Program Generators and the Tools to Make Them

Yannis Smaragdakis, Shan Shan Huang, David Zook
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{yannis|ssh|dzook}@cc.gatech.edu

ABSTRACT

Program generation is among the most promising techniques in the effort to increase the automation of programming tasks. In this paper we discuss the potential impact and research value of program generation, give examples of our research in the area, and outline a future work direction that we consider most interesting. Specifically, we first discuss why program generators have significant applied potential. At the same time we argue that as a research topic, meta-programming tools (i.e., language tools for writing program generators) may be of greater value. We illustrate our views on generators and meta-programming tools with our latest work on the Meta-AspectJ meta-programming language and the GOTECH generator. Finally, we examine the problem of statically determining the safety of a generator and present its intricacies. We limit our focus to one particular kind of guarantee for generated code—ensuring that the generated program is free of compile-time errors. We believe that this research direction will see significant attention and will make a big difference in the mainstream adoption of meta-programming technology.

1. PROGRAM GENERATORS AND META-PROGRAMMING

We first present some general thoughts on program generators. We concentrate on frequently-asked questions about the nature and value of generators, as well as the research promise of the area.

1.1 What Are Program Generators?

A *program generator* (or just *generator*) is a program that generates other programs. This broad definition is often qualified to include constraints such as "the generated program is expressed in a high-level programming language". No definition, however, offers strict boundaries distinguishing generators from traditional compilers, text-generating programs, etc. Thus, what really constitutes a generator is often determined intuitively and does not reflect so much a technical distinction as the emphasis in the development of the "generator".

A related concept is *meta-programming*. Meta-programming can be described as the creation of a program that computes (something about) other programs. Meta-programming tools constitute the general platform for implementing generators in a given language setting.

To understand what generators are we should look at their common uses. In practice, generators are typically compilers for domain-specific languages (DSLs). A domain-specific language is a special-purpose programming language for a particular software domain. A "domain" can be defined either by its technical structure (e.g., the domain of reactive real-time programs, the domain of LALR language parsers) or by real-world application (e.g., the database domain, the telephony domain, etc.). The purpose of restricting attention to specific domains is to exploit the domain features and knowledge to increase automation.

If we view generators as compilers for DSLs, it is worth asking how they differ from general-purpose language compilers. Indeed, the research and practice of program generators is very different from that of general-purpose compilers. A general-purpose language compiler implements a stable, separately defined specification and can take several man-years to develop. In contrast, a generator is typically co-designed with the DSL that it implements. Thus, the emphasis is not on deeply analyzing a program to infer its properties, but on designing the DSL so that the properties are clearly exposed and on having the generator exploit them with as little effort as possible. The effort of implementing a program generator is typically small—comparable to the effort of implementing a software library for the domain. This is largely the result of leveraging the high-level language (commonly called the *object language*) in which the generated programs are expressed.

The above features of program generators (domain-specificity, language/generator co-design, low-effort development) also define the focus of research in the area. Most research in program generators concentrates either on specific domains that are amenable to a generator-based approach or on meta-programming tools that simplify generator implementation.

1.2 Why Care about Generators?

Program generators have been an active research focus since the early days of Computer Science. The main reasons that researchers are attracted to program generators have to do with the inherently interesting conceptual problems of meta-programming and with the potential for practical

PEPM'04, August 24–26, 2004, Verona, Italy.

benefit.

For many researchers, meta-programming is an intellectually fascinating topic. After all, computer scientists are people who find computations interesting. What then can be more interesting than computing about computations? The canonical sensationalist example is self-generating programs. It is easy to be intrigued by programs whose task is to output their own program text. For example, we can have:

```
((lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))))
```

in Lisp or:

```
main(a){a="main(a){a=%c%s%c;printf(a,34,a,34);}";
printf(a,34,a,34);}
```

in C.

At the same time, many researchers hold the belief that software engineering tasks can be substantially automated through program generation. Everyday programming often involves rote coding of pieces of functionality that can be generated mechanically, as long as the domain-specific knowledge is suitably encoded. Most programming practitioners are confident that they can achieve far greater code reuse with the appropriate domain-specific tool. (In fact, we have encountered many cases where people overestimate the potential for automation based on their experience with a small set of examples.) Nevertheless, the question is one of engineering balance between the cost of developing a DSL and the benefit of using it. In the software development arena, program generators compete primarily with lower-effort but also lower-benefit tools, such as domain-specific libraries/APIs. There are currently many thousands specialized software libraries for various domain-specific tasks.[1] Replacing some of them with domain-specific languages implemented as program generators may result in significant simplification of the programming tasks in the domain.

Libraries/APIs can themselves be thought of as crude programming languages. They have their own simplistic syntax (only function call syntax allowed); their own semantic restrictions (arguments need to satisfy some preconditions and calls may affect the state of the system and need to occur in specific ordering patterns) with very little static error checking (other than type checking of function calls in the host language); and their own simple optimization (libraries typically offer multiple hand-specialized versions of operations for different kinds of arguments, such as special-purpose multiplication operators for sparse arrays in a scientific computing library). Users of standard libraries/APIs are often constrained more by the library semantics than by the semantics of the host programming language. This is a common sentiment among users of large parallel processing and scientific computing libraries (like MPI or LAPACK). It is also often expressed by programmers in large projects where each part of the code needs to support the conventions of other parts. In this case, the "domain" is the project itself. A Microsoft programmer once told us "I don't program in C, I program in Word's internal API".

The software engineering benefit of domain-specific languages relative to function libraries/APIs is exactly in addressing the above deficiencies of syntax, safety, and performance. A domain-specific language can offer more concise syntax, increasing the ease of development and maintainance; it can perform static error checking to detect common violations of the library semantics; and it can offer better performance through domain-specific optimizations.

A common argument (especially of the partial evaluation community) is that the advantages of program generation can be expressed entirely on the performance axis. After all, if performance is not a concern, the syntax and safety benefits of a domain-specific language can be achieved with just an interpreter. Subsequently, if better performance is required, it can be achieved by specializing the interpreter. This specialization can occur either via a general-purpose partial evaluator or via program generation. Thus, according to this view, program generation is an alternative to partial evaluation and it is entirely about code specialization. Although it is tempting and often useful to think entirely in performance terms, we should note that this view is not accurate for many, if not most, of the program generators in actual use. In practice, program generators are often employed not for reasons of performance but for reasons dictated by modularity. Many generation tasks are performed to produce code so that it satisfies conventions of existing external libraries, run-time systems, etc. For example, a domain-specific language for telephony can be generating code in the form needed to interface with a specific telephony standard API [**?**]. The GOTECH generator that we will later present yields components that can be loaded in a Java application server. None of these tasks is performance related: programs need to be generated in a specific form in order to interface with external clients.

1.3 The Research Value of Meta-Programming

Although program generators are very interesting in structure and valuable in practice, individual generators are often less than ideal targets for research. The reason has to do with the domain-specificity of generators. Domain knowledge is by far the primary element of a successful generator. Unfortunately, however, domain knowledge is not accessible to non-domain-experts. Since research is concerned with the encoding and transmission of reusable knowledge, the domain-specificity of generators often limits their research impact to narrow communities. Without domain-independent results it would be hard to speak of research in generators as a whole.

In contrast, we believe that meta-programming infrastructure is a much more promising research target, especially for the programming languages and tools research community. Meta-programming tools are the domain-independent part of program generation. They pose all the interesting conceptual research problems of program generation and have a large applied value. In terms of technical problems, meta-programming has a need for better language constructs, type systems and analyses to ensure safety, etc. Additionally, meta-programming stresses the limits of language processing and analysis technology (e.g., parsing) because the generated code fragments are small and often appear out of their final program context [12]. In terms of applied value, the benefit of generators is tied to the ease with which they can be implemented. A successful meta-programming tool is one that significantly simplifies generator implementation. This simplification can be a result of added expressiveness

[1] We are not aware of exact current figures, but a 1997 survey of libraries exported by Microsoft revealed over 1800 different APIs [**?**].

(i.e., easy program generation and transformation) or added safety (i.e., avoiding errors in coding generators).

Perhaps surprisingly, the potential impact of meta-programming tools is lower for mature, well-understood, high-value domains. For such domains, the importance of domain abstractions is so high that better meta-programming infrastructure is unlikely to matter much. For instance, it is hard to imagine that good meta-programming tools would make a difference for parser generation (i.e., generators like Yacc, ANTLR), string and system command processing (i.e., implementations of scripting languages like Perl), or database processing (i.e., languages like SQL). In general, although such systems are technically translators for DSLs, they are not part of the focus of program generation. Meta-programming tools can have a greater impact in less mature domains, i.e., exactly where it is not yet clear whether applying a domain-specific language approach is appropriate. Good meta-programming infrastructure can bridge the implementation effort gap between a library and a generator, thus making program generation cost-feasible for many more domains.

2. META-ASPECTJ AND GOTECH

In this section we discuss our recent work on the Meta-AspectJ meta-programming tool and the GOTECH program generator. Both of these artifacts illustrate well our earlier and later points in this paper, regarding the conceptual and applied value of meta-programming, as well as the kinds of generators that meta-programming tools should support.

2.1 Meta-AspectJ

Meta-AspectJ (MAJ) [14] is a language tool for generating Java and AspectJ programs using code templates, i.e., a quote and unquote operators. Two elements of MAJ are quite interesting. First, we believe that using the AspectJ language as a back-end simplifies the task of writing a generator. Second, MAJ is a technically mature meta-programming tool—in many respects the most advanced meta-programming tool for Java. For instance, MAJ reduces the need to deal with low level syntactic types for quoted entities (e.g., "expression", "statement", "identifier", etc.) through type inference and a context-sensitive parsing algorithm.

2.1.1 What Do Aspects Have to Do with Generation?

The AspectJ [5] language is an extension of Java and the flagship tool of *aspect-oriented programming (AOP)*: a methodology that advocates decomposing software by aspects of functionality. AspectJ supports defining aspects separately from the main application code and subsequently merging them with that code. For the purposes of our discussion, we can view AspectJ as a sophisticated code manipulation tool. With AspectJ, the user can add superclasses and interfaces to existing classes and can interpose arbitrary code to method executions, field references, exception throwing, and more. Complex enabling predicates can be used to determine whether code should be interposed at a certain point. Such predicates can include, for instance, information on the identity of the caller and callee, whether a call to a method is made while a call to a certain different method is on the stack, etc.

Our use of AspectJ in MAJ is not tied to the AOP methodology at all. Instead, the main observation behind Meta-AspectJ is that generators can use AspectJ as a back-end language. AspectJ offers a convenient vocabulary for expressing program inspection and transformation. This functionality is crucial for program generators. A generator often needs to inspect an existing program and generate code for each field, method, field assignment, etc. encountered. At the same time, the generator often needs to transform the original input program—e.g., to make it call the newly generated code. Such transformations are conveniently expressed in the AspectJ vocabulary. For example, consider a simple transformation of a Java program. We may want to take as input a class `C` and add an interface definition clause to it. At the source code level, this means transforming the code of `C` from:

```
class C ... {...}
```

to:

```
class C implements I ... {...}
```

The standard way to effect this code transformation would be to parse the input, find the definition of class `C` and add the `implements` clause as a branch in the abstract syntax tree representing the `C` definition. At the same time, we need to be careful to find the right class `C` (and not, for instance, some local class with the same name), preserve its existing superclasses and interfaces, etc. An alternative way to do this transformation would be to emit the AspectJ code:

```
aspect S { declare parents: C implements I; }
```

The latter way is simpler and relieves us of dealing with many low-level details.

The above example is admittedly simple. For more complicated code transformations (e.g., indirecting all sites where a field value may be changed, transforming all instantiations of a class to instantiations of a different class, etc.) the benefit of expressing them with AspectJ strictly increases. AspectJ offers a convenient level of abstraction in transforming programs. Although AspectJ cannot manipulate low-level code (e.g., one cannot find all the `if` statements or all `for` loops in a program and change them) it can transform almost every externally visible element of a Java class (e.g., methods, fields, type information, etc.). At the same time, AspectJ features a mature bytecode compiler implementation.

2.1.2 MAJ Technically

MAJ extends Java with two code-template operators for creating AspectJ code fragments: `‘[...]` ("quote") and `#[EXPR]` or just `#IDENTIFIER` ("unquote"). (The ellipses, `EXPR` and `IDENTIFIER` are meta-variables matching any syntax, expressions and identifiers, respectively.) The quote operator creates abstract syntax tree representations of AspectJ code fragments. Parts of these representations can be variable and are designated by the unquote operator (instances of unquote can only occur inside a quoted code fragment). For example, the value of the MAJ expression `‘[call(* *(..))]` is a data structure that represents the abstract syntax tree for the fragment of AspectJ code `call(* *(..))`. (This AspectJ expression matches all method calls for any method with any argument list and any return type.) Similarly, the

MAJ expression `‘[!within(#className)]` is a quoted pattern with an unquoted part. Its value depends on the value of the variable `className`. If, for instance, `className` holds the identifier “`SomeClass`”, the value of `‘[!within(className)]` is the abstract syntax tree for the expression `!within(SomeClass)`.

MAJ also introduces a new keyword `infer` that can be used in place of a type name when a new variable is being declared and initialized to a quoted expression. For example, we can write:

```
infer pct1 = ‘[call(* *(..))];
```

This declares a variable `pct1` that can be used just like any other program variable. For instance, we can unquote it:

```
infer adv1 = ‘[Object around() : #pct1 { }];
```

This creates the abstract syntax tree for a piece of AspectJ code defining what should happen (nothing in this case) every time any method gets called.

In the above example, the inferred type of variable `adv1` will be `AdviceDec` (for “advice declaration”), which is one of the types for AspectJ abstract syntax tree nodes that MAJ defines. The full set of permitted type qualifiers contains over 20 types, including `Identifier`, `Modifiers`, `JavaExpr`, `Stmt`, `MethodDec`, `ConstructorDec`, etc. In addition to variable definitions, such types can also be used explicitly in the quote/unquote operators. For instance, the fully qualified version of the `adv1` example would be:

```
AdviceDec adv1 =
  ‘(AdviceDec)[Object around(): #(Pcd)pct1 {}];
```

The inference of qualifiers and types of variables holding abstract syntax trees relieves the user from dealing with the specifics of abstract syntax tree types, while maintaining the static checking of the well-formedness of trees. This feature distinguishes MAJ from other meta-programming tools. Having multiple quote/unquote operators is the norm in meta-programming tools for languages with rich surface syntax (e.g., meta-programming tools for Java [1], C [13], and C++ [3]). For instance, let us examine the JTS tool for Java meta-programming—the closest comparable to MAJ. JTS introduces several different kinds of quote/unquote operators: `exp{...}exp`, `$exp(...)`, `stm{...}stm`, `$stm(...)`, `mth{...}mth`, `$mth(...)`, `cls{...}cls`, `$cls(...)`, etc. Additionally, just like in MAJ, JTS has distinct types for each abstract syntax tree form: `AST_Exp`, `AST_Stmt`, `AST_FieldDecl`, `AST_Class`, etc. Unlike MAJ, however, the JTS user needs to always specify explicitly the correct operator and tree type for all generated code fragments. For instance, consider the JTS fragment:

```
AST_Exp x = exp{ 7 + i }exp;
AST_Stm s = stm{ if (i > 0) return $exp(x); }stm;
```

This written in MAJ is simply:

```
infer x = ‘[7 + i];
infer s = ‘[if (i > 0) return #x;];
```

The inference of type qualifiers, as in MAJ, requires sophistication in the implementation of the meta-programming tool. The meta-programming tool needs to be a full-fledged compiler, with its own type system (instead of a naive pre-processor, translating to an object language and relying on its type system for checking). Additionally, parsing becomes context-sensitive—i.e., the type of a variable determines how a certain piece of syntax is parsed, which puts the parsing task beyond the capabilities of a (context-free) grammar-based parser generator.

To see the above points, consider the MAJ code fragment:

```
infer l = ‘[ #foo class A {} ];
```

The inferred type of `l` depends on the type of `foo`. For instance, if `foo` is of type `Modifiers` (e.g., it has the value `‘[public]`) then the above code would be equivalent to:

```
ClassDec l = ‘[ #(Modifiers)foo class A {} ];
```

If, however, `foo` is of type `Import` (e.g., it has the value `‘[import java.io.*;]`) then the above code would be equivalent to:

```
CompilationUnit l = ‘[ #(Import)foo class A {} ];
```

Thus, to be able to infer the type of the quoted expression we need to know the types of the unquoted expressions. The MAJ type system itself is simple: it has a fixed set of types with a few subtyping relations and a couple of ad hoc conversion rules (e.g., from Java strings to MAJ identifiers). Type inference is quite straightforward: when deriving the type of an expression, the types of its component subexpressions are known and there is a most specific type for each expression. No recursion is possible in the inference logic, since the `infer` keyword can only be used in variable declarations and the use of a variable in its own initialization expression is not allowed in Java.

The types of expressions even influence the parsing and translation of quoted code. Consider again the above example. The two possible abstract syntax trees are not even isomorphic. If the type of `foo` is `Modifiers`, this will result in an entirely different parse and translation of the quoted code than if the type of `foo` is `Import` (or `ClassDec`, or `InterfaceDec`, etc). In the former case, `foo` just describes a modifier—i.e., a branch of the abstract syntax tree for the definition of class `A`. In the latter case, the abstract syntax tree value of `foo` is at the same level as the tree for the class definition.

An added advantage of having a full compiler for the meta-programming tool is that it can emit better error messages. MAJ differentiates between MAJ type errors (i.e., syntactically invalid generated code) and regular parse errors in the MAJ language. It then emits accurate error messages relative to the original MAJ source.

2.2 Applications: The GOTECH Generator

We will next briefly describe our GOTECH generator. GOTECH has two interesting elements. First, it is implemented by producing AspectJ code, thus being a representative of the generators that MAJ can support. Second, it shows the kinds of program manipulations that we need to handle in meta-programming tools. The next section, discussing future directions in statically safe meta-programming, will often refer to examples of tasks from GOTECH.

2.2.1 What Is GOTECH?

GOTECH [11] (for “General Object-To-EJB Conversion Helper”) is a program generator for Java server-side components. GOTECH takes as input a Java program annotated with JavaDoc comments to describe what parts of

the functionality should be remotely executable. It then transforms parts of the program so that they execute over a network instead of running on a local machine. The middleware platform used for distributed computing is J2EE (the protocol for Enterprise Java Beans—EJB). GOTECH takes care of generating code adhering to the EJB conventions (EJB session beans) and makes methods, construction calls, etc. execute on a remote machine. Internally, the modification of the application is performed by generating AspectJ code that transforms existing classes. GOTECH was originally implemented using XDoclet [9]—a simple, text-based meta-programming tool. Subsequently GOTECH has been expressed more safely and conveniently with MAJ.

It is interesting to look at the specific program inspection and generation tasks that GOTECH performs. These include the following (for each Java class that has the appropriate GOTECH input annotation):

- GOTECH generates two isomorphic interfaces—i.e., Java interfaces whose methods correspond one-to-one to the public methods of the class. These are the *local* and *remote* interfaces required for the session bean according to the EJB specification.

- An EJB class (session bean) is generated by cloning the functionality of the original Java class. The session bean declares that it implements the generated local and remote interfaces and modifies the original method signatures to throw the exceptions required by the distribution middleware.

- An AspectJ aspect is generated to intercept all instantiations and calls to methods of the original Java class. All such object creations and client calls are now performed on the EJB class.

- An aspect is generated to make parameter types serializable (i.e., passed by-copy when used as arguments to remote calls) if necessary, according to user-supplied annotations.

2.2.2 Using AspectJ in GOTECH

GOTECH illustrates the value of using AspectJ as a generator back-end. The transformations performed by GOTECH (e.g., changing all client calls, making parameter types serializable) would be much harder to do without AspectJ. On the other hand, AspectJ alone would not be able to handle the GOTECH tasks. The GOTECH activities were previously [8] shown impossible to automate with just AspectJ. For example, AspectJ cannot be used to create an interface isomorphic to the public methods of a given class. Additionally, the aspects used are not static—they need to be custom-generated for the particular input application. The customization is not just with respect to the names of classes that need to be transformed. Instead, complex decisions may need to be made. For instance, the GOTECH logic making method parameter types be serializable is roughly "the type should be serializable, *if* neither it nor its supertypes are already declared to be serializable *and* it is not a primitive type." None of these tests can be expressed in AspectJ but they are easy to compute using arbitrary Java code and reflection. Then the right custom aspect gets generated when applicable.

3. GUARANTEED-LEGAL GENERATORS

In this section we present some thoughts on valuable research directions for meta-programming tools. We discuss the motivation for statically safe program generation, the state-of-the-art, and what needs to be addressed next to support powerful generators.

3.1 What Is Guaranteed-Legal Generation and Why Do We Need It?

As we saw earlier, the value of a MAJ quoted code fragment is an abstract syntax tree for the code fragment. Representing the generated program as abstract syntax trees is a standard technique in *structured* meta-programming tools—to be contrasted with unstructured, or "text-based" tools. It statically ensures that any generated code is syntactically correct by requiring that the tree be well-formed. This property is often described as "the type safety of the generator implies the syntactic correctness of the generated program." Nevertheless, we would like to go further than mere syntax checking. The generated program may still contain semantic errors.

Offering guarantees on the generated code is one of the toughest and most interesting issues in meta-programming. We are interested in *statically safe* program generation: statically checking the generator should enforce the safety of the generated program. Since the space of program safety properties is huge and most of these properties are undecidable, we limit our attention to *language legality* checking. Specifically, we use the term *guaranteed-legal generation* for the process of statically ensuring that the generated program does not suffer from errors typically detected by a conventional compiler. Such statically-detectable semantic errors include type mismatch errors or scoping errors, such as references to undeclared variables, duplicate variable definitions, etc. Thus, a guaranteed-legal generator is one always producing programs that will compile. It is the responsibility of the meta-programming infrastructure used to write the generator to statically ensure that it is guaranteed-legal. Of course, having any kind of sound static safety check means putting restrictions on the expressible programs. Hence, some safe programs will be rejected. The challenge is to design a static checking mechanism that is expressive enough for common program generators (e.g., the GOTECH program manipulations).

It is often debated whether static legality checking is a valuable feature. After all, the generated program will be checked statically before it runs, so why try to catch these errors before the program is even generated? The answer is that static checking is not intended to detect errors in the generated program, but errors in the generator. Although these errors will be detected at compile-time of the generated program, this is at least as late as the run-time of the generator. Thus, static legality checking for generators is analogous to static typing for regular programs. It is a desirable property because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator writer has tested the generator).

To see the problem in an example, consider a program generator that emits programs depending on two input-related conditions:

```
if (pred1())
```

```
  emit( `[int i;] );
...
if (pred2())
  emit( `[i++;] );
```

If for some input `pred2` does not imply `pred1`, then the generator can emit the reference to variable `i` without having generated the definition of `i`. This is an error in the generator and it should be the responsibility of a good meta-programming language tool to prevent such errors. In fact, the error may only occur after the generator writer has tested and widely deployed the generator. The error will then be detected by some random user. The analogy to compilers for general-purpose languages is interesting. Consider a C compiler that for some inputs (perhaps perfectly legal C inputs) produces executable files in an illegal binary format—e.g., illegal ELF or .EXE files or machine code for the wrong architecture. These files do not represent programs and will not fail at run-time: the operating system will reject them at load time. Nevertheless, this is small consolation to the C compiler author: his/her artifact was deployed while containing basic errors and has failed during its run-time. If there existed a language tool whose purpose was to help C compiler writers, it would be desirable to statically check that the compiler will always produce legal executables. Of course, in general meta-programming the problem is both more pronounced (due to the multitude of potential generators relative to the number of C compilers) and harder (due to the richness of target high-level languages relative to the shallow rules for binary executable formats).

3.2 State of the Art and Why We Need More

Static safety checking is a hard property to ensure, even when we limit our attention to language legality checks. Nevertheless, an interesting special case of program generation already offers strong legality guarantees for generated programs. Specifically, multi-stage languages, such as MetaML [10] and MetaOCaml [2], guarantee that the generated program is type-correct by statically checking the generator. In this sense, multi-stage languages represent the state of the art in static safety checking of generators. Nevertheless, as we will see, staging applies severe restrictions on the structure of the generator and prohibits the expression of arbitrary generators, like GOTECH.

Staging language constructs are fairly common in the partial evaluation community. They comprise primitives for quoting, unquoting, and performing run-time evaluation (with an eval/run construct) of code representations. The focus of multi-stage languages, however, is performance. Multi-stage languages serve as assembly languages for partial evaluation. The staging primitives are sufficient for expressing any partial evaluation task, either hand-staged or derived by an automatic partial evaluator through binding time analysis. Staging differs from general program generation, however. The hallmark property of multi-stage languages is the *erasure property*: erasing the staging constructs should yield a meaningful program. Clearly this property does not hold for arbitrary program generation, even when the meta-language and the object language are the same. In MAJ, for instance, we can write:

```
infer x = `[ class A {} ];
```

Removing the quote construct does not result in a legal Java program, however. For an example of a staget program, consider a simple exponentiation function (we use MAJ-style quotes and unquotes but the function is shown in ML-like pseudo-code, to avoid low-level complexities of explicit typing):

```
exp(n,a) =
  if (a == 0) 1
  else  n * exp(n, a-1)
```

If we would like to stage this function with respect to a statically known exponent, the result would be:

```
exp(n,a) =
  if (a == 0) `[1]
  else `[#[n] * #[exp(n, a-1)]]
```

It is easy to see that erasing the staging constructs from the second `exp` function will yield exactly the first one. (As a side note, in this example the two `exp` functions are not exactly equivalent. The latter `exp` is a generator even when all involved values are known: `exp(3,4)` will yield `` `[3*3*3*3*1] ``. The run/eval primitive can then evaluate this expression.)

In order to offer type correctness guarantees on the generated code, multi-stage languages place rigid syntactic restrictions on the generator. The first restriction is that the binding of an identifier should be clear from the generator code. Specifically, the only way to quote a code fragment containing free variables is in the scope of another quote that contains definitions for these variables. That is, we can write:

```
`[ int i;
   #[fun( `[ i++; ])]
 ]
```

In this fragment, `i` occurs free in the inner quote, but is bound to a definition in the outer quote. We cannot, however, write two independent quote expressions containing a separate definition and reference, such as:

```
... `[ int i; ]
... `[ i++; ]
```

This is a serious restriction for arbitrary program generation. Supporting separate variable definition and reference would dissociate the structure of the generator source code from the structure of the generated program, allowing more freedom in writing the generator. The main problem is not one of scoping—one can imagine a namespace construct that ties together independent variable definitions and references. Instead, the greater issue is control flow. Just as we saw in Section 3.1, separating the definitions and references can result in generated programs referencing undeclared variables. Furthermore, the same quoted definition may be used to create multiple actual variable definitions in the generated program. For instance, consider the fragment:

```
while (pred1()) {
  emit( `[int i;] );
  ...
}
...`[i++;]
```

There may be several distinct definitions generated by the quoted fragment `` `[int i;] ``. The problem is not one of

naming conflicts among the `i` definitions. A good meta-programming system should rename the actual variable names (from `i` to some unique identifier for every iteration) to avoid accidental name conflicts with both user-supplied code and generated declarations. (This is part of the *hygiene* problem for meta-programming [6, 4, 7].) More importantly, however, it is not clear which binding of `i` is intended in the generated expression `i++`.

Current systems that offer static legality guarantees also restrict the names used in the generated program. In multi-stage languages, it is not legal to unquote an identifier. Instead, all names used in definitions and references are constants. For instance, it is not possible to write an expression such as:

```
emit( `[int #name; ] );
```

This is a severe limitation. Recall the program manipulations performed in GOTECH. One of the most straightforward ones is to generate isomorphic interfaces for each input class. Such interfaces need to have methods with names that depend on the generator input. In general, program generators commonly emit both definitions and references with names that are not statically known. For instance, a generated program may be calling existing methods of classes, whose names are discovered at generation time.

Allowing non-constant names in quoted expressions is an issue of data flow in the generator. For example, consider a generator that introduces two new names in the same lexical context:

```
emit( `[int #name1; ] );
emit( `[int #name2; ] );
```

For static language legality checking, we need to know that `name1` and `name2` do not hold the same value (or we will end up with a duplicate variable definition in the generated program).

3.3 Open Problems and Promising Directions

Based on the above observations, meta-programming tools need to advance if they are to support realistic guaranteed-legal generators, like GOTECH. There is one general design direction that we are pursuing and we consider particularly promising. Given that the problem is one of data and control flow analysis in the generator, we expect that an easy-to-analyze meta-programming language will be beneficial. For instance, consider a sub-Turing-computable language with controlled iteration and selection. This language can integrate a reflective mechanism (analyzing an input program to extract its elements) with program generation. That is, the language can define iterators/cursors over existing programs. The iterators can range over, say, all fields of a class, all arguments of a method, all classes in a package, etc. All program generation should be predicated on an iterator: copies of the quoted code will be generated for each iteration. For example, we could have a code generation expression such as:

```
#for[f in Field(c), `[ #[Type(f)] #[Name(f}] ; ]]
```

(The `#for`, primitive is part of our invented syntax, as are the usual `` `[...] `` and `#[...]`. `Field`, `Type`, and `Name` are iterator functions.) In this case, several variables are being declared. The generated variable names are not statically known but they depend on existing names. Thus, static checking can be done, based on the assumption that the input program is legal. For instance, the above code fragment can never generate a duplicate definition: the generated names are in a one-to-one mapping with fields of input class `c`, which are guaranteed to be uniquely named. Similarly, when generating references, we can use the iterators to match them to generated definitions. For instance, we can refer to the variables generated by the above fragment in code such as:

```
#for[f in Field(c), `[insert(#[Name(f)])] ]
```

More specifically, we want to define collections of elements of existing programs in a relational view. That is, we will treat the input program as a set of relations (or a single big relation), matching packages and classes, classes and fields, methods and arguments, etc. Each of our iterators will then be defined using a relational calculus query. Subsequently we can use the iterators to determine the control and data flow of program generation. Our controlled-iteration language will be able to interface with a general-purpose language but the analysis will be limited to the controlled-iteration language. For a more complete example, consider a routine generating an isomorphic interface for a give input class. In our syntax, this would be written as:

```
Input [ cl :: Class; ]

pm(c::Class) = m::Method(c) :
                   "public" in Modifier(m);

total() =
`[interface #name[cl, "Isomorphic"]
  {
   #for[m in pm(cl),
    `[#[RetType(m)] #[Name(m)]
      (#for[a in Args(m),`[#[Type(a)] arg_name]])
     ]
    ]
  }
 ] ;
```

The `Input` clause in the above program declares `cl` as an input class relation. Then, `pm` is defined as an iterator over all public methods of the class. Finally, the generator function `total` generates an interface (named after the input class, with the appended suffix "Isomorphic") that contains all public methods of `cl` with identical signatures.

The above description is sketchy—we omitted many of the specifics of the syntax and semantics of the proposed language, other language constructs for conditional generation, etc. Nevertheless, the example is hopefully sufficient to outline the interesting conceptual problems. In order to match references to definitions (e.g., for type checking, or checking for undeclared variable references) we need to know which definitions are generated under the conditions for generating the reference. Recall our problem from Section 3.1:

```
if (pred1())
  emit( `[int i;] );
...
if (pred2())
  emit( `[i++;] );
```

We need to know whether `pred2()` implies `pred1()`. With our controlled language, all generation is predicated on an

iterator. Thus our problem is one of containment of the relations ranged over by two iterators. This is not a decidable problem in the full relational calculus but there are many well-known restrictions that result in decidable containment problems. The restriction could be imposed on the syntax or on the reasoning power (in which case, our checking will reject some programs that would always generate legal code).

We outlined above just one design direction in the space of guaranteed-legal meta-programming. In the end, the right solution will be an engineering tradeoff between expressiveness and safety. Many interesting open issues remain to be explored, even in the immediate vicinity of the ideas we outlined above:

- Can we do a powerful enough data- and control-flow analysis for a general purpose, Turing-complete language that will support static legality checking while allowing to express most common generation tasks?
- For controlled-iteration languages, like the one sketched above, what is an expressive logic that allows decidable containment checking?
- What is a good balance between the run-time complexity of containment checking and the expressiveness of the logic?

4. CONCLUSIONS

5. ACKNOWLEDGMENTS

6. REFERENCES

[1] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.

[2] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2830, pages 57–76. Springer, 2003.

[3] S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, Oct. 1995.

[4] W. Clinger. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[6] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.

[7] Y. Smaragdakis and D. Batory. Scoping constructs for program generators. In *Generative and Component-Based Software Engineering Symposium (GCSE)*, 1999. Earlier version in Technical Report UTCS-TR-96-37.

[8] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[9] A. Stevens et al. *XDoclet Web site, http://xdoclet.sourceforge.net/.*

[10] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[11] E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.

[12] E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2487, pages 299–315. Springer, 2002.

[13] D. Weise and R. F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[14] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In *Proceedings of the 2004 Generative Progamming and Component Engineering (GPCE) Conference*. Springer-Verlag, to appear.