Singularity Design Note

32

# Sing# support for Compile-Time Reflection

*SDN22 describes the ideas behind compile-time reflection. This document describes the actual support provided by the Sing# language to express and apply compile-time reflection.*

## 1. Transforms

The basic language construct added to Sing# to express compile-time reflection is the **transform**. A transform serves two purposes: 1) it acts as a *pattern* to describe where the transform is applicable, and 2) it serves as a *template* describing what code is to be generated in the context where the transform matches. There is a good reason for combining these two seemingly separate aspects (matching and generation) into a combined transform: given the context information from the pattern, we can type check the template to make sure that the generated code is sensible in all contexts in which it could be applied. This is a major benefit over generating MSIL directly via the reflection-emit API.

### 1.1. Example

```
using System;
using System.Collections;

public delegate int Command(string[] args);
public class CommandAttribute : Attribute {
  public CommandAttribute(string name) {}
}

transform ShellCommands {

   public class Shell {

     [Command($name)]
     private static int $$Commands(string[] args);

     implement static Hashtable CommandTable = $BuildCommandTable();

     generate static Hashtable $BuildCommandTable() {
       Hashtable result = new Hashtable();
       forall (; $cmd in $$Commands; ) {
         result [ $cmd.$name ] = new Command( $cmd );
       }
```

 

```
      return result;
    }
  }
}
```

The example defines a transform called ShellCommands in the top-level namespace. The Command delegate and CommandAttribute classes are not part of the transform, but are given here for completeness. These declarations could be defined elsewhere.

This example transform contains a single class pattern with the name Shell. This pattern matches classes with the concrete name “Shell”, provided the visibility is public, and the members of the class match the members of the pattern.
The first member pattern of Shell is $$Commands, which matches private static methods with a string[] parameter and return type int. The name of the method is irrelevant, since we use a meta-variable (identifiers starting with a $ are meta-variables). In this case, the meta-variable starts with $$, denoting a list of members (as opposed to a single member). Thus, this pattern will match any number of methods (including zero). The attribute [Command($name)] is an additional pattern that methods must match in order to be considered part of the $$Commands set. The attribute matches any attribute of the given type with a single argument, where the argument is bound to meta-variable $expr.
The next member pattern of Shell is the field CommandTable. The implement keyword specifies that this is a field already present in the class to be matched, but the matched class specifies the field as “reflective”. Reflective members in actual classes match implement members in patterns. These members serve as link or bind points between original code and generated code. In this scenario, we are assuming that the Shell class is going to use the CommandTable field. Only the initialization of the field is managed by the transform. In this scenario, the field is initialized by the result of calling the method $BuildCommandTable(). The last pattern in Shell is the method $BuildCommandTable. This pattern is marked “generate”, meaning that it should not match anything, since it will be generated in the matched class. The use of a meta-variable $BuildCommandTable is to avoid having to come up with a name that avoids clashing with existing methods in Shell.
The most interesting part of the above transform is the body of $BuildCommandTable. It first creates a hashtable to hold the command delegates. It then iterates over all matched methods $cmd in $$Commands, emitting the body of the forall for each such match. Note the use of the qualified path $cmd.$name, which is used to refer to the attribute argument for a particular method (the one currently bound to $cmd). It is not possible to refer to $name directly, since it only makes sense in the context of a particular method within the set matched by $$Commands.

We can apply the above transform to the following code as follows. Assume the transform above is in source file shelltransform.sg. We first build an assembly containing that transform:

```
% sgc /t:library /debug shelltransform.sg
```

This results in shelltransform.dll containing the transform. Assume the code below is in file shell.sg. In order to apply the transform, the assembly containing shell.sg needs to contain an assembly level attribute requesting the ShellTransform to be applied:

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

```
// This attribute causes the named transform to be applied to this assembly
[assembly:Transform(typeof(ShellCommands))]

public class Shell {

    [Command("ls")]
    static int Ls(string[] args) {
        return 0;
    }

    [Command("rm")]
    static int Rm(string[] args) {
        return 0;
    }

    [Command("mkdir")]
    static int Mkdir(string[] args) {
        return 0;
    }

    reflective static Hashtable CommandTable;

    public static void Main(string[] args) {
        foreach(string name in CommandTable.Keys) {
            Command cmd = (Command)CommandTable[name];
            int result = cmd(args);

            Console.WriteLine("Command '{0}' returned '{1}'", name, result);
        }
    }

}
```

To build an executable from shell.sg applying the transform, we simply compile shell.sg while referencing the assembly containing the transform:

```
% sgc /debug /debugtransform /r:shelltransform.dll shell.sg
```

This results in an executable shell.exe. The /debugtransform option spits out what matches and what does not match during transform application. It is useful for debugging, but not required. Running the resulting executable produces the following output:

```
Command 'ls' returned '0'
Command 'mkdir' returned '0'
Command 'rm' returned '0'
```

## 1.2. Meta-variables

Meta-variables serve to bind matches so that they can be used in generated code or in other matches. This section describes the different kinds of meta-variables recognized in transforms.

### 1.2.1. Single match

Identifiers starting with a single $ sign are used to name a syntactic element that matches exactly once. These variable can appear as type and member names, as type reference names (eg. for return types or parameter types), or for naming attribute arguments.

#### 1.2.1.1. Don't care

A special form of a single match variable is the single underscore '_'. It stands for a single match without binding the match to a name. It's equivalent to a fresh single match variable.

### 1.2.2. List meta-variable

Meta-variables starting with a $$ sign are used to bind collections of matches. They can be used as type and member names (in defining positions), and for attribute names to match multiple attributes. List meta-variables cannot be used in other places.

Elements bound by list meta-variables cannot be directly referenced in generated code. Instead, the **forall** construct is used to iterate over all matches bound by a list meta-variable.

### 1.2.3. Star

Star variables are used in parameter lists and expression lists (in attribute arguments) to match zero or more elements of the sequence. For example, to match methods with 1 or more parameters, we could use the following pattern:

```
void $MM( $T1 arg1, *);
```

# 2. Transform members

At top-level, a transform must contain types, just like a namespace. Namespaces themselves cannot be matched at the moment and can thus not be part of transforms yet.

Inside types, all the usual members can appear, including nested types. Each member inside a transform (including nested members and types at top-level) are of one of three kinds: 1) a pattern, 2) a *generated* member, or 3) an *implemented* member,.

A pattern is a member not preceded by the **implement** or **generate** keyword, and not appearing nested within a generated type. A pattern serves only to match an element in the target assembly, not to generate anything (although for a type, its members might still have such generative parts). A pattern member does not need to define a body (e.g., a method body, or getter/setter definitions). It acts like a signature declaration in interfaces, although, types can also be specified as patterns.

A generated member is prefixed by the **generate** keyword. Such a member does not act as a pattern. In fact it is a negative pattern. It maches if the target context does not yet contain that member, for otherwise a conflict would arise. A generated member is fully specified in the transform and will become the actual definition in the target context. The name of a generated member can be a meta-variable in which case it will not match any concrete name. This is a way to avoid coming up with names that won't clash with names in the target context.

An implemented member serves both as a pattern and as a generator. Implemented members serve as link points between the generated code and the original code. Implemented members only match *reflective* members in the target, where a reflective member is a placeholder in the target prefixed by the **reflective** keyword. The point of reflective members in the target is to allow the target code to reference members that will be generated by a transform.

The next subsections describe in more detail what these concepts mean for the different kinds of members.

## 2.1. Types

### 2.1.1. Type pattern

A type pattern matches a target type, if the name matches, the visibility (private, public, protected, internal), the attribute list in the pattern is a subset of the actual attribute list, the actual base type is a subtype of the pattern base type, and the members of the type pattern match the members of the actual type.

The members of a type pattern can be of any kind themselves (ie., patterns, generated or implemented).

### 2.1.2. Generate

A generated type is a type that is added to the target context. The target context should not contain a type of the same name, or this member won't logically match (negative match). If the generated name is a meta-variable, the name won't match anything in the target context, thus avoiding potential conflicts. The members of a generated type are implicitly all generated, i.e., it does not make sense to have patterns or implemented members inside a generated type.

### 2.1.3. Implement

An implemented type is a type that already appears in the target as a reflective type, but whose context is fleshed out by the transform. It matches only if the target context contains a reflective type and the type matches the target as a pattern (as in 2.1.1). During generation, all implemented and generated members of the implemented type are added to the target context.

## 2.2. Methods

### 2.2.1. Method pattern

A method pattern specifies the method signature and possibly attributes, but no method body. It matches a target method, if the name matches, the visibility matches, the attribute patterns match a subset of the actual attribute list, the return type matches, and the parameter list matches. Method patterns are used to establish the existence of a method in a target type. This is most useful if generated code is going to call such a method.

### 2.2.2. Generate

A generated method defines a method including its signature and body. It matches only the target context does not contain a method with that name. A meta-variable can be used to guarantee that the generated method is distinct from existing methods.

### 2.2.3. Implement

An implemented method defines both the method's signature, including attributes, as well as a body. It matches only target methods marked as reflective and that match the method's signature (as under 2.2.1). The body of the implemented method will appear in the target type for the matched reflective member.

## 2.3. Properties

### 2.3.1. Property pattern

A property pattern specifies the signature of a property, its visibility, type, attributes, and whether it has a getter or setter, but no bodies for getters or setters. A property pattern matches a target property if the name matches, the visibility matches, the attribute patterns match a subset of the actual attribute list, the type of the property matches, and if pattern has a getter, the actual property has a getter, and if the pattern has a setter, the actual property has a setter.

TODO: match getter and setter attributes too.

### 2.3.2. Generate

TODO: test this

### 2.3.3. Implement

TODO: test this

## 2.4. Fields

### 2.4.1. Field pattern

A field pattern specifies a field, its type, visibility and attributes, but no initializer. It matches if the target field has the same visibility, type, and attributes.

### 2.4.2. Generate

A generated field matches if the target context has no field with the given name. A meta-variable can be used to avoid name clashes here. The field can have an initializer only if it is either static, or the context is a generated type.

TODO: support initializers on instance fields.

### 2.4.3. Implement

An implemented field specifies a name, type, visibility, and optionally attributes and an initializer. It matches a target field only if the target field is reflective and has the same visibility, type, and a superset of the attributes. The optional initializer is only allowed on static fields for now.

TODO: support field initializers on instance fields.

## 2.5. Indexers

Not yet implemented

## 2.6. Delegates

Not yet tested

## 2.7. Events

Not yet implemented

## 2.8. Attribute patterns

Attribute patterns can appear on any member. An attribute pattern consists of the name, and optionally, argument patterns. The name can be either a concrete attribute name or a meta-variable (list or single match). The expression list can contain literals, including typeof(T), where T can be a meta-variable, as well as single-match meta-variables and star variables.

## 2.9. Type reference patterns

Wherever type references occur in member signatures, a single-match meta-variable may be used. These meta-variables are bound in the context of either the transform or the closest enclosing list match. (see also section 4.2). Such type meta-variables match any type, except void in method return type positions. Void is only matched by the literal void type itself.

### 2.9.1. Constrained type patterns

In order to match types in type reference positions that satisfy certain subtyping constraints, (e.g., types deriving from a class C, or implementing interfaces I and J), transforms can declare type meta-variables with constraints after the transform name using the following syntax:

```
transform T
   where $A : A1, A2, …
   where $B : B1, B2, …
     …
{  … }
```

The bound variables $A, $B, etc.,, are declared above to only match type occurrences that satisfy the type bounds A1, A2, etc.

Note that each occurrence of a constrained type meta-variable matches independently, ie., two occurrences of $A might match different types.

TODO: Come up with a scheme that allows a different matching strategy where a constrained type matches the same type in all occurrences.

# 3. Using transforms

Currently, the only way to apply transforms is to first separately compile a transform (or multiple transforms) into a separate dll, let's call it X.dll. To apply this transform or transforms to target code, we need to reference X.dll while compiling the target code. Additionally, the target code assembly needs to specify using assembly level attributes, which transforms are to be applied.

The compiler option /debugtransform is useful when debugging transform application. It causes the compiler to emit information about what members match and where they don't to aid in debugging transforms that should apply but don't match.

## 3.1. Transform attributes

The target code needs to specify which transforms to apply using the following assembly level attribute:

```
[assembly:Microsoft.SingSharp.Reflection.Transform( typeof(TransformName) )
```

where "TransformName" is the name of the transform to be applied.

## 3.2. Reflective members

All members including types in the target code can use the "reflective" modifier to mark the member as being implemented by a transform. This allows target code to reference members that have not yet been generated.

## 3.3. Forall iteration

In order to use the set of elements matched by list meta-variables, a special kind of statement can be used in implemented and generated code bodies. A forall statement can appear in any statement context. It's form is

```
forall ( index-decl ; $Elem in $$Collection ; index-increments ) {
  stmt-list
}
```

where *index-decl* allows one to declare compile-time index variables, e.g., $index = 0, and *index-increments* is used to increment the index variable, e.g., $index++. Any meta-variable name can be used for compile-time index variables.
If no index variables are used, the semicolons must still appear. (TODO: fix this syntactic oddity)

The semantics of a forall statement is that it's *stmt-list* is expanded once for each element $Elem in $$Collection. In the scope of the forall statement, meta-variables bound in the context of each element matched in the collection are accessible via qualified names from $Elem. For example, in Section 1.1, the forall body accesses $cmd.$name, where $name is the meta-variable bound to the attribute argument of the particular method bound to $cmd. These qualified access paths are used for all meta-variables that are logically nested under a list-match.

# 4. Matching

A transform matches in a target context, if the type list in the transform matches the set of top-level types in the target assembly. The next section describes how member lists match.

## 4.1. Member list matching

Given a list of member patterns and a list of actual members (or top-level types), the matching proceeds as follows:

1. All non-list patterns are consulted first. Each non-list pattern is either a single-match pattern or a generator:
    a. A single-match pattern must match exactly one target member in the list. Otherwise, the entire match fails

   b. A generate must match no element in the target member list. Otherwise, the entire match fails.

2. After all non-list patterns are determined to succeed, the list matches are applied. For each list matcher, the the set of target elements that are matched are gathered. Note that this means that the same target element can match zero or more non-list patterns and zero or more list patterns.

## 4.2. Match scopes

The matches defined by transforms are mostly linear matches, meaning that each meta-variable occurs only once and matches independently of all other matches. This means that if the same meta-variable occurs in different matching positions in the transform, these occurrences act as if they had been different meta-variables, i.e., each will match individually.

There are some exceptions to this simple scheme,

1. In the scope of a type bound by a meta-variable, the meta-variable can be used in type reference positions to match the same type.
2. Each list meta-variable occurring as a member name or attribute name opens a fresh scope. Under that scope, an occurrence of a meta-variable represents a single match per match of the outer list meta-variable. These nested meta-variables are not directly accessible from outside the scope of the enclosing list meta-variable. They are only accessible via qualified names starting with a bound name iterating over the list meta-variable in forall blocks.