

Rational Metaprogramming

Diomidis Spinellis

Metaprogramming, using programs to manipulate other programs, is as old as programming. From self-modifying machine code in early computers to expressions involving partially applied functions in modern functional-programming languages, metaprogramming is an essential part of an advanced programmer's arsenal.



Also known as generative programming, metaprogramming leverages a computer language's power. Rather than manipulating plain data elements, it manipulates symbols representing various complex operations. However, like all levers, metaprogramming can be a blunt instrument. Small perturbations on the lever's short end (the language) result in large changes in the final product. In metaprogramming, this leverage can lead to unmaintainable code, insidious bugs, inscrutable error messages, and code-injection-attack vulnerabilities. Even when we take away industrial-strength compilers and interpreters, which are also metaprograms, we find metaprogramming wherever we look.

Everyday metaprogramming involves on-the-fly code production. Representative examples include dynamically generated SQL statements and code created for evaluation at runtime in interpreted languages. Metaprogramming also occurs in programs that spew out HTML or XML. Although we can't classify these markup languages as code, their rich syntactic structure qualifies their generation as metaprogramming. Unfortunately, we commonly produce code on the fly by simply pasting

together character strings. This means that it's difficult to verify essential properties of the generated code—such as validity, correctness, and safety—at compile time.

Language extensions

A more powerful type of metaprogramming involves extending existing languages or creating new ones.

In the C programming language, the vehicle for metaprogramming is the preprocessor, and its applications range from the mundane to the bizarre. In 1978, Steven Bourne was using macro definitions to give C the flavor of Algol. Ten years later, Jack Applin entered the International Obfuscated C Code Contest with an entry that calculated a list of prime numbers at compile time. Most commonly, however, the C preprocessor hides tricky or long code sequences behind macros that are easier on the eye. Although the C preprocessor has a functional-programming language at its core, its severe limitations (no recursion and no syntax or type checking of the generated code) make it a tricky, dangerous tool.

In the C++ world, we often use its templates facility for metaprogramming. In contrast to the C preprocessor, these lead to syntactically correct code. Through templates, we can provide type-safe and terse canned implementations of design patterns such as Visitor and Object Factory. However, hijacking a language facility (dauntingly named structural polymorphism and intended for creating more versatile classes) and using it to create elaborate language extensions leads to problems. While I admire the cleverness and skill that hides behind C++ libraries such as Boost (www.boost.org), the fact remains that writing advanced template code is devilishly hard, and even using it can be quite

tricky. This approach's brittleness is apparent in the compiler error messages that can span hundreds of lines if a developer uses a wrong type.

Java and .NET's modern frameworks provide more restrictive extension mechanisms: annotations (in Java) and attributes (in .NET). We can use these to annotate our code, and our metaprograms are then runtime libraries or compiler extensions that act on the annotated code. Unfortunately, although each platform lets us extend the language through a carefully designed API and corresponding tools, we're still severely constrained in extending the language. For instance, we can only use compile-time constants as arguments to Java's annotations.

Specialized languages and tools

The least restrictive form of metaprogramming involves implementing a domain-specific language (DSL) as a simple compiler or interpreter. Although this is a common approach with few inherent limitations, its large start-up cost weighs against the long-term productivity payoff. Tools for a DSL will always be inferior to those for a mainstream language and will always require an expert to maintain them.

So-called wizards that generate code from user replies to canned questions saddle us with an additional problem. Because the original user interaction is typically lost, we're left to maintain the often inscrutable wizard-generated code.

A final alternative involves using a language that's explicitly designed with metaprogramming in mind. Functional languages fall in this category, because in them functions are first-class citizens that we can manipulate in the same way as other data. Using a specialized transformation language, like TXL and (to a lesser extent) XSLT, can also work in some instances. However, in all cases we're left with the impression of operating in a niche area where support, documentation, and trained developers will be hard to come by.

A tall order

I hope to have convinced you by now that although metaprogramming is ubiquitous, the way we go about it leaves a lot to be desired.

In the early 1970s, Brian Kernighan, dis-

mailed code in Fortran (66, I guess), implemented RATFOR, a preprocessor that would take Fortran statements flavored with a block structure and spew out the goto-infested code that was in those days the norm. That made Fortran programming a considerably saner affair—"pleasant" in Kernighan's words. RATFOR presaged the widespread adoption of current block-structured languages, like C and Java.

Over the past few years, I've been dreaming about a similar move toward a rational framework for metaprogramming. Embarrassingly, I've been unable to come up with an acceptable result. However, in the process, I've put together some requirements for a satisfactory solution:

- *Consistent programming and metaprogramming languages.* Designing, learning, and supporting different languages is wasteful. Programmers often shy away from metaprogramming because they have to master two languages and handle their often tricky interactions. By using the same language at all levels, we can reap wide economies of scale.
- *Compile-time objects as first-class citizens.* This includes both code and types. We should be able to generate through code any program that we can write by hand.
- *Closed form.* Manipulations of compile-time objects should always lead to syntactically correct code and valid types.
- *Familiar metaprogramming constructs.* I admit that the functional-program-

ming community has already solved elegantly most of the problems I describe. Yet, despite impressive progress on many fronts, including performance, type-system versatility, library support, and tool availability, most programmers are loath to embrace an unfamiliar and—to their eyes at least—often impenetrable programming paradigm. So, we should base the solution on syntax and programming constructs that most developers already know. If the distance between the metaprogramming theory and the available language constructs is large, it's up to the theorists to bridge it, not the programmers.

- *Familiar code.* The metaprogramming facilities should enforce the principle of least astonishment on the target language. Programs that take advantage of metaprogramming-provided extensions should be readable and easy to understand by developers unfamiliar with the extensions.
- *Parsimony.* A general-purpose language supporting metaprogramming should be simpler than today's modern languages. Metaprogramming should provide many of the language features, just as today's languages rely on (standard) external libraries for much of their functionality.

Although languages like Lisp and Python satisfy some of the goals I've stated, I think that we're still a long way from a satisfactory solution.

Did I miss any requirements? Do you find my goal realistic? How can we go about making it a reality? Please post your responses in the column's blog. 

It's up to theorists to bridge the distance between metaprogramming theory and the available language constructs.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: www.spinellis.gr/tools