Singularity Design Note 22

Compile-Time Reflection

Reflection in C# is a run-time mechanism that enables a program to examine its classes, methods, and interfaces and to produce and execute new code. These two operations, introspection and transformation, are powerful mechanisms for implementing abstractions. However, deferring reflection until a program executes increase the run-time overhead, complicates program analysis, and conflicts with Singularity's closed process model. This note describes compile-time reflection (CTR)—the replacement for reflection in Singularity—which enables many of the same transformations to be applied while a program is being prepared for execution.

1. Code Transformation

Programmers use many mechanisms to manipulate and transform their code, with the general goal of constructing abstractions to concisely express their intensions and improve program understanding and maintainability. The most common mechanism is a macro, which programmatically transform source text before it is read by a compiler. A slightly more expressive approach is source-to-source translation, in which a compiler translates code in a domain-specific language into code in a conventional language. More recently, aspect-oriented programming introduced weaving, a process in which code fragments are inserted at specified points in a program. Other program transformation tools, such as Atom or Vulcan, modify compiled, binary programs. A more recent innovation is run-time code reflection, which enables a running program to inspect its code and data structures and generate and execute new code. This mechanism is widely used in the CLR for tasks such as: producing stubs to marshal data, compiling regular expression pattern matchers, and generating XML parsers.

This continuum between purely static and dynamic mechanisms offers many tradeoffs. At one extreme, static mechanisms, such as macros, are accessible to most programmers, as they operate at the source level and are written in a familiar language. The result of their transformation is source code—readable both by humans and analysis tools—that directly corresponds to the executable program. At the other extreme, dynamic mechanisms can defer transformation until the last possible minute, at point at which all dynamically linked libraries are available and when knowledge of data values permit program specialization to generate specific and efficient code.

Both approaches also have disadvantages. Macros can be difficult to write, test, and debug, particularly macros that are heavily parameterized with configuration options that generate multiple version of a program. Moreover, non-syntactic macros, such as those provided by cpp, can make parsing and analysis of an untransformed program impossible. On the other side, reflection is a low-level mechanism that requires a programmer to be aware of implementation details, such as the instruction set. Reflection also imposes a high-runtime cost, both because of the additional data structures needed during program execution, and also because of optimization opportunities that must be foregone because of potential interactions with reflection. In addition, reflection is difficult to analyze statically, as it can run an arbitrary computation that produces and executes new code, much like the `eval` function in Lisp.

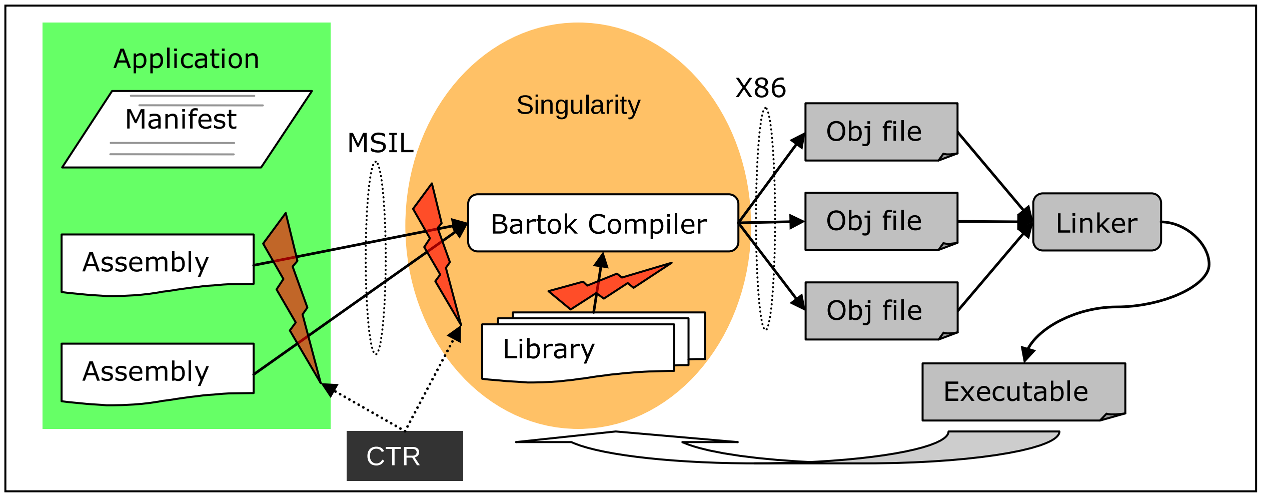
2. Singularity

A primary tenet of Singularity is systems should be analyzed and verified. Because of this goal, we have not implemented reflection in the Bartok compiler or run-time system—both to enable the Bartok compiler to produce efficient, high-quality code and to enable analysis and defect-detection tools to make sound assumptions about program behavior. C#, of course, does not support a source-level macro facility, so Singularity lacks a standard mechanism for transforming code.

This note describes a new mechanism for mechanically transforming system and application code, which provides many of reflection's advantages without undercutting Singularity's goal of improving system analyzability. The mechanism, called Compile-Time Reflection (CTR), provides functionality similar to reflection, without its run-time overhead or the limitations on program analysis. CTR does not, however, provide the syntactic extensibility of source-level macros.

3. CTR

Compile-time reflection is closely tied to Singularity's application abstraction, which introduces new opportunities for introspection and transformation. An application in Singularity is a first-class system abstraction, consisting of code and data resources described by a manifest. Code is delivered as MSIL, so that it can be verified before it is compiled. Conceptually, when an application is executed, its process is created by collecting, verifying, compiling, and linking MSIL from the assemblies and libraries listed in the application's manifest. In practice, of course, this process can occur earlier and the resulting executable is cached for subsequent use.



CTR transforms the MSIL representation. It can examine and modify both program abstractions—namespaces, classes, methods, variables, annotations, etc.—and detailed implementation—MSIL instructions. CTR is a first-class entity in an application, whose manifest lists which CTRs are applied to the application's constituent assemblies. A developer can create a CTR to extend his or her abstraction by augment code that uses the abstraction. For example, the author of a communication library can also provide an CTR with the library code that produces marshalling stubs for classes that are transmitted by the library. The library's manifest lists its CTR and requires that it be applied to any code that uses the library.

CTRs provide most of the functionality of reflection, but require additional effort to incorporate dynamic data. For example, CTRs can produce a regular expression pattern matcher for a regular expression whose value is explicit in an assembly. Dynamically constructed regular expressions require additional effort. They must be compiled at execution time to run in a separate SIP, or they can be interpreted. The first alternative requires additional support to produce a MSIL assembly with the dynamic value, invoke CTR, generate code from the transformed MSIL, and start this pattern matcher in a distinct process. Further experience with CTRs may clarify actual use of dynamic reflection and whether the complexity and overhead of produce code for a new process is feasible.

4. Implementation

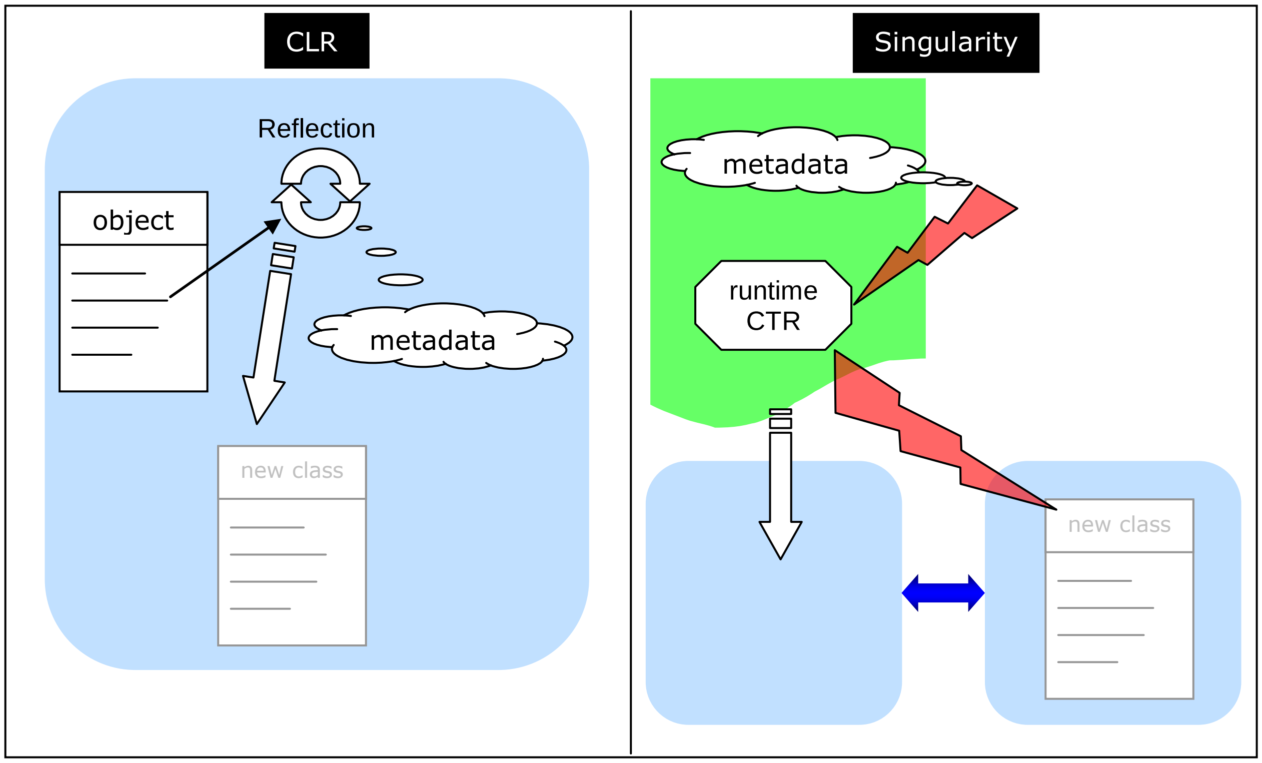
Making CTRs usable tools requires that their abstraction level be raised above the details of an MSIL file. Tools such as Vulcan and Phoenix already provide a low-level interface for analyzing and modifying MSIL, but using them to write transformations requires a moderate amount of study and effort and has a steep learning curve. Moreover, they produce free-standing programs that manipulate MSIL files, not elements of a build process or the application abstraction. The goal of CTRs is to permit a programmer to write transforms in a C#-like notation that requires no knowledge of the details of MSIL or the C# compiler.

To achieve this end, we will follow a three-step plan. The first step is to provide a reflection-like interface for examining and translating MSIL. The second step is to express the generated MSIL in a high-level, C#-like language, so that a user can avoid the System.`Reflection.Emit` interfaces. The third step is to provide a higher-level language for describing the analysis and manipulation of a program.

4.1. Reflection Interface

Implementing an interface similar to the reflection interface in the CLR will enable us to quickly discover how much of the existing reflection code is static and how much is dynamic. The reflection interface is composed of an introspective part and a generative part. The introspection interface allows code to examine the structure of an assembly by enumerating its types, methods, and variables. All of this functionality is easily implemented in CTRs. Similarly, the generative part of this interface, which allows a program to create new types and methods, is easily built on Vulcan and Phoenix.

The distinction between CTRs and reflection becomes clear in code that uses the reflection interface to examine run-time values, either directly by accessing the program's state or indirectly through `FieldInfo` methods. In this case, CTRs must to generate code that defers the translation process until run time, when these values are available. Binding-time analysis can distinguish between values that are statically known and those only available when a program executes.



Since other information is available statically, the program's runtime environment need not carry around the full metadata of the CLR. CTRs can perform partial evaluation on the code that uses reflection, to make the static information in the MSIL available at run time. This partial evaluation may be difficult to perform on code that uses the reflection interface, but will be less difficult to manage with higher-level descriptions of the translation task.

4.2. Improved Code Generation

Producing executable code using reflection is tedious and error-prone because of the low-level manner in which the generated code is specified. The CLR's reflection facility provides operations to construct assemblies and modules and to populate them with classes and methods. However, a method's code is implemented by generating the MSIL instructions one at a time, as they would be produced in the backend of a compiler. This approach, although providing a programmer complete control over the generated code, requires knowledge of the MSIL instruction set and conventions and skill at assembly-language-level programming.

CTR provides a higher-level way in which to describe the generated code, using the Lisp-like syntax from the multi-stage compilation community. This syntax augments C# with *quote* and *unquote* operators, which permits programmers to build up C# code by combining syntactic fragments with literal values. For example, consider this function, which unrolls a loop:

```
public delegate Stmt Body(Expr index);

public Stmt unroll(int n, Body b) {            // Produce code that executes body b
  Stmt c = $< ; >$;                            // n times, with index values 0..n-1
  for (int i = 0; i < n; i++)
    c = $< ‘c ‘b.iteration($< ‘Int(i) $>) >$;
  return c;
}

public Stmt myBody(Expr index) {               // Produce body of loop for iteration
  return $< Debug.Print (‘index); >$;          // index
}

// Example:
Stmt output = unroll(2, new Body(this.myBody)) //$< Debug.Print(0); Debug.Print(1); >$

System.Reflection.EmitCode(output);            // Generate statements above
```

This example should look familiar to anyone who has programmed with macros in Lisp. The quote operators are "$<" and ">$" and the backquote operator, which evaluates an expression in a quoted form and splices its result into the form, is "‘". To aid comprehension, the quoted forms are highlighted in red and the backquoted forms in blue in the sample.

Unlike Lisp, these macros are typed, so that a statement has a different type than an expression. This typing provides a form of static checking that ensures a macro produces code that is syntactically correct. Initially, the following syntactic forms seem sufficient: `Expr, Stmt, Id, Type, Method, Field,` and literal types (`Char, Int, Float`, etc.). **ToDo:** explore whether typing fully ensures syntactic integrity, as in the multi-stage compilation work.

The quoted forms are a domain-specific language, which requires a compiler to translate them into explicit ASTs and produce code generators that ultimate use the System.`Reflection.Emit` interfaces. Initially, the methods that contain these quoted forms are only available and executed before Bartok compiles an assembly. However, binding time analysis of these methods (and the methods that invoke them!) may make it possible to partially evaluate this code and produce run-time code generators that produce code for a new process.

4.3. Improved Pattern Matching

A large amount of code that uses Reflection is performing pattern matching on the representation of a program. This pattern matching is very ad-hoc and generally consists of enumerating and examining all elements in a collection, such as the members in a class. It may be worthwhile extending the CTR language to make these patterns explicit. Consider, for example, the problem of finding the Singularity shell commands, which are methods in a variety of classes labeled with a special attribute:

```
[CommandAttribute("foo")]
void DoFooCommand(..) {…}
```

One way is to inspect each method, looking for a specific attribute:

```
class DemoClass {
        static void Main(Type type) {
            // Iterate through all the methods of the class.
            foreach(MethodInfo mInfo in type.GetMethods()) {
                // Iterate through all the Attributes for each method.
                foreach (Attribute attr in Attribute.GetCustomAttributes(mInfo)) {
                    if (attr.GetType() == typeof(CommandAttribute)) {
                        Shell.commands.Add(new Command(attr.name),
                                           new CommandDelegate(mInfo));
                    }
                }
         }
}
```

A more attractive way is to describe the desired methods:

```
Pattern
[CommandAttribute(<<command_name>>)]
void <<method_name>>(...) {…}
➔
Shell.commands.Add(new Command(<<command_name>>),
                   new CommandDelegate(<<method_name>>));
```