

Run-time Program Generation

Sam Kamin
University of Illinois, Urbana-Champaign
kamin@cs.uiuc.edu

Position paper for inaugural meeting of IFIP Working Group 2.11
St. Emilion, France, May 16-18, 2004

Program generation is a promising approach to improving software engineering – and has been for thirty years.

My research in this area is based on my belief that the approach will reach its full potential only when a facility is created that satisfies two requirements: programs to be generated must be specified at the source level, and the program generation process must be carried out at the binary (or virtual machine code) level.

Source-level specification for program generators is not controversial. However, I would take this a bit further than is usual. Generated code should be specified *in the same language* as static code. Not a restricted subset, but the very same one. In this domain, restrictions are imposed for efficiency or safety reasons. But any unaccustomed restrictions can be frustrating to programmers, and should be viewed with skepticism.

Binary-level operation is achieved in a number of systems, but its importance merits emphasis. Experience teaches that programs distributed in binary are more appealing than those that need to be compiled.¹ The software components community uses the term *deployability*. If program generators were as deployable as other software components, it would enhance their popularity.

A facility satisfying these two requirements can be constructed in a straightforward way by, in effect, building the compiler into each fragment of code that will be included in the generated program. Of course, fragments cannot be compiled to machine code, because contextual information is required; they can only be compiled when contained in a “compilation unit.” But they can be *partially* compiled, producing a function that, when given contextual information, will generate machine code. In this scheme, programs, when delivered to a client, must first be loaded and executed to obtain the final executable, which can then be loaded and executed. (This process can also be done during execution, even multiple times.)

Ordinary compilers cannot be directly used in this way, for an obvious reason: The action of an ordinary compiler on a program fragment that is not a compilation unit – such as an isolated expression or statement – is to issue an error message. There is no useful compilation to perform partially. A program with “holes” will suffer a similar fate. Thus, the compiler needs to be rewritten. Taking a page from denotational semantics, we require that each syntactic construct have a self-contained compilation meaning. That is, for each syntactic construct \oplus , there must be a function **compile** _{\oplus} such that, for any syntactic fragments S_1, \dots, S_n ,

¹ I would argue that the reason for this phenomenon is the relative stability of machine language - even virtual machine language - as compared to any programming language. Machine language is a fixed target, while a programming language is, even if only a little bit, mobile; therefore, a binary is more likely to execute correctly than a source program is to compile and execute correctly.

$$\mathbf{compile}(\oplus[S_1, \dots, S_n]) = \mathbf{compile}_\oplus(\mathbf{compile}(S_1), \dots, \mathbf{compile}(S_n)).$$

Then, each fragment has a meaning, and programs with holes have a well-defined meaning as functions from the missing values to the compiled value. The codomain of **compile**, as noted above, cannot be machine language, but something richer; call it *Code*. Then there must be a function **codegen** mapping *Code* to machine language. When writing such a compiler, as much meaning as possible should be imparted by **compile**, leaving **codegen** relatively simple.

We call this approach *compositional compilation*. When the compiler is written in this style, run-time program generation comes essentially for free.

In our experience, this idea translates well into practice. We have written a full Java compiler (generating Java virtual machine code) in compositional form, and written numerous example programs. The compiler, called Jumbo, is quite easy to use.² Virtually any program that can be written in Java can be “quoted” and generated at run time, because it is compiled by residual code from the exact same compiler as that used for static compilation. In a recent experiment, generating code for marshalling produced an order-of-magnitude speed-up relative to a pure Java implementation.

My research starts from this base, and addresses questions about the efficiency of code generation, appropriate source and target languages, formal specification, safety, and applicability to other languages (particularly those compiled to machine language rather than virtual machine language). My current emphasis is on optimizing run-time program generation by partial evaluation. I also continue to look for interesting examples demonstrating the usefulness of run-time program generation.

² Download from <http://loome.cs.uiuc.edu>.