Multistage programming support in CLI

G. Attardi and A. Cisternino

Abstract: Execution environments such as CLR and JVM provide many features needed by multi-stage programming languages, though there is no explicit support for them. Besides, staged computations are widely used in areas such as Web programming and generative programming. In the paper the authors present a possible CLR extension (which can also be ported to JVM) to provide support for multi-stage languages. The extension is based on CodeBricks – a framework for run-time code generation which allows expressing homogenous transformations of intermediate language as a composition of methods. They discuss the code generation strategy adopted by the framework and how an extension to CLR may improve the performance of multi-stage applications, although CodeBricks can also be implemented using the standard CLR. An informal discussion of how to translate MetaML staging annotations into CodeBricks is provided with a simple example.

1 Introduction

Multistage programming is an emerging paradigm whose ultimate goal is to express programs that are evaluated in stages rather than in a single execution step. The aim of delaying computations is to exploit the amount of information available to a program before its final execution, avoiding unnecessary computation.

Extensions to programming languages such as ML have been proposed to support annotations indicating when portions of a program should be executed. MetaML [1] employed a notation derived from the LISP quasi-quotation [2], though in this case the compiler is able to type-check the program. A simple example [3] of a MetaML program is the following:

```
let even n = (n mod 2) = 0
  square x = x * x
  rec power n x =
    if n = 0 then lift 1
    else if even n
      then <square ~ (power (n/2) x)>
      else <~ x * ~(power (n - 1) x)>
  power72 = run<fun x-> ~ (power 72 <x>)>
in power72 2
```

The program specialises the power function for a given exponent. Four annotations have been added to ML to express multi-stage computations [4]:

–*Brackets* ('<' and '>') are inserted around any expression to delay its computation; another way to think about bracketed expressions is as code values representing the quoted expression.
–*Escape* ('~') is used to 'splice-in' the result of an expression (that should be of code type) in the context of a bracketed term.
–*Run* allows execution of a code fragment, removing brackets and evaluating the resulting expression.
–*Lift* allows lifting a value into a code value that, evaluated under run, produces the same ground value.

An execution stage is represented by an application of operator *run*. Because brackets can be nested, a single expression may require multiple applications of *run* to reduce a term. Each application of *run* is said to be an execution *stage* of the program.

MetaML is not the only programming language that allows expressing staged computations: interpreted languages can express staged programs using the *eval* function; macro systems like the one provided by LISP can also be used for staging. Although many of these do not provide any specific annotation to express staging; it seems that the notation proposed by MetaML is general enough for this class of programs.

1.1 Problem

Despite staged systems that are already employed in key technologies such as Web programming (ASP, PHP), there is no support for support of these languages in execution environments like Microsoft CLR [5] or Sun JVM [6]. More in general, these execution environments do not provide any support to languages with meta-programming facilities even though they provide many of the elements needed.

Besides this lack of support these environments still provide many mechanisms to do runtime code generation as well as reflection mechanisms to introspect programs to some extent. Nevertheless, staged computations may benefit from an explicit support from the execution environments. In particular, program transformations should be homogenous [7], so that they can be iterated across several stages.

Code manipulation at intermediate language level looks promising for staging because it is homogeneous. Moreover, Common Language Runtime (CLR) provides explicit support for generation of IL at runtime, and also within JVM bytecode can be generated at runtime [8]. Bytecode has already been used for implementing MetaOCaml [9], an extension of OCaml with multi-staging, though the OCaml virtual machine provides an instruction set significantly different from MSIL or Java bytecode.

One may ask why it is worth supporting multi-stage programs in CLR. First of all a main goal of CLR is to

IEE Proceedings online no. 20030990
doi: 10.1049/ip-sen:20030990
Paper received 29th June 2003

The authors are with the Dipartimento di Informatica, via Buonarroti, 2, I-56127, Pisa, Italy

support many programming languages, including those with stage annotations. A more compelling reason is that, nowadays, even programs without staging annotations are often multi-stage. Many programs manipulate a program before the runtime: preprocessors, parser generators, installers, verifiers and loaders are but a few examples.

What kind of support should an execution environment such as CLR provide to programming languages supporting multi-staging annotations? Is it possible to provide a general and efficient implementation of this support?

1.2 Contributions

In this paper we propose an extension to CLR to support multi-stage programming languages. The core of our proposal is based on a runtime code generator that allows code generation at intermediate language level, though the programmer perceives the transformation at language level. The code generator allows expressing homogeneous program transformation, used to express staged computations.

Although the whole system can be implemented on top of CLR, exploiting the code generation facilities already provided, the overall performance of the approach can be greatly improved by making aware the execution engine of it. The code generator provides a means for generating methods by composing a body of other methods; it is a pity to have to bake a class in order to be able to get the code running; in addition, the lifetime of generated code is, by design, bound to the application domain wasting either memory or execution speed [Note 1].

We illustrate a general approach to composition of method bodies possessing the interesting property that the execution of composed code is equivalent to a particular execution of the methods used to build it. The code composition is exposed as an extension of the reflection facilities of the runtime.

A less obvious, though important, contribution of this paper is the observation about the role played by metadata and intermediate language. The nature of execution environments, such as CLR, that try to enforce type safety at runtime and provide dynamic loading facilities implies a need for a lot of information in binary files. Moreover, programming languages have great benefits in exposing the types of the runtime into the language (mainly for code reuse). We can exploit this richness to encode into binaries information not intended for the execution environment, but used by other programs that interprets binaries with purposes other than mere execution. If the encoding is done using the semantic objects shared among languages and the execution environment (i.e. types, methods, fields and so on), the programmer would perceive it as if a tool operates on the program source rather than its compiled form.

2 A programming illusion

CLR and JVM are execution systems whose execution is type driven. Programs are expressed using an intermediate language for a (virtual) stack based machine. The execution environment type checks types before their execution enforcing type safety. Types are loaded dynamically when needed.

Note 1: Types and related code can be unloaded only with the application domain containing it. It is possible to allocate application domains on the fly, though cross-domain method invocations involve marshalling of parameters with additional overhead.

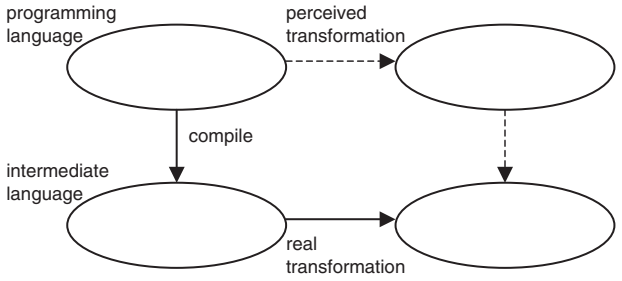


Fig. 1 *In CodeBricks code generation is expressed a method combination providing the illusion that the code manipulation is performed at source level*

We note that the adoption of an intermediate language is a consequence of the constraint imposing code verifiability. Moreover dynamic loading, together with a 'type aware runtime' encourage code reuse in the form of libraries of types like those in the base class library providing an ever increasing amount of basic functionalities (networking, security, windowing, etc.). This reuse is possible only if programming languages targeting these execution environments expose runtime types as a language abstraction. For instance, the common language infrastructure (CLI) [5] defines a set of rules known as Common Type System (CTS), whose aim is to impose additional constraints on type definitions so that types can be used across different languages.

The amount of semantic abstractions shared between programming languages and the runtime has been, for compiled languages, quite small, even in programming languages such as OCaml, where the virtual machine [10] provides abstractions far from the language. Besides, Java bytecode and MSIL retain so much information that types, method calls, fields, and many others are still present in the binary. Thus if we encode information into binaries in terms of these elements present in both the runtime and programming languages the programmer would have the impression that he is expressing something at language level though a tool may operate at binary level.

In this context bytecode manipulation becomes more than a technical trick to achieve some magic, and it has already been exploited in the context of runtime code specialisation [11] and to weave aspects into code [12]. In our case we rely on reflection and on intermediate language patterns in *CodeBricks*, a library for runtime code generation in which the programmer combines methods having the illusion of manipulating the source language though the real code transformation is performed at binary level (Fig. 1).

3 Introduction to CodeBricks

CodeBricks is a framework for code generation that allows programs to manipulate and generate code at the source level while the joining and splicing of executable code is carried out automatically at the intermediate language level. Herein we briefly describe the abstractions provided by the framework for runtime code generation. A more detailed discussion can be found in [13].

The framework is built around the `Code` type, which represents the class of code fragments. A code fragment is formed by three elements:

– an *environment* containing references to objects associated with the fragment
– a *body* of well formed intermediate language instructions
– a *signature* containing the type of input arguments and of the returned value of the fragment.

Code fragments can be obtained from methods as in the following example:

```
public static int add(int i, int j) {
  return i + j;
}
//...
Code codeOfAdd=
    new Code (typeof(MyClass).
      GetMethod ("add"));
```

In this example `codeOfAdd` is a fragment with an empty environment, a body and a signature that are the same of method `add`.

Fragments can be combined together by means of a *Bind* operator which produces new fragments. The code fragment resulting from a *Bind* operation represents a partial application of the fragment source of binding. Consider the following example:

```
Code inc = codeOfAdd.Bind(1,new FreeVar());
```

In this example `inc` is a new code fragment representing the partial application of 1 to the code fragment corresponding to the `add` method [Note 2]. The second argument is left unapplied, using as a placeholder the object of type `FreeVar`. The signature associated with `inc` has only an input argument because the first argument of `codeOfAdd` has been bound to the value 1. Assuming that code fragments were directly executables, our intuition would expect that the following equivalence holds:

```
inc(x) == codeOfAdd(1,x)
```

From the semantics of partial application it follows that

```
add3(x,y,z) == codeOfAdd(codeOfAdd(x,y),z)
```

where

```
Code add3 = codeOfAdd.Bind(
    codeOfAdd.Bind(new FreeVar(),
      new FreeVar()),
    new FreeVar());
```

We note that the composition of code fragments can be type checked during binding, ensuring that the values used to bind arguments are compatible with those indicated in the signature associated with the fragment used as the bind target.

Within CLR it is possible to express higher order values using *delegates* [Note 3]: a delegate represents a class of methods with the same signature. A method can accept a delegate as an argument and call the method it represents inside its body. An example of delegate use is the following:

```
delegate object F(object o);

public static ArrayList
  MapCar(F f,   ArrayList l) {
  ArrayList ret = new ArrayList();
  foreach (object o in l)
    ret.Add(f(o));
  return ret;
}
```

Note 2: We use the expression 'partial application' rather than 'partial evaluation' because the code generator does not partially evaluate the generated code: binding both arguments of `codeOfAdd` to 1 would result in the code that adds 1 to 1 rather than the constant 2.

Note 3: As pointed out in [8], delegates can also be expressed in Java as a particular case of interface.

Code fragments represent higher order values; thus it seems natural to allow splice-in of the body of a fragment in place of calls to an input argument of type delegate. Suppose we want just to clone all the object of a list; we may generate such a code fragment as follows:

```
public static object CloneObject(object o)
{
  return o.Clone();
}
//...
Code  map =  new  Code(typeof(MyClass).
  GetMethod("MapCar"));
Code clone =
  new Code(typeof(MyClass).GetMethod
    ("CloneObject"));
Code mapclone = map.Bind(new Splice
  (clone),
      new FreeVar());
```

In this case we used the `Splice` object to indicate to `Bind` that `clone` should be considered a higher order value to splice in place of the calls to the delegate argument of method `MapCar`. Again, `Bind` is able to type check that the signature of delegate `F` is compatible with the signature of `clone`. We expect the body of `mapclone` would be such that

```
mapclone(l) == map(clone, l) == MapCar
  (newF(CloneObject),l)
```

Code fragments simply represent code values: they are not directly executable. Moreover, the runtime is built around the notion of type, and fragments represent functions. We transform code fragments into delegates for execution. The `Code` class provides a `MakeDelegat` method that builds a delegate from a fragment.

```
delegate int MyDel(int i);
MyDel f = (MyDel) inc.MakeDelegate
  (typeof (MyDel));
int two = f(1);
```

Notice that we should specify the type of the delegate in the conversion. This is because there can be many delegate types for the same signature.

How the conversion of a code fragment into a delegate is performed depends on the implementation and will be discussed later; however, we found that the explicit conversion of code fragments into delegates offers a good integration of code values into the existing type system without requiring the design of the runtime. Moreover the implementation under this assumption can be more efficient than assuming that code fragments were implicitly executable objects: the runtime may delay code generation until the very last moment because it is known when a code fragment should be made executable.

4 Composing IL

In the preceding Section we introduced the use of the *CodeBricks* library and how it can be used for code generation, combining code fragments that, in the end, come from the precompiled bodies of methods. We have also assumed that the generated code is a partial application, providing a means to the programmer of controlling its behaviour. But does there exist a general transformation of bytecode that is suitable for implementing *Bind* operator?

In this section we describe the set of IL transformation schemes adopted by *CodeBricks* to generate the code. In [14]

a formal model of this transformation is introduced; it is proven that only type-safe code can be generated [Note 4] and that the semantics of generated code is one of the partial applications described in the preceding Section.

4.1 Structure of bind operator

Bind is a polymorphic operator performing runtime type checking to ensure that code fragments are correctly combined together. The real signature of the operator is the following:

```
public Code Bind(params object[] args) {...}
```

The C# `params` keyword authorises the compiler into silently turning a list of arguments into an array of objects before calling the method. With this trick we obtained, at least for C# programs, a syntax recalling the standard method invocation.

The first step of binding is type-checking: the signature of the code fragment target of binding is compared with the content of the arguments passed into the array of objects. A new signature is generated for the result fragment: the return type is the same as the target; arguments bound to instances of `FreeVar` are copied into the new signature in the same order because they are left unbound. For all the other arguments passed to bind they are type checked as follows (assuming T is the expected type present in the target's signature):

– the type A of the argument is such that $A <: T$
– the type of the argument is `Code`, the return value in its signature is A, and $A <: T$
– the type of the argument is `Splice`, T is a delegate type with the same signature as the code fragment contained into the argument.

After type checking, the body of the new code fragment is filled with the body of binding's target. Intermediate instructions are just copied unless they are instructions involving arguments or local variables (i.e. `ldarg`, `ldarga`, `starg`, `ldloc`, `stloc`...). When an instruction involving an input argument or a local variable is found the instructions added to the new body depend on the index of the instruction argument. Arguments may be mapped to locals (initialised with the appropriate value, as discussed later). Instructions involving local variables may require a different index for the variable they are referring; this renaming is necessary when bodies of different code fragments are merged together.

4.2 Arguments bound to values

If an input argument is bound to a value, as we did in the increment example, we should replace the instructions referring to that argument with instructions that go to fetch and store the value in the appropriate site.

We recall that a code fragment is characterised by an *environment* which is responsible for containing the values bound with `Bind`. We assume that the environment is implemented as an array of `Object`, although it is not the only possible choice. Under this hypothesis we transform instructions reported in Table 1 (assuming that I is the index of the input argument and j the index of the value into the environment):

Every access to the argument is replaced with a corresponding access to the environment. If the environment

Table 1

`ldarg` *i*	`ldarg.0`
	`ldc.i4` *j*
	`ldelem.i4 // It depends on the type!`
	`// Type conversion instruction needed here:`
	`// either unboxing or type cast`
`ldarga` *i*	`ldarg.0`
	`ldc.i4 j`
	`ldelema`
`starg` *i*	`ldarg.0`
	`ldc.i4 j`
	`stelem.i4 // It depends on the type!`

Table 2: The body of method `add` and the generated code for the increment function using the general [1] and the optimised [2] approaches

`add`	`inc`[1]	`inc`[2]
`ldarg.0`	`ldarg.0`	`ldc.i4.1`
`ldarg.1`	`ldc.i4.0`	`ldarg.0`
`add`	`ldelem.ref`	`add`
`ret`	`unbox`	`ret`
	`System.Int32`	
	`ldind.i4`	
	`ldarg.1`	
	`add`	
	`ret`	

is implemented as an array we should cast values to the appropriate type after loading the value from the array, which is silently passed as the first argument of the code fragment. Otherwise the resulting code would be unverifiable.

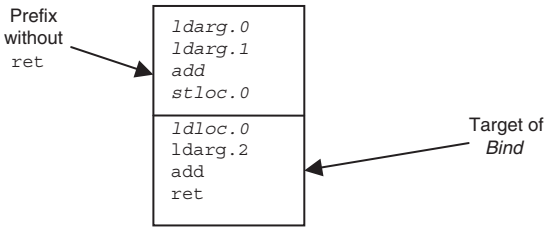
Our current implementation optimises the special case in which an argument is used in a read-only fashion inside the method body and its type is native (i.e. number or string): in this case the value is loaded with a single instruction whose argument is the value itself (or a metadata token for a string); see Table 2.

4.3 Argument bound to a code fragment

Code fragments can be used instead of values given that the return type of the fragment is compatible with the expected type of the input argument. In this case a new local variable is introduced. The body of the code fragment is made the prefix of the resulting body and local variables are renamed in order to avoid capturing.

Instead of the instruction `ret` a `stloc` is generated in the prefixed code and all the instructions referring to the argument in the target body are replaced with instructions referring the local variable.

Applying this strategy the code generated for `add3` would be the following:



Note 4: under the hypothesis that the bodies of methods involved in the code generation are type-safe.

In this case the original body of `add` has been prefixed with another copy of it (it is the result of `codeOfAdd.Bind(codeOfAdd, new FreeVar())`. The result of the first `add` is stored in a local variable and then used in the code of add instead of the first argument. We note that the number of arguments of the resulting fragment is increased by one and the arguments have been renamed accordingly.

In general, if an open code fragment (i.e. with input arguments) is used in a binding, its arguments are *lifted* into the signature of the resulting code fragment replacing the bound argument in the original signature. The process of lifting arguments is very similar to lambda lifting [15], where code fragments play the role of lambda terms. Consider the following code fragment:

```
Code c = codeOfAdd.Bind (new FreeVar(),
  codeOfSub);
```

The arguments of `c` are treated as follows: the first argument maps to the first argument of the `add` method and the remaining two are used as arguments for the `sub` method.

Also in this case our implementation recognises that in this case the local variable can be avoided generating the optimal code.

4.4 Splicing code fragments

If a splice operation is requested, the type of the input argument should be a delegate. In this case the code fragment bound to the argument should have the same signature of the delegate type so that we can inline its body where the delegate is invoked. (See Fig. 2) it is worth noting that the transformation of IL is not limited to a single instruction.

Here we sketch a simple implementation of the transformation, although it cannot deal with nested delegate invocations (a more general approach can be adopted, though, would make the example harder to follow).

The pattern of a delegate invocation (assuming *i* is the index of the *i*th argument of type delegate) is the following:

```
ldarg i
// instructions that load arguments
callvirt Invoke // Invocationof the
delegate
```

The strategy for replacing the method invocation with the body of the bound code fragment is the following:

– the block of instructions responsible for loading arguments (except for `ldarg i`) are preserved
– a local variable of the appropriate type for each argument is introduced
– the `callvirt` instruction is replaced by a `stloc` instruction for each argument
– the body of the code fragment is appended and fixed to ensure that the instructions dealing with arguments refer to the appropriate local variable.

Often arguments passed to a method call are stored in local variables (or input arguments): in this case it is possible to avoid the introduction of the local variables reducing the code size and improving performance.

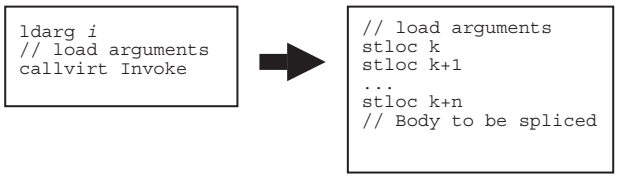


Fig. 2 *A simple schema for splicing code fragments into delegate invocations*

We note that with splice-in it is trivial to write code that behaves like fixing an argument to a code fragment (the case discussed in the preceding Section). Nevertheless, *CodeBricks* has the chance of performing better optimisations if both transformations are possible.

5 Making CLR aware of code fragments

We did the first implementation of *CodeBricks* on the commercial version of CLR. Thus we have generated code dynamically with the classes provided in the `Reflection.Emit` namespace: for each code fragment we emit a new type containing a method whose body corresponds to the one of the fragment. The environment is stored in the class and a standard delegate to the method is returned when the fragment should be turned into an executable entity.

As discussed in [13] we have compared performances of the hand-written compiler for regular expressions included with SSCLI [16] with an equivalent program based on *CodeBricks*. The code generated using *CodeBricks* has produced results close to the hand-written implementation. However, in our preliminary benchmark we were testing the performance of a single code fragment.

The main drawback in using `Reflection.Emit` is that the approach does not scale very well when a large number of different code fragments should be executed. Unfortunately, in program specialisation this is often the case: for instance, the power function can be specialised for each possible value of its exponent.

Moreover, in runtime code generation it is often the case that generated code is needed for a small amount of time, and then could be discarded, freeing memory. Also, type driven execution tends to retain loaded code into memory. SSCLI provides a form of garbage collection known as *code pitching* [17] for code generated by JIT compiler; nonetheless the CLR still retains all the data structures of loaded types (for instance *EEClass*and *MethodTable*). Without a modified runtime the only way to free the memory used for generated code is to unload the application domain, at the price of having to produce the code into a separate domain where the method invocation is more expensive due to required marshalling of arguments.

We believe that the execution environment can be modified in the following areas to improve the support to code fragments:

– methods generation without the need to generate a type for each method
– support for closures: the environment should be associated with the corresponding executable code fragment
– garbage collection of unnecessary code fragments
– debugging of runtime generated code
– reflection interface: method bodies can be exposed as code fragments from *MethodInfo*.

We briefly outline the impact of the modifications on the SSCLI [Note 5].

5.1 Method generation

As already discussed, the design of *CodeBricks* introduces code fragments that are rendered as delegates in order to be

Note 5: Our wish was to include some early experiment on changing SSCLI to support *CodeBricks*. Unfortunately the hard disk of one of the authors crashed, causing the loss of part of the work.

executable. Because of this, the type system is left unchanged by the introduction of *Code* type because the delegate object hides the fact that a code fragment is a method without an associate type.

In the implementation made on top of CLR we use `Reflection.Emit` to generate a type with a method containing the code fragment in it. Then we build the delegate on an instance of that type. A better solution would be to allocate an *EEClass* inside each application domain (the descriptor of a loaded type within the runtime) with a large method table. Each time we should convert a code fragment into a delegate a new method is generated in this private class that is kept open on purpose.

With this change the creation of an executable code fragment would be faster than using the `Emit` package. Moreover, our implementation of *CodeBricks* has been designed to generate the bytes so that the JIT can directly use it without having to call a method for each instruction.

5.2 Support for closures

During the binding process it is possible to bind objects to arguments. These objects are associated with the code fragment within the *environment*, i.e. in its simplest form, an array. The body of a code fragment requires the values stored into the environment for execution.

Delegates seem to be the right type to represent a method with its environment; unfortunately the delegate specification only allows to link a method with an instance of the class to which belongs. If the runtime is aware of the `Code`type, the constraint can be slightly relaxed, allowing the building of delegates using code fragments: the implementation would ensure that the environment of a code fragment would be passed to the generated method as if it is this pointer.

This small change would not break any existing code nor change the overall model of the type system.

5.3 Collection of code fragments

By definition a code fragment can be executed only when converted into a delegate. Thus the lifetime of the delegate object coincides with the time span for which a generated code is needed. We can exploit this fact to make aware the garbage collector of the delegates built around a code fragment.

Our goal is to collect the memory of a compiled code fragment that is not needed anymore. This can easily be achieved if the runtime is aware of the `Code` type: when a code fragment is turned into a delegate the pre-stub [Note 6] for the generated method can retain its origin so that the code generated by the JIT compiler can be stored in a different heap. When the delegate object is collected we know that the code is not required anymore and the collector can reclaim the memory used for it.

We use a weak reference in the code fragment to reference its executable version (once generated) in order to decouple the lifetime of the two objects.

5.4 Debugging of the generated code

A major problem in runtime generation code is its debugging. In systems that generate machine code (or bytecode) there is no chance for the debugger to present any source code. This fact makes debugging difficult in this kind of system, and often the only viable solution is to insert a print instruction in the generated code [Note 7].

It is possible to extend our approach of combining methods to the debugging information. Thus it is possible (perhaps disabling some optimisation) to keep track, for each instruction, of its corresponding instruction in the source code.

Moreover, because the semantics of the generated code is the partial application, we can modify the debugger so that it shows the interpreted version of the application while it is executing the compiled code. We think that this is a novelty in the context of runtime code generation.

5.5 Reflection interface

Execution environments such as CLR or JVM provide a rich set of information about types loaded into the runtime. Programs may inspect types' structure, fields and method signatures. Besides there is no means to get the method body: this is because the only information about method body available to the runtime is its bytecode (that is not really interesting as the source code).

We believe that a method body could be exposed as a code fragment by the runtime. Although it cannot be decomposed in single instructions, it can be used to generate new code, making it useful for the program to have access at runtime to a representation of method bodies.

Moreover, if code fragments are exposed as a property of *MethodInfo* object, then the runtime has the opportunity of book-keeping which methods could be used in the future for generating code.

6 Power example revised

It is not difficult to see that code fragments are tightly related to the bracket operator of MetaML. The splice-in provided by *CodeBricks* can be used to obtain the same effect of MetaML's tilde operator. When a code fragment is turned into a delegate and executed we have the same effect that the run operator has on a bracketed expression in ML. Finally, it is easy to imagine a method using code fragments inside that is used to build a new code fragment which corresponds to a two stages expression.

Thus a MetaML compiler for CLI could rely on *CodeBricks* to compile staging annotations. MetaML adopts a finer grain for deferred code: a single expression can be deferred, whereas in *CodeBricks* only methods can be used to build code fragments. This is not a real issue: every expression can be turned into a function that is invoked in the place where the expression should have been. We have followed this approach to compile the power function shown in the introduction from MetaML to *CodeBricks*. We have used C# instead of IL to show how a MetaML compiler for .NET could have translated the program:

```
delegate int IntExp(int i);

class Power {
    static
    Code one = new Code(typeof(Power).
      GetMethod("One"));
    static
    Code id = new Code(typeof (Power).
      GetMethod("Id"));
    static
```

Note 6: The chunk of code responsible for jitting the code associated with a method.

Note 7: Besides when the code is generated at source level (i.e. with string manipulation and then executed with *eval*) this is not an issue, though there is much more overhead at runtime.

```
    Code sqr = new Code (typeof(Power).
      GetMethod("Sqr"));
    static
    Code mul = new Code (typeof(Power).
      GetMethod("Mul"));

    public static int One(int i) {
      return 1;
    }
    public static int Id(int i) {
      return i;
    }
    public static int Sqr(IntExp f, int x) {
      int v = f(x);
      return v * v;
    }
    public static intMul (IntExp f,IntExp
      g,int x) {
      return f(x) * g(x);
    }
    public static Code Power(int n, Code x) {
      if (n == 0) return one.Bind(x);
      else if (n == 1) return x;
      else if (n%2 == 0)
          return sqr.Bind(Power(n/2,x),
            Code.Free);
      else return mul.Bind(x,Power(n - 1, x),
        Code.Free);
    }
    public static Code Power(int n) {
      return Power(n, id);
    }
    static public void Main(string[] args){
      ((IntExp) power(72) .MakeDelegate
        (typeof (IntExp))) (2);
    }
}
```

7 Conclusions

We presented an extension to CLR to support programming languages with multi-stage annotations. The support is based on code fragments that correspond to delayed computations. Code fragments can be combined together by means of *Bind* operator, which combines sequences of IL instructions and values so that the generated code is semantically equivalent to the partial application of its constituents. We have shown the base transformation we use for generating code and how a combination of methods provides the illusion of manipulating source code language rather than IL.

Although an implementation of *CodeBricks* for standard CLR is possible, a runtime aware of *CodeBricks* would provide better use of memory and faster code generation with small changes. This is crucial because, as witnessed by the power example, even simple meta-programs may involve several code fragments that may impose a significant overhead on the runtime.

We showed with a simple example a possible translation schema of staged annotations from MetaML to *CodeBricks*. We believe that the *CodeBricks* framework is expressive enough to support multi-stage languages, though further research is required in this direction.

CodeBricks currently handles only static methods; we plan to extend it to handle instance methods as well, solving issues of protection in accessing member fields. We should also extend the code transformation model to include exception handling.

8 References

1 Taha, W., and Sheard, T.: 'Multi-stage programming with explicit ennotations'. Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantic Based Program Manipulations (PEPM)', Amsterdam, 12–13 July 1997, pp. 203–217

2 Steele, G.L.: 'Common Lisp: the Language' (Digital Press, Woburn, MA, 1990, 2nd edn.)

3 Calcagno, C., Taha, W., Huang, L., and Leroy, X.: 'Implementing multi-stage languages using ASTs, Gensym, and Reflection', *Lect. Notes Comput. Sci.*, **2830**

4 Taha, W.: 'Multistage programming: its theory and applications'. PhD thesis, Oregon Graduate Institute of Science and Technology, Available at ftp://cse.ogi.edu/pub/tech-reports/1999/99-TH-002.ps.gz, accessed October 2003

5 ECMA 335, 'Common language infrastructure (CLI)', http://www.ecma.ch/ecma1/STAND/ecma-335.htm, accessed October 2003

6 Lindholm, T., and Yellin F.: 'The Java™ Virtual Machine Specification' (Addison-Wesley, Reading, MA, 1999, 2nd edn.)

7 Sheard, T.: 'Accomplishments and research challenges in meta-programming', *Lect. Notes Comput. Sci.*, 2001, **2196**, pp. 2–44

8 Sestoft, P.: 'Runtime code generation with JVM and CLR', http://www.dina.dk/~sestoft/rtcg/rtcg.pdf, accessed October 2003

9 Calcagno, C., Taha, W., Huang, L., and Leroy, X.: 'A bytecode-compiled, type-safe, multi-stage language', http://citeseer.nj.nec.com/460583.html, accessed October 2003

10 'OCaml VM', http://pauillac.inria.fr/~lebotlan/docaml_html/english/, accessed October 2003

11 Masuhara, H., and Yonezawa, A.: 'Run-time bytecode specialization: a portable approach to generating optimized specialized code'. Proc. Symp. on Programs as Data Objects (PADO), Aarhus, Denmark, 21–23 May 2001, pp. 138–154

12 Tanter, E., Ségura-Devillechaise, M., Noyé, J., and Piquer, J.: Altering Java semantics via bytecode manipulation, *Lect. Notes Comput. Sci.*, 2002, **2487**, pp. 283–298

13 Attardi, G., Cisternino, A., and Kennedy, A.: 'Code bricks: code fragments as building blocks'. Proc. SIGPLAN Workshop on Partial Evaluation and Semantic-based Program Manipulation (PEPM), San Diego, CA, USA, 2003, pp. 66–74

14 Cisternino, A.: 'Multi-stage and Meta-programming support in strongly typed execution engines'. Phd thesis, TD-5/03, Dipartimento di Informatica, Universita diPisa, May 2003, available at http://www.di.unipi.it/phd/tesi/tesi-2003/PhDthesis-Cisternino.ps.gz, accessed October 2003

15 Fischbach, A., and Hannan, J.: 'Specification and correctness of lambda lifting', *Lect. Notes Comput. Sci.*, 2000, **1924**, pp. 108–128

16 'Microsoft Shared Source CLI', http://msdn.microsoft.com/net/sscli, accessed October 2003

17 Stutz, D., Neward, T., and Shilling, G.: 'Shared source CLI essentials' (O'Reilly, Sebastopol, CA, 2003)