CodeBricks: Code Fragments as Building Blocks

Giuseppe Attardi
Dipartimento di Informatica
via Buonarroti 2
I-56127 Pisa, Italy
+39 (050) 221-2744
attardi@di.unipi.it

Antonio Cisternino
Dipartimento di Informatica
via Buonarroti 2
I-56127 Pisa, Italy
+39 (050) 221-3149
cisterni@di.unipi.it

Andrew Kennedy
Microsoft Research Ltd
7, JJ. Thompson Av.
CB Cambridge, UK
akenn@microsoft.com

ABSTRACT
We present 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 code/VM level. The framework introduces a data type `Code` to represent code fragments: methods/operators from this class are used to reify a method from a class, producing its representation as an object of type `Code`. `Code` objects can be combined by partial application to other `Code` objects. `Code` combinators, corresponding to higher-order methods, allow splicing the code of a functional actual parameter into the resulting `Code` object. *CodeBricks* is a library implementing the framework for the .NET Common Language Runtime. The framework can be exploited by language designers to implement metaprogramming, multistage programming and other language features. We illustrate the use of the technique in the implementation of a fully featured regular expression compiler that generates code emulating a finite state automaton. We present benchmarks comparing the performance of the RE matcher built with *CodeBricks* with the hand written one present in .NET.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Algorithms, Performance, Design, Languages.

Keywords
Generative programming, program transformation, program generation, metaprogramming, reflection, multistage programming, Domain Specific Language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'03, June 7, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-667-6/03/0006…$5.00.

1. INTRODUCTION
Generative programming [2] allows solving software problems in families, by automatically generating the code for each instance of a problem: implementations are built from code patterns and lower-level components, according to directives expressed through metadata.

The technique is employed in several tools routinely used by programmers. For instance, GUI application frameworks generate the code for creating the elements of an interface and the complex event generation/dispatch mechanism required by the framework; IDL compilers produce proxy-stub code; compiler generators such as YACC [12] produce parsing programs; configuration utilities select and combine the code that makes an application; many so called "wizards" perform a similar task of producing code on behalf of the user. In these cases, generative programming techniques are embedded in the tools. Sometimes the tools provide a specific language (e.g. IDL) or a notation to drive their operation, that can be as sophisticated as to become a full Domain Specific Language (e.g. YACC for parsing).

Such means of generative programming are embedded in tools and built in ad hoc fashion, but tools to build generative programs would be useful also to programmers, allowing them to control the structure and content of the generated code and to avoid using a proprietary scripting language.

Generative programming (GP) techniques can be made available either as tools *external* to a programming language, or as facilities *internal* to the language or exploited *within the programming environment*.

Tools external to a language include preprocessors, and program generators like Active Server Pages (ASP) pages, or XML-based code generation using XSLT transformations [13][16]. We will show examples of these approaches in the related work section.

Manipulating or generating programs within the language requires a language with metalanguage capabilities, i.e. at least the ability to represent programs in the language itself. Languages like Lisp [20], MetaML [21], `C [16] and DynJava [15], provide such facilities. C++ provides an unusual solution with template metaprogramming [5], where generated programs are expressed as parameterized types, and code is produced by the compiler through inlining. It has been even suggested that the most relevant facilities of Aspect Oriented Programming can be emulated by suitable conventions of usage of C++ templates.

The execution of generated programs requires a language processor, i.e. either an interpreter or a compiler. When using an interpreter,

runtime code generation (RTCG) is easier, since code in high-level language is simpler to produce, however performance of the generated program may not be satisfactory.

Generating machine code is more complicated and requires either access to a compiler or at least to a compiler back-end to produce binary or intermediate code. For instance in `C [16] the Vcode [6] runtime code generation library is used to produce binary code. Compilation however introduces an overhead whose cost can be amortized only if the generated code is used repeatedly.

In this paper we present 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 code/VM level.

The framework relies on the fact that programs compiled for an intermediate language, such as the JVM bytecode or the .NET Common Intermediate Language (CIL), retain enough information about high-level types so that a correspondence between high-level and low-level manipulations can be retained. Consequently type checking and verification can be performed, guaranteeing that the code generated by our framework is in fact well typed [2]. When an intermediate language common to several languages is used, as in our current implementation based on CIL, code fragments even from different languages can be combined.

Differently from other approaches to metaprogramming that operate at the source level and require a language processor to be available at run time to run the resulting program, our approach operates directly on precompiled executable versions of programs. A similar technique of combining precompiled binary fragments is used in Tempo templates [4], which are filled in at run time with constant values.

In our framework code manipulation primitives are provided through a library rather than as a language extension, allowing cross language code generation as well as a more robust and maintainable approach than language extensions.

Programs can be specialized more than once, for instance as more information becomes available to the program. Since no language processor is involved, specialization can happen at different stages through the lifetime of a program, even beyond the program build: for instance at installation time, at load time, or on the client side in a multi-tier application. If we can realistically assume that each stage is capable of running code for the same VM, the framework can be used as support for real multistage programming.

The framework has been implemented as a class library, called *CodeBricks*, on the Microsoft .NET platform [22]. Code generation is fast since no compiler invocation is involved and produces efficient code.

The benefits of the approach can be summarized as follows:

- the generated code is expressed and manipulated in a high-level language
- efficient binary code is produced and code generation overhead is minimal
- no high-level language processor is required to run the generated code
- the solution is not tied to a programming language
- generated code can be involved in further code generation and can itself generate code, providing full multi stage capabilities.

As an example of use of the framework we will show how to perform transformations corresponding to partial evaluation.

To illustrate the brick composition metaphor, we present the implementation of a regular expression compiler, assembling small bricks of code that represent typical patterns in the finite state automaton being produced. The performance of the generated code will be compared to that of a handcrafted compiler.

2. Code Objects

The ability of manipulating a program within a program requires a data type for representing programs as well as operators to handle it. In principle a language may expose a representation of its programs through the mechanisms of reflection, but this is often not sufficient since reflection in compiled languages typically exposes at most information about the signature of a function or method (Java, C#). Our framework introduces hence a specific data type for representing code fragments and a single operator called `Bind` to combine them.

CodeBricks provides a special type, `Code`, for representing parametric code fragments. `Code` objects behave like functional objects: they have a function signature and can be applied to arguments of types compatible with their signature.

Consider the following example:

```
public class Test {
  static int add(int i, int j) {
    return i + j;
  }
};

Type t = typeof(Test);
MethodInfo m = t.GetMethod("add");
Code cadd = new Code(m);
int n = cadd(3, 4);
```

The constructor `Code(m)` creates a `Code` object for the method `Test.add`, obtained through reflection. The `Code` object `c` has the same signature as the method from which it was created; hence it can be applied to two integers returning their integer sum.

The `Bind` operator allows binding the parameters of a `Code` object to constants or to other code objects, in particular to free variables. For instance

```
Code d = cadd.Bind(1, Code.FreeVar);
```

corresponds to a function of one integer argument that adds one to it (the integer constant 1 is implicitly lifted into a `Code` object).

The semantics of `Bind` can be explained in terms of application and lambda-lifting [7].

Definition. *Lambda lifting* is a program transformation to remove free variables. An expression containing a free variable is replaced by a function applied to that variable.

The `Bind` operator combines application and lambda-lifting as follows:

1. application: the code object function is applied to the supplied arguments provided they are compatible with the function signature types.
2. lambda lifting: all free variables present in the expression are lifted into bound variables of the resulting function.

More precisely, one may consider a code object as representing a lambda expression. For instance the code object `cadd` corresponds to the expression:

$\lambda x\, y.\ x + y$

Binding

```
cadd.Bind(1, Code.FreeVar);
```

corresponds to performing the application:

$(\lambda x\, y.\ x + y)\ 1\ z$

where *z* is a new free variable, obtaining

$1 + z$

and then lambda lifting to obtain:

$\lambda z.\ 1 + z$

Similarly,

```
cadd.Bind(Code.FreeVar, Code.FreeVar);
```

produces

$\lambda z.\ z + z$

and hence the combination

```
cadd.Bind(1, cadd.Bind(Code.FreeVar, Code.FreeVar));
```

produces

$\lambda z.\ 1 + (z + z)$

If more than one free variable is needed, they can be created as instances of class `FreeVar`:

```
Code x = new FreeVar();
Code y = new FreeVar();
Code d = cadd.Bind(x, y); // λx y.x+y
Code e = d.Bind(x, d); // λx y.x+x+y
```

The use of free variables in `Bind` produces a similar effect to partial application:

Definition. *Partial application* consists in the application of a curried function $f : A_1 \rightarrow \dots A_n \rightarrow R$ to its first m $(m \leq n)$ arguments yielding a new function $g : A_{m+1} \rightarrow \dots A_n \rightarrow R$. Assuming that $g = f(x_1, \dots, x_m)$, g is defined by $g(x_{m+1}, \dots, x_n) = f(x_1, \dots, x_n)$.

A full account would involve the use of a typed lambda calculus, since *CodeBricks* deals with typed expressions and functions as well as with imperative features.

3. CODE COMPOSITION

Code composition occurs when an argument of a code object is bound to another code object. For instance a three-way add method can be built from the two way version:

```
Code cadd3 = cadd.Bind(Code.FreeVar, cadd);
// λxyz.x+(y+z)
```

The second argument of `cadd` is bound to a code object (`cadd` itself) with two arguments. These arguments are not bound, hence lifting turns them into arguments of the resulting code object, that will have three arguments: the first corresponds to the first argument of the outer `cadd`, the second and third to the arguments of the inner `cadd`.

Having established the signature of the resulting code object, its machine code is produced by inlining the machine codes of its components, performing suitable renaming, relocation and introduction of temporary variables.

The resulting `add3` is a code object equivalent to the following method:

```
public static int add3(int i, int j, int k) {
  int l = j + k;
  return i + l;
}
```

While in this case introducing a local variable might seem redundant, in general it is required in order to preserve the order of evaluation. In fact, since code objects passed to `Bind` may involve side-effects, *CodeBricks* specifies that they are dealt in left-to-right order, i.e. in the generated code the code corresponding to arguments appears and is run in the same order as they appear in the call to `Bind`.

3.1 Higher Order Code Objects

A code combinator is created from a higher-order method that accepts a function object as parameter. A code object *C* that is passed as the function object parameter to a code combinator, can be used in two ways:

- issuing a call to *C*
- splicing the code of *C* in the place of its usage.

To denote that a code object should be spliced, *CodeBricks* provides the constructor `Spliced()` to be used instead of `Code()`. For example suppose one needs to build a loop around a certain action: one creates first a code combinator for a method like this:

```
public delegate void Cmd(int i);

public static void For(Cmd c, int par) {
  for (int i = 0; i < par; i++) c(i);
}
```

The .NET notion of delegate is used here to represent a functional type[1].

To repeat a certain body within the above loop involves using a code combinator from the `For()` method above and applying it to a spliced code object for the method `Body()`:

```
public delegate void Cmd(int);

public class Test {
  public static void For(Cmd body, int count) {
    for (int i = 0; i < count; i++)
      body(i);
  }
  public static void Body(int j) {
    Console.WriteLine("iteration: {0}", j);
  }
  public static void Main(string[] args) {
    Type TestT = typeof(Test);
    Code c = new Code(TestType.GetMethod("For"));
    Code b = new Spliced(TestType.GetMethod("Body"));
    Code d = c.Bind(b, new FreeVar());
    d(3);
  }
}
```

The resulting code object `d` behaves like the following method:

```
public static void d(int count) {
  for (int i = 0; i < count; i++)
    Console.WriteLine("iteration: {0}", i);
}
```

[1] In Java one could use an interface with one invocation method.

The effect is similar to something like this in `C quasi-quote notation:

```
void cspec body(int vspec i) {
   return
      `{ Console.WriteLine("iteration: {0}", i); };
   }
void cspec d(int count) {
   return
      `{ int i;
         for (i = 0; i < count; i++)
            @body(i); };
   };
```

`vspec` is the type of a dynamically generated variable, that can be used to refer to variable declared within the scope of dynamic code, like the index `i` of the for loop in the example. DynJava allows instead declaring the free variables used in a statement specification. In both cases though new source code is produced that needs to be passed to a compiler for compilation, while in *CodeBricks* code objects are built out of complete and well formed code fragments that the compiler has already compiled. *CodeBricks* though could be used to implement a DynJava compiler [2].

4. FORMAL MODEL

In [2] a formal model is presented of an abstract intermediate language for the CLR called Baby IL (BIL). BIL is a deterministic, single-threaded, class based object oriented language that covers most of the features of the real CIL. Code objects are added to BIL, whose semantics is given through a function that interprets code objects. The semantics of the `Bind` operator is defined in terms of BIL as a transformation on code objects.

From the semantics two theorems have been proved:

- interpreting a code object is equivalent to interpreting the code object produced by `Bind`
- the code produced by `Bind` is type safe.

5. IMPLEMENTATION

CodeBricks has been implemented as a class library for the Microsoft .NET CLR, using the code generation facilities of the Reflection.Emit package.

Code objects are a kind of metaobjects representing lambda expressions, built from primitive lambda expressions (methods) by means of function application to other code objects or to constant values.

When such a lambda expression is applied to some arguments by the use of `Bind`, the lambda expression must be turned into a closure to record the bindings of those arguments. Hence a code object retains an environment to represent closures.

Code generation in *CodeBricks* is performed lazily, at the first occurrence of an invocation of a code object. Therefore a code objects maintains the whole tree representation of the lambda expression, and after code has been generated, it caches the IL code produced.

Finally the signature of the lambda expression must be retained, in order to perform type checking during code generation.

5.1 Representation of Code Objects

A code object is hence a triple:

Code = <*Environment*, *Signature*, *MachineCode*>

Signature is the signature of the associated function: list of input parameters types and return type. The *MachineCode* is the IL code that implements the code object. Machine code refers to input parameters through their index in the signature *Signature*. *Environment* is a function that associates values to local variable names.

Atomic code objects are created from a reified method object. The *MachineCode* for them is just the IL for the associated method. This code is read from an assembly, through a reader for .NET assemblies that we developed [3].

Each `Code` object contains a representation of the whole tree of bind expressions through which it was created. *Atomic* code objects appear on the leaves of the tree. The tree is traversed when converting a code object into an executable object that is exposed as a delegate with signature *Signature*.

5.2 Argument binding

Argument binding in general requires creating a closure. However, in the case when the argument is a value type, the binding can be directly stored in the code by introducing an assignment to a local variable. For instance, in:

```
public class Test {
  public static void foo(int i) {
    Console.WriteLine("Argument is {0}", i);
  }
};

Code c = new Code(typeof(Test).GetMethod("foo"));
Code d = c.Bind(1);
```

the generated code would look like this:

```
public static void foo1() {
  int i = 1;
  Console.WriteLine("Argument is {0}", i);
}
```

An optimized version of the code would just use the constant `1` instead of the variable `i`. Such optimizations can be performed while generating code, as discussed in [2].

Binding a reference type rather than a value type requires instead the use of an *environment*, i.e. an array of object references. Storing the reference in the environment also ensures that it remains visible to the garbage collector and not collected throughout the lifetime of the code object.

Code generated for code objects using reference constants refers to them indirectly through the environment. For instance, from:

```
public class Test {
  public class Bar { … }

  public static void foo(Bar b) { … b … }
};

Code c = new Code(typeof(Test).GetMethod("foo"));
Bar b = new Bar();
Code d = c.Bind(b);
```

the generated code for `d` would look like:

```
public static void d(object[] env) {
  … (Bar)env[0] …
}
```

5.3 Code generation

Code generation in *CodeBricks* does not involve invoking a compiler at run time: performing bindings requires simple linear

rewritings of sequences of IL instructions, already available as produced by the compiler for the methods present in the atomic code objects. The overhead of code generation at runtime is hence minimal, and can be recovered in a quite smaller number of uses than, e.g., in `C.

The tree within a `Code` object also holds the mapping between arguments and locals. During code generation the code stream is transformed redirecting the access to arguments and locals according to this mapping.

Arguments are used as placeholders in the IL associated with code objects. The code generation process uses operations on arguments as placeholders where values or code objects are placed. Consider again code object `cadd`, whose associated CIL code is:

```
ldarg.0
ldarg.1
add
ret
```

In the following code object

```
Code cinc = cadd.Bind(1, Code.FreeVar);
```

the constant value 1 can be used instead of the value of argument 0, since argument 0 of `cadd` is read-only and has a value type:

```
ldc.i4.1
ldarg.0
add
ret
```

Arguments that remain open, being bound to a free variable, are renumbered: in this case argument 1 becomes argument 0.

When a code object is passed as argument to `Bind()`, its arguments are lifted in the signature of the new code object. For example, in:

```
Code cadd3 = cadd.Bind(cadd, new FreeVar());
```

the CIL associated with `cadd3` is:

```
ldarg.0
ldarg.1
add
stloc.0
ldloc.0
ldarg.2
add
ret
```

Code generation consists in generating the code for the first argument of `cadd`; the value returned is stored into a local variable that is used instead of any reference to argument 0 in `cadd`.

Higher order arguments in code objects are dealt as follows. Consider the C# method `Foo`:

```
delegate int F(int i, int j);
public static int Foo(int i, F f) {
  return f(i, 1) + i;
}
```

whose compiled CIL is:

```
ldarg.1
ldarg.0
ldc.i4.1
callvirt F::Invoke
ldarg.0
add
ret
```

The higher order argument `f` of `Foo` (argument 1) is loaded on the operand stack, then the arguments to be passed to `it` (argument 0 and constant 1) and finally the method `Invoke` is called to perform the call. When a `Spliced` object is used as actual parameter for a higher order argument, the code generator replaces the call to `Invoke` with the code of the actual method parameter, using local variables to simulate the arguments array needed by the method.

The CIL transformation performed by *CodeBricks* is sound and typesafe. It is also efficient since it consists in merging and filtering CIL instruction streams, applying simple transformations to arguments and local variables.

The quality of the generated code is not yet optimal, however standard code optimizations are delegated to the JIT compiler, exploiting the two-stage compilation model of the .NET framework. Analysis of the generated code has shown the presence of frequent code patterns that the current JIT does not handle, but for which specific optimizations might be added, significantly improving the performance of the final code.

6. APPLICATIONS

We present two applications of our code generation technique.

In the first example we show how to perform partial evaluation on the expression computing the power of integers.

The second example is an application of code generation in the domain of regular expressions. We have rewritten a compiler for regular expressions using *CodeBricks*.

Finally we discuss potential uses of *CodeBricks* in the implementation of Domain Specific Languages.

6.1 Partial Evaluation

We show how to handle in *CodeBricks* a classical example of partial evaluation: generating a specialized version of the x^y function for a given value of y. The process of partial evaluation is driven by the program itself, rather than being dealt by an automated tool. It is conceivable though to build a partial evaluator that manipulates the expression stored within a `Code` object to automate the task. The function `Power` generates a code object that computes the power function for a given exponent without loops.

The algorithm relies on the following equivalences:

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

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

that lead to the following recursive function:

```
static int power(int x, int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return x;
  else if (n % 2 == 0) {
    int v = pow(x, n / 2);
    return v * v;
  } else // y is odd and > 1
    return x*pow(x, n - 1);
}
```

The code generator has the same pattern: method `Power()` is identical to `power()` except that it uses `Bind` instead of application.

```
class PowerGen {
   static Code one =
        new Code(typeof(Power).GetMethod("One"));
   static Code id  =
        new Code(typeof(Power).GetMethod("Id"));
   static 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(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) {
      Code x = new FreeVar();
      return Power(id.Bind(x), n);
    }
}
```

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 `FreeVar` object to its first argument.

The code generated by `Power(5)` corresponds to the following method:

```
public static int power5(int x) {
 int l1 = x;       // λx.x = Power(x, 1)
 int l2 = l1 * l1; // λx.x * x = λx.x^2 = Power(x, 2)
 int l3 = l2 * l2; // λx.(x^2) * (x^2) = Power(x, 4)
 return x * l3;    // λx.x * (x^2)^2 = Power(x, 5)
}
```

6.2 Regular Expression Compiler

Using *CodeBricks* we implemented a regular expressions compiler, modelled after the one provided in the .NET Base Class Library present within the SSCLI [14] in directory

`fx/src/regex/system/text/regularexpressions`

Regular expressions matching in .NET can be performed either in interpreted or compiled mode. Class `RegexRunner` scans the input and coordinates the pattern matching process.

The RE matcher uses a Domain Specific Language for RE matching (that we call REML) to code a non-deterministic finite state automaton matching a given regular expression.

The RE parser generates for instance the following REML program for the pattern *xy|xx*:

```
 1: RegexCode.Lazybranch 8
 2: RegexCode.Setmark
 3: RegexCode.Lazybranch 6
 4: RegexCode.Multi 0
 5: RegexCode.Goto 7
 6: RegexCode.Multi 1
 7: RegexCode.Capturemark 0 -1
 8: RegexCode.Stop
 9: (RegexCode.Lazybranch | RegexCode.Back) 8
10: (RegexCode.Setmark | RegexCode.Back)
11: (RegexCode.Lazybranch | RegexCode.Back) 6
12: (RegexCode.Capturemark | RegexCode.Back) 0
```

The program consists in two parts: the first part tries to match in turn the two alternatives (*xy* at line 4, *xx* at line 6) of the pattern while annotating two stacks used for backtracking. The second part of the program (lines 9-12) contains instructions to perform various cases of backtracking.

Given a REML program, we can compile it using *CodeBricks*. The compiled program may look like this: it fetches the next instruction, indicated by the program counter `pc`, and executes it:

```
int pc = 1;
while (pc != -1) {
  switch (pc) {
  case 0: // Backtracking logic
    EnsureStorage();
    pc = runtrack[runtrackpos++];
    break;
  case 1: // RegexCode.Lazybranch 14
    runtrack[--runtrackpos] = runtextpos;
    runtrack[--runtrackpos] = 9;
    pc = 2;
    break;
  case 2: // RegexCode.SetMark
    //...
  }
}
```

The matcher program starts at instruction 1, instruction 0 is used to perform backtracking, i.e. the address of the next instruction is popped from the backtrack stack and determines the next instruction.

The following code combinator is used to generate the main loop:

```
public static void _Pump(RegexRunner re, IRI sw) {
  int pc = 1;
  while ((pc = sw(re, pc)) != -1);
}
```

`IRI` is a delegate type representing methods with two arguments (a `RegexRunner` and an integer) returning an integer.

The inner switch is made out of bricks corresponding to each instruction. The code for an instruction is just a specialization for the given operand of the code used by the interpreter for that REML instruction. For instance, the `RegexCode.Multi` instruction tries to match a character sequence and is implemented in the interpreter as follows:

```
case RegexCode.Multi:
  if (!Stringmatch(runstrings[Operand(0)]))
    break;
  Advance(1);
  continue;
```

`runstrings` is an array of constant strings, appearing in the pattern. The `Stringmatch` method compares the input with its argument.

Since the string is a known constant we can unroll the loop within `Stringmatch`. The following code is used in the *CodeBricks* version of the compiler for generating the code brick for an instance of the `Multi` instruction:

```
public static
int _CharCheck(RegexRunner re, int pos, int ch) {
  return re.runtext[re.runtextpos + pos] == ch ?
         0 : -1;
}
public static Code CharCheck = new
   Code(typeof(CBCompiler).GetMethod("_CharCheck"));

public static int _CharCheckMid(RegexRunner re, int pos,
int ch, IR rest) {
  return re.runtext[re.runtextpos + pos] == ch ?
         rest(re) : -1;
}
public static Code CharCheckMid = new
   Code(typeof(CBCompiler).GetMethod("_CharCheckMid"));

public static int _MultiSkel(RegexRunner re, int len,
                             IR checkstring, int go) {
  if (len > re.runtextend - re.runtextpos) return 0;
  if (checkstring(re) == -1) return 0;
  runtextpos += len;
  return go;
}
public static Code MultiSkel = new
   Code(typeof(CBCompiler).GetMethod("_MultiSkel"));
```

```
public Code Multi(int operand, int instnum) {
  string s = runstrings[operand];
  Code c = CharCheck.Bind(new FreeVar(), 0, (int)s[0]);
  for (int i = 1; i < s.Length; i++)
   c = CharCheckMid.Bind(new FreeVar(), i, s[i], c);

  return
   MultiSkel.Bind(new FreeVar(), s.Length,
                  new Spliced(c), instnum);
}
```

The `MultiSkel` brick provides the skeleton of the operand's implementation. `checkstring` is a higher order argument used as placeholder for testing the string. `CharCheck` and `CharCheckMid` are used to unroll the tests needed for matching the string. Method `Multi` combines the bricks generating the unrolled version of sequence matching.

The compiler iterates over the REML program for a given pattern invoking the appropriate code brick generator. The structure of the compiler is almost the same as the interpreter except that code is generated instead of performing the computation.

The source of the original .NET RE compiler is more than 3,000 lines of code. The version written with *CodeBricks* is about 1,400 lines of code and generates quite similar CIL code. To see where this difference comes from, consider for instance how the `GetMark` opcode is handled by the .NET RE compiler:

```
case RegexCode.GetMark:
  ReadyPushTrack();
  PopStack();
  Dup();
  Stloc(_textposV);
  DoPush();
  Track();
break;
```

Method `PopStack` emits code in the following way:

```
internal void PopStack() {
   _ilg.Emit(OpCodes.Ldloc_S, _stackV);
   _ilg.Emit(OpCodes.Ldloc_S, _stackposV);
   _ilg.Emit(OpCodes.Dup);
   _ilg.Emit(OpCodes.Ldc_I4_1);
   _ilg.Emit(OpCodes.Add);
   _ilg.Emit(OpCodes.Stloc_S, _stackposV);
   _ilg.Emit(OpCodes.Ldelem_I4);
}
```

This sequence corresponds to the following single C# instruction:

```
stack[stackpos++];
```

The functions `ReadyPushTrack`, `DoPush` and `Track` behave similarly. The equivalent C# code produced for the `GetMark` opcode is the following:

```
int s = stack[stackpos++];
track[--trackpos] = s;
textpos = s;
track[--trackpos] = state;
```

This sequence is used as a single brick in the *CodeBricks* version of the RE compiler, relying on the C# compiler to expand it into a sequence of CIL instructions.

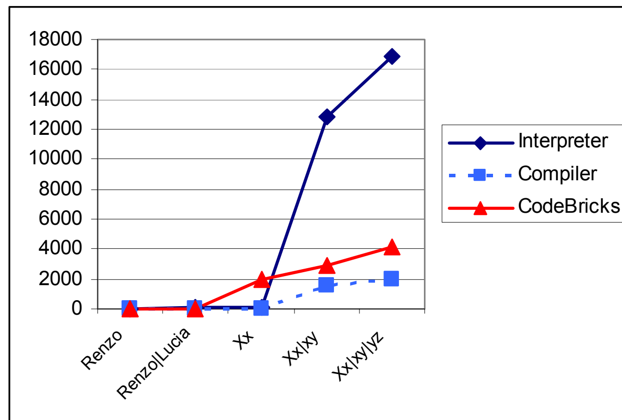
6.3 Benchmarks

We compared the .NET RE matcher, in both interpreted and compiler versions, with the one produced with *CodeBricks* in the task of searching a file of about 7.5 MB, made of six copies of Manzoni's novel "I Promessi Sposi".

The test program stops at the first match: the patterns including the string "Xy" never match and are used to force the matcher to scan the whole file.

We tested several RE patterns, measuring for each pattern the compilation time and the times for the first and second run. We tested so far the worst case for *CodeBricks*, using code compiled in debug mode without optimization against the output of the handcrafted compiler, which is hand optimized.

The following table reports the times in milliseconds for the second run of various patterns.



The *CodeBricks* version is slower than the hand-written one in this setup. This is mostly due to the lack of a number of trivial optimization that could be applied to the generated CIL (redundant load/store, etc.) and also by the debug setting. We feel confident that with such optimizations we will be able to match the performance of the handcrafted compiler.

We also report that *CodeBricks* compilation time is 2 times faster.

These early experiments confirm our goal of achieving comparable performance to low-level code generation, while performing code manipulation at the high-level language.

6.4 Domain Specific Embedded Languages

The previous example can be considered as a special case of implementation of a Domain Specific Embedded Languages (DSEL). Hudak [8] suggested embedding a DSL into another language as a way to simplify the implementation of a DSL, exploiting the fact that a large part of a DSL is not domain specific, so it could rely on the facilities of the host language. Lisp macros have often been used for this purpose. The Jakarta Tool Suite [1] allows building DSLs in an extensible superset of Java that provides Abstract Syntax Tree constructors and hygienic macros. A software generator uses the DSL to create the program of interest.

Embedding a DSL into an existing host language allows inheriting its standard mechanisms and facilities, including compilers and tools, and to integrate fully with the host language. However the domain specific parts of the language are not handled as well as in a native implementation, since the compiler has no knowledge of the domain and the new features are handled as a layer on top of the existing language, often losing significantly in performance. Techniques like partial evaluation [9] might be helpful to eliminate interpretation steps by transforming further the generated code into code operating at the innermost level. Unfortunately partial evaluation is not suitable yet for fully automated code generation tasks.

The goal of embedding a special purpose language in a general purpose language can be achieved also through the technique of

template metaprogramming [5]. Exploiting C++ templates, one can write template meta-programs that are executed during compilation by the type checker. The technique can be used to perform code selection and code generation at compile time. Blitz++ [24] provides a sophisticated DSL for matrix algebra built through template metaprogramming. The library achieves high performance in matrix computations by performing a number of optimizations (unrolling, fusion, partial evaluation) that rely on detailed knowledge of the domain and by specializing code to each particular combination of the arguments to a function.

In [2], Cisternino shows how to use *CodeBricks* to implement a matrix DSL similar to Blitz++, except that specialization is performed at run-time instead of compile-time.

7. RELATED WORK

Program manipulation and run-time code generation has a long history. We mention just a few notable cases of programming languages or systems that provide such facilities.

In Lisp [20] programs are represented as S-expressions and macros can exploit the whole power of the language to manipulate and produce programs. Macros are written using a notation for quasi-quotation that involves a quasi-quote operator (`), an eval operator (,) and a list splice operator (@). For instance a while construct can be defined as follows:

```
(defmacro while (test . body)
        `(do () (,test) ,@body))
```

`C [16] and DynJava [15] provide a similar mechanism for the C language and Java respectively, based on a quasi-quote (`) and eval operators (@ and $). For example:

```
int cspec s1 = `4; int cspec c2 = `5;
int cspec c = `(@c1 + @c2);
```

`cspec` is the type of a dynamic scoped specification, and `c` is the additive composition of the two previous `cspecs`.

Tempo [4] performs partial evaluation on C programs, based on annotations by the user about which parameters are to be considered constants. Specialization can also be done at runtime, by generating binary code templates containing holes to be filled with constants at runtime. *CodeBricks* code objects are more flexible than Tempo's templates, since they can be combined with arbitrary code objects, not just constants and the process can be repeated.

MetaML [21] extends the language ML with a code type, used to represent programs. MetaML allows the programmer to construct, combine, and execute code fragments in a type-safe manner.

Kamin [10] introduces code objects for building software components. He uses both a string representation and a binary form, with an associated abstract machine.

Thiemann [23] suggests an approach based on code splicing as an alternative to non-portable solutions like RTCG and JIT compilation. However, in his approach the transformations are done at the source level and the results are metalevel expressions; such metalevel expressions are said to be "compiled", but in reality are just passed to the interpreter, through eval or apply. As a consequence, the performance of the generated program, degrades, sometimes significantly, even with respect to normal interpretation. On the contrary, we believe that *CodeBricks* addresses properly both portability and efficiency.

7.1 External Tools to the Language

Program manipulations are often preformed by tools external to a language.

Preprocessors and source-to-source transformation tools work on the program text or abstract syntax tree and produce new source programs. Besides programming language preprocessors, a popular generator tool is Active Server Pages (ASP) that provides a notation for embedding programming constructs within the text to be produced, like in the following example:

```
<% for (var i = 3; i != 8; ++i) { %>
   <font size=<%=i %>>Hello</font><br>
<% } %>
```

The `<% %>` enclose JScript statements; the first is a `for` statement, wrapping the text to be output. The `<%= %>` block outputs the value of the expression inside the block – in this case, the loop variable. Everything else is output verbatim.

An XML-based code generation approach has also been suggested [13][16], using XSLT transformations to generate programs. For instance, given the following XML definition:

```
<Enumerations>
 <EnumDef name = "ActionEnum">
 <choices>
  <choice name = "start" value = "1"/>
  <choice name = "stop" value = "2"/>
  <choice name = "pause" value = "3"/>
 </choices>
 </EnumDef>
</Enumerations>
```

the switch statement on the right can be synthesized from the XSLT script on the left:

<table>
<tr>
<td><pre>switch(enumValue) {
<xsl:for-each
   select='choices/choice'>
  case <xsl:value-of
   select='@value'/>:
    return "<xsl:value-of
   select='@name'/>;
</xsl:for-each>
  default:
    return null;
}</pre></td>
<td><pre>switch(enumValue)
 {
  case 1:
    return "start";
  case 2:
    return "stop";
  case 3:
    return "pause";
  default:
    return null;
}</pre></td>
</tr>
</table>

7.2 Programming Environment Facilities

Programming environments obviously perform program manipulations, but systems based on more and more sophisticated code manipulation techniques are being proposed to handle the complex requirements of software development.

Intentional Programming [19] relies on a programming environment that supports language extensibility through syntactic rewrites. A whole language can be expressed as a collection of cooperating transformations.

Aspect Oriented Programming [11] relies on creating programs by weaving pieces of code that express crosscutting concerns.

8. CONCLUSIONS

We presented a framework for dynamic code generation that allows expressing code objects by composing code fragments written in the same language.

CodeBricks relies on an IL that provides enough type information so that the generated code is type safe and verifiable. Code transformations are expressed at the source level but performed on the IL code. Since such code fragments are precompiled, code

generation is quite fast. If the abstract machine is common to several languages as in .NET, the ability of building code objects from compiled methods allows also cross-language code generation.

We have reported our experience in building a RE compiler that has shown the effectiveness of the approach and satisfactory performance results.

CodeBricks only provides basic mechanisms, and as such it may appear more cumbersome to use with respect to the quasi-quote notation provided by languages like Lisp, MetaML, `C, DynJava. Indeed we envisage *CodeBricks* to be used by language designers to implement language extensions (e.g. MetaC#) or domain specific languages, rather than to be used directly.

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 envisage applications of our framework in various areas of generative programming: compiler writing, in particular for DSL; aspect oriented programming.

We plan to investigate in particular the use of the framework to provide support for multi-stage programming, exploiting the fact that program transformations are performed at the binary level without requiring a language processor (interpreter or compiler) and hence can be applied even at completely separate execution stages.

9. ACKNOWLEDGMENTS

We wish to thank Don Syme, Claudio Russo, Nick Benton, Luca Cardelli, Cedric Fournet, Peter Sestoft and Geoff Shilling for their contributions in discussions and suggestions about our work. Worked reported herein was done in part while the second author was visiting Microsoft Research, Cambridge.

We thank an anonymous reviewer for suggesting a nice concise description of our approach.

10. REFERENCES

[1] Batory, D., Lofaso, B., Smaragdakis, Y. JTS: A tool suite for building GenVoca generations. In Proceedings of 5th International Conference on Software Reuse, IEEE/ACM, 1998.

[2] Cisternino, A. Multi-stage and Meta-programming support in strongly typed execution engines. PhD Thesis, Dipartimento di Informatica, Università di Pisa, 2003.

[3] Cisternino. A. CLIFileReader library. http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103.

[4] C. Consel, L. Hornof, F. Noël, J. Noyé, S. Thibault, and N. Volanschi. *Tempo: Specializing systems applications and beyond*. ACM Computing Surveys, Symposium on Partial Evaluation, 30 (3), 1998.

[5] Czarnecki, K. a nd Eisenecker, U.W. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, 2000.

[6] Engler, D.R. *VCODE: a retargetable, extensible, very fast dynamic code generation system*, in Programming Language Design and Implementation, 1996.

[7] A. Fischbach, and J. Hannan. *Specification and Correctness of Lambda Lifting*. W. Taha (Ed.), SAIG 2000, LNCS 1924, pp. 108-128, Springer-Verlag, Berlin, 2000.

[8] Hudak, P. Modular Domain Specific Languages and Tools. http://www.cs.chalmers.se/Cs/Grundutb/Kurser/afp/Papers/dsel-hudak.ps

[9] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[10] Kamin, S., Callahan, M., Clausen, L. *Lightweight and Generative Components II: Binary-Level Components*. W. Taha (Ed.), SAIG 2000, LNCS 1924, pp. 28-49, Springer-Verlag, Berlin, 2000.

[11] Kiczales, G., et al. *Aspect Oriented Programming*. In Proceedings ECOOP'97 – Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 1997, M. Aksit and S. Matsuoka (Eds.), LNCS 1241, Springer-Verlag, Berlin and Heidelberg, Germany, 1997.

[12] Levine, J., Mason, T., Brown, D. *Lex & yacc 2nd Edition*. O'Reilly, 1992.

[13] Metacoder. See Web site http://metacoder.info.

[14] Microsoft Corp. Shared Source CLI. http://msdn.microsoft.com/net/sscli/

[15] Y. Oiwa, H. Masuhara, A. Yonezawa. DynJava: type safe dynamic code generation in Java. *JSST Workshop on Programming and Programming Languages*, PPL2001, Tokyo, 2001.

[16] M. Poletto, W. C. Hsieh, D. R. Engler, and M. Frans Kaashoek. `C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2), 1999, 324-369.

[17] Sarkar, S., Cleveland, C. Code Generation Using XML Based Document Transformation. http://www.theserverside.com/resources/articles/XMLCodeGen/xmltransform.pdf, 2001.

[18] Sestoft, P. Runtime Code Generation with JVM and CLR. http://www.dina.dk/~sestoft/rtcg/rtcg.pdf, 2002.

[19] Simonyi, C. The Death of Computer Languages, the Birth of Intentional Programming. NATO Science Committee Conference, 1995.

[20] Guy L. Steele, *Common Lisp the Language*, 2nd edition, Digital Press, 1990.

[21] Taha, W., Sheard, T. *Multi-stage programming with explicit ennotations*. In Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam, p. 203-217. ACM, 1997.

[22] The .NET Common Language Runtime. See Web site http://msdn.microsoft.com/net.

[23] Thiemann P., *Higher-Order Code Splicing*, European Symposium on Programming, ESOP '99, LNCS 1576, Amsterdam, March 1999.

[24] Veldhuizen, T.L. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments* (ISCOPE'98), Springer-Verlag, 1998.