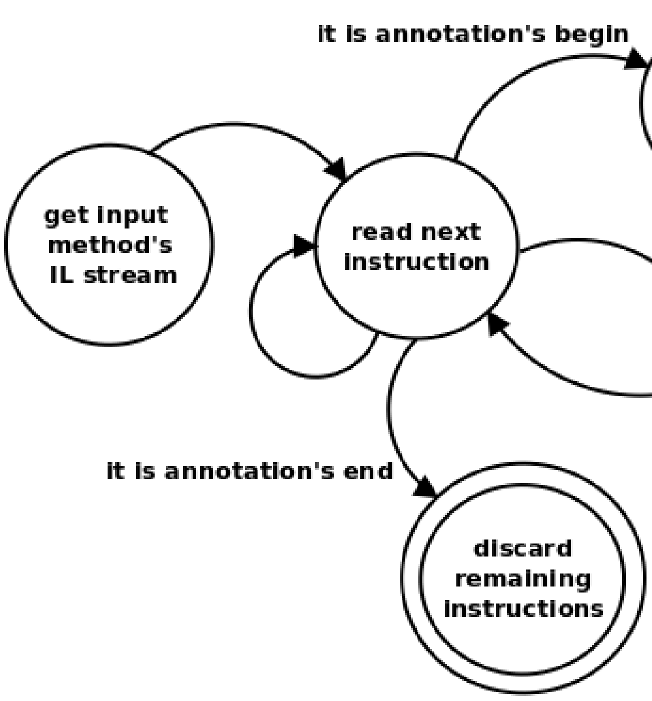


Figure 3.2: Extrusion flow chart

When building a new "Code" object by performing an extrusion operation on any method's annotations, we produce a new method that could return a value, just like any other method.

However the whole thing turns to be a little more complicated because of the next problem.

The ***return value*** problem

It could be possible for an annotated code fragment to behave just like a function[2] if it would be possible for the fragment itself to return a value.

This matter raises a non-trivial question whose solution brings to different choices and further problems.

The problem concerns the return value and it is plain to see that we have at least three alternatives to consider:

1. A variable specified on the annotation signature.

[2]Here *function* denotes the common prorgramming languages' concept of procedure returning a value rather than the more general mathematical notion.

2. What actually the annotation returns, considering real "return" instructions contained inside the annotated fragment, if there are any.

3. No return value (void).

Being the most intuitive choice, we decided to choose the second option, that is to generate - as result of an extrusion operation - a method which returns a value whenever the extruded annotated fragment contains one or more *return* instruction. Among other things, implementing this feature has requested to solve the problem described on Section **??**.

Obviously, such an operation is possible because the annotated fragment's return value is always known: it is equal to the container method's return value.

Anyhow, an annotated fragment containing *return* instructions could lead to the generation of an incorrect method, at least from a semantic point of view. In fact, it would be possible for the user to annotate a portion of code whose execution's flows do not always return a value.

For instance, consider the following method:

```
public static bool harmless() {
  int i = SomeValue();
  [Code] {
    if (i == 0)
      return true;
  }
  return false;
}
```

Extruding the annotation contained in the method above would bring to the generation of an invalid method; in fact, for any values of the generated method's argument different from 0, the method would return no values at all.

Anyway, this detail is left to the programmer, who will be always able to decide whether an annotation can be safely extruded or not.

Furthermore, an automatic tool could retrieve this information from annotation's user-provided attribute data.

3.4.3 Inclusion

The inclusion operation allows to inject (to insert) the IL code produced by any "Code" object[3] *before* or *after* a specified annotation provided that its

[3]Since we are able to build a Code object out of an annotation, we actually allows to include an annotation too. Anyway, the Code object can be obtained otherwise, increasing

signature is compatible with that specified by the annotation itself.

For both options to work, if the *Code* value that we want to include has any parameters, then the "target" annotation must have a compatible signature, otherwise the operation must be aborted notifying the user about the error (see again **??**).

As already said, we must decide where the code will be included and, as a matter of fact, there are only four different places:

- Before annotation's begin: the specified Code's IL code will be inserted starting after the last instruction before the call to the annotation's *begin* placeholder.

- After annotation's begin: the specified Code's IL code will be inserted starting after the call to the annotation's *begin* placeholder.

- Before annotation's end: the specified Code's IL code will be inserted starting after the last instruction before the call to the annotation's *end* placeholder.

- After annotation's end: the specified Code's IL code will be inserted starting after the call to the annotation's *end* placeholder.

Anyway, both the second and the third option ...

There could be another option, that is to include a Code's IL code both before *and* after an annotation at the same time, but this could be accomplished otherwise by mean of annotations composition (see Section **??**).

Of course, the implementation must provide the user - and actually does it - with a way to choose one of the remaining two alternatives.

On the next figure a flow diagram of a simplified inclusion algorithm is shown.

Local variables belonging to the included code fragment must become input method's local variables, therefore they must be added to the list of the input method's local variables. Obviously, in this way their index will change, thus it will be necessary to update the parameter of any *load* or *store* instructions, belonging to the included code fragment, with a new correct index.

To assure inclusion operation's correctness we must also check the compatibility of the included Code's signature and the annotation's signature. If they are not compatibles, then we have to abort the operation and notify the user by throwing an exception.

inclusion genericity and power.

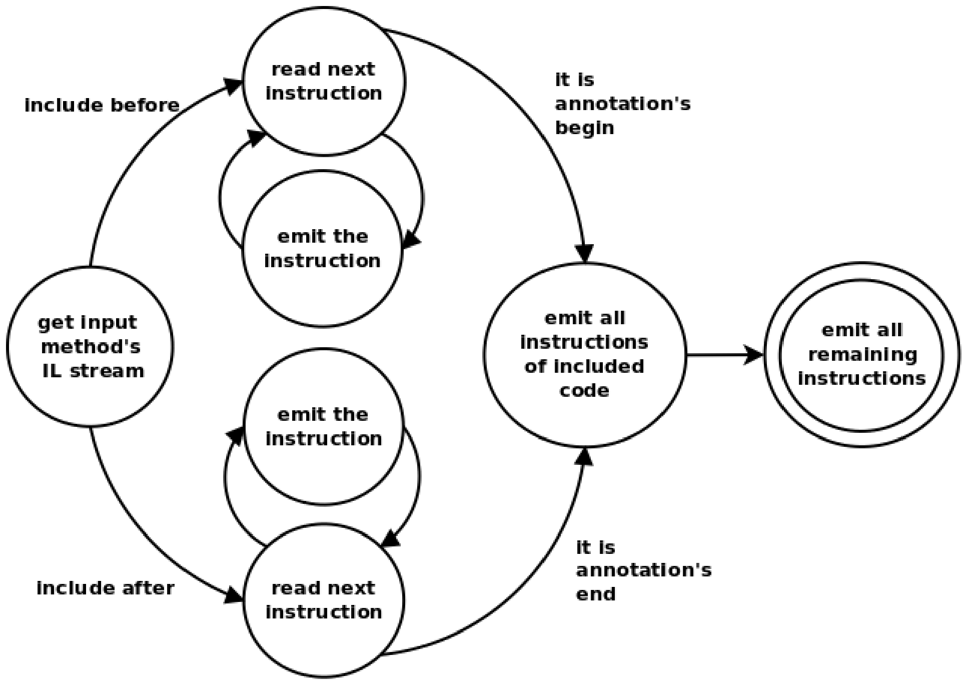


Figure 3.3: Inclusion flow chart

For the two signatures to be compatible they must have the same number of arguments and each argument's type of a signature must be compatible with the respective other's one.

Nevertheless, granted the validity of the Code object's IL code that is being included, the method built by the inclusion operation will always be valid too.

For simplicity, we analyze the type compatibility problem related to the signature's return value only whereas the real implementation will handle the whole signature. Furthermore, the solution described applies to both issues.

Types compatibility and types conversion

As already said above, we have to check type-compatibility of annotation's and inclusion's signature. We must face the same problem when we are going to include a method which returns a value, because we should specify which variables to assign the return value itself. Either situations can lead to a type conflict or, better said, a type compatibility problem. For example, let us suppose the user has specified a variable whose type is `float` to hold the included method's return value whose type is `string`: this is an user's mistake and we have to notify him about it.

But there are more subtle cases that should not be considered user's mistakes. Indeed, if in the previous example the returned value's type was `int` instead of `string` then there were no compatibility problems because an `int` value can be directly assigned to a `float` variable.

We must recognize situations like these and handle them accordingly; when necessary, the implementation will generate code realizing any necessary type conversion. For a more detailed discussion about types and conversions see Chapter 8, Part I of [?].

3.4.4 Copy

The copy operation permits to copy (to duplicate) an annotated code fragment of a method *M* before or after another annotated code fragment of the same method *M*.

For what concerns the implementation of this operation, it is very similar to that of the inclusion, though more simple since there is no need to check for equal numbers of variables and compatibility of types (compatible signature). Indeed, being the annotation belonging to the same method, we can safely duplicate the code fragment without any particular care. As always, of course, the semantic of the generated code is leaved to the programmer.

An example of use of the copy operation can be found in **??**.

On the next figure a flow diagram of a simplified copy algorithm is shown.

3.5 Possible optimizations

An effective improvement could be obtained by removing all calls to annotations' boundaries placeholders (*Annotation.Begin()* and *Annotation.End()*) at the cost of disallowing any feasible successive operations on the resulting method because, obviously, we would lose any information about annotations' boundaries.

For such a reason, in the implementation we have decided to discard only the placeholders belonging to the particular annotation interested by an operation, conserving calls to placeholders belonging to any other annotations. Besides, this is an implementation detail and it will be also feasible to let the user decide whether to remove or to keep the placeholders.

For the *removal* operations there is another chance to perform an optimization regarding emitted IL code.

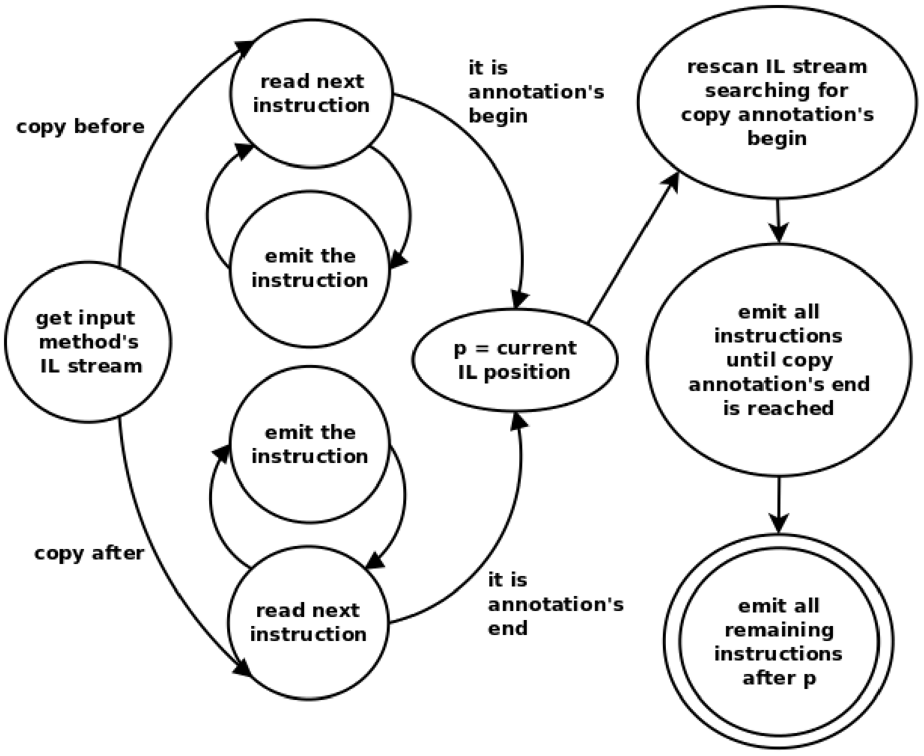

copy before
read next instruction
it is annotation's begin
rescan IL stream searching for copy annotation's begin
emit the instruction
get input method's IL stream
p = current IL position
emit all instructions until copy annotation's end is reached
emit the instruction
copy after
it is annotation's end
read next instruction
emit all remaining instructions after p

Figure 3.4: Copy flow chart

Indeed, removal implementation could be refined to discard unused local variables, if there are any, to produce an optimized IL code; actually, it could be possible for one or more method's variables to be referenced inside an annotated fragment only, therefore a method resulting from the removal of such a fragment would not need anymore that variables, so they could be discarded safely.

Anyway, there would not be any execution speed increase, rather we would get just a slightly smaller code size. Concerning the implementation, we do not remove useless local variables.

3.6 Composing annotations

One of the most remarkable results of our work is a chance to *compose* annotations that is applying an arbitrary numbers of consecutive operations on an annotated method.

For instance, we could be interested to realize a transformation such as the following:

$$methodB = Extrude_X(PreInclude_Y(Remove_Z(methodA)));$$

Of course, the semantics of the final method resulting from a composition of operations like the above will depend by their order.

Actually, our implementation will be capable of performing this kind of transformation, allowing the user to specify a list of operations to be carried out sequentially on the same method.

However - just to make things easier - our implementation will not allow all feasible combination of operations; more precisely, though the user will be allowed to specify more than one *extrusion* operation, only the first one of them in the order will be actually realized. Furthermore -excepted for the *copy* operation- the order of the annotations involved in an operation must be increasing; more formally, called *op* an operation and *an* an annotation:

$$valid\_operations\_list = \{op_0(an_i), \ldots, op_{n-1}(an_j), op_n(an_k)\} \mid i \leq j \leq k$$

This last constraint is due only to implementation choices and could be dropped in future.

Composition could be effectively used as a way to realize the weaving (see Section **??**) step of the compiler of an aspect oriented programming language.

Nevertheless, we think that annotation composition is a very interesting subject deserving further research and we will talk again about this argument on Section **??**.

3.7 Transforming a fragment in a method

Although we have chosen CodeBricks as a mean to create a method from a compositions of annotated code fragments, there can be other ways to accomplish the same task.

As a proof of concept, now we want to outline such an alternative approach which is based on lambda expressions.

We have shown on **??** that a lambda expression is actually an anonymous method contained in a compiler-generated class; the purpose of the class is to hold all the variables of the calling method (i.e. the method where the lambda expression has been defined) that are shared with the anonymous method (i.e. the lambda expression).

This same strategy is well suited for our goal too; indeed, taking advantage of a library such as Cecil or maybe CodeBricks again, we could bake at runtime a class containing:

- the method resulting from the compositions of annotated code fragments, as an instance method
- each variables that is not a local of a code fragment, as a class' field (variable lifting)

Then, to call the generated method, we could mimic the .NET and Mono implementation which produce a class derived from *System.MulticastDelegate* which is responsible to actually invoke the generated method.

System.MulticastDelegate is a special class that can be derived only by the compiler or other IL-level tools, just like ours.

This solution would be quite elegant and would also remove the current limitation to static methods of CodeBricks.

Unfortunately, there would be a little complication due to the need for some native code (platform dependent code) mainly dealing with class instances and method pointers. Hence, in the interest of maximum portability between different implementations of the runtime, and on different architectures, we have opted not to pursue this approach.

Chapter 4

Implementation

“The greatest way to live with honor in this world is to be what we pretend to be.”
Socrates

4.1 Target framework and platform

Both [a]C# and CodeBricks have been developed using the .NET framework which is available on the Microsoft Windows platform only.

On the other hand, for any programming language or framework, in particular a standardized one, it should be a fundamental feature to be as much independent as possible from any vendor-specific platform or operating system.

For such a reason, we would have preferred to concretize our work on a environment different from the Windows one.

Fortunately, in the last few years, the Mono project brought an implementation of the ECMA-335 standard on many other platforms, mainly UNIX-like systems.

The most famous and widespread UNIX-like operating system is Linux, which comes in a plethora of -often slightly- different distributions[1].

Therefore, we have chosen to develop an implementation of this thesis' idea using Mono in a Linux environment.

Now, let us spend some more words about this choice.

[1]A Linux distribution is basically a Linux kernel together with an amount of programs just like those forming the graphical user interface (Gnome, KDE, ICEwm and so on) and the most common system utilities. Different distributions differ mainly on their own kernel configuration details and bundled software packages.

4.1.1 About Open Source

In the last few years the so-called "Open Source" community has become wider and wider by an enthusiastic crowd of people which is constantly contributing to the design and development of a big variety of software, whose complexity ranges from that of a simple text editor to that of a huge application server, such as the Apache Web Server[2].

It could be quite easy to find a number of major drawbacks related to the kind of development process mentioned above.

However, we would like to focus attention instead on what -as a matter of fact- the open source may have revealed to have a significant impact on future software development:

- sharing ideas is a great chance to get improvements and stability faster whereas hiding ideas (closed source) could be difficult if not quite impossible.

- although design and planning are two undoubtable crucial things, it is often highly preferable to have a not optimal and still incomplete but working code than just nothing.

- professional developers deeply appreciate to have more control on what is running on a computer. Also normal users prefer to choose among different softwares of the same typology, rather than just being coerced to buy the only available one, even when it is a good one.

According to these considerations, together with an interest for testing new promising technologies, we have decided to work on open source platforms and tools.

A summary of the main open source softwares that have been used throughout this thesis is shown in table **??**.

4.1.2 Motivation

There are mainly two reasons that drove us to choose *Linux* [**?**] as implementation platform and consequently to work on porting CodeBricks to Mono:

- Giving a real proof about platform independence and generality of the concepts involved in this thesis.

[2]The Apache HTTP Server Project is an open-source HTTP server for modern operating systems including UNIX and Windows NT. Apache has been the most popular web server on the Internet since April 1996

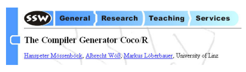
Mono	A free open source *Unix* implementation of the *.NET* framework	
MonoDevelop	An on-going effort to produce a complete IDE for Mono and Gnome	
Coco	A powerful and easy to use compiler compiler	 SSW General Research Teaching Services The Compiler Generator Coco/R Hanspeter Mössenböck, Albrecht Wöß, Markus Löberbauer, University of Linz
Gentoo Linux	A special flavour of the Linux operating system tuned on performance	
Gnome	A free, usable, stable, accessible desktop environments for the Unix-like family of operating systems	

Table 4.1: Exploited Open Source software

- Abstracting as much as possible from language and operating system specific issues in order to let us focus our attention on all possible applications of this work.

Interoperability and customization are two relevant aspects of modern software design -impacting on both quality and efficiency- where metaprogramming can be used effectively. So if our strategy were limited to a specific language or architecture, it would lose much of its effectiveness and applicability.

Despite using a dialect of C# to develop our idea, we have to point out that it has been an arbitrary choice based on opportunity reasons such as simplicity, popularity and widespread of that language, thus the main concept discussed on this thesis is independent from any particular language or platform. Actually, we could have used any other .NET language, granted that the language supports annotations, without needing to introduce any significant design differences. Moreover, as an extreme example, it could even be feasible to design a C compiler supporting annotations and a library performing annotation-based code compositions.

Besides, last but not least from a technical point of view, we had a good chance to experiment with a real open source GNU[3] implementation of the ECMA-335 standard and test its real usability on a non trivial case study.

4.2 Porting CodeBricks to Linux

Here we like to show the steps that have been necessary to accomplish the porting of CodeBricks on the Linux/Mono platform.

CodeBricks, though written in C#, has been developed on the .NET framework which is currently available on the Microsoft Windows platform only.

Moreover, CodeBricks' implementation takes advantage of Windows-specific operating system's memory mapping functionalities to read and to modify both assemblies' code and data. For such a reason, we had to write a small amount of new C# code to provide the same functionalities on the Linux platform too; anyway, after we started this thesis, the Mono project provided these low-level functionalities in their standard APIs. This work is described in the next sections and it is all that is needed to allow the compilation of the CodeBricks library on the Linux/Mono platform.

[3]GNU is a so-called "recursive acronym" standing for "GNU is Not Unix"; GNU is the name of the most common and used free software license in the world.

Furthermore, we fixed some minor bugs which were related to IL code analysis and generation.

4.2.1 CLIFileRW

As said above, CodeBricks needs to read assembly's content primarily in order to retrieve methods' IL code streams as well as their associated metadata.

.NET (and Mono) assemblies are stored on a PE[4] [?] format file. Decoding information stored on a PE format is quite complex but, fortunately, we can rely on an tool already available to accomplish the task that is *CLIFileRW*.

CLIFileRW [?] is a library written in C# capable of reading the raw format of a binary CLI file which is a collection of compiled types.

As for fast access to binary data is a key requirement, especially for what concerns CodeBricks' purposes, it uses the MapView (see paragrah **??**) class to obtain a memory mapping of the file, this way enabling top-speed scanning of IL instructions streams.

Besides adding conditional directives to support correct CLIFileRW compilation on Mono, we also fixed two small bugs located in method "ILInstruction.Normalize"; one was related to the lack of the `stloc_S` instruction's handling, simply consisting of a lack of trivial code. The other was due to a detail of reflection implementation (*System.Reflection* namespace): any method's reference is represented by an instance of the *MethodInfo* class, with the only exception of constructors which are represented by an instance of the *ConstructorInfo* class; the latter case was simply missing.

We have been pleased noting that also Mono provides a library, although it is no more actively maintained (latest release on October 2007), named *Cecil* which is thought to enable generation and inspection of programs and libraries in the ECMA CIL format. With Cecil, it is possible to load existing managed assemblies, browse all the contained types, modify them on the fly and save back to the disk the modified assembly. Cecil has support for extracting the CIL bytecode and does not need to load the assembly or have compatible assemblies to introspect the images.

Unfortunately, at the time this thesis is written, Cecil is at an early stage of development thus it is still incomplete.

Furthermore, because of the tight interaction between CodeBricks and CLIFileRW, we preferred to port CLIFileRW on Linux/Mono too, this way avoiding a great amount of work on CodeBricks to adapt it to use Cecil.

[4]Portable Executable, a executable file format, introduced by Microsoft WindowsNT operating system, which is intended to be architecture-independent.

4.2.2 Unix Memory Mapping

Current CodeBricks implementation uses the CLIFileRW library which, in turn, has a memory mapped approach for reading of CLI files' binary data format.

Memory mapping is a well-known technique to achieve best performance when it is needed to read an arbitrary size of contiguous data such as database table or a big matrix, a task that usually involves many sequential or random access to disk which would be, obviously, very expensive in terms of time, therefore greatly degrading performances.

Memory mapping is sometimes used also to realize Interprocess Communication (IPC), being possible for a memory mapped area to be shared among different concurrent processes.

The POSIX[5] standard, part of which are implemented on the Linux operating system, offers the `mmap()` system call which permits to map a device or a file (either the whole file or a portion of it) into memory; the file is mapped in multiples of the operating system's page size that, in turn, is obtained through the `getpagesize()` system call.

The `munmap` system call, the natural companion of `mmap`, releases the memory mapped area as well as any other OS' resources allocated to support this kind of service.

As we will see next, in order to be able to perform a few minimal tests and debugging, it has been useful to have the `perror()` system call too, that prints a message describing an eventual error encountered during the last call to a system or library function.

An example of how memory mapping works it is provided by the C listing below

Listing 4.1: Unix Memory Mapping example

```c
#include <syscall.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <strings.h>

#define FILE_PATH   "./test.txt"
#define TEST_STRING "hello world!"
```

[5]POSIX (http://www.opengroup.org/onlinepubs/9699919799/) is a set of related standards specified by the IEEE

```
11
12 int main()
13 {
14   int pagesize = getpagesize();
15   printf("Page size is %i\n", pagesize);
16   int fd = open(FILE_PATH, O_RDWR);
17   int length = pagesize * 1;
18   void* mmf = mmap(0, length, PROT_READ | PROT_WRITE, MAP_SHARED
        , fd, 0);
19   perror(NULL);
20   close(fd);
21   memcpy(mmf, TEST_STRING, strlen(TEST_STRING));
22   msync(0, length, MS_SYNC);
23   munmap(mmf, length);
24   return 0;
25 }
```

A pointer to a text file is retrieved and then used to get a memory mapping of the file's content. After closing the file and disposing the file pointer, mapped memory is directly overwritten with a simple string.

Finally, mapped memory is released and the actual file's content is modified as expected.

Every system calls cited above are used:

- Line 14: we get the system dependent memory mapping unit size
- Line 16: retrieving file's descriptor with write access using open
- Line 18: we ask the system to map the first page of the file's content calling mmap
- Line 19: *perror* is invoked to check for any errors
- Line 20: closing file with close
- Line 21: overwriting memory (file's content) using memcpy
- Line 22: with *msync* the current memory content is writed back on actual file
- Line 23: munmap to release the memory mapped area

4.2.3 PInvoke

The *platform invoke* mechanism is used in VES when it is necessary to execute native or, generally speaking, any compiled code whose source it is not available for direct compilation.

The .NET (and Mono of course) framework provides the *Platform Invoke* (or *PInvoke*) features with the `DllImport` attribute to allow calling functions packaged inside DLLs.

This attribute, which takes the name of the DLL as its first argument, is placed before a function declaration for each DLL entry point that will be used.

The signature of the function must match the name of a function exported by the DLL but we can perform some (correct) implicit type conversion by defining the `DllImport` declarations in terms of “our” language types.

The result is a managed entry point for each native DLL function that contains the necessary transition code (that is called *thunk*) and data conversions.

We can then call the external functions defined into the DLL through these entry points.

The reserved keyword `unsafe` is required because invoking an external function causes the virtual machine to execute native code that cannot be verified (unmanaged).

All of this work was necessary because at the time we started to port CodeBricks to Linux these features were still not directly available through Mono’s classes; not surprisingly, now Mono exposes these functionalities, as many other Unix services, in the `Mono.Unix` and `Mono.Unix.Native` namespaces.

4.2.4 MapView

The MapView class provides access to a memory mapped file. It contains a number of functions to speed up both reading and generation of IL code. The original version was able to work on the *Win32*[6] platform only, because it took advantage of native Win32 memory mapping functions. To make it usable on the Unix platform too, we have done some little adjustments to introduce Unix memory mapping support shown in the previous Sections.

Cross platform compilation and compatibility has been obtained making use of standard preprocessor directives such as `#IF`, `#ELIF` and `#ENDIF` check-

[6]Win32 is a common name standing for the common API used in Microsoft’s operating systems architecture.

ing for a symbol defined at compile-time. It is possible to see their usage on listing **??**.

There is a slight difference between Win32 and Linux memory mapping implementation; on Win32 we must use two system calls, *CreateFileMapping()* and *MapViewOfFile()*. The former creates or opens a named or unnamed file-mapping object for the specified file, whereas the latter actually maps a view of a file into the address space of the calling process.

On the other hand, on Linux a call to *mmap()* only is requested to map a file in memory.

Of course, also for releasing resources when memory mapping is no more needed, the same difference holds: a call to *UnMapViewOfFile()* and then a call to *CloseHandle()* on Win32 whereas *munmap()* only is enough on Linux.

4.2.5 Putting it together

The libraries containing all the functions mentioned above are:

- *libm.so*[7] for `mmap()`, `munmap()`
- *libc.so* for `perror()`, `getpagesize()`

In the end, we have written the following code:

Listing 4.2: PInvoke: using external "native" functions

```
[DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
public static unsafe extern int getpagesize();

[DllImport("libm", CharSet=CharSet.Auto, SetLastError=true)]
public static unsafe extern IntPtr mmap(IntPtr start, int length
    ,
                                          int prot, int flags,
                                          int fd, int offset);

[DllImport("libm", CharSet=CharSet.Auto, SetLastError=true)]
public static unsafe extern int munmap(IntPtr start, int length)
    ;

[DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
public static unsafe extern int perror(IntPtr s);
```

[7]On the Linux operating system, dynamic linked libraries ("DLL") are called libraries and the extension for the related binary files is ".so" (*s*hared *o*bject) instead of ".dll"; the binary format of these files is ELF [**?**]

Of course, it is a very small chunk of code besides it is crucial to actually build a working port of CodeBricks on Mono.

4.3 Extending [a]C#

4.3.1 Our Goal

At this time, we have prepared all the necessary base components to start describing the main task which is, in a few words, enabling the programmer to perform -at runtime- some operations on a method based on the knowledge of its internal annotations only.

At first, we tried to get a minimal running implementation of the idea, leaving any feasible optimization to a successive moment. This initial version has served as a test bench to find errors and possible refinements.

First of all, we tried to imagine the easiest way to store, at IL level, informations about bindings specified by the user at language level: it turned out to be a further use of “dummy” methods (placeholders) with an ad-hoc signature.

Initially, we adopted two different methods:

- A placeholder to keep trace of bounded variables.
- A placeholder to retrieve which variable has to hold an inclusion’s return value.

The first working implementation was actually based upon these two methods even if, though being effective, it needed to be refined.

Once obtained an initial working version of the code, we started to think about possible enhancements regarding two different questions:

- Can we avoid to insert two different placeholders ?
- Is there any way to optimize emitted IL code ?

The answer is positive for both questions even though, especially for the latter issue, we have been always more focused on effectiveness rather than on performance, so the optimizations will be far from being definitive.

4.3.2 Modifying language grammar

The original [a]C# compiler has been implemented writing an *attributed grammar*[8] to be parsed by *Coco/R* [**?**].

[8]An attribute grammar is a collection of productions whose right sides are enriched with explicit actions written in a general purpose language such as C or Java.

Coco/R is a compiler generator (a compiler compiler), which takes an attributed grammar (from now on *ATG*) of a source language and generates a scanner and a parser for this language.

The scanner works as a deterministic finite automaton.

The parser uses recursive descent and $LL(1)$ conflicts can be resolved by a multi-symbol lookahead or by semantic checks, thus the class of accepted grammars is $LL(k)$ for an arbitrary k.

A main change to the grammar was necessary to allow bindings specification on an annotation declaration.

Writing an ATG is a quite frustrating task because the unavoidable mixing of grammar production and C# code quickly turns to be hardly understandable. Moreover, we were forced to keep an eye focused on backward compatibility because, obviously, we did not want to break validity of any existing [a]C# code.

We chose the following convention for signature specification:

- an asterisk mark (*) before each variable.
- an ampersand mark (&) to designate the variable used to hold an inclusion's return value.

Here's an example

Listing 4.3: Annotation's signature specification

```
[Code("normal argument", *a, *b, &c)] {
  // some code...
}
```

Another modification will be described in full details on Section **??**.

To view the complete [a]C# ATG including all the changes that have been introduced, see listing [**??**] in the appendix.

4.3.3 Annotations inside lambda expressions

In Section **??**, we have introduced the lambda expression syntax and we have claimed the feasibility of supporting annotation declarations inside lambda's body. Actually, there would not be any particular difficulties in modifying the C# 3.0 ATG accordingly. Not even the lambda's bytecode parsing to retrieve annotations would be very much complicated, since as we have seen in Section **??**, we are still dealing with normal methods; the only real issue could likely be related to a particular compiler's (Mono's, .NET's, etc.) implementation of things such as name mangling or hidden class details.

Rather, we simply argue that in most cases annotating a lambda expression would not be very useful since:

- by their very nature, lambda expressions tend to be very short and concise whilst annotations can be used profitably especially in long and complex methods.
- lambda expressions -in C# 3.0, at least- are only a programming style so we would not be adding anything really different or new to this thesis' results.
- handling all conceivable forms of code annotation in a C# code is beyond our scope.

For these reasons, we did not implemented neither the syntactic nor the runtime support to annotations inside lambda expressions.

4.3.4 Extending language runtime support

After modifying language grammar, it is indispensable to ensure that language runtime will correctly support all the new features that have been introduced.

Consequently, there are two different things to deal with:

- To enable the runtime to infer annotation's free variables (or annotation's *signature*) as well as annotation's local variables.
- To provide any necessary enhancements for supporting the interactions with CodeBricks that will be needed to allow constructing bricks from annotations.

4.3.5 Retrieving annotated fragment signature

The first indispensable thing to do is inferring the annotated fragment signature: we must know what are free and local variables (see Section **??**) used within the fragment.

As seen, for what concerns an annotation, there are two distinct cases to analyze:

- variables declared outside the annotated fragment that are visible and accessed inside it, together with any method's arguments referred inside the fragment: annotation's free variables.

- variables declared inside the annotated fragment: annotation's local variables.

To find out which of the two sets an annotation fragment's variable belongs to, we have to describe how method's variables are defined at IL level.

When a method's source code is compiled, information about where a variable has been declared is lost, being not, by any means, relevant for compiled code execution. Instead, the compiled method is composed of a preamble where, among other things, there is a list of pairs *(index, datatype)* and each of them corresponds to one of method's local variables.

Method's arguments can be located looking at the instructions that load their values on the stack: `ldarg`.

Therefore we must rely on an analysis of any *load* and *store* instructions (such as `ldloc` or `stloc`) to get back the information about variables declaration. Indeed, the following is a property we can exploit during the analysis.

Using the notation of Section **??**:

$$v \in locals \leftrightarrow \exists store_v \in In_A \wedge \neg(\exists load_v \in Pre_A \vee \exists store_v \in Pre_A) \quad (4.1)$$

$$v \in free \leftrightarrow v \notin locals \vee (v \in args \wedge \exists ldarg_v \in In_A) \quad (4.2)$$

Using this property, we can write a program to retrieve both local and free annotation's variables by scanning a method's IL code.

See the next (quite silly) example

Listing 4.4: Annotation's variables

```
public static double GoldenRatio() {
  double g = 0;
  [Code] {
    int a = 1;
    int b = 5;
    int c = 2;
    g = (a + Math.Sqrt(b)) / c;
  }
  return g;
}
```

and relative IL code; we remind that obviously it is not the real assembly content but just a more human-readable form, produced by the disassembler (*monodis.exe*), of the actual IL's bytes stream.

GoldenRatio's IL

```
.maxstack 5
.locals init (
        float64        V_0,
        int32        V_1,
        int32        V_2,
        int32        V_3)
IL_0000: ldc.r8 0.
IL_0009: stloc.0
IL_000a: ldc.i4.0
IL_000b: call void class ACS.Annotation::Begin(int32)
IL_0010: ldc.i4.1
IL_0011: stloc.1
IL_0012: ldc.i4.5
IL_0013: stloc.2
IL_0014: ldc.i4.2
IL_0015: stloc.3
IL_0016: ldloc.1
IL_0017: conv.r8
IL_0018: ldloc.2
IL_0019: conv.r8
IL_001a: call float64 class System.Math::Sqrt(float64)
IL_001f: add
IL_0020: ldloc.3
IL_0021: conv.r8
IL_0022: div
IL_0023: stloc.0
IL_0024: ldc.i4.0
IL_0025: call void class ACS.Annotation::End(int32)
IL_002a: ldloc.0
IL_002b: ret
```

Reading the IL, instruction labelled *IL_0011* (`stloc.1`) is the first *store* instruction on variable `V_1` and that instruction is placed *after* the annotation's begin placeholder (*IL_000b*) so we can deduce that *V_1* is an annotation's local variable whilst variable `V_0` is an annotation's free variable because there is one *store* instruction *before* the annotation's begin placeholder (*IL_0009*, `stloc.0`).

See method *AnnotationTree.GetAnnotationSignature()* (**??**) for actual implementation of the analysis.

The next picture shows basic steps of the algorithm that has been implemented.

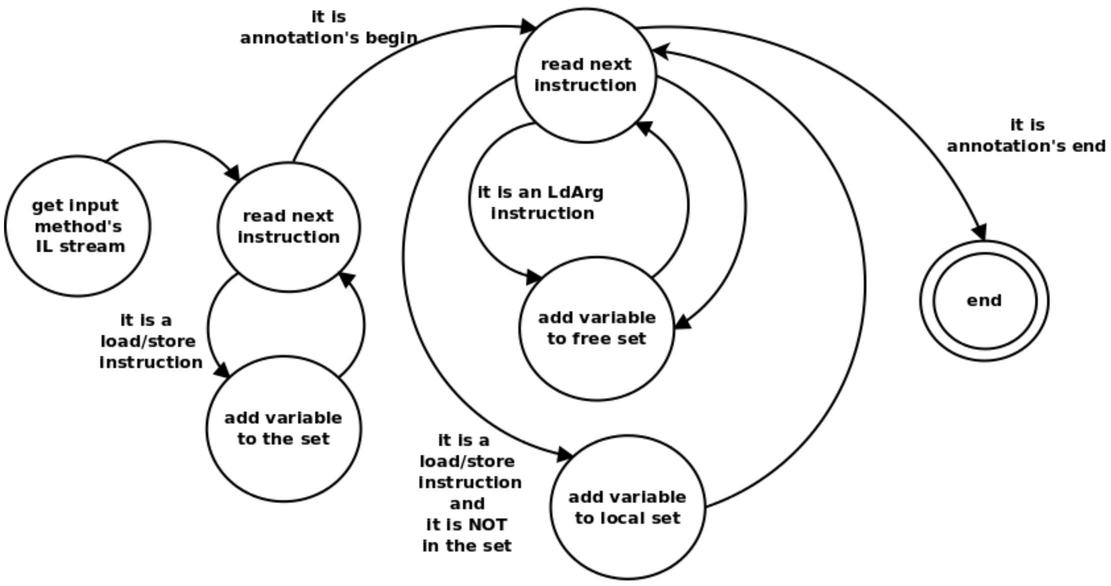

it is annotation's begin
read next instruction
it is annotation's end
get input method's IL stream
read next instruction
it is an LdArg instruction
it is a load/store instruction
add variable to free set
end
add variable to the set
it is a load/store instruction and it is NOT in the set
add variable to local set

Figure 4.1: Annotation's signature retrieval

4.3.6 Retrieving the annotated fragment

We have defined an override of *Annotation.Begin()* which has a richer signature intended to solve the problems described on Section **??**:

```
public static void Begin(object[] p, object r, int idx) {}
```

The first parameter is used to hold the annotation's bindings. The [a]C# compiler collects variables named on the signature specification and uses them as the elements of the array. This way, a static analysis of the final IL code, generated by the C# compiler, will be able to retrieve, at runtime, the indexes of the mentioned variables. In fact, we can look at the *LdLoc* instructions placed between the array creation instructions and the call to *Annotation.Begin()*; the arguments of these *LdLoc* instructions are the indexes we are searching for.

The second argument - r - will hold annotation's return variable.

The last argument - *idx* - is the same of the original *Annotation.Begin(int32)* that is the annotation's index.

Of course, a call to this new version of *Annotation.Begin()* will be inserted by [a]C# compiler only when a signature has been actually specified, otherwise the compiler will insert the old *Annotation.Begin(int32)* call.

Now we provide an example of the whole process, starting from an [a]C# source code to the final IL code.

Here is an hypothetic method containing an [a]C# annotation:

Listing 4.5: The new *Begin()*

```
1  public static double e(int prec) {
2    int n = prec;
3    double r = 0;
4    [Code(*n, &r)] {
5      double i = 1 + (1 / n);
6      r = Math.Pow(i, n);
7    }
8    return r;
9  }
```

On line 3 we have specified a binding for the annotated fragment's n variable.

The following is the previous method translated in standard C#:

Listing 4.6: New *Begin* at work

```
1  [Code (ACSIndex=0)]
2  public static double e(int prec) {
3    int n = prec;
4    double r = 0;
5    ACS.Annotation.BeginMeta();
6    ACS.Annotation.Begin(new object[]{n}, r, 0);
7    {
8      double i = 1+(1/n);
9      r=Math.Pow(i, n);
10   }
11   ACS.Annotation.End(0);
12   return r;
13 }
```

Lines 5 and 6 are responsible for producing "metadata" IL code: the first argument of *Annotation.Begin()* is an array of one element and that element is the n variable.

Next box shows IL code that would be produced for the method above by the C# compiler:

Begin in IL

```
.maxstack 6
.locals init (
        int32        V_0,
        float64        V_1,
        float64        V_2)
IL_0000: ldarg.0
IL_0001: stloc.0
IL_0002: ldc.r8 0.
IL_000b: stloc.1
IL_000c: call void class ACS.Annotation::BeginMeta()
IL_0011: ldc.i4.1
IL_0012: newarr [mscorlib]System.Object
IL_0017: dup
IL_0018: ldc.i4.0
IL_0019: ldloc.0
IL_001a: box [mscorlib]System.Int32
IL_001f: stelem.ref
IL_0020: ldloc.1
IL_0021: box [mscorlib]System.Double
IL_0026: ldc.i4.0
IL_0027: call void class ACS.Annotation::Begin(object[], object, int32)
IL_002c: ldc.i4.1
IL_002d: ldc.i4.1
IL_002e: ldloc.0
IL_002f: div
IL_0030: add
IL_0031: conv.r8
IL_0032: stloc.2
IL_0033: ldloc.2
IL_0034: ldloc.0
IL_0035: conv.r8
IL_0036: call float64 class System.Math::Pow(float64, float64)
IL_003b: stloc.1
IL_003c: ldc.i4.0
IL_003d: call void class ACS.Annotation::End(int32)
IL_0042: ldloc.1
IL_0043: ret
```

All instructions from IL_000c to IL_0027 are the metadata we are interested on. More precisely, the analysis of this portion of code will discover the index of variable n, which has been specified on annotation's signature ($*n$), by reading instruction IL_0019's argument (*ldloc*) which is 0.

Instead, the index corresponding to the variable r, specified as return (&r), is retrieved by reading the argument's value (1) of the last *ldloc* instruction before the *Annotation.Begin()* invocation (IL_0020).

Annotation.BeginMeta()

If the fragment's annotation declaration includes any binding specification, we need to store information about the binding. At this purpose we introduce another method - *Annotation.BeginMeta()* - call to delimit the portion of IL code "holding" this kind of data.

Indeed, as suggested by its name, the *Annotation.BeginMeta()* method is a placeholder used to mark the starting of an IL code section that we can consider to be metadata. The ending of this portion of code will be marked by *Annotation.Begin()*. For such a reason, our version of the [a]C# compiler produces a call to *Annotation.BeginMeta()* just before the call to *Annotation.Begin()*.

This way, when analyzing IL code resulting from compilation of an [a]C# source, we will be able to easily distinguish among "real" IL code and the IL code that will be generated to load on the stack *Annotation.Begin()*'s arguments, as shown on the preceding Section.

Undoubtfully, we could avoid to use this marker, relaying on different information to locate this "metadata" portion of IL code, but such a strategy would require multiple scans of IL (together with an even more complex implementation of *Code.FillMethodBody()*, see Section **??**), slowing down analysis and code generation. This is the reason why we preferred to introduce this new placeholder.

4.3.7 Patching AnnotationTree.GetCustomAttributes()

We implemented a conservative extension of the *AnnotationTree.GetCustomAttributes()* method to make it able to retrieve correct annotation trees of methods' body that contains annotated fragments which have been written using both the original and new [a]C# grammar.

The first step was to replace the following line

```
if (param.Equals(BEGIN)) {
```

with this

```
if (param.Equals(BEGIN) || param.Equals(BEGINEX)) {
```

this way allowing *AnnotationTree.GetCustomAttributes()* to recognize the new annotation placeholder too.

The `BEGIN` and `BEGINEX` are two constants holding a reference to the original *Annotation.Begin(int32)* and the new *Annotation.Begin(object[], object, int32)* respectively.

The *or* condition was necessary because, though having the same name, the two methods are completely different so they have different references too.

As an intermediate step, we have to show a problem related to the [a]C# source-to-source compiler that we have discovered while developing this thesis. Fixing of this bug has required another much more consistent update to *Annotation.GetCustomAttributes()* which is described on the next Section.

4.3.8 The unreachable code problem

After having described what modifications were needed to apply to the [a]C# parser, we want to show a small though subtle [a]C# bug that was, as any other respectable one, a little hard to identify.

Let us consider the case one wants to write a method such as the following that, by the way, it is not so unlikely:

Listing 4.7: [a]C# bug

```
public bool check(int i) {
  [Code] {
    if (i > 0)
      return true;
    else
      return false;
  }
}
```

Compiling it, [a]C# correctly produces the method below

Listing 4.8: [a]C# bug fixed

```
public bool check (int i) {
  ACS.Annotation.Begin(0);
  {
    if (i>0)
      return true;
    else
      return false;
  }
  ACS.Annotation.End(0);
}
```

This as simple as harmless method will be affected by an [a]C# bug; it is possible to discover it -as we did actually- just looking to the IL code generated by the C# compiler for the *check(int32)* method.

We have to remind that [a]C# is a source-to-source compiler which relies on the standard C# compiler to produce final IL code.

```
IL code for ''check''
IL_0000:  ldc.i4.0
IL_0001:  call void class ACS.Annotation::Begin(int32)
IL_0006:  ldarg.1
IL_0007:  ldc.i4.0
IL_0008:  ble IL_000f
IL_000d:  ldc.i4.1
IL_000e:  ret
IL_000f:  ldc.i4.0
IL_0010:  ret
```

It is immediately evident the lack of the placeholder method marking annotation's end.

The reason why the *ACS.Annotation.End(int32)* call is missing appears to be quite simple: the effect shown is due to a C# compiler trivial optimization known as dead code elimination[9].

Actually, when compiling the method above (listing **??**), the C# compiler prints a warning message notifying the presence of "unreachable" code, that is to say a portion of code that a static analisys has recognized as never executable. Indeed, as one can see, either branches of the `if` construct bring to a `return` instruction which causes an immediate jump out of the method thus avoiding the last *Annotation.End(0)* call to be ever reached by the execution flow. This is the reason why the compiler does not emit any IL instructions corresponding to that last portion of code; therefore it reduces IL code size simply discarding the unreachable code. Unfortunately, doing so, it wastes the correct functioning of [a]C# runtime support which is no more able to find the annotation's end marker thus it becomes impossible for it to correctly recover the method's annotations tree; in a few words, in such a situation, the [a]C# runtime fails when asked to retrieve the method's annotation tree.

[9]Dead code elimination is an optimization technique consisting on the removal of any piece of code that can never be executed.

Although this bug is related to [a]C# only, implementing the feature described on Section **??** required to fix the bug.

Furthermore, the particular solution of this problem that is provided by this thesis does not involve any change to the algorithms presented on Sections **??**, **??**, **??** and **??**.

Being too complex to check for unreachable code sections originated by *return* instructions directly from the ATG, we chose an alternative solution, based on adding "preventing" calls to a special placeholder method in the generated C# source wherever they are needed.

In fact, the only reasonable way to avoid the compiler's optimizer to remove a call to *Annotation.End()* preceding a *return* instruction is to move the call itself before the *return* instruction. Of course, an annotated fragment may contain an arbitrary numbers of *return* instructions, so it is necessary to insert a call to a new special placeholder before each of them.

The new special placeholder, *Annotation.EndEx()*, is the following:

```
public static void EndEx(int idx) {}
```

The method differs from *Annotation.End()* only by the name. Anyway, the meaning of this new placeholder is different from that of *Annotation.End()*; each call to *Annotation.EndEx(i)*, for a given *i*, means that, in case we will not found a call to *Annotation.End(i)* matching a previous call to *Annotation.Begin(i)*, the annotation's end has to be considered the first `ret` instruction following the *last* call to *Annotation.EndEx(i)*.

All of this can be done directly adding a new small patch to the ATG and by modifying the annotations retrieval algorithm.

In fact, an adjustment of the ATG is needed for the generation of proper calls to *Annotation.EndEx()*, but we must also adapt the [a]C# runtime to make it aware of this change, otherwise the absence of any *Annotation.End(i)* would break the annotated fragments retrieval algorithm correctness used by the method *Annotation.GetCustomAttributes()* of the "Annotation" class, which is crucial to the whole system we are talking about; as a result, what we must ensure is that the algorithm will check all the calls to *Annotation.EndEx(i)* and will act consequently.

We accomplished the task through the mean of a conservative modification to the behaviour of the method mentioned above.

To get more clarity, let us observe how was the original algorithm flow

1. for each method's IL instructions:

    - if the current instruction is a call to *Annotation.Begin(i)* then a new annotation object is pushed on a stack and inserted on the annotations tree structure. The annotation's starting position is set.

- if the current instruction is a call to *Annotation.End(i)*, the annotation object on the top of the stack is updated with the annotation's ending position. The object is removed from the stack.

2. annotations tree is returned.

With our patch, now the algorithm becomes the following:

1. for each method's IL instructions:
    - if the current instruction is a call to *Annotation.Begin(i)* then a new annotation object is pushed on a stack and inserted on the annotations tree structure. The annotation's starting position is set.
    - if the current instruction is a call to *Annotation.End(i)*:
        - if α is *null* or the index in the annotation object stored in α is not equals to i, the annotation object on the top of the stack is updated with the annotation's ending position, the object is removed from the stack and a reference to the object is stored in α.
        - otherwise the annotation object stored in α is updated with the annotation's ending position.
    - if the current instruction is a call to *Annotation.EndEx(i)*
        - search for the position of the first `ret` instruction that is the possible annotation's ending position.
        - if α is *null* then the annotation object on the top of the stack is updated with the annotation's ending position and removed from the stack. A reference to the object is stored in α.
        - otherwise if the annotation index i it is the same of that stored in α then that annotation object is updated. If not, we search the stack for the corresponding annotation object: if found it is updated, removed from the stack and a reference to it is stored in α.
2. annotations tree is returned.

The figure below shows the main steps of the algorithm:

Now we can see how the new version of the [a]C# compiler will actually translate the example **??**:

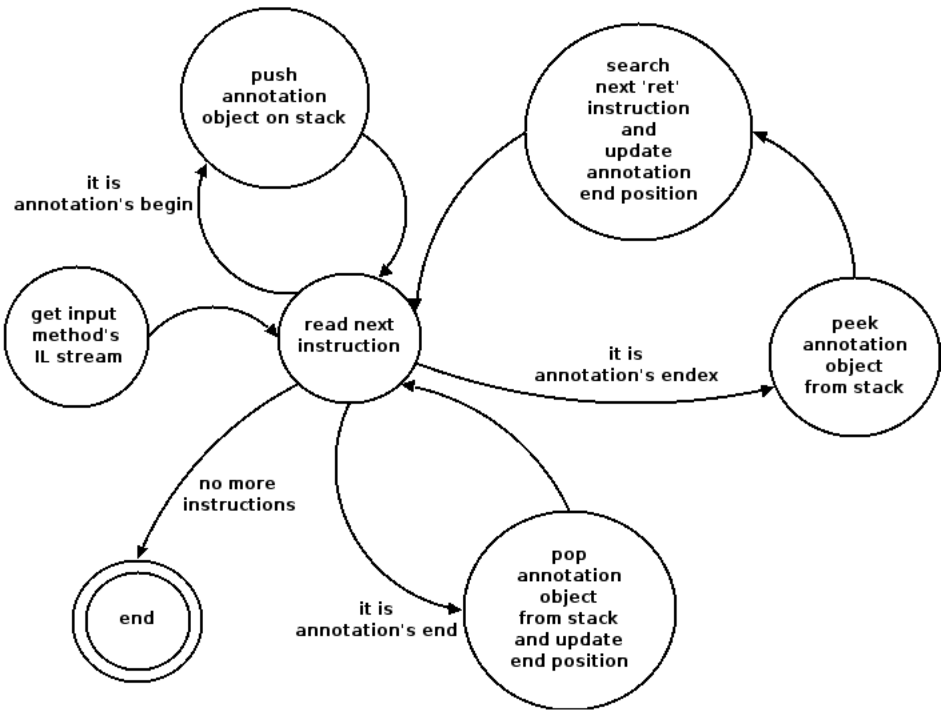

push annotation object on stack
search next 'ret' instruction and update annotation end position
it is annotation's begin
get input method's IL stream
read next instruction
it is annotation's endex
peek annotation object from stack
no more instructions
end
it is annotation's end
pop annotation object from stack and update end position

Figure 4.2: Annotations' retrieval

```
                    New C# code for ''check''
public bool check (int i){
        ACS.Annotation.Begin(0);
        {
        if (i>0)
        {
                ACS.Annotation.EndEx(0);
                return true;
        }
         else
        {
                ACS.Annotation.EndEx(0);
                return false;
        }
        ACS.Annotation.End(0);
}
```

and its compiled version

New IL code for ''check''
```
IL_0000: ldc.i4.0
IL_0001: call void class ACS.Annotation::Begin(int32)
IL_0006: ldarg.1
IL_0007: ldc.i4.0
IL_0008: ble IL_0015
IL_000d: ldc.i4.0
IL_000e: call void class ACS.Annotation::EndEx(int32)
IL_0013: ldc.i4.1
IL_0014: ret
IL_0015: ldc.i4.0
IL_0016: call void class ACS.Annotation::EndEx(int32)
IL_001b: ldc.i4.0
IL_001c: ret
```

The last (third) call to *Annotation.End(*0*)* is still missing, but we now have two new calls - those on *IL_000e* and *IL_0016* - that allows to infer the real annotation's ending position. Indeed, the algorithm described above will correctly set the annotation's ending position as the instruction number *IL_001c*.

4.3.9 Handling errors

A good programming practice suggests to define a new type of exception for each new kind of possible error arising from program usages; following this simple principle we have defined the *OperationException* (see listing **??**) that extends the *System.Exception* class and provide two convenient constructors override, allowing to specify an error message as well as the particular annotation object causing the error.

It will be shown on next Sections which situations require to throw an instance of these exceptions.

4.4 Extending CodeBricks

The hardest part of the development work related to this thesis consisted in extending the CodeBricks library.

Indeed, all the operations on annotated code fragments described in this thesis are performed by the CodeBricks' class *Code*.

We have defined a new constructor for this class and we have enhanced its IL code generation algorithm.

4.4.1 A new constructor for *Code*

The class *Code* is the real core of CodeBricks. An instance of Code represents a code value which is a piece of executable code.

In the original version, a new Code instance can be created by passing to its constructor a descriptor of a method (an instance of *System.Reflection.MethodInfo*).

We have extended the Code class adding a new constructor:

```
public Code(MethodInfo m, Operation[] operations)
```

The constructor allows to specify a list whose each element is one of the operations that have been described in this thesis; these operations will be applied sequentially to the method *m* (see again Section **??**).

The *Operation* class is a representation of any one of such operations.

The *Operation* constructors have three different signatures:

- The following constructor is used when defining either an *extrusion* or an *removal* operation of the annotation *at*.

  ```
  public Operation(AnnotationTree at, CodeAttribute.Operations
      operation)
  ```

- The constructor below is used to define an *inclusion* operation. The last parameter represents the code to be included.

  ```
  public Operation(AnnotationTree at, CodeAttribute.Operations
      operation, Code code)
  ```

- This last constructor is used to define a *copy* operation. The second parameter is the annotation to be copied.

  ```
  public Operation(AnnotationTree at, AnnotationTree copy,
     CodeAttribute.Operations operation)
  ```

All constructors, of course, allow to specify an annotation as first argument as well as the operation to be performed. More precisely - because annotations can be nested - the first argument is an instance of the class *AnnotationTree* which is a tree where each node is an annotation; the constructors operate on the annotation corresponding to the root node only.

The new constructors initialize a new Code instance by carrying out the following tasks:

- Retrieval of both annotated fragment's free variables (parameters) and local variables.

- In case of extrusion, initialization of data related to the annotation's signature to allow method generation.
- In case of inclusion, check of signatures compatibility (see Section **??**). If bindings specified on target annotation are invalid then the constructor will raise an exception.

If no errors are detected, the new Code instance will represent the code value deriving from the applied operations. However, it will be still possible for errors to be detected at actual code generation time, that is when the *Code.FillMethodBody()* method will be invoked. We have to remind that CodeBricks adopts a lazy approach so IL code generation is delayed until the user really needs to execute the method represented by the Code object.

4.4.2 IL code generation

The *System.Reflection.Emit* namespace, contained in the standard Mono (and .NET) framework, provides all necessary classes to realize runtime code generation. These classes are used throughout CodeBricks to generate at runtime methods related to code values.

The most important class is *ILGenerator* which enables to emit a stream of IL instructions forming the method's body and handles many low level issues such as instruction coding; its usage can be noted in the source code of the *Code.FillMethodBody()* method (Section **??**).

4.4.3 Enhacing *Code.FillMethodBody* method

As said, a Code object is a runnable piece of code. To allow execution, the Code class generates a method using the facilities provided by *System.Reflection.Emit* namespace.

The method generation final process is carried out by *FillMethodBody()*, which produces the actual IL code forming the method's body.

In a few words, *FillMethodBody()* scans the input method's IL stream instruction by instruction, modifies instruction's arguments when needed and then and re-emits the instruction in a new stream. The new stream will be the new method's body.

Because we want to manipulate the input method's IL stream by applying the operations described in this thesis, the *FillMethodBody()* method is the place where to put the actual implementation of these operations. For such a reason, almost all the relevant source code that has been written for this thesis is located here.

Most of our efforts have been dedicated to keep the IL code generation's cost linear in terms of IL instructions and to minimize memory usage.

Describing the *FillMethodBody()* new behaviour is quite complex so we will imagine it to be composed of four parts:

1. all necessary IL instructions related to local variables declarations are emitted.
2. original method's instructions are analyzed one by one.
3. depending on the kind of the current operation, we check if it is time to perform the operation.
4. if the execution flow reaches this part, then the current instruction of the input method's IL stream is re-emitted in the new method's

In part 1 in particular, we had to add the code dealing with an aspect of the *extrusion* operation: we are going to produce a method with a different signature from the input method's one, so we have to emit proper instructions to load and store new method's arguments. Therefore, here are emitted the instructions to load each "extruded" method's arguments and to store their value on the respective local variables.

In part 2, given the annotation interested by the current operation - we remind that *FillMethodBody()* must perform a list of operations - we search for the annotation's begin placeholder and retrieve the associated metadata, if any, that are relative to annotation's bindings.

This is done by searching for annotations' placeholders *Annotation.BeginMeta()* and *Annotation.Begin()*, and by analyzing the IL instructions between these two method calls.

Part 3, in summary, performs one of the the following possible actions:

- *removal*: IL instructions enclosed by the annotation are discarded.
- *inclusion*: the IL instructions of the included Code object are emitted.
- *extrusion*: the IL instructions enclosed by the annotation are emitted.

The inclusion operation requires various tasks to be accomplished:

- any `Ldarg` instruction (load of a method's parameter) is replaced by a `Ldloc` with proper local variable's index.
- any `Ldloc`, `Ldloca`, `Stloc` instructions' parameters are replaced with proper local variable's index.

- any ret (return) requires to solve the type compatibility problem (see ??) and therefore to emit necessary additional IL code instructions or to throw a runtime exception.

Part 4 simply reads the instruction on the input IL stream and re-emit it on the new IL stream; in this part we have corrected the jump relocation code that now takes advantage of the right label mechanism instead of the previous what was based on absolute offsets.

This was a fundamental adjustment to allow any code manipulation, in particular any insertion and removal of IL instructions; actually, the ILGenerator class, that is used to produce IL code, must be considered as an input to the just-in-time (JIT) compiler which performs, at runtime, some optimizations on the IL code produced by ILGenerator, just like code motion[10] or loop unrolling[11]. At this purpose, when emitting branch instructions, it is required to specify labels instead of absolute offsets, to facilitate JIT's duty.

Another correction consisted in a bug fix: the old *FillMethodyBody()* did not handle correctly a .NET implementation detail related to constructors.

However, the real method execution flow is not, by any means, so linear. Indeed, to avoid a complete re-writing of *Code.FillMethodBody()*, we introduced our modifications inside the original main `while` statement which scans the input method's instructions.

Furthermore, we added the code dealing with details like type casting and exceptions throwing.

It is interesting to note that the code we developed is executed only when an operation on an annotation is actually required, thus it does not produce any feasible side-effects on previous Codebricks' behaviour.

4.4.4 Implementing type casting

As for the *inclusion* operation, we have seen how it is possible for the user to specify an annotation's signature, allowing to declare desired bindings between annotation's free variables and the external environment.

The problem is that the types of bound variables can be different from that of respective annotation's free variables.

Obviously, for the inclusion operation to be correct, it is necessary for each annotation free variables' type to be assignment-compatible with the bound

[10]Loop-invariant code motion is a technique consisting on moving a piece of code, that is contained inside a loop, before or after the loop itself, without affecting the program's semantics.

[11]Loop unrolling is a technique which duplicates the body of a loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps.

variable's type; if not, we must abort the operation throwing an exception to notify the user about a type-incompatibility problem.

In both cases, first of all we have to discover whether these two types are type-compatible or, more precisely, whether it is permitted to assign a value, whose type is that of the value returned by the included method, to the user-specified variable whose type may be different. For full details see Chapter 8, Part I of [**?**].

To inspect variable's types, the implementation must deal with both object types and value types.

Neither Mono nor .NET gives any direct support to decide whether two value types are compatible or not.

That is the reason why value types compatibility problem has been solved simply using an hard-coded conversion table.

Here is a representation of the conversion table as extracted from [**?**].

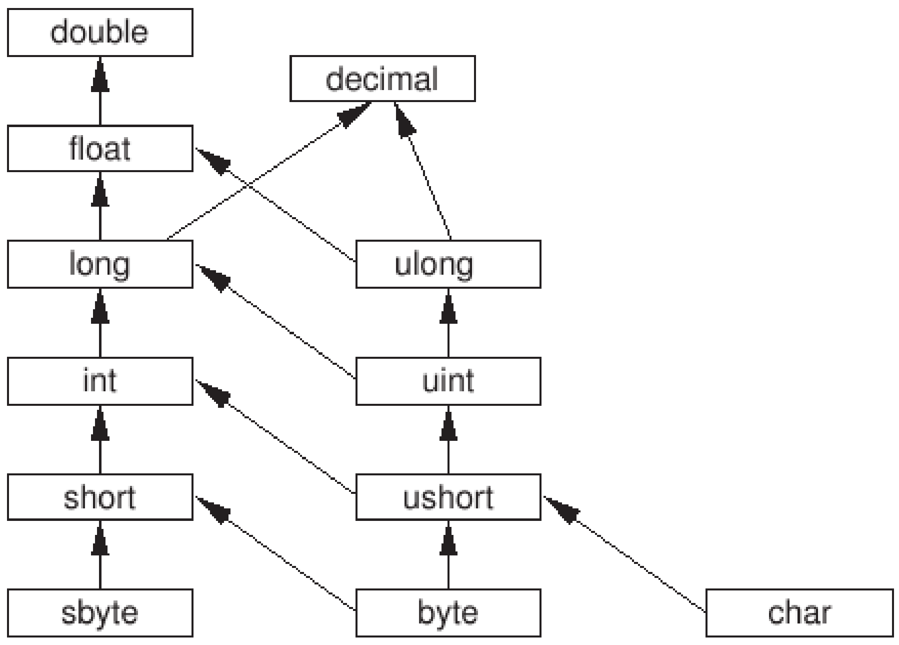


Figure 4.3: Value types of C#

When the two types interested by the conversion are object types, then the problem is easily resolved by *Type.IsAssignableFrom(Type from)*, provided by the System namespace, which simply check the class' hierarchy for the relation above to hold.

Anyway, when we deal with an object type and a value type, the case is more complex because it requires to check for the existence of two special methods belonging to the class whose value is to be assigned:

- *op_Explicit(R)*: it represents an explicit cast operator.
- *op_Implicit(R)*: it represents an implicit cast operator.

To find these methods, we need once again to take advantage of reflection to introspect a class; to this purpose we have developed the code listed below:

Listing 4.9: Finding custom cast operators

```
 1 private static MethodInfo ExistsCustomCast(Type from, Type to,
      bool _explicit) {
 2   String name = "op_Implicit";
 3   if (_explicit)
 4     name = "op_Explicit";
 5   MethodInfo cast = from.GetMethod(name, new Type[]{from});
 6   if (cast != null)
 7     if (cast.ReturnType.Equals(to))
 8       return cast;
 9   return null;
10 }
```

The method searches the “from” class type for a method named either *op_Implicit* or *op_Explicit* with the correct signature that is a signature consisting of one only argument of type “to”: see line number 5. If actually found, a reference to that method is returned for IL generation purposes.

4.4.5 Considerations about performances

Another reason that has been motivating the choice for CodeBricks as implementation base was its remarkable performances both on IL code analysis and generation.

The algorithm performing the operations on annotations has a linear cost in terms of the number of method’s IL instruction ($O(n)$ where n is the sum of all the input method’s IL instructions and all possible other included code’s IL instructions). As a result, we do not have introduced any performance worsening.

Furthermore, for what concerns generated IL code, we tried to follow as much as possible ECMA’s [?] guidelines and suggestions to improve IL execution speed.

Finally, the small IL code size increase due to the insertion of metadata and dummy methods performed by the [a]C# compiler is not relevant.

Moreover an [a]C#'s annotations-aware JIT could easily discard this kind of "useless" method calls.

4.5 Difficulties experienced

Working on a fresh open-source project is not always a so simple task. What follows is just a brief summary of the main problems we had to face throughout the development of this thesis:

- We adopted *Monodevelop* [**?**] as *IDE*[12] though, as by now, it is not much more than an editor; actually, it lacks many common development tools just like assisted code refactoring or source code navigation.
- Mono documentation, as many other open source products, is far away from being complete and exhaustive.
- As a matter of fact, we were not able to exploit any debugging tool though, whilst we was writing this thesis, a Mono's debugger project was already started. On the other hand, Monodevelop's integrated debugger was still missing.
- At the time we started developing this thesis, Mono's support to *.NET* 2.0 framework (therefore things like generics[13] and generic collections implementation) was still incomplete. We initially forced Monodelevop to use *gmcs.exe* instead of *mcs.exe* compiler though this was not encouraged by the Mono team; fortunately, since its version 0.11, it is possible to choose the compilation's target runtime version.
- there was no documentation on the existing CodeBricks, CLIFileRW and [a]C# source code except for some comments.

In summary, the whole thing turned to be an exciting programming effort.

Nevertheless, we still think open source philosophy is a wonderful example of what can be done through collaboration on a worldwide community.

[12]**I**ntegrated **D**evelopment **E**nvironment

[13]Generics are a language feature very similar to C++ template programming.

Chapter 5

Test drive

"But, in case signals can be neither be seen or perfectly understood, no captain can do very wrong if he places his ship alongside that of the enemy."
Vice Admiral Horatio Nelson

5.1 Examples

To bring a proof of real on-field runtime annotation manipulation usability, we present a few examples exploiting the technique that has been described in this thesis in three different application areas. The first one is about code reuse whereas the second example shows a demonstrative implementation of a minimal runtime aspect oriented programming system. The last example demonstrates annotation manipulation's effectiveness when the technique is applied to code specialization.

5.1.1 From interactive to batch

As an example of metaprogramming realized through the means of annotations, we show a transformation which, given an interactive (in terms of its input and output) program, produces an equivalent "batch"[1] program whose input has been replaced with constant values or values produced automatically, and whose output has been redirected to one or more desired targets.

To realize such transformation, our program will rely on the following property: *all input program's source code sections dealing with input, output and debug have been annotated with well-known attributes.*

Well-know attributes has been defined as follows:

[1]Batch processing is the execution of a series of programs ("jobs") on a computer without human interaction, when possible.

Listing 5.1: Input/Output Attributes

```
using System;
using ACS;

public class IOAttribute : CodeAttribute
{

  public enum Redirection
  {
    CONSOLE,
    FILE,
    LOG
  }

  public Redirection Redirect;

}

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class InputAttribute : IOAttribute
{

}

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class OutputAttribute : IOAttribute
{

}
```

Attribute's name should be self-explanatory.

In this example, the input program asks the user for a string and prints out an SHA-512[2] encoding of the supplied string, together with information about total time spent in encoding. The program terminates when the user enters the "exit" string.

Here is the source:

Listing 5.2: Input program

```
using System;
using ACS;
```

[2]The **SHA** (Secure Hash Algorithm) family is a set of related cryptographic hash functions.

```
using System.IO;
using System.Text;
using System.Security.Cryptography;

public class Interactive {

  public static HashAlgorithm hash = SHA512.Create();
  public static string EXIT = "exit";

  public static string Encrypt(string str)
  {
    MemoryStream ms = new MemoryStream();
    byte[] buf = Encoding.UTF8.GetBytes(str);
    ms.Write(buf, 0, buf.Length);
    Decoder dec = Encoding.UTF8.GetDecoder();
    int n = dec.GetCharCount(buf, 0, buf.Length);
    char[] chars = new char[n];
    dec.GetChars(hash.ComputeHash(ms), 0, buf.Length, chars, 0);
    return new String(chars);
  }

  public static void Main() {
    string input = null;
    do {
      [Input(*input, &input)]
      {
        Console.WriteLine("Type a string to encrypt or 'exit' to
             quit: ");
        input = Console.ReadLine();
      }

      DateTime start = DateTime.Now;
      string output = Encrypt(input);
      TimeSpan timeElapsed = DateTime.Now - start;
      string elapsed = timeElapsed.ToString();

      [Output(Redirect = OutputAttribute.Redirection.LOG, *
         elapsed)]
      {
        Console.WriteLine(String.Format("Elapsed time: {0}",
            elapsed));
      }

      [Output(Redirect = OutputAttribute.Redirection.FILE, *
          output)] {
        Console.WriteLine(String.Format("Encrypted string is
            '{0}'", output));
      }
    } while (!input.Equals(EXIT));
  }

}
```

As we can see, *Main()* methods contains three annotated fragments: the first fragment has been annotated with an *Input* attribute whereas the second and third have been annotated with *Output* attributes. Anyway, these two annotations differ on their *Redirect* named parameter: one has been set to the "Log" constant's value while the other to the "FILE" constant's value.

Apart from these annotations, the Interactive's *Main()* is a normal method with a standard input/output behaviour. Now we want it to be transformed, at runtime, in a batch program (without user interaction) whose input will consist on a sequence of random[3] numbers and whose output will be stored on a "output.sha512" file. Furthermore, the program's log output will be stored on a "log.txt" file.

The program performing the transformation is quite simple, its key concept being the definition of three methods:

- *GenerateString* which returns a string composed by random characters.
- *Log* which writes its string argument to a predefined log file.
- *Output* which writes its string argument to a predefined text file.

Here follow the transformer program's source code:

Listing 5.3: Transformer program

```
using System;
using System.Reflection;
using System.Collections;
using System.Text;
using System.IO;
using ACS;

```

[3]these numbers will be pseudo-random numbers, of course.

```
public class Batch {

  public static string GenerateString()
  {
    Random rand = new Random();
    string s = rand.Next(1000).ToString();
    if (s.Equals("999"))
      s = Interactive.EXIT;
    return s;
  }

  public static void Log(string s)
  {
    StreamWriter log = new StreamWriter(new FileStream("log.txt"
        , FileMode.Append));
    log.WriteLine(s);
    log.Close();
  }

  public static void Output(string s)
  {
    StreamWriter output = new StreamWriter(new FileStream("
        output.sha512", FileMode.Append));
    output.WriteLine(s);
    output.Close();
  }

  delegate void Void();
  delegate void Str(string i);

  public static void Main(string[] args)
  {
    Assembly asm = Assembly.GetExecutingAssembly();
    Type program = asm.GetType("Interactive");
    MethodInfo main = program.GetMethod("Main");
    Code In = new Code(typeof(Batch).GetMethod("GenerateString")
        );
    Code Out = new Code(typeof(Batch).GetMethod("Output"));
    Code Log = new Code(typeof(Batch).GetMethod("Log"));
    AnnotationTree[] io = Annotation.GetCustomAttributes(main);
    ArrayList operations = new ArrayList();

47      foreach (AnnotationTree annot in io)
48      {
49        if (annot.Node[0] is InputAttribute)
50        {
51          operations.Add(new Operation(annot,
52            CodeAttribute.Operations.INCLUDE_BEFORE, In));
53          operations.Add(new Operation(annot, CodeAttribute.
              Operations.REMOVE));
54        }else if (annot.Node[0] is OutputAttribute)
55          if (((IOAttribute)annot.Node[0]).Redirect == IOAttribute
              .Redirection.FILE)
56          {
57            operations.Add(new Operation(annot,
58                CodeAttribute.Operations.INCLUDE_BEFORE, Out));
59            operations.Add(new Operation(annot, CodeAttribute.
                Operations.REMOVE));
60          }else if (((IOAttribute)annot.Node[0]).Redirect ==
              IOAttribute.Redirection.LOG)
61          {
62            operations.Add(new Operation(annot,
63                CodeAttribute.Operations.INCLUDE_BEFORE, Log));
64            operations.Add(new Operation(annot, CodeAttribute.
                Operations.REMOVE));
65          }
66      }
67
68      Console.WriteLine("operations # = {0}", operations.Count);
69      Code batch = new Code(main,
70          (Operation[])operations.ToArray(typeof(Operation)));
71      Delegate batchMain = batch.MakeDelegate(typeof(Void), "boh.
          dll");
72      batchMain.DynamicInvoke(null);
73    }
74
75  }
```

As first, lines from 0 to 0, the program loads the input program's assembly to retrieve a reference to its main method. Then this method is inspected to retrieve the annotated fragments that are supposed to have been defined inside it. At this time, on line 0 the program creates three new Code instances bound to the methods discussed above. After this necessary setup, the program scans all retrieved annotations to build a list of operation on the annotations, applying the following rules (lines from 0 to 0):

- if the annotation type is **Input**, then the annotation must be replaced with the Code instance bound to *GenerateString*.
- if the annotation type is **Output**, then the annotation must be replaced with the Code instance bound to *Output* or to *Log* according to the "Redirect" annotation's property value.

Finally, the program creates a new Code instance bound to the input program's main method but, this time, the computed operations list is passed to the the Code constructor too. As a result, we will get a new *Main()* whose interactive portions have been replaced by different non-interactive ones.

As last, the resulting code is executed by dynamically invoking the method that will be generated by the call to `MakeDelegate()`.

5.1.2 Runtime AOP

As discussed on **??**, AOP is now widely used as a means to insert arbitrary cross-cutting source code into another existing source code. Indeed, all AOP system just like AspectJ [**?**] [**?**], perform their actions at compile time and they do not allow to inject code inside method's bodies.

Instead, here we present a simple example proving how it is possible to use annotation's manipulation to realize AOP at runtime and, at the same time, to remove the above restraint.

We start by defining an "AOP" attribute:

Listing 5.4: AOP Attribute

```
using System;
using ACS;

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class AOPAttribute : CodeAttribute {

  public enum Advices
  {
    PRE,
    POST,
    AROUND,
    SWEEP
  }

  public Advices Advice;
  public String Id;
  public String Group;

}
```

The `Advice` property has to be considered as an hint for the AOP runtime engine on applying an AOP strategy. Property's value intuitive meaning is as follows:

- *SWEEP*: the annotated fragment could be removed. This is the case of any debugging-related or even logging-related code. It worth noting that this particular feature is not allowed by AspectJ.
- *PRE*: an advice should be inserted immediately before the annotated fragment. This resembles the AspectJ's `before` advice modifier.
- *POST*: an advice should be inserted immediately after the annotated fragment. This resembles the AspectJ's `after` advice modifier.
- *AROUND*: an advice should be inserted immediately after the annotated fragment. This resembles the AspectJ's `around` advice modifier.

Then, supposed we a have a method executing the following (silly) computation,

Listing 5.5: Not-annotated DoSomething

```
public static void DoSomething()
{
  int v = rnd.Next(0, 100);
  string s;
  Console.WriteLine("v = {0}", v);
  v *= v;
  v += 1;
  v = v \% 2;
  s = v == 0 ? "even" : "odd";
  Console.WriteLine("v is {0}", s);
}
```

we can decorate some method's statements with our newly defined AOP attribute:

Listing 5.6: AOP-annotated DoSomething

```
public static void DoSomething()
{
  int v = rnd.Next(0, 100);
  string s;

  [AOP(Advice = AOPAttribute.Advices.SWEEP, Group = "debug")]
  {
    Console.WriteLine("v = {0}", v);
  }

  [AOP(Advice = AOPAttribute.Advices.POST, Id = "square")]
  {
    v *= v;
  }

  [AOP(Advice = AOPAttribute.Advices.AROUND, Id = "increment")]
  {
    v += 1;
  }

  v = v \% 2;

  [AOP(Advice = AOPAttribute.Advices.PRE, Group = "debug", Id =
     "parity")]
  {
    s = v == 0 ? "even" : "odd";
  }

  [AOP(Advice = AOPAttribute.Advices.SWEEP, Group = "debug")]
  {
    Console.WriteLine("v is {0}", s);
  }
}
```

Doing so, we enable a runtime AOP system, to be discussed later, to apply transformations to the method; in other words, the engine will be able to retrieve all necessary information to perform AOP weaving at runtime.

We have written a simple Gtk# GUI program encapsulating a very minimal runtime AOP system suited for a quick demonstration. The program simply starts a thread continuously executing the above method. The program's GUI allows the user switch on and off the AOP engine and see its effects. Moreover, the GUI also allows to choose an AOP strategy different from that "hard-coded" on the annotations; this is to remark runtime AOP's benefits with respect to common compile-time AOP.

Our minimalistic AOP engine will be capable of:

- injecting advices according to the hints coded in annotation attribute's properties.
- injecting advices before, after or around any annotation whose "Id" property has a value chosen by the user.
- removing any annotation whose annotation's "Group" property has a value chosen by the user.

Of course, it would be feasible -and necessary- for a real AOP engine to perform much more operations. Anyway, the operations listed above are enough to show runtime AOP's promising features.

The program's main thread simply loops on a call to a delegate; at program's startup, that delegate (handler) points to the original *DoSomething()* method. When the user presses on the "Turn on AOP" GUI's button, the AOP engine creates a new Code instance by applying different annotation's operations on *DoSomething()*, then it set the handler to the new delegate returned by the usual *Code.MakeDelegate()*. On the contrary, when the user clicks on the "Turn off AOP" button, the handler is set to *DoSomething()* again, therefore the original program's behaviour is restored.

We provided this example to focus attention on AOP's most important features:

- enabling the user to switch, at runtime, that is without re-compilation, between AOPized and non AOPized program's behaviour.
- allowing the user to dynamically change both aspects and advices, at any time, still avoiding program's restart or re-compilation.

As said, the AOP engine implemented in the example is quite trivial but, on the other hand, only a few lines of code has been necessary to build it, further proving fragments' annotation usefulness.

Our AOP engine's core implementation is presented below:

Listing 5.7: AOP-annotated DoSomething

```
Code advice = new Code(typeof(RuntimeAOP).GetMethod("Advice"));
MethodInfo mi = ((Handler)RuntimeAOP.DoSomething).Method;
AnnotationTree[] annotations = Annotation.GetCustomAttributes(mi
    );
ArrayList operations = new ArrayList();
foreach (AnnotationTree annot in annotations)
{
  if (annot.Node[0] is AOPAttribute)
  {
    AOPAttribute aopa = annot.Node[0] as AOPAttribute;
    if (choice.ActiveText.Equals(HARD_CODED))
      switch(aopa.Advice)
      {
        case AOPAttribute.Advices.PRE:
          operations.Add(new Operation(annot,
          CodeAttribute.Operations.INCLUDE_BEFORE, advice));
          break;
        case AOPAttribute.Advices.POST:
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.INCLUDE_AFTER, advice));
          break;
        case AOPAttribute.Advices.AROUND:
          operations.Add(new Operation(annot,
          CodeAttribute.Operations.INCLUDE_BEFORE, advice));
          operations.Add(new Operation(annot,
          CodeAttribute.Operations.INCLUDE_AFTER, advice));
          break;
        case AOPAttribute.Advices.SWEEP:
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.REMOVE));
          break;
      }
    else if (choice.ActiveText.Equals(ADD_BEFORE + aopa.Id))
      operations.Add(new Operation(annot,
        CodeAttribute.Operations.INCLUDE_BEFORE, advice));
    else if (choice.ActiveText.Equals(ADD_AFTER + aopa.Id))
      operations.Add(new Operation(annot,
        CodeAttribute.Operations.INCLUDE_AFTER, advice));
    else if (choice.ActiveText.Equals(REMOVE_GROUP + aopa.Group)
       )
      operations.Add(new Operation(annot,
        CodeAttribute.Operations.REMOVE));
    else if (choice.ActiveText.Equals(ADD_AROUND + aopa.Id))
    {
      operations.Add(new Operation(annot,
        CodeAttribute.Operations.INCLUDE_BEFORE, advice));
      operations.Add(new Operation(annot,
        CodeAttribute.Operations.INCLUDE_AFTER, advice));
    }
```

```
48   }
49 }
50 Code aopized = new Code(mi, (Operation[]) operations.ToArray(
       typeof(Operation)));
51 AOPHandler = aopized.MakeDelegate(typeof(Handler)) as Handler;
```

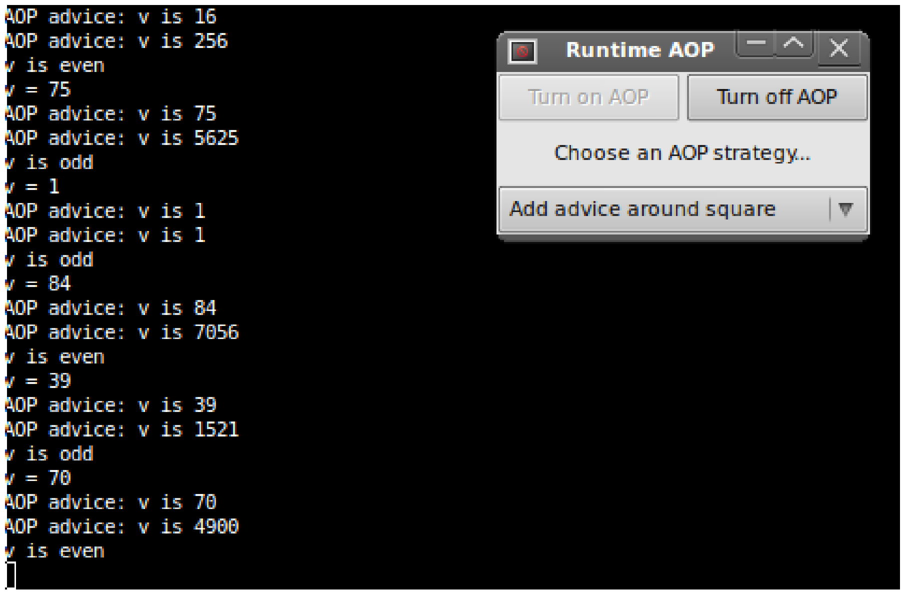
On line 1 the Code instance bound to the *Advice()* method is retrieved (*Advice()* standing for the common AOP's advice concept, indeed it could be whatever else methods, given an annotation's compatible signature).

Lines 2 and 3 retrieve *DoSomething* (but we could use any other body-annotated method) annotation's.

From lines 5 to 49, the program builds different annotation's operations, depending on user settings exposed by program's GUI.

Finally, line 50 creates the new Code instance representing the AOPized *DoSomething()* and line 51 produces related delegate (and IL code, of course).

A screenshot of the annotation based demo application is shown on figure ??. On that figure, we can see the effect of the runtime weaving operation,



```
AOP advice: v is 16
AOP advice: v is 256
v is even
v = 75
AOP advice: v is 75
AOP advice: v is 5625
v is odd
v = 1
AOP advice: v is 1
AOP advice: v is 1
v is odd
v = 84
AOP advice: v is 84
AOP advice: v is 7056
v is even
v = 39
AOP advice: v is 39
AOP advice: v is 1521
v is odd
v = 70
AOP advice: v is 70
AOP advice: v is 4900
v is even
```

Figure 5.1: Annotation-based AOP demo

which is selected in the user interface.

5.1.3 SVG Rendering

Usually, graphics rendering of any complex object, like a 2D vectorial image or 3D model, requires the program to store in memory a data structure representing the object itself (in most cases, some kind of tree), then to access the data structure as many times as needed to draw the object on screen. Although this approach can have acceptable performance when the data-structure is relatively small, that is the object is quite simple, it becomes even more inefficient as object's complexity increases, because of memory consumption and memory access delays.

Using runtime metaprogramming, we can compile, at runtime, a code (a method, in the end) which can render the object without accessing any object's data structure; the object data structure is virtually compiled in, as if it was hard-coded by the user.

Unfortunately, runtime code generation is a quite complex and bug-prone task.

For such a reason, here we show how annotations manipulation can be used to hide those complexity, demanding all runtime code generations details to the annotation manipulation's API.

The example consists of a simple renderer for a dummy SVG[4]-like vectorial image description language. Our example-targeted ESVG simply allows to declare only two different 2D objects, that is rectangles and circles; each of them has attributes describing their position, dimensions and color.

Here is an ESVG example:

Listing 5.8: ESVG sample

```
<?xml version="1.0"?>
<esvg>
  <rectangle x="35" y="77" width="11" height="29" color="1" />
  <circle x="48" y="96" width="27" height="6" color="1" />
</esvg>
```

The example program builds a simple GTK# GUI which is composed by:

- a main drawing area, where the ESVG's content is displayed
- one button to let the user choose the ESVG file to be opened
- a drop-down list to change among three different rendering styles
- a button allowing to switch between the standard iterative drawing algorithm and the “compiled” one

When the user chooses an ESVG file, the program parses file's content and draws its representation on the rendering area, using one of the above approaches depending on the related button's state.

A screenshot of the demo program's window is shown on figure **??**.

Now we can proceed to analyze how the compiled drawing code can be automatically generated by the program, exploiting the annotations manipulation runtime support.

We have created two classes responsible for drawing respectively a rectangle and a circle based on these parameters:

- x, y : cartesian coordinates of the upper left corner or the rectangle or the center of the circle
- w, h: width and height of the rectangle or the bounding rectangle for the case of the circle

[4]SVG is a language for describing two-dimensional graphics and graphical applications in XML. SVG 1.1 is a W3C Recommendation.

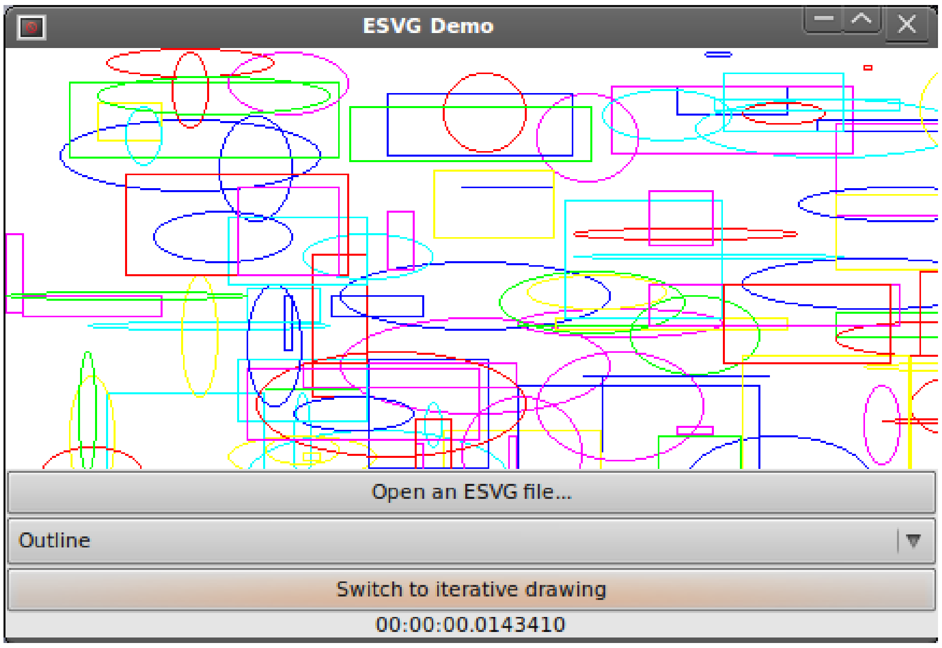


Figure 5.2: The SVG-like viewer

- color: index of the color to be used
- opt: the drawing style to be applied (outline, dashed, filled)

In each of these classes, the key portions of the respective method performing the actual drawing have been annotated using a custom attribute ("DrawAttribute"). Having a look to the *DrawRect* drawing method (the circle's one *DrawCircle* is almost identical), we can notice how the various fragments have been annotated. It is worth to remark that -notwithstanding the presence of the annotations- the flow of the method is quite natural and straightforward.

Listing 5.9: Annotated DrawRect

```
public static void DrawRect(int x, int y, int w, int h, int
    color, int opt)
{
  int lx = (int) (x / 100.0 * window.VisibleRegion.Clipbox.Width
      );
  int ly = (int) (y / 100.0 * window.VisibleRegion.Clipbox.
      Height);
  int lw = (int) (w / 100.0 * window.VisibleRegion.Clipbox.Width
      );
  int lh = (int) (h / 100.0 * window.VisibleRegion.Clipbox.
      Height);
  [Draw]
  {
    if (opt == (int) DrawAttribute.Options.DASHED)
    {
      [Draw(option=DrawAttribute.Options.DASHED)]
      {
        gc.SetDashes(0, new sbyte[]{1, 2}, 2);
        gc.SetLineAttributes(1, LineStyle.OnOffDash, CapStyle.
            Butt, JoinStyle.Miter);
        gc.Foreground = Render.COLORS[color];
        window.DrawRectangle(gc, false, lx, ly, lw, lh);
      }
    }
    else if (opt == (int) DrawAttribute.Options.FILL)
    {
      [Draw(option=DrawAttribute.Options.FILL)]
      {
        gc.Foreground = Render.COLORS[color];
        window.DrawRectangle(gc, true, lx, ly, lw, lh);
      }
    }
    else if (opt == (int) DrawAttribute.Options.OUTLINE)
    {
      [Draw(option=DrawAttribute.Options.OUTLINE)]
      {
        gc.Foreground = Render.COLORS[color];
        window.DrawRectangle(gc, false, lx, ly, lw, lh);
      }
    }
  }
}
```

The *option* property of the custom annotation class *DrawAttribute* is used to mark the fragments of code that performs the three different drawing styles. The two remaining outer annotations have not been marked for simplicity but they could have been decorated properly, of course. For the ease of the explanation, let we call:

- R the outer annotation of the method *DrawRect*
- $r_{dashed}, r_{fill}, r_{outline}$ the three inner annotations of R, respectively

- C the outer annotation of the method *DrawCircle*
- $c_{dashed}, c_{fill}, c_{outline}$ the three inner annotations of C, respectively

To obtain at runtime a method ables to draw an ESVG content as if it was hard-coded, the main program performs the following steps once during the initialization:

1. $\forall r_i | i \in \{dashed, fill, outline\}$ creates a new “Code” object M_{r_i} by applying two operations on *DrawRect*: copy of r_i before R and then removal of R
2. $\forall c_i | i \in \{dashed, fill, outline\}$ creates a new “Code” object M_{c_i} by applying two operations on *DrawCircle*: copy of c_i before C and then removal of C

then, when the user opens an ESVG file:

1. creates the list of operations on annotations L
2. for each shape encountered in the ESVG file:
   - if the shape is :
     - a rectangle, then a new “Code” object N_{r_i} is created by applying the operator *Bind* (see **??**) to r_i where i is the currently user-selected drawing style
     - a circle, then a new “Code” object N_{c_i} is created by applying the operator *Bind* (see **??**) to c_i where i is the currently user-selected drawing style

     Each time, the arguments passed to the CodeBrick's binding operator are the parameters (position, dimensions and color) of the shape. The effect of the binding is that N_{r_i} or N_{c_i} represent fragments of code with no free variables, since the arguments of N_{r_i} or N_{c_i} (Ldarg instructions) have been substituted with constants (Ldc instructions).
   - add N_{r_i} or N_{c_i} to L
3. creates a new “Code” object O by applying the list of operations L to the method “CompiledDraw” of the program's main class. This method is only a placeholder, or better a container where the runtime generated code can be injected.

In the end, the delegate produced by O is used to draw the ESVG content. As we can imagine, the IL code of this delegate will not contain any branch instructions and the parameters of the various rectangle and circle elements will be loaded as constants, thus avoiding direct memory accesses to the elements' data structures.

We have run the program on two different platform: .NET 3.5 on Microsoft Windows XP and Mono 2.4.2.3 on Ubuntu Linux 9.10. Both systems were actually hosted in a virtual machine created by VirtualBox 3.1.4[5] for MacOSX 10.6.2[6] running on an Intel Core Duo 2 @2.53GHz / 4GB RAM computer. Here follows the experimental results:

Table 5.1: Performance comparison of the ESVG example on different platforms

	Compiled			**Generated**		
Iterations	10	100	1000	10	100	1000
Mono 2.4 on Ubuntu 9.10	0.384	5.316	41.566	0.370	4.338	38.400
.NET 3.5 on Windows XP	0.951	10.208	94.711	0.941	9.630	89.510

The performance gain of the generated code with respect to the code compiled from source is little but significant. In such a regard, we remind that the speed-up -though slight- has been achieved only by exploiting the annotation manipulation support. Since we do not ever made any particular attempt to optimize the code produced by the annotation manipulation support, we believe that further improvements are feasible and likely to improve both efficiency and speed of the generated code.

[5]`http://www.virtualbox.org/`

[6]`http://www.apple.com/it/macosx/what-is-macosx/`

Chapter 6

Conclusion

"The whole problem with the world is that fools and fanatics are always so certain of themselves, and wiser people so full of doubts."
Sir Bertrand Russell

6.1 Results

We have developed a metaprogramming technique exploiting annotations that are placed inside methods' body by the programmer. We recall that we use the term annotation to denote the metadata used to mark a block of instructions -a statement- that we call *annotated fragment*.

Although suited for the [a]C# language, the technique could be easily implemented also to target other virtual machine-based languages, such as Java [?] (for which a library to manipulate bytecode is already available: BCEL[1]), since it is always possible to extend their grammar and runtime support.

The technique permits to carry out a few types of operations on a method's body, producing another method having a different semantic. The primitive operations available are:

- inclusion of the instructions of a method before or after another method's annotation.
- extrusion of the instructions of an annotated fragment to produce a new method.
- removal of the instructions of an annotated fragment from the body of the method which the fragment belongs to.

[1]BCEL is library to manipulate Java binary class files (`http://jakarta.apache.org/bcel/`)

- copy of the instructions of an annotated fragment before or after an annotation of the method which the fragment belongs to.

The primitives listed above can be composed to perform more complex operations. For instance, we can generate a new method by applying an inclusion and then a removal of two different annotated fragments in a single step.

As stated before, since our technique works on the IL level, it can be implemented on virtually all the languages supported by the Mono or .NET platforms.

We have showed an example of application of the technique to each one of the following software engineering areas:

- code reuse: the example demonstrates how to programmatically transform a program which need a user-provided input, into a batch program which takes its input from another source, such as a file.

- code specialization: the application shows how it is possible to automatically produce -at runtime- a specialized code performing a certain task; in this case, the example exploits the technique to render an SVG-like content on screen. Also, a performance comparison of the hand-written code versus the specialized code has been presented.

- aspect oriented programming: the example program performs the fundamentals code-weaving operations on itself, at runtime. The program shows how the technique can be used to build a generic runtime aspect-oriented programming support.

We have extended the [a]C# annotations syntax and runtime support to enable the user specify an annotated fragment's signature, that is the binding of the fragment's free variables with respect to an external environment.

Furthermore, we have enriched [a]C# runtime support making it able to retrieve both annotated fragment's free and local variables.

In addition, we have realized a porting of the CodeBricks and CLIFileRW libraries to the Linux/Mono platform. We are currently engaged on porting these libraries and the present work on the MacOSX/Mono platform too.

Whilst our main aim was to develop our technique on the Linux/Mono platform, we have been successfully able to run it on the Windows/.NET platform.

Finally, we have fixed some minor bugs that we have discovered in [a]C# compiler as well as in the CLIFileRW and CodeBricks libraries.

6.2 Future works

The [a]C# [?] language is an extension of the C# language designed to allow the programmer to insert metadata -in the form of annotated statements-into the body of methods.

We have realized an extension to the [a]C# runtime support which allows to perform metaprogramming operations on annotated [a]C# methods.

Anyway, the implementation developed in this thesis, though being actually usable, is still incomplete. Primarily lacks are:

- as by now, CodeBricks' Code class works only on static methods. As a consequence, our work inherited this limitation. Further efforts on CodeBricks are needed to handle instance methods too.
- some C# language features such as *out* parameters should be treated correctly and there are some aspects, just like IL code analysis, that would probably deserve more attention.
- although all the test presented in this thesis run as expected, we neither have conducted an exhaustive test suite nor we have a formal correctness proof.

Additionally, it would be useful allowing to remove or not the placeholders related to a given operation, at least. Besides, a further option could be provided to remove any placeholders. Besides, there are some feasible optimizations -such as those presented on Section **??**- that could be applied to the existing algorithms.

Finally, it could be interesting to study the behavior of the JIT optimizer, in order to produce a bytecode better suited to JIT optimizations.

However, we are confident that the runtime code manipulation technique presented in this thesis could be integrated in any virtual machine based and annotation-capable programming language as an effective metaprogramming tool.

Chapter 7

Appendix

"Appear weak when you are strong, and strong when you are weak."
Sun Tzu

7.1 Sources

7.1.1 [a]C# Runtime

The listing below is the complete source code of the [a]C# runtime.

We have added the method *AnnotationTree.GetAnnotationSignature()* which is responsible for inferring an annotated fragment's free and local variables.

We have created the *TypesConversion* class too, which contains the definition of an hash-table representing all legal conversions among C# language's value-types.

Other modifications affected mainly the *Annotation.GetCustomAttributes()* method; the method builds a tree whose each node is a representation of one of the annotated fragments contained inside the method passed as argument.

Listing 7.1: [a C#]Runtime

```
/////////////////////////////////////////////////////////////////
///
/// Modified by: Nicola Giordani (nicola@dinosoft.it)
///
/// Added some code to allow building a Code
/// out of an [a]C# annotation
///
/////////////////////////////////////////////////////////////////

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Collections;
#if UNIX
using System.Collections.Generic;
```

```
#endif
using CLIFile;

namespace ACS {
  [AttributeUsage(AttributeTargets.Method, AllowMultiple=true, Inherited=true)]
  public class CodeAttribute : Attribute {

    private int idx;
    internal ILCursor.State beg;
    internal long end;
    internal bool returns = false;
    internal long meta = -1;

    public int ACSIndex {
      get { return idx; }
      set { idx = value; }
    }

    public ILCursor ILInstructions {
      get {
        return ILCursor.RestoreCursor(beg);
      }
    }

    public long BeginPosition {
      get {
        return beg.pos;
      }
    }

    public long EndPosition {
      get {
        return end;
      }
    }

    public long Start() {
      if (meta == -1)
        return beg.pos - 1;
      else
        return meta;
    }

    public long MetaPosition {
      get {
        return meta;
      }
      set {
        meta = value;
      }
    }

    public bool Returns {
      get {
        return returns;
      }
    }

    public int MetaLength() {
      if (meta == -1)
        // ldloc + call = 6
        return 12;
      else
        return (int) (beg.pos - meta) + 5 + 5 + 1;
    }

    public int Length() {
      return Size() + MetaLength();
    }

    public int Size() {
      return (int) (end - beg.pos) - 5;
    }

    public enum Operations {
      INCLUDE_BEFORE,
      INCLUDE_AFTER,
      EXTRUDE,
      REMOVE,
      COPY_BEFORE,
      COPY_AFTER
    }

  }

  public delegate void TreeVisitor(AnnotationTree t, int vsitedChild);

  public sealed class AnnotationTree : IEnumerable {
    internal CodeAttribute[] node;
    internal AnnotationTree[] children;

    private object[][] signature;

    internal AnnotationTree() {
    }

    public CodeAttribute[] Node {
      get { return node; }
    }

    public AnnotationTree[] Children {
      get { return children; }
    }

    public AnnotationTree this[int idx] {
      get { return children[idx]; }
    }

    public void Visit(TreeVisitor v) {
      InternalVisit(this, v);
    }

    public CodeAttribute[] GetAttributes(Type t) {
      ArrayList a = new ArrayList();
      foreach (CodeAttribute c in node)
        if (c.GetType() == t)
          a.Add(c);
      return (CodeAttribute[])a.ToArray(t);
    }

    public struct InputArgument {
      public bool IsLocal;
      public int Num;
      public bool ByRef;
      public bool ReadOnly;
      public InputArgument(bool loc, int n, bool byref, bool ro) {
        IsLocal = loc;
        Num = n;
        ByRef = byref;
        ReadOnly = ro;
      }
    }

    public InputArgument[] BlockSignature() {
      ILCursor ilc = node[0].ILInstructions;
      ArrayList ret = new ArrayList();
      // This map is used id -> retid, locals are -id.
      Hashtable map = new Hashtable();
      while (ilc.Position < node[0].end) {
        // FIXME: Check the stack behavior
        ILInstruction il = ILInstruction.Normalize(ilc.Instr);
        if (il.op.Equals(OpCodes.Ldarg)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(il.par.iv)) {
            inp = (InputArgument)ret[(int)map[il.par.iv]];
          } else {
            inp = new InputArgument(false, il.par.iv, false, true);
            map[il.par.iv] = ret.Add(inp);
          }
        }

        if (il.op.Equals(OpCodes.Ldarga)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(il.par.iv)) {
            inp = (InputArgument)ret[(int)map[il.par.iv]];
            inp.ByRef = true;
            inp.ReadOnly = false;
            ret[(int)map[il.par.iv]] = inp;
          } else {
            inp = new InputArgument(false, il.par.iv, true, false);
            map[il.par.iv] = ret.Add(inp);
          }
        }

        if (il.op.Equals(OpCodes.Starg)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(il.par.iv)) {
            inp = (InputArgument)ret[(int)map[il.par.iv]];
            inp.ReadOnly = false;
            ret[(int)map[il.par.iv]] = inp;
          } else {
            inp = new InputArgument(false, il.par.iv, false, false);
            map[il.par.iv] = ret.Add(inp);
          }
        }

        if (il.op.Equals(OpCodes.Ldloc)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(-il.par.iv)) {
            inp = (InputArgument)ret[(int)map[-il.par.iv]];
          } else {
            inp = new InputArgument(true, il.par.iv, false, true);
            map[-il.par.iv] = ret.Add(inp);
          }
        }

        if (il.op.Equals(OpCodes.Ldloca)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(-il.par.iv)) {
            inp = (InputArgument)ret[(int)map[-il.par.iv]];
          inp.ByRef = true;
          inp.ReadOnly = false;
          ret[(int)map[-il.par.iv]] = inp;
        } else {
          inp = new InputArgument(true, il.par.iv, true, false);
          map[-il.par.iv] = ret.Add(inp);
        }
      }

      if (il.op.Equals(OpCodes.Stloc)) {
        InputArgument inp = new InputArgument();

        if (map.ContainsKey(-il.par.iv)) {
          inp = (InputArgument)ret[(int)map[-il.par.iv]];
          inp.ReadOnly = false;
          ret[(int)map[-il.par.iv]] = inp;
        } else {
          inp = new InputArgument(true, il.par.iv, false, false);
          map[-il.par.iv] = ret.Add(inp);
        }
      }
      ilc.Next();
    }
    return (InputArgument[])ret.ToArray(typeof(InputArgument));
  }

  public object[][] GetAnnotationSignature(MethodInfo mi) {
    MethodTableCursor mc = CLIFileReader.FindMethod(mi);
    mc.Goto(mi.MetadataToken & 0xFFFFFF);
    ILCursor milc = mc.MethodBody.ILInstructions;

    long blockbegin = node[0].beg.pos;
    long blockend = node[0].end;

    IList<int> idx = new List<int>();
    ArrayList realparameters = new ArrayList();
    ArrayList parameters = new ArrayList();
    ArrayList locals = new ArrayList();

    ParameterInfo[] mp = mi.GetParameters();
    IList<LocalVariableInfo> lvi = mi.GetMethodBody().LocalVariables;
    Dictionary<int, LocalVariableInfo> dlvi = new Dictionary<int, LocalVariableInfo>()
        ;

    foreach (LocalVariableInfo v in lvi)
      dlvi.Add(v.LocalIndex, v);

    bool more = true;
    while (milc.Position < blockend && more) {
      ILInstruction il = ILInstruction.Normalize(milc.Instr);
      if (il.op.Equals(OpCodes.Ldloc) ||
          il.op.Equals(OpCodes.Ldloca) ||
          il.op.Equals(OpCodes.Stloc)) {
        if (milc.Position < blockbegin) {
          if (!idx.Contains(il.par.iv))
            idx.Add(il.par.iv);
        }else {
          LocalVariableInfo v = dlvi[il.par.iv];
          if (idx.Contains(il.par.iv)) {
            if (v != null)
              if (!parameters.Contains(v))
                parameters.Add(v);
          }else
            locals.Add(v);
        }
        }else if (milc.Position > blockbegin) {
          if (il.op.Equals(OpCodes.Ldarg)  ||
              il.op.Equals(OpCodes.Ldarga) ||
              il.op.Equals(OpCodes.Starg))
            if (!realparameters.Contains(mp[il.par.iv]))
              realparameters.Add(mp[il.par.iv]);
        }
        more = milc.Next();
      }
      object[][] r = new object[3][];
      r[0] = parameters.ToArray(typeof(LocalVariableInfo)) as LocalVariableInfo[];
      r[1] = locals.ToArray(typeof(LocalVariableInfo)) as LocalVariableInfo[];
      r[2] = realparameters.ToArray(typeof(ParameterInfo)) as ParameterInfo[];
      return r;
    }

  public LocalVariableInfo[] AnnotationParameters(MethodInfo m) {
    if (signature == null)
      signature = GetAnnotationSignature(m);
    return signature[0] as LocalVariableInfo[];
  }

  public ParameterInfo[] AnnotationRealParameters(MethodInfo m) {
    if (signature == null)
      signature = GetAnnotationSignature(m);
    return signature[2] as ParameterInfo[];
  }

  public LocalVariableInfo[] AnnotationLocals(MethodInfo m) {
    if (signature == null)
      signature = GetAnnotationSignature(m);
    return signature[1] as LocalVariableInfo[];
  }

  public void GenerateMethodBody(InputArgument[] args, ILGenerator ilg) {
/*      ILCursor ilc = node[0].ILInstructions;
      Hashtable labels = new Hashtable();

      Hashtable map = new Hashtable();
      for (int i = 0; i < args.Length; i++)
        map[args[i].IsLocal ? -args[i].Num : args[i].Num] = i;

      while (ilc.Position < node[0].end) {
        // FIXME: Check the stack behavior
        ILInstruction il = ILInstruction.Normalize(ilc.Instr);
        if (il.op.Equals(OpCodes.Ldarg)) {
        }

        if (il.op.Equals(OpCodes.Ldarga)) {
          InputArgument inp = new InputArgument();

          if (map.ContainsKey(il.par.iv)) {
            inp = (InputArgument)ret[(int)map[il.par.iv]];
            inp.ByRef = true;
            inp.ReadOnly = false;
            ret[(int)map[il.par.iv]] = inp;
          } else {
            inp = new InputArgument(false, il.par.iv, true, false);
            map[il.par.iv] = ret.Add(inp);
          }
        }

        if (il.op.Equals(OpCodes.Starg)) {
          InputArgument inp = new InputArgument();

```

```
            if (map.ContainsKey(il.par.iv)) {
              inp = (InputArgument)ret[(int)map[il.par.iv]];
              inp.ReadOnly = false;
              ret[(int)map[il.par.iv]] = inp;
            } else {
              inp = new InputArgument(false, il.par.iv, false, false);
              map[il.par.iv] = ret.Add(inp);
            }
          }

          if (il.op.Equals(OpCodes.Ldloc)) {
            InputArgument inp = new InputArgument();

            if (map.ContainsKey(-il.par.iv)) {
              inp = (InputArgument)ret[(int)map[-il.par.iv]];
            } else {
              inp = new InputArgument(true, il.par.iv, false, true);
              map[-il.par.iv] = ret.Add(inp);
            }
          }

          if (il.op.Equals(OpCodes.Ldloca)) {
            InputArgument inp = new InputArgument();

            if (map.ContainsKey(-il.par.iv)) {
              inp = (InputArgument)ret[(int)map[-il.par.iv]];
              inp.ByRef = true;
              inp.ReadOnly = false;
              ret[(int)map[-il.par.iv]] = inp;
            } else {
              inp = new InputArgument(true, il.par.iv, true, false);
              map[-il.par.iv] = ret.Add(inp);
            }
          }

          if (il.op.Equals(OpCodes.Stloc)) {
            InputArgument inp = new InputArgument();

            if (map.ContainsKey(-il.par.iv)) {
              inp = (InputArgument)ret[(int)map[-il.par.iv]];
              inp.ReadOnly = false;
              ret[(int)map[-il.par.iv]] = inp;
            } else {
              inp = new InputArgument(true, il.par.iv, false, false);
              map[-il.par.iv] = ret.Add(inp);
            }
          }
        }
        */
      }

      private static void InternalVisit(AnnotationTree t, TreeVisitor v) {
/*        if (!v(t, -1))
          return;

        for (int i = 0; i < t.children.Length; i++) {
          InternalVisit(t, v);
          v(t, i);
        }
        */
      }

      #region IEnumerable Members

      public System.Collections.IEnumerator GetEnumerator() {
```

```
      return children.GetEnumerator();
    }

    #endregion
  }

  public class Annotation {

    public static void Begin(int idx) {}
    public static void Begin(object[] p, object r, int idx) {}
    public static void End(int idx) {}
    public static void EndEx(int idx) {}
    public static void BeginMeta() {}

    private class IntermediateNode {
      internal AnnotationTree node = new AnnotationTree();
      internal ArrayList children = new ArrayList();
      internal int idx;
    }

    // we get a reference to placeholder methods: the first two differs only
    // on parameters so we have to specify their signatures
    private static MethodInfo BEGIN = typeof(ACS.Annotation).GetMethod("Begin", BindingFlags.Static | BindingFlags.Public,
        null, new Type[1] {typeof(int)}, null);
    private static MethodInfo BEGINEX = typeof(ACS.Annotation).GetMethod("Begin", BindingFlags.Static | BindingFlags.Public,
        null, new Type[3] {typeof(object[]), typeof(object), typeof(int)}, null);
    private static MethodInfo BEGINMETA = typeof(ACS.Annotation).GetMethod("BeginMeta", BindingFlags.Static | BindingFlags.Public);
    private static MethodInfo END = typeof(ACS.Annotation).GetMethod("End", BindingFlags.Static | BindingFlags.Public);
    private static MethodInfo ENDEX  = typeof(ACS.Annotation).GetMethod("EndEx", BindingFlags.Static | BindingFlags.Public);

    public enum ACSMethods {
      BEGIN,
      BEGINEX,
      BEGINMETA,
      NONE
    }

    public static AnnotationTree[] GetCustomAttributes(MethodInfo m) {
      return GetCustomAttributes(m, null);
    }

    public static AnnotationTree[] GetCustomAttributes(MethodInfo m, bool recurse) {
      return GetCustomAttributes(m, recurse ? new Hashtable() : null);
    }

    private static AnnotationTree[] GetCustomAttributes(MethodInfo m, Hashtable env) {
      if (env != null && env.ContainsKey(m))
        return env[m] as AnnotationTree[];

      CodeAttribute[] tmp = (CodeAttribute[])m.GetCustomAttributes(typeof(CodeAttribute), true);
      if (tmp == null || tmp.Length == 0)
        return new AnnotationTree[]{};

      CLIFile.MethodBody mb = CLIFileReader.FindMethod(m).MethodBody;
      ILCursor ilc = mb.ILInstructions;
      Stack s = new Stack();
      s.Push(new IntermediateNode());
      ArrayList[] attribs = new ArrayList[tmp.Length];
      // Assumptions: sequential indexing, duplicates indicate groups
      foreach (CodeAttribute c in tmp) {
        if (attribs[c.ACSIndex] == null)
          attribs[c.ACSIndex] = new ArrayList();
        attribs[c.ACSIndex].Add(c);
      }

      IntermediateNode prev = null;

      while (ilc.Next()) {
        int par;

        if (OpCodes.Ldc_I4_M1.Value <= ilc.Instr.op.Value && ilc.Instr.op.Value <=
             OpCodes.Ldc_I4_8.Value) {
          par = ilc.Instr.op.Value - OpCodes.Ldc_I4_0.Value;
        } else if (ilc.Instr.op.Equals(OpCodes.Ldc_I4_S)) {
          par = ilc.Instr.par.sv;
        } else if (ilc.Instr.op.Equals(OpCodes.Ldc_I4)) {
          par = ilc.Instr.par.iv;
        } else if (env != null && (ilc.Instr.op.Equals(OpCodes.Call) || ilc.Instr.op.
             Equals(OpCodes.Callvirt))) {
          object param = ilc.Instr.ResolveParameter(ilc.FileReader);
          if (param is MethodInfo) {
            AnnotationTree[] ann = GetCustomAttributes(param as MethodInfo, env);
            if (ann != null && ann.Length > 0)
              ((IntermediateNode)s.Peek()).children.AddRange(ann);
          }
          continue;
        } else continue;

        // Check for a Call of Begin and End
        ilc.PushState();
        if (ilc.Next())
          if (ilc.Instr.op.Equals(OpCodes.Call)) {
          object param = ilc.Instr.ResolveParameter(ilc.FileReader);
          ilc.PopState();
          if (param is MethodInfo) {
            if (param.Equals(BEGIN) || param.Equals(BEGINEX)) {
              ilc.Next();
              foreach (CodeAttribute c in attribs[par]) {
                c.beg = ilc.CursorState;
              }
              IntermediateNode node = new IntermediateNode();
              ((IntermediateNode)s.Peek()).children.Add(node.node);
              node.node.node = (CodeAttribute[])attribs[par].ToArray(typeof(
                   CodeAttribute));
              node.idx = par;
              s.Push(node);
            } else if (param.Equals(END)) {
                IntermediateNode n;
                if (prev != null)
                  if (par == prev.idx)
                    n = prev;
                  else
                    n = (IntermediateNode)s.Pop();
                else
                  n = (IntermediateNode)s.Pop();
                if (n.idx != par) { // Look for it: it is an intersection!
                  Stack save = new Stack();
                  save.Push(n);
                  while (s.Count > 0 && ((IntermediateNode)s.Peek()).idx != par) save.
                       Push(s.Pop());
                  if (s.Count == 0) throw new Exception("Internal ACS error!");
                  n = (IntermediateNode)s.Pop();
                  while (save.Count > 0) s.Push(save.Pop());
                }
                foreach (CodeAttribute c in n.node.node)
                  if (c.end == 0)
                    c.end = ilc.Position;
                n.node.children = (AnnotationTree[])n.children.ToArray(typeof(
                     AnnotationTree));
                ilc.Next();
            } else if (param.Equals(ENDEX)) {
                do {
                  ilc.Next();
                } while (!ilc.Instr.op.Equals(OpCodes.Ret));
                IntermediateNode n;
                if (prev == null) {
                  n = (IntermediateNode) s.Pop();
                  prev = n;
                }else if (prev.idx != par) {
                  while ((n = (IntermediateNode)s.Peek()).idx != par)
                    s.Pop();
                  prev = n;
                }else
                  n = prev;
                foreach (CodeAttribute c in n.node.node) {
                  c.end = ilc.Position;
                  c.returns = true;
                }
                n.node.children = (AnnotationTree[])n.children.ToArray(typeof(
                     AnnotationTree));
                if (ilc.EOF && ((IntermediateNode)s.Peek()) == n)
                  s.Pop();
            }
          }
        }
      }
      if (s.Count != 1 || ((IntermediateNode)s.Peek()).node.node != null)
        throw new Exception("Internal error in ACS!");
      AnnotationTree[] ret = (AnnotationTree[])(((IntermediateNode)s.Pop()).children.
           ToArray(typeof(AnnotationTree)));
      if (env != null)
        env[m] = ret;
      return ret;
    }

    public static ACSMethods CheckMethod(ILCursor c) {
      if (c.Instr.op.Equals(OpCodes.Call)) {
        object param = c.Instr.ResolveParameter(c.FileReader);
        if (param is MethodInfo)
          if (param.Equals(BEGIN))
            return ACSMethods.BEGIN;
          else if (param.Equals(BEGINEX))
            return ACSMethods.BEGINEX;
          else if (param.Equals(BEGINMETA))
            return ACSMethods.BEGINMETA;
          else
            return ACSMethods.NONE;
        else
          return ACSMethods.NONE;
      }else
        return ACSMethods.NONE;
    }

  }

  public class TypesConversion {

    private static Hashtable types;

    static TypesConversion() {
      types = new Hashtable();

      types.Add(typeof(char), new Type[]{typeof(ushort),
                                          typeof(int),
                                          typeof(uint),
                                          typeof(long),
                                          typeof(ulong),
                                          typeof(float),
                                          typeof(double),
                                          typeof(decimal)});

      types.Add(typeof(sbyte), new Type[]{typeof(short),
                                           typeof(int),
                                           typeof(long),
                                           typeof(float),
                                           typeof(double),
                                           typeof(decimal)});

      types.Add(typeof(short), new Type[]{typeof(int),
                                           typeof(long),
                                           typeof(float),
                                           typeof(double),
                                           typeof(decimal)});

      types.Add(typeof(int), new Type[]{typeof(long),
                                         typeof(float),
                                         typeof(double),
                                         typeof(decimal)});

      types.Add(typeof(long), new Type[]{typeof(decimal),
                                          typeof(float),
                                          typeof(double)});

      types.Add(typeof(float), new Type[]{typeof(double)});

      types.Add(typeof(double), new Type[]{});

      types.Add(typeof(byte), new Type[]{typeof(ushort),
                                           typeof(uint),
                                           typeof(ulong),
                                           typeof(short),
                                           typeof(int),
                                           typeof(long),
                                           typeof(float),
                                           typeof(double),
                                           typeof(decimal)});

      types.Add(typeof(ushort), new Type[]{typeof(uint),
                                           typeof(ulong),
                                           typeof(int),
                                           typeof(long),
                                           typeof(float),
                                           typeof(double),
                                           typeof(decimal)});

      types.Add(typeof(uint), new Type[]{typeof(ulong),
                                           typeof(long),
                                           typeof(float),
                                           typeof(double),
                                           typeof(decimal)});


      types.Add(typeof(ulong), new Type[]{typeof(float),
                                          typeof(double),
                                              typeof(decimal)});

      types.Add(typeof(decimal), new Type[]{});
    }

    public static bool CanConvert(Type from, Type to) {
      if (types.ContainsKey(from))
        foreach (Type t in types[from] as Type[])
          if (t.Equals(to))
            return true;
      return false;
    }

    private static MethodInfo ExistsCustomCast(Type from, Type to, bool _explicit) {
      String name = "op_Implicit";
      if (_explicit)
        name = "op_Explicit";
      MethodInfo cast = from.GetMethod(name, new Type[]{from});
      if (cast != null)
        if (cast.ReturnType.Equals(to))
          return cast;
      return null;
    }

    public static MethodInfo ExistsCustomImplicitCast(Type from, Type to) {
      return ExistsCustomCast(from, to, false);
    }

    public static MethodInfo ExistsCustomExplicitCast(Type from, Type to) {
      return ExistsCustomCast(from, to, true);
    }

  }

}
```

7.1.2 Code's new constructors

Here follows a part of the CodeBricks source code which contains the definition of the Code class' constructors implemented in this thesis.

Listing 7.2: Code's new constructors

```
public Code(MethodInfo m, AnnotationTree annotation, CodeAttribute.Operations operation,
     Code code) :
  this(m, new Operation[1]{ new Operation(annotation, operation, code)}) { }

public Code(MethodInfo m, Operation operation) :
  this(m, new Operation[1] { operation }) { }

public Code(MethodInfo m, AnnotationTree annotation, CodeAttribute.Operations operation)
     :
  this(m, annotation, operation, null) { }

public Code(MethodInfo m, Operation[] operations) : this(m)
{
  info.retType = typeof(void);
  this.operations = operations;
  ArrayList newlocals = new ArrayList();
  newlocals.AddRange(info.locals);
  foreach (Operation op in operations)
  {
  op.fragmentLocals = op.annotation.AnnotationLocals(m);
  ParameterInfo[] realparameters = op.annotation.AnnotationRealParameters(m);
  op.fragmentRealParameters = realparameters.Length;
  LocalVariableInfo[] lvi = op.annotation.AnnotationParameters(m);
  op.fragmentParameters = lvi.Length;
  CodeParameterInfo[] parameters = null;
  if (op.operation == CodeAttribute.Operations.EXTRUDE)
  {
    parameters = new CodeParameterInfo[op.fragmentParameters + realparameters.Length];
    for (int i = 0; i < parameters.Length; i++)
    if (i >= realparameters.Length)
      parameters[i] = new CodeParameterInfo(Code.free, lvi[i - realparameters.Length].
           LocalType,i);
    else
      parameters[i] = new CodeParameterInfo(Code.free, realparameters[i].ParameterType,i
           );
    ((MethodReader)this.Builder).info.parameters = parameters;
    info.parameters = parameters;
    if (op.fragment.Returns)
      info.retType = m.ReturnType;
    break;
  }
  else if (op.operation == CodeAttribute.Operations.INCLUDE_AFTER ||
       op.operation == CodeAttribute.Operations.INCLUDE_BEFORE)
  {
    if (op.code == null)
      throw new ArgumentException("Code parameter cannot be null when including !");
    else if (op.code.Arity > 0 && op.code.Arity != op.fragmentParameters && op.constants
         .Length == 0)
      throw new OperationException("Annotation parameters count ("
                                   + op.fragmentParameters + ") differs from
                                        inclusion parameters count ("
                                   + op.code.Arity + ") !", op);
    else
    {
      info.maxStack += op.code.MaxStack;
                        op.reloc = newlocals.Count;
                        foreach (CodeLocalInfo cli in op.code.info.locals)
                            newlocals.Add(new CodeLocalInfo(cli.Type));
    }
  }else if (op.operation == CodeAttribute.Operations.COPY_BEFORE ||
        op.operation == CodeAttribute.Operations.COPY_AFTER)
                {
                }
        }
        info.locals = newlocals.ToArray(typeof(CodeLocalInfo)) as CodeLocalInfo[];
}
```

7.1.3 Code's new "Code.FillMethodBody"

The source code presented here shows the implementation of the whole *Code.FillMethodBody()* method. The method implements the algorithms discussed in this thesis. In particular, it is possible to see the implementation of the operations on annotations: *extrusion*, *removal* and *inclusion*.

Listing 7.3: Code's new "FillMethodBody"

```
    public void FillMethodBody(ILGenerator ilg)
    {
        // Declare locals
        LocalBuilder[] locals = new LocalBuilder[info.locals.Length];
        for (int i = 0; i < info.locals.Length; i++)
            locals[i] = ilg.DeclareLocal(info.locals[i].Type);
        //mtb.CreateMethodBody(ils.ILStream, ils.Length);

        AssemblyILStore ils = new AssemblyILStore(this);
        ILCursor c = ils.ILCursor;
        c.TrackTargets();
        Hashtable labels = new Hashtable(c.Targets.Length);
        Hashtable ilabels = new Hashtable();
        Hashtable poslabels = new Hashtable(c.Targets.Length);
        Hashtable iposlabels = new Hashtable();
        foreach (Target t in c.Targets)
        {
            labels.Add(t, ilg.DefineLabel());
            poslabels.Add(t.Position, labels[t]);
        }

        ArrayList jumps = new ArrayList();
        if (operations != null)
        {
            foreach (Operation op in operations)
                if (op.operation == CodeAttribute.Operations.EXTRUDE)
                {
                    if (op.fragmentRealParameters > 0)
                    {
                        for (int i = op.fragmentRealParameters; i < info.parameters.
                             Length; i++)
                            switch (i)
                            {
                                case 1:
                                    ilg.Emit(OpCodes.Ldarg_1);
                                    ilg.Emit(OpCodes.Stloc, i - op.
                                         fragmentRealParameters);
                                    break;
                                case 2:
                                    ilg.Emit(OpCodes.Ldarg_2);
                                    ilg.Emit(OpCodes.Stloc, i - op.
                                         fragmentRealParameters);
                                    break;
                                case 3:
                                    ilg.Emit(OpCodes.Ldarg_3);
                                    ilg.Emit(OpCodes.Stloc, i - op.
                                         fragmentRealParameters);
                                    break;
                                default:
                                    if (i < 256)
                                    {
                                        ilg.Emit(OpCodes.Ldarg_S, (byte)i);
                                        ilg.Emit(OpCodes.Stloc_S, (byte)(i - op.
                                             fragmentRealParameters));
                                    }
                                    else
                                    {
                                        ilg.Emit(OpCodes.Ldarg, i);
                                        ilg.Emit(OpCodes.Stloc, i - op.
                                             fragmentRealParameters);
                                    }
                                    break;
                            }
                    }
                    else
                    {
                        for (int i = 0; i < info.parameters.Length; i++)
```

```
                            switch (i)
                            {
                                case 0:
                                    ilg.Emit(OpCodes.Ldarg_0);
                                    ilg.Emit(OpCodes.Stloc_0);
                                    break;
                                case 1:
                                    ilg.Emit(OpCodes.Ldarg_1);
                                    ilg.Emit(OpCodes.Stloc_1);
                                    break;
                                case 2:
                                    ilg.Emit(OpCodes.Ldarg_2);
                                    ilg.Emit(OpCodes.Stloc_2);
                                    break;
                                case 3:
                                    ilg.Emit(OpCodes.Ldarg_3);
                                    ilg.Emit(OpCodes.Stloc_3);
                                    break;
                                default:
                                    if (i < 256)
                                    {
                                        ilg.Emit(OpCodes.Ldarg_S, (byte)i);
                                        ilg.Emit(OpCodes.Stloc_S, (byte)i);
                                    }
                                    else
                                    {
                                        ilg.Emit(OpCodes.Ldarg, i);
                                        ilg.Emit(OpCodes.Stloc, i);
                                    }
                                    break;
                            }
                        }
                    }
        }

        bool done = true;
        int current = 0;
        Operation operation = null;
        int b = 0;
        int brTarget = 0;
        long delEnd = -2;
        int idx = -1;
        int instnum = 0;
        while (c.Next())
        {

            if (c.Label != null)
            {
                if (operation != null && operation.including)
                    ilg.MarkLabel((Label)ilabels[c.Label]);
                else
                    ilg.MarkLabel((Label)labels[c.Label]);
            }

            if (operations != null)
                if (done)
                {
                    Operation oldOp = operation;
                    if (current < operations.Length)
                    {
                        operation = operations[current];
                    }
                    if (oldOp != null)
                    {
                        if (operation.fragment.ACSIndex == oldOp.fragment.ACSIndex)
                        {
                            operation.bindings = oldOp.bindings;
                            if (operation.operation == CodeAttribute.Operations.INCLUDE_BEFORE ||
                                operation.operation == CodeAttribute.Operations.INCLUDE_AFTER ||
                                operation.operation == CodeAttribute.Operations.COPY_BEFORE ||
                                operation.operation == CodeAttribute.Operations.COPY_AFTER)
                                c.PopState();
                        }
                    }
                    if (current < operations.Length)
                        current++;
                    else
                        operation = null;
                    done = false;
                    b = 0;
                }

            if (operation != null)
            {
                ILInstruction ni = ILInstruction.Normalize(c.Instr);

                Annotation.ACSMethods acsMethod = Annotation.CheckMethod(c);
                if (acsMethod == Annotation.ACSMethods.BEGINMETA)
                {
                    foreach (Operation o in operations)
                        if (o.fragment.ACSIndex - 1 == idx)
                        {
                            o.fragment.MetaPosition = c.Position;
                            break;
                        }
                }
                else if (acsMethod == Annotation.ACSMethods.BEGIN ||
                                    acsMethod == Annotation.ACSMethods.BEGINEX)
                    idx++;

                if (acsMethod != Annotation.ACSMethods.NONE)
                    if (c.Label != null)
                    {
                        for (int no = current; no < operations.Length; no++)
                            if (operations[no].operation == CodeAttribute.Operations.REMOVE)
                            {
                                ilg.Emit(OpCodes.Nop);
                                try
                                {
                                    ilg.MarkLabel((Label)labels[c.Label]);
                                }
                                catch (Exception e) { }
                                break;
                            }
                    }

                if (!operation.including)
                {

                    if (operation.fragment.MetaPosition != -1)
                        if (c.Position < operation.fragment.BeginPosition)
                        {
                            if (c.Position > operation.fragment.MetaPosition)
                            {

                                if (operation.bindings == null)
                                {
                                    if (ni.op.Equals(OpCodes.Ldc_I4))
                                        if (ni.par.iv != operation.fragmentParameters)
                                            throw new OperationException(
                                                String.Format("Wrong annotation bindings declaration ! ({0} != {1})",
                                                                ni.par.iv, operation.fragmentParameters),
                                                                operation);
                                        else
                                            operation.bindings = new int[ni.par.iv];
                                }

                                if (operation.operation == CodeAttribute.Operations.INCLUDE_BEFORE ||
                                operation.operation == CodeAttribute.Operations.INCLUDE_AFTER)
                                    if (ni.op.Equals(OpCodes.Ldloc))
                                    {
                                        if (b < operation.fragmentParameters)
                                            operation.bindings[b++] = ni.par.iv;
                                        else
                                        {
                                            operation.annotationReturnLocal = ni.par.iv;
                                            operation.annotationReturnType = locals[ni.par.iv].LocalType;
                                        }
                                    }
                            }
                        }
                }

                if (operation.including)
                {
                    if (ni.op.Equals(OpCodes.Ldarg))
                    {
                        if (operation.constants.Length > 0)
                        {
                            Console.WriteLine("hack {0} {1}", ni.par.iv, operation.constants.Length);
                            ilg.Emit(OpCodes.Ldc_I4, operation.constants[ni.par.iv]);
                        }
                        else
                        {
                            int loc;
                            try
                            {
                                loc = operation.bindings[ni.par.iv];
                            }
                            catch (NullReferenceException e)
                            {
                                loc = ni.par.iv;
                            }

                            //int loc = operation.bindings[ni.par.iv];
                            if (!EmitCastCode(ilg, this.info.locals[loc].Type, operation.code.info.parameters[ni.par.iv].ParameterType))
                                throw new OperationException("Incompatible types !",
                                    operation, new InvalidCastException("Local variable type ("
                                                                                                +
this
.
info.locals[loc].Type
+ ") is not compatible with inclusion's parameter type ("
+ operation.code.info.parameters[ni.par.iv].ParameterType + ")"));
Console.WriteLine("{2}, {0} {1}", ni.par.iv, loc, ni.op);
if (loc == 0)
    ilg.Emit(OpCodes.Ldloc_0);
else if (loc == 1)
    ilg.Emit(OpCodes.Ldloc_1);
else if (loc == 2)
```

```
                        ilg.Emit(OpCodes.Ldloc_2);
                    else if (loc == 3)
                        ilg.Emit(OpCodes.Ldloc_3);
                    else if (loc <= 255)
                        ilg.Emit(OpCodes.Ldloc_S, (sbyte)loc);
                    else
                        ilg.Emit(OpCodes.Ldloc, loc);
                }
                continue;
            }
            else if (ni.op.Equals(OpCodes.Ldarga))
            {
                int loc = operation.bindings[ni.par.iv];
                Console.WriteLine("{2}, {0} {1}", ni.par.iv, loc, ni.op);
                if (loc <= 255)
                    ilg.Emit(OpCodes.Ldloca_S, (sbyte)loc);
                else
                    ilg.Emit(OpCodes.Ldloca, loc);
                continue;
            }
            else if (ni.op.Equals(OpCodes.Ldloc))
            {
                int loc = ni.par.iv + operation.reloc;
                Console.WriteLine("{2}, {0} {1}", ni.par.iv, loc, ni.op);
                if (loc == 0)
                    ilg.Emit(OpCodes.Ldloc_0);
                else if (loc == 1)
                    ilg.Emit(OpCodes.Ldloc_1);
                else if (loc == 2)
                    ilg.Emit(OpCodes.Ldloc_2);
                else if (loc == 3)
                    ilg.Emit(OpCodes.Ldloc_3);
                else if (loc <= 255)
                    ilg.Emit(OpCodes.Ldloc_S, (sbyte)loc);
                else
                    ilg.Emit(OpCodes.Ldloc, loc);
                continue;
            }
            else if (ni.op.Equals(OpCodes.Ldloca))
            {
                int loc = ni.par.iv + operation.reloc;
                Console.WriteLine("{2}, {0} {1}", ni.par.iv, loc, ni.op);
                if (loc <= 255)
                    ilg.Emit(OpCodes.Ldloca_S, (sbyte)loc);
                else
                    ilg.Emit(OpCodes.Ldloca, loc);
                continue;
            }
            else if (ni.op.Equals(OpCodes.Stloc))
            {
                int loc = ni.par.iv + operation.reloc;
                Console.WriteLine("{2}, {0} {1}", ni.par.iv, loc, ni.op);
                if (loc == 0)
                    ilg.Emit(OpCodes.Stloc_0);
                else if (loc == 1)
                    ilg.Emit(OpCodes.Stloc_1);
                else if (loc == 2)
                    ilg.Emit(OpCodes.Stloc_2);
                else if (loc == 3)
                    ilg.Emit(OpCodes.Stloc_3);
                else if (loc <= 255)
                    ilg.Emit(OpCodes.Stloc_S, (sbyte)loc);
                else
                    ilg.Emit(OpCodes.Stloc, loc);
                continue;
```

```
                    }
                    else if (ni.op.OperandType == OperandType.InlineBrTarget)
                    {
                        Console.WriteLine("{0}, {1}", c.Instr.op, ni.par.iv);
                        ilg.Emit(c.Instr.op, (Label)iposlabels[(long)(c.Position + 5 +
                            ni.par.iv)]);
                        continue;
                    }
                    else if (ni.op.OperandType == OperandType.ShortInlineBrTarget)
                    {
                        Console.WriteLine("{0}, {1}", c.Instr.op, ni.par.sbv);
                        ilg.Emit(c.Instr.op, (Label)iposlabels[(long)(c.Position + 5 +
                            ni.par.sbv)]);
                        continue;
                    }
                    else if (ni.op.Equals(OpCodes.Ret))
                    {
                        if (operation.code.ReturnType != typeof(void))
                            if (operation.annotationReturnType == null)
                                ilg.Emit(OpCodes.Pop);
                            else
                                if (EmitCastCode(ilg, operation.code.ReturnType,
                                    operation.annotationReturnType))
                                {
                                    Console.WriteLine("Stloc {0}", operation.
                                        annotationReturnLocal);
                                    ilg.Emit(OpCodes.Stloc, operation.
                                        annotationReturnLocal);
                                }
                                else
                                    throw new OperationException("Incompatible signature
                                         !", operation, new InvalidCastException("Local
                                        variable type ("
                                                                                    +
                                                                                        operation
                                                                                        .
                                                                                        annotationReturn
                                                                                    +
                                                                                        "
                                                                                        )
                                                                                        is
                                                                                        not
                                                                                        compatible
                                                                                        with
                                                                                        included
                                                                                        return
                                                                                        type
                                                                                        (
                                                                                        "
                                                                                    +
                                                                                        operation
                                                                                        .
```

```
code
.
ReturnTy

+

"
)
"
)
)
;
                        }

                        if (c.EOF)
                        {
                            operation.including = false;
                            c = operation.tmpc;
                            operation.tmpc = null;
                            done = true;
                        }

                    }
                    else if (operation.copying)
                    {

                        if (c.Position == operation.copy.EndPosition)
                        {
                            operation.copying = false;
                            done = true;
                            continue;
                        }

                    }
                    else if (operation.operation == CodeAttribute.Operations.REMOVE)
                    {

                        if (c.Position >= operation.fragmentPosition)
                            if (c.Position <= operation.fragment.EndPosition + 1)
                                continue;
                            else if (c.Position == operation.fragment.EndPosition + 1)
                            {
                                done = true;
                                continue;
                            }

                    }
                    else if ((operation.operation == CodeAttribute.Operations.INCLUDE_BEFORE
                             && c.Position == operation.fragmentPosition + 1) ||
                                     (operation.operation == CodeAttribute.Operations.
                                          INCLUDE_AFTER
                                      && c.Position == operation.fragment.EndPosition))
                    {

                        c.PushState();
                        delEnd = operation.fragment.EndPosition;
                        operation.tmpc = c;
                        c = new AssemblyILStore(operation.code).ILCursor;

                        c.TrackTargets();
                        ilabels.Clear();
                        iposlabels.Clear();
                        foreach (Target t in c.Targets)
                        {
```

```
                        ilabels.Add(t, ilg.DefineLabel());
                        if (!iposlabels.Contains(t.Position))
                            iposlabels.Add(t.Position, ilabels[t]);
                    }
                    /*
                            while (c.Next())
                            {
                                Console.WriteLine("skipping {0}..{1} {2}", c.Position, operation.code.extrusionBegin, c.Instr.op);
                                if (c.Position >= operation.code.extrusionBegin)
                                    break;
                            }
                    */
                    operation.including = true;
                    continue;

                }
                else if ((operation.operation == CodeAttribute.Operations.COPY_BEFORE
                         && c.Position == operation.fragmentPosition) ||
                         (operation.operation == CodeAttribute.Operations.COPY_AFTER
                         && c.Position == operation.fragment.EndPosition) && operation.copying == false)
                {

                    c.PushState();
                    //delEnd = operation.fragment.EndPosition;
                    c.Reset();
                    c.Next();
                    while (c.Position <= operation.copy.BeginPosition)
                        c.Next();
                    operation.copying = true;
                    //continue;

                }
                else if (operation.operation == CodeAttribute.Operations.EXTRUDE)
                {

                    if (c.Position >= operation.fragment.EndPosition)
                    {
                        if (c.Position > brTarget)
                        {
                            if (!ni.op.Equals(OpCodes.Ret))
                                ilg.Emit(OpCodes.Ret);
                            break;
                        }
                    }
                    else if (c.Position <= operation.fragmentPosition + 1)
                    {
                        continue;
                    }
                    else
                    {
                        if (ni.op.Equals(OpCodes.Br))
                            if ((c.Position + ni.par.iv) >= operation.fragment.EndPosition)
                                throw new OperationException("The annotation contains a jump outside of it!", operation);
                    }
                }

                if (!operation.including)
                {
                    if (operation.fragment.MetaPosition != -1)
                    {
                        if (c.Position >= operation.fragment.MetaPosition)
```

```
                            if (c.Position <= operation.fragment.BeginPosition)
                                continue;
                    }
                    else
                    {
                        if (c.Position >= operation.fragmentPosition)
                            if (c.Position <= operation.fragment.BeginPosition)
                                continue;
                    }
                }

            }

            if ((operation != null && !operation.including)
                    || operation == null)
                if (c.Position == delEnd ||
                        c.Position == delEnd + 1)
                    continue;

            ++instnum;
            /*
    if (c.Label != null)
        ilg.MarkLabel((Label)labels[c.Label]);
            */
            /*
                    if (c.Label != null) {
                      if (!labels.ContainsKey(c.Label))
                        labels.Add(c.Label, ilg.DefineLabel());
                      ilg.MarkLabel((Label)labels[c.Label]);
                    }
            */
            object par;
            if (operation != null)
                if (operation.including)
                    par = c.Instr.ResolveParameter(operation.code.info.AssemblyReader);
                else
                    par = c.Instr.ResolveParameter(this.info.AssemblyReader);
            else
                par = c.Instr.ResolveParameter(this.info.AssemblyReader);
            /*
            if (par is Target) {
              if (!labels.ContainsKey(par))
                labels.Add(par, ilg.DefineLabel());
              par = labels[par];
            } else if (par is Target[]) {
              Target[] t = par as Target[];
              Label[] a = new Label[t.Length];
              for (int i = 0; i < a.Length; i++) {
                if (!labels.ContainsKey(t[i]))
                  labels.Add(t[i], ilg.DefineLabel());
                a[i] = (Label)labels[t[i]];
              }
              par = a;
            }
            */

#if DEBUG
        Console.WriteLine(c.Position + ": " + c.Instr.op + " " + par);
#endif
            switch (c.Instr.op.OperandType)
            {
                case OperandType.InlineBrTarget:
                    brTarget = c.Instr.par.iv;
                    ilg.Emit(c.Instr.op, (Label)poslabels[(long)(c.Position + 5 + c.
                         Instr.par.iv)]);
                        break;
                    case OperandType.ShortInlineBrTarget:
                        brTarget = c.Instr.par.sbv;
                        ilg.Emit(c.Instr.op, (Label)poslabels[(long)(c.Position + 5 + c.
                            Instr.par.sbv)]);
                        break;
                    case OperandType.InlineI:
                        ilg.Emit(c.Instr.op, c.Instr.par.iv);
                        break;
                    case OperandType.InlineField:
                        ilg.Emit(c.Instr.op, (FieldInfo)par);
                        break;
                    case OperandType.InlineI8:
                        ilg.Emit(c.Instr.op, c.Instr.par.lv);
                        break;
                    case OperandType.InlineMethod:
                        if (par is MethodInfo)
                            ilg.Emit(c.Instr.op, (MethodInfo)par);
                        else
                            ilg.Emit(c.Instr.op, (ConstructorInfo)par);
                        break;
                    case OperandType.InlineNone:
                        ilg.Emit(c.Instr.op);
                        break;
                    /*
                             case OperandType.InlinePhi:
                               throw new Exception("Unsupported");
                    */
                    case OperandType.InlineR:
                        ilg.Emit(c.Instr.op, c.Instr.par.dv);
                        break;
                    case OperandType.InlineSig:
                        throw new Exception("Unsupported");
                    case OperandType.InlineString:
                        ilg.Emit(c.Instr.op, (string)par);
                        break;
                    case OperandType.InlineSwitch:
                        {
                            Label[] lab = new Label[c.Instr.par.iv];
                            object[] sw = (object[])par;
                            for (int i = 0; i < lab.Length; i++)
                                lab[i] = (Label)labels[sw[i]];
                            ilg.Emit(c.Instr.op, lab);
                            break;
                        }
                    case OperandType.InlineTok:
                        if (par is Type) ilg.Emit(c.Instr.op, (Type)par);
                        else if (par is MethodInfo) ilg.Emit(c.Instr.op, (MethodInfo)par);
                        else if (par is FieldInfo) ilg.Emit(c.Instr.op, (FieldInfo)par);
                        else throw new Exception("Unsupported");
                        break;
                    case OperandType.InlineType:
                        ilg.Emit(c.Instr.op, (Type)par);
                        break;
                    case OperandType.InlineVar:
                        ilg.Emit(c.Instr.op, c.Instr.par.sv);
                        break;
                    case OperandType.ShortInlineI:
                        ilg.Emit(c.Instr.op, c.Instr.par.sbv);
                        break;
                    case OperandType.ShortInlineR:
                        ilg.Emit(c.Instr.op, c.Instr.par.fv);
                        break;
                    case OperandType.ShortInlineVar:
                        ilg.Emit(c.Instr.op, c.Instr.par.bv);
                        break;
                    }
                }
#if DEBUG
            Console.WriteLine("Inst. num: {0}", instnum);
#endif
        }
```

7.1.4 Unix Memory Mapping

The following is the source code which is responsible for exposing to the Mono environment the Linux OS' low level memory mapping functions; these functions were needed to port the CLIFileRW and CodeBricks library on the Linux/Mono platform.

Listing 7.4: Unix Memory Mapping

```
/////////////////////////////////////////////////////////////////////////////
//  Author: Nicola Giordani (nicola@dinosoft.it)
//
//  This is to support memory mapping of files on the unix platform
//  It uses the mmap(), msync() and munmap() functions provided by libm and
//  the getpagesize() function provided by libc to get os' page allocation size
/////////////////////////////////////////////////////////////////////////////

#if UNIX

using System;
using System.Runtime.InteropServices;

namespace MappedView {

  public enum MMapFlags {
    PROT_READ = 0x1,     /* Page can be read.  */
    PROT_WRITE  =   0x2,     /* Page can be written.  */
    PROT_EXEC   =   0x4,     /* Page can be executed.  */
    /* Sharing types (must choose one and only one of these).  */
    MAP_SHARED  =   0x01,    /* Share changes.  */
    MAP_PRIVATE = 0x02,    /* Changes are private.  */
    MS_ASYNC  = 1,       /* Sync memory asynchronously.  */
    MS_SYNC = 4        /* Synchronous memory sync.  */
  }

  public unsafe class UnixMapping {

    public static readonly int AllocationGranularity = 0;
    public static readonly long MaskBase = 0;
    public static readonly long MaskOffset = 0;

    static UnixMapping() {
      AllocationGranularity = (int)Math.Log(getpagesize(), 2);
    // Assume a power of 2!
    MaskBase = (1 << AllocationGranularity) - 1;
    MaskOffset = ~MaskBase;
    }

  [DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern int fopen(char* file, char* m);

  [DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
```

```
    public static unsafe extern int perror(IntPtr s);

  [DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern int memcpy(IntPtr dest,
                           IntPtr src,
                                          int n);

  [DllImport("libc", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern int getpagesize();

  [DllImport("libm", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern IntPtr mmap(IntPtr start,
                                           int length,
                                           int prot,
                                           int flags,
                                           int fd,
                                           int offset);

  [DllImport("libm", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern int munmap(IntPtr start,
                                          int length);

  [DllImport("libm", CharSet=CharSet.Auto, SetLastError=true)]
    public static unsafe extern int msync(IntPtr start,
                                         int length,
                                         int flags);

  }

}

#endif
```

7.2 Examples sources

7.2.1 From Interactive To Batch

Listing 7.5: IOAttribute

```
using System;
using ACS;

public class IOAttribute : CodeAttribute
{

  public enum Redirection
  {
    CONSOLE,
    FILE,
    LOG
  }

  public Redirection Redirect;

}

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class InputAttribute : IOAttribute
{

}

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class OutputAttribute : IOAttribute
{

}
```

Listing 7.6: Interactive

```
using System;
using ACS;
using System.IO;
using System.Text;
using System.Security.Cryptography;

public class Interactive {

  public static HashAlgorithm hash = SHA512.Create();
  public static string EXIT = "exit";

  public static string Encrypt(string str)
  {
    MemoryStream ms = new MemoryStream();
    byte[] buf = Encoding.UTF8.GetBytes(str);
    ms.Write(buf, 0, buf.Length);
    Decoder dec = Encoding.UTF8.GetDecoder();
    int n = dec.GetCharCount(buf, 0, buf.Length);
    char[] chars = new char[n];
    dec.GetChars(hash.ComputeHash(ms), 0, buf.Length, chars, 0);
    return new String(chars);
  }

  public static void Main() {
    string input = null;
    do {
      [Input(*input, &input)]
      {
        Console.WriteLine("Type a string to encrypt or 'exit' to quit: ");
        input = Console.ReadLine();
      }

      DateTime start = DateTime.Now;
      string output = Encrypt(input);
      TimeSpan timeElapsed = DateTime.Now - start;
      string elapsed = timeElapsed.ToString();

      [Output(Redirect = OutputAttribute.Redirection.LOG, *elapsed)]
      {
        Console.WriteLine(String.Format("Elapsed time: {0}", elapsed));
      }

      [Output(Redirect = OutputAttribute.Redirection.FILE, *output)] {
        Console.WriteLine(String.Format("Encrypted string is '{0}'", output));
      }
    } while (!input.Equals(EXIT));
  }

}
```

Listing 7.7: Batch

```
using System;
using System.Reflection;
using System.Collections;
using System.Text;
using System.IO;
using ACS;

public class Batch {

  public static string GenerateString()
  {
    Random rand = new Random();
    string s = rand.Next(1000).ToString();
    if (s.Equals("999"))
      s = Interactive.EXIT;
    return s;
  }

  public static void Log(string s)
  {
    StreamWriter log = new StreamWriter(new FileStream("log.txt", FileMode.Append));
    log.WriteLine(s);
    log.Close();
  }

  public static void Output(string s)
  {
    StreamWriter output = new StreamWriter(new FileStream("output.sha512", FileMode.
        Append));
    output.WriteLine(s);
    output.Close();
  }

  delegate void Void();
  delegate void Str(string i);

  public static void Main(string[] args)
  {
    Assembly asm = Assembly.GetExecutingAssembly();
    Type program = asm.GetType("Interactive");
    MethodInfo main = program.GetMethod("Main");
    Code In = new Code(typeof(Batch).GetMethod("GenerateString"));
    Code Out = new Code(typeof(Batch).GetMethod("Output"));
    Code Log = new Code(typeof(Batch).GetMethod("Log"));
    AnnotationTree[] io = Annotation.GetCustomAttributes(main);
    ArrayList operations = new ArrayList();

    foreach (AnnotationTree annot in io)
    {
      if (annot.Node[0] is InputAttribute)
      {
        operations.Add(new Operation(annot,
          CodeAttribute.Operations.INCLUDE_BEFORE, In));
        operations.Add(new Operation(annot, CodeAttribute.Operations.REMOVE));
      }else if (annot.Node[0] is OutputAttribute)
        if (((IOAttribute)annot.Node[0]).Redirect == IOAttribute.Redirection.FILE)
        {
          operations.Add(new Operation(annot,
              CodeAttribute.Operations.INCLUDE_BEFORE, Out));
          operations.Add(new Operation(annot, CodeAttribute.Operations.REMOVE));
        }else if (((IOAttribute)annot.Node[0]).Redirect == IOAttribute.Redirection.LOG)
        {
          operations.Add(new Operation(annot,
              CodeAttribute.Operations.INCLUDE_BEFORE, Log));
```

```
            operations.Add(new Operation(annot, CodeAttribute.Operations.REMOVE));
        }
    }

    Console.WriteLine("operations # = {0}", operations.Count);
    Code batch = new Code(main,
        (Operation[])operations.ToArray(typeof(Operation)));
    Delegate batchMain = batch.MakeDelegate(typeof(Void), "boh.dll");
    batchMain.DynamicInvoke(null);
  }

}
```

7.2.2 Runtime AOP

Listing 7.8: AOPAttribute

```
using System;
using ACS;

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class AOPAttribute : CodeAttribute {

  public enum Advices
  {
    PRE,
    POST,
    AROUND,
    SWEEP
  }

  public Advices Advice;
  public String Id;
  public String Group;

}
```

Listing 7.9: RuntimeAOP

```
using System;
using System.Collections;
using System.Threading;
using System.Reflection;
using ACS;
using Gtk;

public class RuntimeAOP
{

  public static readonly string HARD_CODED = "Hard-coded";
  public static readonly string ADD_BEFORE = "Add advice before ";
  public static readonly string ADD_AFTER = "Add advice after ";
  public static readonly string ADD_AROUND = "Add advice around ";
  public static readonly string REMOVE_GROUP = "Remove group ";

  public static bool alive;
  public static Window app;
  public delegate void Handler();
  public static Handler handler;
  public static Handler AOPHandler;
  public static Thread thread;
  public static Random rnd = new Random();

  public static Button on;
  public static Button off;
  public static ComboBox choice;

  public static void Main()
  {
    Application.Init();
    new RuntimeAOP();
    Application.Run();
    thread.Join();
  }

  public RuntimeAOP()
  {
      app = new Window("Runtime AOP");
    HBox hb = new HBox(true, 2);
    on = new Button("Turn on AOP");
    on.Sensitive = false;
    on.Clicked += new EventHandler(AOPon);
    off = new Button("Turn off AOP");
    off.Clicked += new EventHandler(AOPoff);
    off.Sensitive = false;
    hb.PackStart(on);
    hb.PackEnd(off);
    VBox vb = new VBox(true, 3);
    vb.PackStart(hb);

    AnnotationTree[] annotations = Annotation.GetCustomAttributes(typeof(RuntimeAOP).
         GetMethod("DoSomething"));
    ArrayList items = new ArrayList();
    items.Add(HARD_CODED);
    foreach (AnnotationTree annot in annotations)
      if (annot.Node[0] is AOPAttribute)
      {
        AOPAttribute aopa = annot.Node[0] as AOPAttribute;
        if (aopa.Group != null)
        {
          String s = REMOVE_GROUP + aopa.Group;
          if (!items.Contains(s))
            items.Add(s);
        }
        if (aopa.Id != null)
        {
          items.Add(ADD_BEFORE + aopa.Id);
          items.Add(ADD_AFTER + aopa.Id);
          items.Add(ADD_AROUND + aopa.Id);
        }
      }
    choice = new ComboBox((string[])items.ToArray(typeof(string)));
    choice.Changed += new EventHandler(setAOP);
    choice.Active = 0;
    vb.PackStart(new Label("Choose an AOP strategy..."));
    vb.PackEnd(choice);
    app.Add(vb);
    app.SetDefaultSize(200, 75);
    app.DeleteEvent += new DeleteEventHandler(OnAppDelete);
    app.ShowAll();

    ThreadStart ts = new ThreadStart(RuntimeAOP.Run);
    thread = new Thread(ts);
    alive = true;
    handler = RuntimeAOP.DoSomething;
    thread.Start();
  }

  static void AOPon(object obj, EventArgs args)
  {
    handler = AOPHandler;
    on.Sensitive = false;
    off.Sensitive = true;
  }

  static void AOPoff(object obj, EventArgs args)
  {
    handler = RuntimeAOP.DoSomething;
    on.Sensitive = true;
    off.Sensitive = false;
  }

  static void setAOP(object obj, EventArgs args)
  {
    on.Sensitive = true;
    Code advice = new Code(typeof(RuntimeAOP).GetMethod("Advice"));
    MethodInfo mi = ((Handler)RuntimeAOP.DoSomething).Method;
    AnnotationTree[] annotations = Annotation.GetCustomAttributes(mi);
    ArrayList operations = new ArrayList();
    foreach (AnnotationTree annot in annotations)
    {
      if (annot.Node[0] is AOPAttribute)
      {
        AOPAttribute aopa = annot.Node[0] as AOPAttribute;
        if (choice.ActiveText.Equals(HARD_CODED))
          switch(aopa.Advice)
          {
            case AOPAttribute.Advices.PRE:
              operations.Add(new Operation(annot,
                CodeAttribute.Operations.INCLUDE_BEFORE, advice));
              break;
            case AOPAttribute.Advices.POST:
              operations.Add(new Operation(annot,
                CodeAttribute.Operations.INCLUDE_AFTER, advice));
              break;
            case AOPAttribute.Advices.AROUND:
              operations.Add(new Operation(annot,
                CodeAttribute.Operations.INCLUDE_BEFORE, advice));
              operations.Add(new Operation(annot,
                CodeAttribute.Operations.INCLUDE_AFTER, advice));
              break;
            case AOPAttribute.Advices.SWEEP:
              operations.Add(new Operation(annot,
                CodeAttribute.Operations.REMOVE));
              break;
          }
        else if (choice.ActiveText.Equals(ADD_BEFORE + aopa.Id))
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.INCLUDE_BEFORE, advice));
        else if (choice.ActiveText.Equals(ADD_AFTER + aopa.Id))
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.INCLUDE_AFTER, advice));
        else if (choice.ActiveText.Equals(REMOVE_GROUP + aopa.Group))
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.REMOVE));
        else if (choice.ActiveText.Equals(ADD_AROUND + aopa.Id))
        {
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.INCLUDE_BEFORE, advice));
          operations.Add(new Operation(annot,
            CodeAttribute.Operations.INCLUDE_AFTER, advice));
        }
      }
    }
    Code aopized = new Code(mi, (Operation[]) operations.ToArray(typeof(Operation)));
    AOPHandler = aopized.MakeDelegate(typeof(Handler)) as Handler;
  }

  void OnAppDelete(object o, DeleteEventArgs args)
  {
    alive = false;
    Application.Quit();
  }

  public static void Advice(int arg)
  {
    Console.WriteLine("AOP advice: v is {0}", arg);
  }

  public static void Run()
  {
    while (alive)
    {
      handler();
      Thread.Sleep(1000);
    }
  }

  public static void DoSomething()
  {
    int v = rnd.Next(0, 100);
    string s;

    [AOP(Advice = AOPAttribute.Advices.SWEEP, Group = "debug")]
    {
      Console.WriteLine("v = {0}", v);
    }

    [AOP(Advice = AOPAttribute.Advices.POST, Id = "square")]
    {
      v *= v;
    }

    [AOP(Advice = AOPAttribute.Advices.AROUND, Id = "increment")]
    {
      v += 1;
    }

    v = v % 2;

    [AOP(Advice = AOPAttribute.Advices.PRE, Group = "debug", Id = "parity")]
    {
      s = v == 0 ? "even" : "odd";
    }

    [AOP(Advice = AOPAttribute.Advices.SWEEP, Group = "debug")]
    {
      Console.WriteLine("v is {0}", s);
    }
  }

}
```

7.2.3 SVG Rendering

Listing 7.10: Render

```
using System;
using System.Collections;
using System.Xml;
using System.Reflection;
using ACS;
using Gtk;
using Gdk;

public class Render
{

  private static readonly string X = "x";
  private static readonly string Y = "y";
  private static readonly string WIDTH = "width";
  private static readonly string HEIGHT = "height";
  private static readonly string COLOR = "color";
  private static readonly string RECT = "rectangle";
  private static readonly string CIRCLE = "circle";
  private static readonly string COMPILED = "Switch to compiled drawing";
  private static readonly string ITERATIVE = "Switch to iterative drawing";
  public static Gdk.Color[] COLORS;
  private static Gdk.Color white;

  public delegate void Drawer();
  public static Drawer Draw;
  public static Drawer compiled;
  private DrawingArea da;
  private Gtk.Window win;
  private Button comp;
  private Label msg;
  private ComboBox options;
  private ArrayList shapes;
  private int opt = (int) DrawAttribute.Options.OUTLINE;

  private MethodInfo compileddraw;
  private AnnotationTree empty;
  private MethodInfo mrect;
  private AnnotationTree rext;
  private AnnotationTree[] rint;
  private Code rectOutline;
  private Code rectFill;
  private Code rectDashed;
  private MethodInfo mcircle;
  private AnnotationTree cext;
  private AnnotationTree[] cint;
  private Code circleOutline;
  private Code circleFill;
  private Code circleDashed;

  private delegate void ExtrudedRect(int i, int j, int k, int l, int m, int n);
  private delegate void ExtrudedRect2(int i, int j, int k, int l, int m, int n);
  private delegate void ExtrudedCircle(int i, int j, int k, int l);
  private delegate void ExtrudedCircle2(int i, int j, int k, int l, int m, int n);
  private delegate void NoArgs();

  public static void Main()
  {
    Application.Init();
   new Render();
    Application.Run();
  }

  public Render()
  {
    win = new Gtk.Window("ESVG Demo");
    da = new DrawingArea();
    msg = new Label();
    Button op = new Button("Open an ESVG file...");
    op.Clicked += OpenSVG;
    comp = new Button(COMPILED);
    comp.Sensitive = false;
    comp.Clicked += Switcher;
    VBox vb = new VBox(false, 0);
    options = ComboBox.NewText();
    options.AppendText("Outline");
    options.AppendText("Filled");
    options.AppendText("Dashed");
    options.Active = 0;
    options.Changed += Option;
    vb.PackStart(da);
    vb.PackEnd(msg, false, false, 0);
    vb.PackEnd(comp, false, false, 0);
    vb.PackEnd(options, false, false, 0);
    vb.PackEnd(op, false, false, 0);
    win.Add(vb);
    win.SetDefaultSize(400, 300);
    win.DeleteEvent += OnWinDelete;
    da.ExposeEvent += OnExposed;
      win.ShowAll();

    Colormap colormap = Colormap.System;
    COLORS = new Color[6];
    COLORS[0] = new Gdk.Color(255, 0, 0);
    COLORS[1] = new Gdk.Color(0, 255, 0);
    COLORS[2] = new Gdk.Color(0, 0, 255);
    COLORS[3] = new Gdk.Color(0, 255, 255);
    COLORS[4] = new Gdk.Color(255, 0, 255);
    COLORS[5] = new Gdk.Color(255, 255, 0);
    white = new Gdk.Color(255, 255, 255);
    colormap.AllocColor(ref COLORS[0], true, true);
    colormap.AllocColor(ref COLORS[1], true, true);
    colormap.AllocColor(ref COLORS[2], true, true);
    colormap.AllocColor(ref COLORS[3], true, true);
    colormap.AllocColor(ref COLORS[4], true, true);
    colormap.AllocColor(ref COLORS[5], true, true);
    colormap.AllocColor(ref white, true, true);

    Shape.window = da.GdkWindow;
    Shape.window.Background = white;
    Shape.gc = new Gdk.GC(da.GdkWindow);

    shapes = new ArrayList();
    Draw = IterativeDraw;

    compileddraw = typeof(Render).GetMethod("CompiledDraw");
    empty = Annotation.GetCustomAttributes(compileddraw)[0];

    mrect = typeof(Rectangle).GetMethod("DrawRect");
    rext = Annotation.GetCustomAttributes(mrect)[0];
    rint = rext.Children;
    foreach (AnnotationTree at in rint)
    {
      DrawAttribute d = at.Node[0] as DrawAttribute;
      string file = "rect_" + d.option + ".dll";

      Code r = new Code(mrect, new Operation[]{new Operation(rext, at, CodeAttribute.
           Operations.COPY_BEFORE), new Operation(rext, CodeAttribute.Operations.REMOVE)
           });
      r.MakeDelegateEx(typeof(ExtrudedRect), file);
      r = Code.FromAssembly(file);

      if (d.option == DrawAttribute.Options.DASHED)
        rectDashed = r;
      else if (d.option == DrawAttribute.Options.FILL)
        rectFill = r;
      else if (d.option == DrawAttribute.Options.OUTLINE)
        rectOutline = r;
    }

    mcircle = typeof(Circle).GetMethod("DrawCircle");
    cext = Annotation.GetCustomAttributes(mcircle)[0];
    cint = cext.Children;
    foreach (AnnotationTree at in cint)
    {
      DrawAttribute d = at.Node[0] as DrawAttribute;
      string file = "circle_" + d.option + ".dll";

      Code c = new Code(mcircle, new Operation[]{new Operation(cext, at, CodeAttribute.
           Operations.COPY_BEFORE), new Operation(cext, CodeAttribute.Operations.REMOVE)
           });
      c.MakeDelegateEx(typeof(ExtrudedRect), file);
      c = Code.FromAssembly(file);

      if (d.option == DrawAttribute.Options.DASHED)
        circleDashed = c;
      if (d.option == DrawAttribute.Options.FILL)
        circleFill = c;
      if (d.option == DrawAttribute.Options.OUTLINE)
        circleOutline = c;
    }

    // why we need this ?
    Code render = new Code(compileddraw);
    }

  private void OpenSVG(object obj, EventArgs args)
  {
    FileChooserDialog fc = new FileChooserDialog("Open SVG file",
                                win,
                                  FileChooserAction.Open,
                                  "Cancel",ResponseType.Cancel,
                                  "Open",ResponseType.Accept);
    if (fc.Run() == (int)ResponseType.Accept)
    {
      shapes.Clear();
      XmlTextReader xml = new XmlTextReader(fc.Filename);
      while (xml.Read())
      {
        if (xml.IsEmptyElement) {
          int x = Int32.Parse(xml[X]);
          int y = Int32.Parse(xml[Y]);
          int w = Int32.Parse(xml[WIDTH]);
          int h = Int32.Parse(xml[HEIGHT]);
          int c = Int32.Parse(xml[COLOR]);
          if (xml.Name.Equals(RECT))
            shapes.Add(new Rectangle(x, y, w, h, c));
          else if (xml.Name.Equals(CIRCLE))
            shapes.Add(new Circle(x, y, w, h, c));
        }
      }
      xml.Close();
      comp.Sensitive = true;
    }
    fc.Destroy();
  }

  public Drawer Compile(ArrayList list)
  {
    Code cr,cc;

    if (options.Active == 1)
    {
      cr = rectFill;
      cc = circleFill;
    }else if (options.Active == 2)
    {
      cr = rectDashed;
      cc = circleDashed;
    }else
    {
      cr = rectOutline;
      cc = circleOutline;
    }

    ArrayList operations = new ArrayList();
    foreach (Shape s in list)
    {
      if (s is Rectangle)
      {
        Rectangle r = s as Rectangle;
        operations.Add(new Operation(empty,
            CodeAttribute.Operations.INCLUDE_BEFORE,
            cr.Bind(r.x, r.y, r.width, r.height, r.color, 0)));

      }else if (s is Circle)
      {
        Circle c = s as Circle;
        operations.Add(new Operation(empty,
            CodeAttribute.Operations.INCLUDE_BEFORE,
            cc.Bind(c.x, c.y, c.width, c.height, c.color, 0)));
      }
    }
```

```
      operations.Add(new Operation(empty, CodeAttribute.Operations.REMOVE));
      Code render = new Code(compileddraw, (Operation[])operations.ToArray(typeof(
           Operation)));
      return render.MakeDelegate(typeof(Drawer)) as Drawer;
//      return render.MakeDelegateEx(typeof(Drawer), "drawer.dll") as Drawer;
//      return render.MakeDelegate(typeof(Drawer)) as Drawer;
    }

    private void Switcher(object obj, EventArgs args)
    {
      if ((obj as Button).Label.Equals(COMPILED))
      {
        (obj as Button).Label = ITERATIVE;

        compiled = Compile(shapes);
        Draw = compiled;
        DateTime start = DateTime.Now;
        for (int i=0; i<1000; i++)
          Draw();
        TimeSpan timeElapsed = DateTime.Now - start;
        Console.WriteLine("Compiledraw = {0}", timeElapsed.ToString());

        Draw = IterativeDraw;
        start = DateTime.Now;
        for (int i=0; i<1000; i++)
          Draw();
        timeElapsed = DateTime.Now - start;
        Console.WriteLine("IterativeDraw = {0}", timeElapsed.ToString());

        Draw = compiled;
      }
      else
      {
        Draw = IterativeDraw;
        (obj as Button).Label = COMPILED;
      }
    }

    private void Option(object obj, EventArgs args)
    {
      if (options.Active == 1)
        opt = (int) DrawAttribute.Options.FILL;
      else if (options.Active == 2)
        opt = (int) DrawAttribute.Options.DASHED;
      else
        opt = (int) DrawAttribute.Options.OUTLINE;
      Draw = Compile(shapes);
      win.QueueDraw();
    }

    public void IterativeDraw()
    {
      foreach (Shape s in shapes)
        s.Draw(opt);
    }

    public static void CompiledDraw()
    {
      [Draw]
      {
        Console.WriteLine("debug");
      }
    }

    public void OnExposed(object o, ExposeEventArgs args)
```

```
  {
    DateTime start = DateTime.Now;
    Draw();
    TimeSpan timeElapsed = DateTime.Now - start;
    msg.Text = timeElapsed.ToString();
  }

  private void OnWinDelete(object o, DeleteEventArgs args)
  {
    Application.Quit ();
  }

}
```

Listing 7.11: DrawAttribute

```
using System;
using ACS;

[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
public class DrawAttribute : CodeAttribute
{

  public enum Options : int
  {
    DASHED = 1,
    COLOR = 2,
    OUTLINE = 4,
    FILL = 8
  }

  public Options option;

}
```

Listing 7.12: Shape

```
using Gdk;

public abstract class Shape
{

  public static Window window;
  public static Gdk.GC gc;

  public int x;
  public int y;
  public int width;
  public int height;
  public int color;

  public Shape(int x, int y, int width, int height, int color)
  {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.color = color;
  }

  public abstract void Draw(int opt);

}
```

Listing 7.13: Rectangle

```
using System;
using Gdk;

public class Rectangle : Shape
{

  public Rectangle(int x, int y, int w, int h, int color) : base(x, y, w, h, color) {}

  public override void Draw(int opt)
  {
    DrawRect(x, y, width, height, color, opt);
  }

  public static void DrawRect(int x, int y, int w, int h, int color, int opt)
  {
    int lx = (int) (x / 100.0 * window.VisibleRegion.Clipbox.Width);
    int ly = (int) (y / 100.0 * window.VisibleRegion.Clipbox.Height);
    int lw = (int) (w / 100.0 * window.VisibleRegion.Clipbox.Width);
    int lh = (int) (h / 100.0 * window.VisibleRegion.Clipbox.Height);
    [Draw]
    {
      if (opt == (int) DrawAttribute.Options.DASHED)
      {
        [Draw(option=DrawAttribute.Options.DASHED)]
        {
          gc.SetDashes(0, new sbyte[]{1, 2}, 2);
          gc.SetLineAttributes(1, LineStyle.OnOffDash, CapStyle.Butt, JoinStyle.Miter);
          gc.Foreground = Render.COLORS[color];
          window.DrawRectangle(gc, false, lx, ly, lw, lh);
        }
      }
      else if (opt == (int) DrawAttribute.Options.FILL)
      {
        [Draw(option=DrawAttribute.Options.FILL)]
        {
          gc.Foreground = Render.COLORS[color];
          window.DrawRectangle(gc, true, lx, ly, lw, lh);
        }
      }
      else if (opt == (int) DrawAttribute.Options.OUTLINE)
      {
        [Draw(option=DrawAttribute.Options.OUTLINE)]
        {
          gc.Foreground = Render.COLORS[color];
          window.DrawRectangle(gc, false, lx, ly, lw, lh);
        }
      }
    }
  }

}
```

Listing 7.14: Circle

```
using System;
using Gdk;

public class Circle : Shape
{

  public Circle(int x, int y, int w, int h, int color) : base(x, y, w, h, color)
  {

  }

  public override void Draw(int opt)
  {
    DrawCircle(x, y, width, height, color, opt);
  }

  public static void DrawCircle(int x, int y, int w, int h, int color, int opt)
  {

    int lx = (int) (x / 100.0 * window.VisibleRegion.Clipbox.Width);
    int ly = (int) (y / 100.0 * window.VisibleRegion.Clipbox.Height);
    int lw = (int) (w / 100.0 * window.VisibleRegion.Clipbox.Width);
    int lh = (int) (h / 100.0 * window.VisibleRegion.Clipbox.Height);
    [Draw]
    {
      if (opt == (int) DrawAttribute.Options.DASHED)
      {
        [Draw(option=DrawAttribute.Options.DASHED)]
        {
          gc.Foreground = Render.COLORS[color];           gc.SetDashes(0, new sbyte[]{1,
               2}, 2);
            gc.SetLineAttributes(1, LineStyle.OnOffDash, CapStyle.Butt, JoinStyle.Miter)
               ;
            window.DrawArc(gc, false, lx, ly, lw, lh, 0, 360*64);
        }
      }
      if (opt == (int) DrawAttribute.Options.FILL)
      {
        [Draw(option=DrawAttribute.Options.FILL)]
        {
          gc.Foreground = Render.COLORS[color];
          window.DrawArc(gc, true, lx, ly, lw, lh, 0, 360*64);
        }
      }
      if (opt == (int) DrawAttribute.Options.OUTLINE)
      {
        [Draw(option=DrawAttribute.Options.OUTLINE)]
        {
          gc.Foreground = Render.COLORS[color];
          window.DrawArc(gc, false, lx, ly, lw, lh, 0, 360*64);
        }
      }
    }
  }


}
```

Bibliography

[1] Cisternino, A. Multi-stage and Meta-programming support in strongly typed execution environments. PhD thesis, Università di Pisa, May 2003. Available at http://www.di.unipi.it/phd/tesi/tesi_2003/PhDthesis_Cisternino.ps.gz

[2] Leone, M., Lee, P., Lightweight Run-Time Code Generation, Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation, 97-106, 1994.

[3] Veldhuizen, T.L. Arrays in Blitz++. In Proceedings of 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Springer-Verlag, 1998.

[4] Kennedy, K., Syme D., the Design and Implementation of Generics for the .NET Common Language Runtime, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA. June 2001.

[5] Bracha, G., Odersky, M., Stoutamire, D., and Walker P., Making the future safe for the past: Adding Genericity to the Java Programming Language, Proceedings of the Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Vancouver, October 1998.

[6] Fischbach. A., and Hannan, J. Specification and Correctness of Lambda Lifting, W. Taha (Ed.), SAIG 2000, LNCS 1924, pp. 108-128, Springer-Verlag, Berlin, 2000.

[7] De Bruijn, N.G., Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation, Indag.Mat., 34:381-392, 1972.

[8] Lee, P., Leone, M., Optimizing ML with Run-Time Code Generation, SIGPLAN Conference on Programming Languages Design and Implementation, 137-148, 1996.

[9] OCaml VM, http://pauillac.inria.fr/~lebotlan/docaml_html/english/.

[10] Leone, M., Lee, P., A Declarative Approch to Run-Time Code Generation, Workshop on Compiler Support for System Software (WCSSS), February 1996.

[11] Czarnecki, K., Eisenecker, U.W., Generative Programming - Methods, Tools, and Applications, Addison Wesley, Reading, MA, 2000.

[12] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Code Bricks: Code Fragments as Building Blocks. In Proceedings of 2003 SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'03), pages 66-74, San Diego, CA, USA, 20.

[13] Taha, W., Sheard, T. Multistage programming with explicit annotations. In Proceedings of ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations. PEPM'97, Amsterdam, p.203-217. ACM, 1997. At http://oops.tercom.ru/dml/.

[14] Naor, M., and Nissim, K., Certificate revocation and certificate update, in Proceedings of the 7th USENIX Security Simposium (San Antonio, Texas), Jan 1998.

[15] Lomov, D., and Moskal, A., Dynamic Caml v. 0.2, run-time code generation library for objective caml, Technical Report, Sankt PetersBurg State University, Russia, May 2002,

[16] Stroustrup, B., The Design and Evolution of C++. Addison-Wesley, Reading, MA, 1994.

[17] Taha, W., Multistage Programming: Its Theory and Applications, PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Available at ftp://cse.ogi.edu/pub/tech-reports/1999/99-TH-002.ps.gz

[18] Tanter, E., Ségura-Devillechaise, M., Noyé, J.Piquer, J., Altering Java Semantics via Bytecode Manipulation, in Proceedings of Generative Programming and Component Engeneerring (GPCE), LNCS 2487, 283-298, 2002.

[19] Winskel, G., The Formal Semantics of Programming Languages: An Introduction, MIT Press, Cambridge, MA, 1993.

[20] Cousineau, Guy and Michel Mauny. The Functional Approach to Programming. Cambridge, UK: Cambridge University Press, 1998.

[21] V. Gervasi and G. A. Galilei. Software manipulation with annotations in Java. In E. Boerger and A. Cisternino, editors, Advances in Software Engineering, number 5316 in LNCS. Springer-Verlag, 2008.

[22] ECMA International - http://www.ecma-international.org/

[23] Common Language Infrastructure (CLI) - Standard ECMA-335, June 2005, http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[24] http://www.ecma-international.org/publications/standards/Ecma-335.htm - Part III

[25] Shared Source CLI Essentials, Ted Neward, David Stutz, Geoff Shilling, ISBN:059600351X, O'Reilly, 2003.

[26] Introducing Microsoft .NET, 2nd edition, David S. Platt, ISBN:0735615713, Microsoft Press, 2002.

[27] A proposal for garbage-collector-safe C compilation, HJ Boehm, D Chase, The Journal of C Language Translation, 1992.

[28] http://www.monodevelop.org/

[29] Cisternino. A. CLIFileRW library, http://www.codeplex.com/clifilerw

[30] [a]C#: C# with a Customizable Code Annotation Mechanism, Cazzola, W., Cisternino, A., Colombo, D. In Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05), pages 1274-1278, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[31] JUnit Pocket Guide, Kent Beck, ISBN:0596007434, O'Reilly, 2004.

[32] NUnit Pocket Reference, Bill Hamilton, ISBN:0596007396, O'Reilly, 2004.

[33] Understanding The Linux Kernel, ISBN:0596005652, O'Reilly, Daniel Plerre Bovet, Marco Cesati, 2005

[34] Mono, A Developer's Notebook, Niel M. Bornstein, Edd Dumbill, ISBN:0596007922, O'Reilly, 2004.

[35] C# Language Specification, Standard ECMA-334, http://www.ecma-international.org/publications/standards/Ecma-334.htm

[36] C# in a Nutshell, 2nd Edition, Ben Albahari, Peter Drayton, Ted Neward, ISBN:0596005261, O'Reilly, 2003.

[37] A high-level modular definition of the semantics of C#. Egon Borger, Nico G. Fruja, Vincenzo Gervasi, Robert F. Stark. Theoretical Computer Science, 336(2/3):235-284, May 2005.

[38] http://www.ssw.uni-linz.ac.at/Coco/

[39] Learning Python, 2nd Edition, David Ascher, Mark Lutz, ISBN:0596002815, O'Reilly, 2003.

[40] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[41] Programming PHP, Rasmuf Lerdorf, Kevin Tatroe, ISBN:1565926102, 2002.

[42] The Java Language Specification, 3rd Edition - Gilad Bracha, James Gosling, Bill Joy, Guy Steele, ISBN:0321246780, Addison Wesley, Jun 2005.

[43] http://www-128.ibm.com/developerworks/linux/library/l-metaprog1.html

[44] Common LISP, Guy Steele, ISBN:1555580416, Digital Press, 1984.

[45] http://en.wikipedia.org/wiki/COFF

[46] Ruby in a Nutshell, Yukihiro Matsumoto, ISBN:0596002149, O'Reilly, 2001.

[47] http://www.xerox.com/innovation/parc.shtml

[48] Aspect-Oriented Programming (1997) Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, Proceedings European Conference on Object-Oriented Programming.

[49] http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[50] An Overview of AspectJ, Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold.

[51] Gregor Kiczales. AspectJ: aspect-oriented programming using Java technology. JavaOne, June 2000.

[52] Cristian Dittamo, Tecniche di parallelizzazione di programmi sequenziali basate su annotazioni. Master Thesis, University of Pisa, Department of Computer Science, 2006.