Reflective Program Generation with Patterns

Manuel Fähndrich
Microsoft Research
maf@microsoft.com

Michael Carbin
Stanford University
mcarbin@cs.stanford.edu

James R. Larus
Microsoft Research
larus@microsoft.com

Abstract

Runtime reflection facilities, as present in Java and .NET, are powerful mechanisms for inspecting existing code and metadata, as well as generating new code and metadata on the fly. Such power does come at a high price though. The runtime reflection support in Java and .NET imposes a cost on all programs, whether they use reflection or not, simply by the necessity of keeping all metadata around and the inability to optimize code because of future possible code changes. A second—often overlooked—cost is the difficulty of writing correct reflection code to inspect or emit new metadata and code and the risk that the emitted code is not well-formed.

In this paper we examine a subclass of problems that can be addressed using a simpler mechanism than runtime reflection, which we call compile-time reflection. We argue for a high-level construct called a transform that allows programmers to write inspection and generation code in a pattern matching and template style, avoiding at the same time the complexities of reflection APIs and providing the benefits of staged compilation in that the generated code and metadata is known to be well-formed and type safe ahead of time.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Languages, Design, Verification

Keywords Reflection, Generative Programming, Patterns, Templates, C#

1. Introduction

The ability to reflect over a program's metadata (types and members) as well as code—commonly called reflection—enables both highly dynamic applications (such as runtime upgrading of code), as well as generative programming. Today's mainstream platforms such as Sun's JVM and Microsoft's .NET execution environment support reflection.

Reflection appears to the programmer as an API that provides access to metadata and code, as well as facilities to emit new metadata and code. Using these APIs is not easy. Programmers usually need to be familiar with basic programming language implementation techniques to successfully master the construction of new types and code using reflection. Even just writing reflection code that inspects metadata is tedious to write and tricky to get right.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010...$5.00

We are interested in providing main-stream programmers with a very simple form of generative programming that avoids the complications of reflection APIs and statically guarantees the well-formedness of the generated code and metadata.

A natural starting point for such an investigation is to provide metadata pattern matching facilities and then let programmers pass the pattern matching results to code generating templates. However, in order to statically guarantee the well-formedness of the generated code and metadata, patterns must contain quite a bit of information about the matching context. For example, it might be necessary to know that a certain type is a subtype of another type, that a type contains (or does not contain) a certain member (e.g., a constructor), or that a method has a certain set of formal arguments. These requirements not only force patterns to be precisely specified, but also mandate the transmission of all this contextual information from the pattern to the generation template. Our insight is that these two needs are best addressed by combining patterns and generation templates into a single construct that we call a *transform*.

In this paper we describe our design called *Compile-Time Reflection* (CTR). CTR aims to provide a powerful, yet accessible replacement for complicated reflection APIs by adopting the following principles:

- CTR allows inspection of static metadata, but not code nor runtime values. While this choice limits the possible applications, it 1) allows for static compilation and full compiler support, 2) enables static checks that prove the well-formedness of all subsequently generated code, and 3) allows generative programming in environments that don't have runtime reflection mechanisms (e.g. [12]).
- We focus on the generation and addition of new metadata and code, not the modification of existing elements. This makes it easier to reason about the behavior of the final generated code and its well-formedness.
- CTR does not provide a programmatic reflection API. Both metadata inspection and code generation are done via patterns and templates that look like ordinary C# code. We believe that complex reflection APIs present a significant hurdle for programmers who want to quickly and easily write code generators.
- New syntactic constructs are kept to an absolute minimum. We avoid the explicit quoting and unquoting conventions of many generative and macro programming approaches as they can often make writing code generators unwieldy. Tight C# integration means that programmers need only learn the proper use of meta-variables and a few new, but intuitive, keywords.
- Transforms are applied during target application compilation. This allows the compiler to help programmers by emitting errors if a compilation would produce missing types, members,

or code due to patterns not matching or transforms failing to generate all expected code.

- Transforms can be compiled and distributed like regular code. Because of our close C# integration, transforms can be compiled to standard MSIL. This allows transforms to be distributed along with libraries and other such distributions with ease. Moreover, since the well-formedness properties of transforms are checked at transform compile time, our safety guarantees still hold.

The rest of the paper is organized as follows: Section 2 gives an overview of transforms using a simple example. Sections 3 and 4 describe the general form of transform patterns and generation constructs. Section 5 defines in more detail how transforms match target contexts. Section 6 provides an informal discussion of our safety and well-formedness claims. Section 7 shows how transforms can be used to help implement a software transactional memory system and automate process startup. Section 8 contains more details on our implementation and Section 9 discusses related work. Shortcomings of our design are explored in Section 10.

2. Simple example

Consider the following generative programming problem: To automate the generation of unit test harnesses for our code base, we want to mark entry points for unit testing using attributes in our code and then generate methods that invoke all unit tests. Here's the transform that achieves this task:

```
 1 public class UnitTestEntry : Attribute {
 2   public UnitTestEntry(string name) {}
 3 }
 4
 5 transform GenerateUnitTestHarness {
 6
 7   public class $$C {
 8     public $$C();
 9
10     [UnitTestEntry($name)]
11     void $$TestMethods();
12
13     generate public static void UnitTest() {
14       $$C c;
15       forall ( $m in $$TestMethods ) {
16         c = new $$C();
17         Console.WriteLine("Invoking test '{0}'", $m.$name);
18         c.$m();
19       }
20     }
21   }
22
23   generate class ModuleUnitTest {
24     public static void Main() {
25       forall ( $TC in $$C ) {
26         $TC.UnitTest();
27       }
28     }
29   }
30 }
```

We can apply this transform to the unit below.

```
using System;
using Microsoft.SingSharp.Reflection;

[assembly:Transform(typeof(GenerateUnitTestHarness))]

public class ClassA {
    public ClassA() {}

    [UnitTestEntry("A1")]
    void MethodA1() {}
    void MethodA2() {}
    [UnitTestEntry("A3")]
    void MethodA3() {}
    void MethodA4() {}
}

public class ClassB {
    public ClassB() {}

    void MethodB1() {}
    [UnitTestEntry("B2")]
    void MethodB2() {}
    void MethodB3() {}
    [UnitTestEntry("B4")]
    void MethodB4() {}
}
```

When invoking the generated method ModuleUnitTest.Main in the final executable, the following output is produced:

```
Invoking test 'A1'
Invoking test 'A3'
Invoking test 'B2'
Invoking test 'B4'
```

We now explain the elements of this example in detail. Lines 1–3 define a standard C# attribute called UnitTestEntry that takes a string argument. We use this attribute in our units to mark methods that serve as unit test entry points. The string argument can be used to give each test a descriptive name.

2.1 Patterns

Each transform is named. Our example transform starts on line 5 and is called GenerateUnitTestHarness. The transform has two pattern members (here two classes). The first class $$C is on lines 7–21 and the second class ModuleUnitTest on lines 23–29. The first class pattern has a meta-variable $$C, meaning it matches classes of any name. Furthermore, the meta-variable is a multi-match variable (variable starting with two $$ signs) as opposed to a single-match variable (starting with a single $). The first pattern thus matches a list of classes, where each class's members must match the members in the $$C pattern. The pattern $$C has three members, a constructor pattern, a method pattern $$TestMethods, and a static method generation pattern UnitTest. Note that within the scope of class pattern $$C, the name $$C refers to exactly one element of the eventual set of matches for $$C. Thus, the constructor pattern matches only one constructor, not a set (not that a class could have multiple nullary constructors anyway).

The $$TestMethods method pattern matches any number of instance methods that have the UnitTestEntry attribute, return **void** and take no parameters. First note that the pattern involving the attribute is not a match by name. The pattern does not match all attributes called UnitTestEntry. Instead, it refers directly to the declaration of UnitTestEntry used during the compilation of the transform. This attribute need not be defined in the same compilation unit as the transform. It could have been defined in a separate compilation unit referenced by the compilation of the transform. The type checking of the transform however guarantees that the attribute type exists. This is in contrast to reflection based code which is always dependent on matching names as character strings.

Second, note that the expression meta-variable $name is bound within the context of a single method of $$TestMethods, meaning that each method matched by $$TestMethods can have a distinct name. This is determined by the fact that $name does not appear in a matching position outside of the $$TestMethods scope.

The method generation pattern UnitTest matches exactly when a class under consideration does *not* have such a method. In other words, generation patterns are anti-patterns that match if they don't clash with existing members. This guarantees that we don't generate a member that is already present in a target context. Similarly, the class generation pattern ModuleUnitTest matches only if there is no top-level class called ModuleUnitTest in the target compilation unit.

2.2 Generation

Let's turn our attention to the body of method generation pattern UnitTest. It contains ordinary C# code with the extension of the use of meta-variables and the **forall** construct that allows iteration over all matches of multi-match meta-variables. Recall that since UnitTest is within the context of class pattern $$C, type $$C refers to a single type, not the list. Thus, the body starts out declaring a local variable of type $$C. The next statement is a **forall** iteration statement. It generates a sequence of blocks, one for each match in the set of matches being iterated. Here, we iterate over all methods $m in the set $$TestMethods. For each such method, we create a fresh object of type $$C and assign it to local c. We then write a message to the console describing which test is being executed. Note the use of $m.$name to refer to the string argument of the UnitTestEntry attribute of the matched method $m. We use this qualified path to access such dependent information. Finally, we invoke the test method on c using the usual method invocation syntax, albeit using a bound meta-variable for the method name.

2.3 Well-formedness

Consider what the compiler needs to know about the matching context in order to guarantee that the body of UnitTest is well-formed. The declaration of local c requires knowing that type $$C exists and is distinct from **void**. Clearly, we are in the context of a class $$C and thus $$C cannot be **void**. In fact our design is such that meta-variables never match type **void**, only the literal type **void** matches **void**. We have found this to be the most useful approach in practice. The **forall** construct binds a single method $m of the set $$TestMethods. We know from the method pattern $$TestMethods, that $m thus is an instance method of $$C taking no arguments. This information is sufficient to check the method call on line 18. On line 16, we exploit the fact that the constructor pattern establishes the existence of a constructor of $$C taking no parameters. Finally, on line 17, we know that there was an attribute on $m and we thus have a binding for $m.$name.

Observe that all these constraints naturally follow from the patterns themselves. In this example, we did not need to further constrain the match to guarantee well-formedness. Clearly, the constructor pattern was motivated by the need to construct an object of type $$C. If we matched static test methods instead, that pattern could have been omitted.

The well-formedness of the generated class ModuleUnitTest can be argued similarly. It generates a single Main method which iterates over all classes $TC in the multi-match pattern $$C and calls $TC's UnitTest method. Note that in order to check the validity of this method call, we are again using the knowledge gleaned from the combination of the pattern and generation templates in that we know for a fact that we generate the static method UnitTest for each class $TC. Were we to use an approach that would allow one to splice together generated code from a variety of sources, the necessary reasoning would be much more complicated. This insight, we believe, is a major contribution of this work.

2.4 Transform application

Figure 1 shows the general model how to compile and apply transforms. Given a set of source files T containing one or more

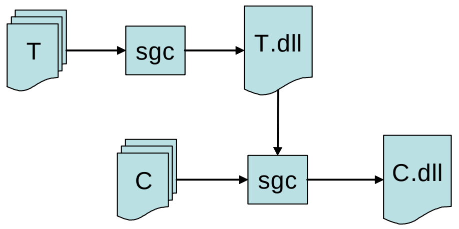

Figure 1. Transform compilation and application

transforms, our compiler (sgc) checks the transforms for well-formedness and produces a library T.dll containing a compiled form of the transforms. To apply any of the transforms in T.dll, the programmer adds a [Transform(t)] attribute into his client code C, specfiying which transform t is to be applied (see the unit test code on line 4). The compilation of the client code C thus references the assembly containing the transforms (T.dll). After compilation of the client code C, the desired transform is matched against the entire compilation unit C. If it matches unambiguously, the effects of the transform are applied at the MSIL level, prior to writing the resulting library C.dll. Otherwise, the compiler emits a message explaining why the transform could not be applied.

3. Transforms

A transform is a combination of patterns and code[1] to be generated. As the introductory example illustrated, this combination allows the pattern to naturally provide the constraints required to check the well-formedness of the generated code. This section describes in more detail the elements of transforms and their well-formedness conditions.

3.1 Meta-variables

Meta-variables serve both to bind elements to be matched and to refer to such matches. Transforms contain two kinds of meta-variables, single-match variables, starting with a single $ sign, and multi-match variables, starting with a double $$ sign. As their names suggest, a single-match variable matches a single element, whereas a multi-match variable matches zero or more elements.

3.1.1 Variable scopes

Each meta-variable in a transform is assigned a scope. Nested scopes are introduced at each multi-match meta-variable and extend for the subtree matched by that list. These sub-scopes exist to maintain the correlation between sub-parts of each match of a multi-match variable. The scope of a multi-match meta-variable is its own scope.

The binding occurrence of a multi-match variable is either a member[2], an attribute name, or the formals of a method pattern. Multi-match variables can only be referenced from within their own scope (if referring to a type), or in a context where a list is expected (such as a **forall** iteration).

The scope of a single-match meta-variable is the outermost scope in which it occurs in a binding position. A binding position is any occurrence other than in a method body or in a signature of a generation pattern. For example, the GenerateUnitTestHarness transform in Section 2 has three scopes. The outermost scope is the transform itself. Inside that scope is the scope of multi-match variable $$C. Within that scope is the scope of multi-match variable

[1] To avoid repeating the phrase "code and metadata", we use the word "code" to mean both declarations of types and members, as well as code bodies when used in the context of generation.

[2] Classes and structs are considered members since they can appear nested inside other types.

$$TestMethods. This is also the scope of the single-match variable $name.

Single-match variables are also used to range over the elements of a multi-match in **forall** constructs. Such single-match variables cannot be referenced outside the binding **forall** construct.

3.2 Member patterns

A transform is made up of type member patterns such as classes and structs at top-level. Each type pattern in turn is made up of general member patterns corresponding to C# entities, such as fields, methods, events, properties, and nested types. We distinguish between ordinary member patterns not prefixed by **generate** or **implement**, and generation or implementation patterns. The next sections describe ordinary patterns, followed by generation- and implementation-patterns. The matching of entire member lists is described in Section 5.

3.2.1 Fields

A field pattern consists of a name (possibly a meta-variable), a type reference pattern, a visibility (public, protected, internal, private), an optional static modifier, and optional attribute patterns, but no initializer. It matches a target field if all pattern parts match the corresponding actual characteristics.

3.2.2 Methods

A method pattern consists of a name, a type reference pattern for the return type, a formal argument list pattern, a visibility, an optional static modifier, and optional attribute patterns for the method and or the return type, but no method body. It matches a target method if all the pattern parts match the corresponding actual characteristics.

3.2.3 Properties

A property pattern consists of a name, a type reference pattern, a visibility, an optional static modifier, and optional attribute patterns. In addition, the property pattern indicates the presence of a getter and/or setter, like abstract property declarations do, without method bodies. Property patterns match if all pattern parts match the corresponding actual characteristics, including the presence of a getter and/or setter if the pattern requires it.

3.2.4 Types

An ordinary type pattern is either a class or struct declaration. It consists of a name, visibility, optional attribute patterns, and a list of member patterns. Class patterns additionally indicate if they match abstract classes, a possible base class and interface constraints in the form of type reference patterns. A type member pattern matches a target type if the type kind (struct or class) matches, and all pattern parts match the corresponding characterstics. Additionally, class patterns match only if the actual type's base class is a subtype of the base class pattern and the actual class implements (possibly indirectly) all interfaces in the class pattern.

3.2.5 Implementation patterns

When writing code in conjunction with transforms, one often needs to be able to reference a member that will be generated by a transform. To provide a clear interface between original code and generated code that allows such forward references, our design provides the **reflective** qualifier to introduce such forward member declarations. Reflective declarations are similar to extern declarations, i.e., they do not provide bodies for methods, or initializers for fields.

Reflective members are matched by implementation patterns, i.e., member implementation patterns prefixed by the **implement** qualifier. An implementation pattern actually provides a definition of the forward declared member, such as a field initializer, a method body, getter and/or setter methods for properties. (Implementation type patterns do not differ from ordinary type patterns.)

An implementation pattern matches a target member only when the ordinary member pattern matches and in addition, the target member was declared as **reflective** . Thus, the linking points between non-generated and generated code are always formed by **reflective** members and corresponding **implement** patterns.

Consider a slight variation of the GenerateUnitTestHarness that does not generate a Main method, but a RunTests method. In order for a unit to contain code that calls the RunTests method, the unit uses a **reflective** place holder for this method and the transform uses an implementation pattern for this method.

```
transform GenerateUnitTestHarness2 {
  public class $$C {
      ...
  }
  public class $UnitTestClass {
    implement public static void RunTests() {
      forall ( $TC in $$C ) {
        $TC.UnitTest();
      }
    }
  }
}
```

This transform matches units that provide a class with a reflective method called RunTests.

```
using System;
using Microsoft.SingSharp.Reflection;

[assembly:Transform(typeof(GenerateUnitTestHarness2))]
public class ClassA {
  ...
}

public class ClassB {
  ...
}

public class UnitTesting {
  reflective public static void RunTests();

  public static void Main(string args) {
    // invoke unit tests if argument switch is selected

     ... UnitTesting.RunTests(); ...
  }
}
```

Class pattern $UnitTestClass matches class UnitTesting and implements the RunTests method. The UnitTesting class (or any other part of that compilation unit) can thus refer to the RunTests method and call it when desired.

3.2.6 Generation patterns

Whereas implementation patterns match only if the target context anticipates the implementation of such a member, a generation pattern is used to generate a member that the target module does not already have and does not refer to directly. A member generation pattern thus has the same parts as an ordinary member pattern, but it is prefixed by the **generate** modifier and contains an implementation. The implementation is either a method body, a field initializer, property getters and setters, or class and struct members.

The name of a member generation pattern can be a single-match meta-variable, which is then interpreted as a fresh identifier generator, guaranteed to not clash with existing member names. Otherwise, a generation pattern represents an anti-match, i.e., it matches a context only if it does not clash with any existing members in the target context according to the rules of the .NET platform.

For type generation patterns, all members of the type are implicitly considered generation patterns and implementation patterns are disallowed.

3.2.7 Scope patterns

Scope patterns are a special construct that has no equivalent in ordinary C# members. A scope pattern describes a set of subsets of members in the current scope, allowing the same kind of transformation on each subset. As an example, consider the following generative programming problem: Properties (or getter and setter methods in Java) are commonly used to abstract over field accesses. Most properties are backed up directly by a field in the containing type and programmers have to write boiler plate code for such patterns as follows[3]:

```
class C {
   T _XbackingField;
   public T XProperty {
      get { return this._XbackingField; }
      set { this._XbackingField = value; }
   }
}
```

This practice is useful in software engineering in that code evolution is easier to handle. If the property set and get operations change over time, only the methods have to be updated, not all clients. To ease generation of such default boilerplate code though, programmers might want the backing field and the setter and getter generated automatically by writing just the property as follows:

```
class C {
   [AutoProperty]
    reflective  public T XProperty { get; set; }
}
```

The idea of course is to use a transform to generate the backing field, as well as getter, and setter methods. Scope patterns allow us to express such a transform easily:

```
transform AutoPropertyTransform {
  public class $$C {
    scope $$AutoProperties {
      [AutoProperty]
      implement public $T $Property {
         get { return this.$backingField; }
         set { this.$backingField = value; }
      }
      generate $T $backingField;
    }
  }
}
```

A scope consists of a list of member patterns. The example contains a property implementation pattern and a field generation pattern. A scope matches any number of times in the member context in which it appears. Thus, the above transform would generate a backing field and setter and getter implementations for every reflective property with an [AutoProperty] attribute in every public class.

Note the natural way this transform solves this simple task. We know of no traditional generative programming approach that solves this as elegantly as the scope construct.

3.3 Type reference patterns

A type reference pattern corresponds to a type reference in C#. Here we consider only two forms, named types (class or struct) and array types. A named type pattern is either a literal type reference, which will match exactly that type, a meta-variable, which matches any type, or a type reference to a type within the transform, which matches whatever that type member matches. An array type reference pattern is then simply an array type whose element type is a type reference pattern.

In practice, it is often necessary to constrain the super types of a type reference pattern. We use syntax similar to that of generic constraints to achieve this effect as in the following example:

```
transform Test
  where $T : ICollection
{
  class $C {
    public $T GetCollection ();
  }
}
```

The **where** clause constrains a type reference meta-variable $T to implement the ICollection interface. Thus, the method pattern GetCollection only matches methods returning a type deriving from ICollection .

3.4 Attribute patterns

Attribute patterns occur wherever attributes can be written in C# programs. An attribute pattern consists of a named type reference pattern, and an expression list pattern. An attribute pattern matches a single attribute in the target context, unless the type reference pattern is a multi-match meta-variable, in which case it can match zero or more times.

3.4.1 Expression list patterns

An expression list pattern for attributes consists of a sequence of expression patterns. An expression pattern is either a base literal (allowed to occur as an attribute argument, such as numbers, booleans, and strings, typeof), or a meta-variable. We restrict an expression list pattern to contain at most one multi-match meta-variable to avoid matching ambiguity.

3.5 Formal argument patterns

Formal argument lists patterns consist of a list of type name pairs, with optional attribute patterns for each parameter. The names of formals are immaterial for matching purposes in our current design, so no meta-variables need to be employed for formals. Alternatively, the entire sequence of formals can be matched with a single multi-match meta-variable.

4. Generation

As we have seen in the previous sections, code is generated by **implement** or **generate** members. For simple templates, these members contain ordinary C# code with meta-variables. However, to generate a piece of code for every match in a multi-match list, our design provides a **forall** construct.

4.1 Statement iteration

To generate a block of statements for each match in a multi-match, the **forall** block construct is used. The general form is:

```
  forall ( $m in [Path].$$C ) {
    statement−list
  }
```

where the meta-variable $m will range over each match in the multi-match variable $$C. Within the body of the **forall** block, meta-variables nested within $$C's scope are accessible by qualifying their name with $m (see example in Section 2).

[3] **value** is the implicit parameter of the setter in C#

4.2 Member iteration

At first, it might seem useful to provide **forall** iteration at the member level to generate, say, a member (or collection of members) for each match in a particular multi-match. However, such a construct poses problems in that it is not clear how to refer to the generated members outside the **forall** construct. Instead, we can observe that the **scope** construct can span arbitrary contexts and permits us to solve such tasks directly, as the next example shows. The following transform generates delegation methods in a class to forward calls onto a delegate target object stored in a field.

```
transform Delegation {
  scope $$D {
    class $C {
      [DelegationTarget]
      private $DelegateTarget $target;

      generate $T $Method($$formals) {
        return this.$target.$m();
      }
    }

    class $DelegateTarget {
      public $T $Method($$formals);
    }
  }
}
```

Scope $$D gathers all matches consisting of a class $C having a field annotated with a [DelegationTarget] attribute. The field type should match a class $DelegateTarget and this class should have a method with arbitrary arguments and return type. For each such match in scope $$D, we generate a method in $C that simply invokes the target method using the field as the target instance and the same parameters. Thus, this transform will generate forwarding methods for all classes, all fields with the [DelegationTarget], and all methods in the target class. We abuse our syntax to name the generated method with the same name as the target method by reusing the meta-variable $Method.

This example illustrates not only the power of the **scope** construct, but also a limitation of our design. The delegation passes the parameters to the delegate target unchanged. If we wanted to transform the actual arguments before calling the target method, we would need a way to construct arbitrary argument lists. We have not yet explored the issues surrounding inspection and construction of formal and actual argument lists.

5. Matching

So far we have described matching of individual parts, such as names, members, and type references. What remains to be defined however is how member lists are matched against lists of member patterns.

Member lists are matched in three places, 1) at the transform top-level, where only type patterns are matched, at member pattern lists of nested types within the transform, and within scope patterns.

Matching a list of member patterns against a member list produces a set of all matches such that each match is a mapping from patterns to members. Within a match, there exists a unique substitution of all meta-variables that makes the patterns match their corresponding member.

5.1 Matching algorithm

We use a brute force algorithm to enumerate all possible mappings of patterns to members to find all matchees. Matching of member lists proceeds in three phases:

```
bool FindNextMatch(int[] candidates,
                   ref int patternIndex,
                   MemberMatcher[] matchers,
                   Member[] members,
                   MatchEnvironment matchEnv)
  requires candidates.Length == matchers.Length;
{
  if (patternIndex > candidate.Length) {
    // found a successful match for all matchers
    return true;
  }
  // increment the current match candidate at patternIndex
  while (GotoNextCandidate(candidates,
                           patternIndex,
                           members.Length))
  {
    int memberIndex = candidates[patternIndex];
    MatchEnvironment nestedEnv = matchEnv.NewUndoScope();
    Member member = members[memberIndex];
    if (matchers[patternIndex].Matches(member, nestedEnv)) {
      // found a match at current pattern index
      // find matches for remaining patterns
      patternIndex++;
      if (FindNextMatch(candidates, ref patternIndex,
                        matchers, members, nestedEnv)) {
        // yes, can complete the match
        return true;
      }
      // no try another member at this level.
    }
  }
  // no more matches at this level
  return false;
}

bool GotoNextCandidate(int[] candidates, int patternIndex,
                       int count) {
  do {
    int memberIndex = ++candidates[patternIndex];
    if (memberIndex >= count) {
      return false; // no more candidates for this pattern
    }
  } while (DuplicateIndex(candidates, patternIndex));
  return true;
}
```

Listing 1. Member matching algorithm

1. Find a match environment that satisfies all single-match member matchers (ordinary single-match member patterns and implementation patterns)
2. Check that given this match environment, none of the generation members clash with existing members
3. Given this match environment, record match lists for all multi-match member patterns individually against all members. The reason multi-matchers are only considered last is that they do not determine the match since they always match (in the worst case zero times) and given our scoping rules, there cannot be any meta-variables nested within a multi-match that is also used outside the multi-match, unless it is being determined by a matcher in step 1.

If all three steps succeed, we have found one possible match. To find all matches, simply repeat without considering combinations in step one that have already been tried.

Listing 1 contains the pseudo-code for finding all matches for step 1. The FindNextMatch method is initially called with an array where candidates[0]=−1 and patternIndex = 0, meaning that we start the search by considering members starting with the next member (0) for pattern 0. The helper method DuplicateIndex checks if the new candidate member index is a member we have already used for preceeding patterns. This step is optional, but we find the matching semantics more intuitive if single-match (must match) patterns don't overlap.

In the context of a transform or type member list, matching only succeeds if there's a unique match for the member list matching. For scopes on the other hand, all matches are considered.

6. Static safety

In this section, we describe what safety guarantees the generated code satisfies and informally argue why safety follows from type-checking of transforms.

There are two guarantees we want from the compilation of a transform: 1) that the transform represents a well-formed pattern that can be interpreted unambiguously by the matching algorithm, and 2) whenever a transform matches in a compilation unit, the resulting .NET assembly produced by applying the transform passes the verifier.

The well-formedness of a transform checks that all referenced types and members are defined or are meta-variables with non-ambiguous binding scopes.

The well-formedness of the result of applying a transform can be split into two components: 1) method-body verification, and 2) metadata verification. The former requires that the code of each method body is type safe, and that all referenced types and members exist and have the expected signatures. The metadata verification requires well-formedness on the type and member structure, such as a non-cyclic inheritance hierarchy, implementation of abstract and interface methods, as well as absence of conflicting member signatures.

Our approach to guarantee type safety of generated method bodies uses the same principle as type checking in the context of generic type parameters [14], or more appropriately, type checking of functors in ML [18]. When type checking a functor body in ML, the functor argument signatures are added to the typing environment and are indistinguishable from other typing assumptions in the context. We use the same principle for typing transforms. Patterns give rise to typing assumptions, either about concretely named classes and structs, or about type meta-variables. In the latter case, we treat the types similar to type variables in a traditional C# type checker, except that we have more detailed constraints on these type variables than is expressible in standard C#. In other words, we type check the generated method bodies of a transform as code parameterized by types. A formal approach would have to establish a substitution lemma, showing that typing is preserved under substitutions satisfying the type constraints.

This approach also encompasses the **forall** construct by treating the bound meta-variable as a generic parameter as well. Scoping guarantees a single such parameter is sufficient as a witness for all elements in the eventual multi-match list.

Guaranteeing the well-formedness of the metadata part of an assembly during transform checking time leads to a design trade-off between usefulness and early-checking. Of the three rules for metadata well-formedness (acyclic inheritance, proper implementation of all abstract/interface members, and no clashing members), we check the first two at transform checking time, but leave the last one to transform application time. The first two issues are ruled out because transforms cannot modify the existing inheritance hierarchy except for the addition of new subtrees in the hierarchy.

The problem of checking for clashing members is best seen in the following simple example:

```
transform AorB {
    [CaseA]
    class $A {
        generate void FooBar () { ... case A code }
    }

    [CaseB]
    class $B {
        generate void FooBar () { ... case B code }
    }
}
```

The transform above adds one of two FooBar methods to classes depending on whether they have the [CaseA] or [CaseB] attribute. If the transform checking were conservative, it would have to reject the above, as a class could have both [CaseA] and [CaseB] attributes at the same time and would thus end up with clashing FooBar members. Clearly, the writer of the above transform is establishing a usage rule for the transform in that only one of the two attributes should appear on a class. Our methodology does not allow capturing such conventions. We do not want to rule out applications such as the above. Therefore, we made the trade-off to check for member clashes at transform application time. Given these observations, it is clear that transforms without **generate** members having concrete names are guaranteed not to cause clashes.

Our implementation is structured in such a way that the normal semantic checks performed during compilation take care of most of the semantic checks for the well-formedness of the code generated by transforms. For example, a type meta-variable really is represented as a type during compilation, with all characteristics specified in the pattern. Thus, uses of this type are automatically checked against the knowledge given by the pattern. If the pattern is under-constraining the desired usage criteria, the compiler will detect it and emit an error. This approach has the advantage of reusing the existing compiler functionality without having to duplicate it, as well as reducing the chances of missing certain checks. For example, scoping and lookup rules are complicated and easy to get wrong, and member visibility checking might be overlooked.

7. Applications

7.1 Software transactional memory

Herlihy has published a software transactional memory implementation named SXM. SXM is based on C# and uses C#'s runtime reflection capabilities to provide an easily accessible software transactional memory model; no language or runtime modifications are required [8]. Users annotate a type as atomic, designating that reads and writes to its fields should be recorded and treated transactionally with respect to other concurrent transactions. To support this easily, SXM mandates that accesses to atomically handled fields be redirected through C# properties. At runtime, SXM locates the properties of atomic types and wraps their implementation with the appropriate calls to the SXM runtime. SXM must also add additional shadow fields and backup and restore methods to facilitate object rollback on aborted transactions.

SXM is the quintessential example of the utility of our approach. SXM's transformations rely solely on static information. While many problems require runtime information, a purely dynamic approach to reflection places an undue burden on SXM, and other such applications, where dynamic information isn't required.

In addition to performance penalties, writing the appropriate code to emit wrappers is a complex process. First, there is a non-trivial amount of API work required to simply find the annotated types and prepare the appropriate metaobjects for code generation. Second, Herlihy has expressed that generating bytecode with C#'s Reflection.Emit capabilities is tedious, time-consuming, and bug prone when not automated. Reflection.Emit places a hefty burden on the programmer's abilities to both write and document his intentions. As a result, SXM's reflection implementation comprises almost 500 lines of well documented and formatted C# code whereas an equivalent implementation with CTR, which is given in Appendix A, is roughly 60 lines of code.

7.2 Process startup boiler-plate

Our work on compile-time reflection was motivated in the context of the Singularity project [12]. Singularity is a research operating system built almost entirely in managed code. Its runtime system is a stripped version of .NET. In particular, it does not support runtime reflection. We use compile-time reflection to build process startup boiler-plate code from declarative specifications of a process' startup arguments [21]. The generated code retrieves process arguments through a uniform kernel API, casts the arguments to their appropriate declared type, and populates a startup object for the process containing a field per parameter. This transform is used in the regular build process of more than 100 test applications.

8. Implementation

We have built an experimental compiler extension to C#. Transforms can be separately compiled into (non-executable) .NET assemblies [1]. The IL representation of a transform is a class and the transform member patterns are represented as members with additional information represented as attributes. Type meta-variables are represented as dummy class types within the transform. Attribute patterns are serialized into pattern attributes. For binding purposes, we generate static fields in the transform representing expression meta-variables. The forall statement construct is represented as a dummy try-all block with extra information serialized into attributes.

To apply a transform during compilation of another assembly, we simply reference the dll containing the previously compiled transform representation and use an assembly level attributes to specify which transforms to apply to the compilation (see Figure 1). For debugging purposes, it is possible to have the compiler output detailed information of what is and what isn't matched by a transform, as well as instances of ambiguous matches.

Due to C#'s rich type system of structs and classes, we use a specialization phase after instantiating code templates to handle corner cases such as the necessary insertion of boxing or unboxing operations on struct values.

9. Related work

Reflective Program Generators. Our work is closest to other work that shares the desire to make program generation easy while guaranteeing type safety and other well-formedness conditions of the generated code. Genoupe [5] is a C# extension allowing programmers to define program generators and apply them at compile-time. Our work can be seen as overcoming some of Genoupe's most serious limitations: it cannot guarantee many well-formedness conditions of the generated code because parameters (such as types, etc) do not carry enough constraints to allow such checking. In contrast, our patterns serve the dual purpose of describing the matching context and recording the necessary constraints for static checking. A second serious limitation of Genoupe is that it can only generate new programming elements, not add to existing ones. The combination of patterns and generators within a single transform makes extension very natural in our approach.

SafeGen [11] is a very ambitious system that uses first-order logic formulae to express patterns and templates to generate code. The formulae in their approach are similar to our patterns in that they express not only what is to be matched, they serve at the same time as pre-conditions to check the safety of the generated code. Safegen's formulae are more expressive than our patterns, albeit more difficult to write. We believe that formulae are best suited for complicated matches and we consider augmenting our patterns with formulae as possible future work.

Multi-staged programming. The research on MetaML [22] was the first work to propose designing languages that enable programmers to write program generators that are guaranteed to generate well-formed programs. Apart from this very important benefit, multi-staged programs are still complex to write and in particular hard to read, since they manipulate code as data and often make heavy use of quoting and unquoting or constructor syntax. Work on typed macros [6] has shown, however, that quoting can be avoided in most contexts, except for explicit recursive macros. Semantically, CTR can be viewed as a typed generative macro system, where we are able to avoid the need for quasi-quotes because no recursive macros are expressible. Furthermore, CTR acts as a staged program which is provided with the program under transformation as a data object that can be inspected. Note that expressing CTR directly in a multi-staged program would require dependent types to reflect constraints obtained from inspection into the typing of generated code [19]. Furthermore, multi-staged programming languages generally disallow inspection of code as it can lead to a loss of equational reasoning and/or static type-safety [23, 6].

Although only a two stage language, compile-time reflective ML [9] has the same characteristics as multi-stage programming in this context.

XML Processing Languages. The explosion in popularity of XML as a universal mechanism for describing and exchanging data has led to a number of XML processing and transformation languages. The challenges researchers have faced in designing manageable, yet capable languages for processing and generating XML are much akin to our own. XQuery provides general mechanisms for pattern matching, iterating, and generating XML elements [3]. Hosoya and Pierce have devised XDuce, a statically typed XML processing language that integrates XML patterns with a strict notion of typing to guarantee the type-safety of generated XML [10]. While these languages provided the initial insight on the utility of patterns for code generation, the generality and regularity of XML's structure enables these languages to provide complex pattern and matching structures that would be hard formalize and difficult implement in a large, mainstream language like C#.

Language Extensiblity. Several popular languages and language extensions provide extensible compilation mechanisms [2, 4, 17]. These mechanisms enable programmers to do everything from simple textual macro expansion to general AST manipulation and language semantics modification. OpenJava and OpenC++ provide programmers with a Meta-Object Protocol (MOP) for the extension and manipulation of Java and C++ classes, respectively. These MOPs allow programmers to devise custom, first-class annotations for types and members and then give an alternate or extended semantics to these annotated elements. Just as in our approach, programmers can generate new members and implement interfaces (similar to our **reflective** keyword). These MOPs also go further in allowing programmers to specify alternate semantics for type instance creations sites and field references for annotated types and fields. However, programmers must generate and manipulate explicit ASTs with a compile time reflection API in order to implement their desired functionality.

While our design does not approach the full generality of MOPs, we believe that programmers can combine our patterns with C#'s attributes to provide extended semantics for several of C#'s program elements. More importantly, our construction lets programmers write generative implementations in C# rather than through the manipulation of ASTs. Further, our design provides guarantees on well-formedness at pattern compilation time rather than at application or generated code compilation time.

Aspect-Oriented Programming. Aspect-Oriented Programming proposes that crosscutting concerns–design decisions that span module boundaries–can be coalesced into *aspects* [16]. Aspects allow programmers to supplement or replace *join points* (code execution events) with *advice* (code fragments). Programmers write patterns to specify *pointcuts* (sets of join points) to which they'd like to contribute advice. AOP languages, such as AspectJ, provide programmers with models to contribute advice to join points such as method calls, field references, and exception handler executions [15]. While aspects are typically weaved into the existing code base statically, aspect execution is a dynamic consideration. Thus, AspectJ provides some functionality to both constrain and capture the dynamic matching context.

AOP approaches differ from our goal in that they seek to give programmers tools to, primarily, modify existing program behavior. We, on the other hand, have sought a purely generative approach. Programmers that use our system can only generate new functionality or implement "fill-in-the-blank" functionality as specified by the **reflective** keyword. Thus, unlike AOP, our goal steers away from many of the violation of modularity and encapsulation claims that are often lodged at AOP approaches. Further, our approach is purely static and metadata introspective. While querying static and dynamic values is a useful tool for code generation, conditional/dependent code generation places an extra burden on the well-formedness claims of generative systems.

LogicAJ 2 extends the principles of AspectJ's take on AOP with richer pointcut patterns [24]. In LogicAJ 2, programmers can specify arbitrary code structures as join points. Much akin to our approach, programmers specify patterns as Java code with the addition of meta-variables that can abstract and capture arbitrary code elements or lists of such elements. However, this flexibility comes at a price; the level of well-formedness provided by the system is unclear.

Runtime Code Generation. Runtime code generation is another similar domain. Runtime code generators, such as Jumbo [13] and 'C [20], provide programmers with a rich set of language constructs that allow for flexible code generation at runtime. These systems further improve on bytecode emission constructs by allowing programmers to specify code fragments at the original source language level (i.e. Java for Jumbo and C for 'C). Both systems utilize quoting and unquoting operators to facilitate compositional code generation. Moreover, Jumbo permits higher-order generation of code that generates code through nested quoting and an AST-like API. 'C further provides static typing facilities to ensure that dynamically generated code is well-formed.

We view these systems as somewhat orthogonal to what our design seeks to achieve; these systems seek to provide programmer guided optimization and runtime extensibility. On the other hand, our approach aims to allow programmers to glue together metadata level program elements in a user-defined, property driven manner. As a result, we believe that the enormous flexibility given by such systems is unnecessary to achieve our goal.

10. Future Work

The expressiveness of our patterns is limited at the moment. We have no conditional (zero or one match) patterns or patterns that must match one or more times. We already mentioned the possible addition of formulae in the style of SafeGen to augment the matching power of our patterns, while maintaining the same overall design. A formula is akin to a pure transform (a transform without generation or implementation parts).

Another open issue that needs to be addressed is how to compose multiple transforms when applied to the same code.

11. Conclusion

The transform design described in this paper addresses two problems we see in generative programming and compile-time reflection systems. First, writing programs to inspect code and generate new code should be made simple and intuitive so that programmers do not have to be experts in compiler implementation techniques to use it. Our design uses patterns and templates that look mostly like ordinary code to achieve this. Our powerful **scope** construct enables concise descriptions of non-trivial patterns and templates. Second, we do not want to forgo safety of the generated code in overcoming the first problem. Our insight is that patterns describing the interesting matching contexts also serve as constraints that enable the checking of well-formedness of the generated code. Our approach does trade-off expressiveness in order to achieve these goals.

We believe that our approach is an excellent point in the design trade-off between simplicity, safety of generated code, and expressiveness.

References

[1] Partition III: CIL Instruction Set. ECMA Standard 335 http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[2] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C.* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[3] Don Chamberlin. XQuery: A Query Language for XML. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 682–682, New York, NY, USA, 2003. ACM Press.

[4] Shigeru Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, October 1995.

[5] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In Glück and Lowry [7], pages 327–341.

[6] Steven Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.

[7] Robert Glück and Michael R. Lowry, editors. *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*. Springer, September 2005.

[8] Maurice Herlihy. SXM: C# Software Transactional Memory. http://www.cs.brown.edu/˜mph/SXM/, May 2005.

[9] J. Hook and T. Sheard. A semantics of compile-time reflection. Technical Report 93-019, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[10] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 50–62, New York, NY, USA, 2005. ACM Press.

[11] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In Glück and Lowry [7], pages 309–326.

[12] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[13] Sam Kamin. Routine run-time code generation. In *Proceedings of 2003 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 208–220, October 2003.

[14] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–12, 2001.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[17] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[18] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[19] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002.

[20] Massimiliano Poletto, Wilson C. Hsieh, Dawson 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):324–369, March 1999.

[21] Michael Spear, Tom Roeder, Orion Hodson, Galen Hunt, and Steven Levi. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. In *Proceedings of the Eurosys 2006 Conference*, pages 45–57, Leuven, Belgium, April 2006. ACM.

[22] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[23] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Boston, 2000.

[24] Malte Appeltauer Tobias Rho, Gnter Kniesel. Fine-grained generic aspects, workshop on foundations of aspect-oriented languages (foal'06), aosd 2006. Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), March 20-24, 2006, Bonn, Germany, Mar 2006.

A. STM Transform

We assume the following declarations for interface IRecoverable and class TransactionManager, similar to Herlihy's implementation.

```
interface IRecoverable {
  void Backup();
  void Restore ();
}

class TransactionManaager {
  public Transaction GetCurrentTransaction ();

  /// returns a conflicting transaction or null
  public Transaction OpenForRead(IRecoverable object,
                                 Transaction t );

  /// returns a conflicting transaction or null
  public Transaction OpenForWrite(IRecoverable object,
                                  Transaction t );

  public void ResolveConflict (Transaction us,
                               Transaction them);
}
```

The transform below then implements transactable properties for atomic classes using a scope pattern to add a current and backup field for each property. For each atomic class the Backup and Restore methods of the IRecoverable interface are also implemented by the transform by calling all Backup (respectively Restore) methods for individual properties.

```
using TM = TransactionManager;

transform MakeTransactional {
  [Atomic]
  class $$C : IRecoverable {

    scope $$TransactionalProperties {

      [Atomic]
      implement public $T $Property {
        get {
          Transaction me = TM.GetCurrentTransaction();
          Transaction conflict = null;

          while(true) {
            lock( this ) {
              conflict = TM.OpenForRead(this, me);
              if ( conflict == null) {
                return $currentValue;
              }
            }
            TM.ResolveConflict(me, conflict );
          }
        }

        set {
          Transaction me = TM.GetCurrentTransaction();
          Transaction conflict = null;

          while(true) {
            lock( this ) {
              conflict = TM.OpenForWrite(this, me);
              if ( conflict == null)
                $currentValue = value;
            }
            TM.ResolveConflict(me, conflict );
          }
        }
      }

      generate $T $currentValue;
      generate $T $backupValue;

      generate void $BackupProp() {
        $backupValue = $currentValue;
      }

      generate void $RestoreProp() {
        $currentValue = $backupValue;
      }
    } // end scope

    implement void IRecoverable.Restore() {
      forall ( $Property in $$TransactionalProperties ) {
        $Property.$RestoreProp();
      }
    }

    implement void IRecoverable.Backup() {
      forall ( $Property in $$TransactionalProperties ) {
        $Property.$BackupProp();
      }
    }
  }
}
```