

Program Generation Research at University of Pisa

Giuseppe Attardi, Antonio Cisternino

Dipartimento di Informatica, via Buonarroti, 2, I-56127, Pisa, Italy
{attardi,cisterni}@di.unipi.it

Abstract. In this paper we describe research activities at University of Pisa about Program Generation. The main focus of our research is to find an appropriate runtime support for program manipulation and staging so that multi-staging and meta-programming are not tied to a particular language neither to have some interpreter at runtime. We already have CodeBricks, a library to perform code generation by combining pre-compiled methods. Next step will be to build a generic program transformation system in order to express persistent multi-stage computations.

1 Background: CodeBricks

CodeBricks allows manipulating abstractions of code fragments, which contain Intermediate Language (IL) code, but retain enough information about high-level types to be able to perform type checking and verification.

CodeBricks allows performing transformations on the code abstractions at the IL while retaining the illusion of manipulating source programs. When using a common intermediate language like ECMA CIL, as in our implementation of the library, code fragments produced by different languages can be used together.

CodeBricks provides a mean for a programmer to generate low level code, by letting the compiler to take care of the nitty-gritty details, and assembling fragments of code that resemble building blocks providing suitable primitives.

The approach gives the programmer detailed control on the process of code generation, while being able to perform specializations and optimizations in the code produced.

For instance a *Domain Specific Language* can be embedded within a general purpose language, by having specific code to be produced for the Domain Specific parts of the language.

CodeBricks is also able to express multi-staged computations. Manipulating code objects is a mean to produce the various versions of the program that will run at different stages.

Performing such transformations at the IL level, allows stages to be run on different language processors, rather than within a single processor as in traditional source-level approaches (e.g. MetaML).

2 Ongoing Research

The ultimate goal of our research is to build a complete framework to support programming languages featuring meta-programming and multi-staging. We expect that the resulting framework will be:

- Language independent
- Usable without special programming extensions (through an appropriate API)
- Multi-stage: provide persistent multi-stage program transformation
- Efficient (comparable with hand coded low level program generation)

We decided to target Microsoft .NET (though the approach can be easily ported to other platforms like Java, or even C under certain assumptions) because CLI has become a standard and the platform is multi-language. The already developed CodeBricks library is at the center of the framework: it provides a mean to manipulate code fragments in intermediate language format though the programmer perceives the manipulation at language level. This primitive mechanism helps to achieve the first two goals. After early performance tests it turned out that CodeBricks library is itself efficient: the code generation process is as fast as hand coded program generation. Nonetheless we should guarantee that the overall process is efficient, not just the code generation phase.

2.1 Functional Inlining

The code generation model implemented by CodeBricks is based on the notion of application with inlining: methods are reified to code objects (essentially a body with a signature and an environment) and their arguments bound to values or other code objects. Generated code behaves as the application of the method to the specified arguments. This approach has been borrowed from C++ template meta-programming and is radically different than the more popular model based on some form of quasi-quotation.

Although undisputedly effective and intuitive, quasi-quotation introduces also a new level of complexity into code because it forces the programmer to deal with its code at two different levels. The programmer should figure out in her mind how the meta-program will combine all the code fragments in the resulting program. Moreover, because quasi-quotation notation is usually designed to be concise, binding of names can be difficult to understand due to the mix of names of source and object programs.

Besides, relying on functional application plus inlining (*functional inlining*) allow expressing code fragments in a more scalable way. First of all quasi-quotation is handled by early stages of a programming runtime (usually the reader) whereas application is a deeper notion of programming languages. This implies that programming tools can provide better support to code generation expressed as function application assuming inlining rather than syntax-based transformations. The distinction among formal and actual parameters avoids confusing different levels of computations in the same expression. Finally the application notation focuses on the

semantics of generated code (by means of inlining) while providing control over the generated code: these two aspects are kept separated by this notation whereas in quasi-quotation are often tangled together.

It is worth noting that we refer to functional inlining as a mean to express code composition (where method bodies are akin to quasi-quoted terms and arguments comma expressions). Some mechanism to delay the execution of a functional inlining is required in order to be comparable with quasi-quotation mechanism.

We plan to extend C# (MetaC#?) with the notion of partial application as a mean to defer a computation and combine code fragments in a way similar to CodeBricks but with a better syntax. We believe that language designers should take into account functional inlining as a real alternative to quasi-quotation to express program manipulation. This language will provide an example of a programming language with meta-programming capabilities that relies on our runtime-level framework.

2.2 [a]C#

We have extended C# language (the extension is called annotated C#) in order to put custom annotations (in the same style of custom attributes) inside method bodies. With C# it is possible to annotate a method body like in the following example:

```
class MyAnnotationAttribute : ACS.CodeAttribute { }
class AnotherAnnotationAttribute : ACS.CodeAttribute {}

class Example {
    public static void Main(string[] args) {
        // Code without annotation
        [MyAnnotation] {
            // Code under the MyAnnotation attribute
            [MyAnnotation, AnotherAnn] {
                // Code inside a nested annotation here
            }
        }
        [MyAnnotation] // Single statement here
    }
}
```

Blocks tagged by braced annotations put marks into binaries that can be retrieved through an extended version of Reflection API. Code annotations allow decomposing method bodies (they are a sort of inverse operation of combination of methods provided by CodeBricks).

Annotated blocks can be retrieved and manipulated with the following operators:

- *Extrusion*: the code annotated is used to generate a new method whose arguments are the free variables
- *Injection*: code can be inserted immediately before and after an annotation
- *Replacement*: the code inside an annotation is replaced by another code specified

The language is still under development, and it is available from the CodeBricks project page (<http://codebricks.sscli.net>).

2.3 Assembly Rewriting Engine (ARE)

In order to achieve cross-stage persistence we should assume that after the execution of a computation stage should be possible to dump the new program on disk. This is a crucial mechanism because it could be used to implement essential program transformations such as installation or configuration.

We are designing a rewriting system that analyzes an assembly, finds annotated portion of code (with a label indicating the current stage), executes them and generates a new assembly.

Code annotations provide a mean to specify which portions of the code should be executed. The evaluation of annotated blocks will produce code bricks that will take their place. This provides a general mechanism to express homogeneous program transformations at the binary level. Nonetheless all the annotations and code bricks are perceived as part of the source program by programmers, providing control without knowledge of the language runtime.

3 Conclusions

We believe that meta-programming and multi-staged would lead to a better software infrastructure: multi-staging is a natural way to describe nowadays software deployment, with the fundamental difference that provides a linguistic means to express in the program itself rather than scattered in a puzzle of different systems.

We also believe that wide acceptance of these programming mechanisms in the main stream of programming requires that these operations are supported by the runtime rather than the language.

Our framework seems to be general enough to provide meta-programming and multi-stage support to a wide range of languages. Nevertheless we are also trying to build a language that directly exposes functional inlining instead of reducing quasi-quotation to it.

MetaC# will be an example of a compiler of a language targeting the runtime-framework in order to support meta-programming facilities.

Bibliography

1. *Microsoft Shared Source CLI*, <http://msdn.microsoft.com/net/sscli>.
2. ECMA 335, *Common Language Infrastructure (CLI)*, <http://www.ecma.ch/ecma1/STAND/ecma-335.htm>.

3. Taha, W., Sheard, T. *Multi-stage programming with explicit annotations*. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM' 97, Amsterdam, p. 203-217. ACM, 1997.
4. Calcagno, C., Taha, W., Huang, L., Leroy, X., *Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection*, paper available at <http://www.cs.rice.edu/~taha/publications/preprints/2003-04-09-B.pdf>.
5. Taha, W., *Multistage Programming: Its Theory and Applications*, PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Available at <ftp://cse.ogi.edu/pub/tech-reports/1999/99-TH-002.ps.gz>.
6. Sestoft, P., *Runtime Code Generation with JVM and CLR*, <http://www.dina.dk/~sestoft/rtcg/rtcg.pdf>.
7. Calcagno, C., Taha, W., Huang, L., Leroy, X., *A Bytecode-Compiled, Type-safe, Multi-Stage Language*, <http://citeseer.nj.nec.com/460583.html>.
8. Masuhara, H., and Yonezawa, A., *Run-time Bytecode Specialization: A Portable Approach to Generating Optimized Specialized Code*, In proceedings of Programs as Data Objects, Second Symposium, PADO 2001.
9. Tanter, E., Ségura-Devillechaise, M., Noyé, J., Piquer, J., *Altering Java Semantics via Bytecode Manipulation*, in proceedings of Generative Programming and Component Engineering (GPCE), LNCS 2487, 283-298, 2002.
10. Attardi, G., Cisternino, A., Kennedy, A., *Code Bricks: Code Fragments as Building Blocks*, in proceedings of 2003 SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), 66-74, San Diego, CA, USA, 2003.
11. Cisternino, A. *Multi-stage and Meta-programming support in strongly typed execution engines*. PhD Thesis, TD-5/03, Dipartimento di Informatica, Università di Pisa, May 2003, available at http://www.di.unipi.it/phd/tesi/tesi_2003/PhDthesis_Cisternino.ps.gz.
12. Attardi, G., Cisternino, A., *Multistage programming support in CLI*, In IEE Proceedings Software, Vol. 150, No. 5, 275-281, October 2003.
13. Sheard, T., *Accomplishments and research challenges in meta-programming* (invited talk), in LNCS 2196, 2-44, 2001.