A Personal Outlook on Generator Research (A Position Paper)

Yannis Smaragdakis

College of Computing, Georgia Institute of Technology
Atlanta, GA 30332, USA
yannis@cc.gatech.edu

http://www.cc.gatech.edu/~yannis/

Abstract. If we want domain-specific program generation to form the basis of a strong, long-lived research community, we need to recognize what its potential impact might be and why the promise has not been fulfilled so far. In this chapter, I review my past work on generators and I present a collection of personal opinions on the symptoms convincing me that there is room for improvement in the generators research community. Then I analyze the causes of these symptoms, some of which are inherent, while some others can be overcome. A major cause of difficulty is the inherent domain-specificity of generators that often makes research work be less valuable to other generator writers who are unfamiliar with the domain. I propose directions on what should be considered promising research for the community, what I believe are useful principles for generator design, and what community building measures we can take.

1 Introduction

This chapter is a personal account of my past work and current thoughts on research in software generators and the generators research community.

As an opinion piece, this chapter contains several unsubstantiated claims and (hopefully) many opinions the reader will find provocative. It also makes liberal use of the first person singular. At the same time, whenever I use the first person plural, I try to not have it mean the "royal 'we' " but instead to speak on behalf of the community of generators researchers.

There are two ways to view the material of this chapter. The first is as a threat-analysis for the area of domain-specific program generation. Indeed, a lot of the discussion is explicitly critical. For instance, although I believe that domain-specific program generation has tremendous potential, I also feel that the domain-specificity of the area can limit the potential for knowledge transfer and deep research. Another way to view this chapter, however, is as an opportunity-analysis: based on a critical view of the area, I try to explicitly identify the directions along which both research and community-building in software generators can have the maximum impact.

I will begin with a description of my background in generators research. This is useful to the reader mainly as a point of reference for my angle and outlook.

2 My Work in Generators

A large part of my past and present research is related to software generators. I have worked on two different transformation systems (Intentional Programming and JTS), on the DiSTiL generator, on the Generation Scoping facility, on C++ Template libraries and components, and on the GOTECH framework for generation of EJB code.

2.1 Transformation Systems

Transformation systems are infrastructure for generators. They are usually tools for doing meta-programming: writing programs that manipulate other programs. Transformation systems typically include a language extension facility (e.g., a macro language and ways to add concrete syntax), and a transformation engine: a way to specify manipulations of syntax entities and semantic information.

I have worked on two different transformation systems: Intentional Programming (IP) and JTS. IP [1, 11, 12] was a long term project at Microsoft Research that intended to provide a complete and mature language extensibility environment. IP's goal was to accommodate an ecology of transformations that would be provided by different sources and would not need to be designed to cooperate. (The term "ecology" refers to a biological metaphor that has transformations or language features be the analogue of genes and programming languages be the analogue of organisms, through which transformations propagate and evolution takes place.) As part of my work on IP, I implemented code template operators (a quote/unquote facility for manipulating program fragments as data) and a pattern language for transformations. Two interesting results of my IP work were the *generation scoping* facility and the *DiSTiL* generator, both of which I describe later in more detail.

JTS [3] is an extensible Java parser and syntax analyzer. It enables syntactic extensions to the language and arbitrary syntax tree transformations. JTS provides a standard programmatic interface for manipulating syntax trees, but also offers a full pattern language and a macro facility. JTS has served mostly as an experimentation platform for language extension ideas (e.g., additions of templates and module-like constructs in Java [17]) and domain-specific languages (e.g., the P3 generator [4]).

2.2 DiSTiL

The DiSTiL domain specific language [12] is an extension to C that allows the user to compose data structure components to form very efficient combinations of data structures. The language has a declarative syntax for specifying data structure operations: the user can define traversals over data through a predicate that the data need to satisfy. The DiSTiL generator can then perform optimizations based on the static parts of the predicate. Optimizations include the choice of an appropriate data structure, if multiple are available over the same data.

The following (slightly simplified) DiSTiL source code fragment shows the main elements of the language, including the data structure definition (`typeq1` and `cont1` definitions), cursor predicate (`curs1` definition) and traversal keywords (`foreach`, `ref`).

```
struct phonebook_record {...};   // C definition

typeq (phonebook_record, Hash(Tree(Malloc(Transient)))) typeq1;
Container (typeq1, (Hash (phone), Tree (name))) cont1;
Cursor (cont1, name > "Sm" && name < "Sn", ascending(name))
  curs1;                          // DiSTiL definitions

...
foreach(curs1)
  ... ref(curs1, name) ...
// DiSTiL operations mixed with C code
```

This example code shows a data structure organizing the same data in two ways: using a hash table (the `Hash` component in the above code) on the "phone" field and using a red-black tree (`Tree`) on the "name" field of the data records. The data are stored transiently in memory (`Transient`) and allocated dynamically on the heap (`Malloc`). The cursor shown in the example code is defined using a predicate on the "name" field. Therefore, DiSTiL will generate efficient code for all uses of this cursor by using the red-black tree structure. Should the cursor predicate or the data structure specification change in the future, DiSTiL will generate efficient code for the new requirements without needing to change the data structure traversal code.

2.3 Generation Scoping

Consider the generation of code using code template operators `quote` (`‘`) and `unquote` (`$`). The generator code may contain, for instance, the expression:

```
‘(if ((fp = fopen($filename, "r")) == NULL)...)
```

The question becomes, what are the bindings of free variables in this expression? What is the meaning of `fp` or even `fopen`? This is the scoping problem for generated code and it has been studied extensively in the *hygienic macros* literature [5, 8] for the case of pattern-based generation. Generation scoping [16] is a mechanism that gives a similar solution for the case of programmatic (i.e., not pattern-based) generation. The mechanism adds a new type, `Env`, and a new keyword, `environment`, that takes an expression of type `Env` as an argument. `environment` works in conjunction with `quote` and `unquote`—all generated code fragments under an `environment(e)` scope have their variable declarations inserted in `e` and their identifiers bound to variables in `e`. For example, the following code fragment demonstrates the generation scoping syntax:

```
Env e = new Env(parent);
...
environment(e)
  return `{ FILE *fp; ... }
...
environment(e)
  return `{ if ((fp = fopen($filename, "r")) == NULL)
              FatalError(FILE_OPEN_ERROR);
            ...
          }
```

In the above example, a new environment, `e`, gets created. Then, a variable, `fp`, is added to it, just by virtue of quoting its declaration under `e`. Any subsequent code generated under environment `e` will have its identifiers bound to variables in `e`. For instance, the `fp` identifier in the last quoted code fragment will be bound to the `fp` variable generated earlier. This binding is ensured, even if there are multiple `fp`s visible in the same lexical scope in the generated program, by consistently renaming the `fp`s in the generated code to a unique name.

The advantage of generation scoping is that it ensures that scoping is what the generator programmer intends. The scoping of an identifier in a generated fragment is not determined merely by its location in the final generated program. This allows the arbitrary mixing and matching of generated code fragments without worrying about name conflicts. The sophisticated part of the generation scoping implementation is that it needs to recognize what parts of the generated code correspond to binding instances (e.g., the declaration `FILE *fp`) and produce code that adds them in the right scoping environment maintained during generation. Environments can be organized hierarchically—in the above example, environment `e` is a child of environment `parent`. An identifier is looked up in the current environment, then this environment's parent, etc. Hierarchies of environments allow generation scoping to mimic a variety of scoping arrangements (e.g., lexical scoping but also ad hoc namespaces) in the generated program.

Generation scoping simplified significantly the implementation of DiSTiL. DiSTiL is a component-based generator—the generated code is produced by putting together a large numbers of smaller code fragments. Thus, the same code is often generated in the same lexical scope but with different intended identifier bindings. This was made significantly easier by generation scoping, as each generation-time component only needed to maintain a single environment, regardless of how the code of all components ended up being weaved together.

2.4 C++ Templates Work

Advanced work with C++ templates is often closely related to generators. C++ templates offer a Turing-complete compile-time sub-language that can be used to perform complex meta-programming [6]. In combination with the C++ syntactic extensibility features, like operator overloading, C++ templates offer an extensible language environment where many domain-specific constructs can be

added. In fact, several useful implementations of domain-specific languages [19], especially for scientific computing, have been implemented exclusively as C++ template libraries. These domain-specific languages/libraries perform complex compile-time optimizations, akin to those performed by a high-performance Fortran compiler.

Much of my work involves C++ templates although often these templates are not used for language extensibility. Specifically, I have proposed the concept of a *mixin layer* [13, 14, 17]. Mixin layers are large-scale components, implemented using a combination of inheritance and parameterization. A mixin layer contains multiple classes, all of which have a yet-unknown superclass. What makes mixin layers convenient is that all their component classes are simultaneously instantiated as soon as a mixin layer is composed with another layer. Mixin layers can have a C++ form as simple as the following:

```
template <class S> class T : public S {
  class I1: public S::I1 {...};
  class I2: public S::I2 {...};
  class I3: public S::I3 {...};
};
```

That is, a mixin layer in C++ is a class template, `T`, that inherits from its template parameter, `S`, while it contains nested classes that inherit from the corresponding nested classes of `S`. In this way, a mixin layer can inherit entire classes from other layers, while by composing layers (e.g., `T<A>`) the programmer can form inheritance hierarchies for a whole set of inter-related classes (like `T<A>::I1`, `T<A>::I2`, etc.).

My C++ templates work also includes FC++ [9, 10]: a library for functional programming in C++. FC++ offers much of the convenience of programming in Haskell without needing to extend the C++ language. Although the novelty and value of FC++ is mostly in its type system, the latest FC++ versions make use of C++ template meta-programming to also extend C++ with a sub-language for expressing lambdas and various monad-related syntactic conveniences (e.g., comprehensions).

2.5 GOTECH

The GOTECH system is a modular generator that transforms plain Java classes into Enterprise Java Beans—i.e., classes conforming to a complex specification (J2EE) for supporting distribution in a server-side setting. The purpose of the transformation is to make these classes accessible from remote machines, i.e., to turn local communication into distributed. In GOTECH, the programmer marks existing classes with unobtrusive annotations (inside Java comments). The annotations contain simple settings, such as:

```
/**
  *@ejb:bean name = "SimpleClass"
```

```
*          type = "stateless"
*     jndi-name = "ejb/test/simple"
*     semantics = "by-copy"
*/
```

From such information, the generator creates several pieces of Java code and meta-data conforming to the specifications for Enterprise Java Beans. These include remote and local interfaces, a deployment descriptor (meta-data describing how the code is to be deployed), etc. Furthermore, all clients of the annotated class are modified to now make remote calls to the new form of the class. The modifications are done elegantly by producing code in the AspectJ language [7] that takes care of performing the necessary redirection. (AspectJ is a system that allows the separate specification of dispersed parts of an application's code and the subsequent composition of these parts with the main code body.) As a result, GOTECH is a modular, template-based generator that is easy to change, even for end-users.

2.6 Current Work

Some of my yet unpublished work is also relevant to generators research and language extensibility. In particular, the MAJ system is a meta-programming extension to Java for creating AspectJ programs. Specifically, MAJ adds to Java code template operators (a quote and unquote constructs) for structured (i.e., statically syntax-checked) generation of AspectJ code. The application value of MAJ is that it allows the easy creation of generators by just producing AspectJ programs that will transform existing applications. In general, I believe that using aspect-oriented technologies, like AspectJ, as a back-end for generators is a very promising approach.

Another one of my current projects is LC++. LC++ extends C++ with a full logic sub-language, closely resembling Prolog. The extension is implemented using template meta-programming, i.e., as a regular C++ library without needing any modifications to existing compilers.

3 What Are the Difficulties of Generator Research?

After describing my past and current work in generators, I take off my researcher hat and put on my GPCE'03 PC co-Chair hat. The GPCE conference (Generative Programming and Component Engineering) is trying to build a community of people interested in work in program generation.[1] Clearly, I advertise GPCE

[1] The name also includes "Component Engineering" which is a closely related area. The main element behind both generators and component engineering is the domain-specificity of the approaches. Some domains are simple enough that after the domain analysis is performed there is no need for a full-blown generator or language. Instead, an appropriate collection of components and a straightforward composition mechanism are a powerful enough implementation technique.

in this chapter, but at the same time I am hoping to outline the reasons why I think GPCE is important and why the generators community needs something like what I imagine GPCE becoming. All of my observations concern not just GPCE but the overall generators community. At the same time, all opinions are, of course, only mine and not necessarily shared among GPCE organizers. When I speak of "generators conferences", the ones I have in mind are ICSR (the International Conference on Software Reuse), GPCE, as well as the older events DSL (the Domain-Specific Languages conference), SAIG (the workshop on Semantics, Applications and Implementation of Program Generation), and GCSE (the Generative and Component-based Software Engineering conference). Of course, much of the work on generators also appears in broader conferences (e.g., in the Object-Oriented Programming or Automated Software Engineering communities) and my observations also apply to the generators-related parts of these venues.

Is there something wrong with the current state of research in generators or the current state of the generators scientific community? One can certainly argue that the community is alive and well, good research is being produced, and one cannot improve research quality with strategic decisions anyway. Nevertheless, I will argue that there are some symptoms that suggest we can do a lot better. These symptoms are to some extent shared by our neighboring and surrounding research communities—those of object-oriented and functional programming, as well as the broader Programming Languages and Software Engineering communities. I do believe, however, that some of the symptoms outlined below are unique and the ones that are shared are even more pronounced in the generators community. By necessity, my comments on community building are specific to current circumstances, but I hope that my comments on the inherent difficulties of generator research are general.

3.1 Symptoms

Relying on Other Communities. The generators community is derivative, to a larger extent than it should be. This means that we often expect technical solutions from the outside. The solution of fundamental problems that have direct impact to the generators community is often not even considered *our* responsibility. Perhaps this is an unfair characterization, but I often get the impression that we delegate important conceptual problems to the programming languages or systems communities. A lot of the interactions between members of the generators community and researchers in, say, programming languages (but outside generators) take the form of "what cool things did you guys invent lately that we can use in generators?".

Although I acknowledge that my symptom description is vague, I did want to state this separately from the next symptom, which may be a cause as well as an expression of this one.

Low Prestige. The generators community is lacking in research prestige. Specific indications include the lack of a prestigious, high-selectivity publication outlet,

and the corresponding shortage of people who have built a career entirely on doing generators work. Most of us prefer to publish our best results elsewhere. Of course, this is a chicken-and-egg problem: if the publication outlets are not prestigious, people will not submit their best papers. But if people do not submit their best papers, the publication outlets will remain non-prestigious. I don't know if GPCE will overcome this obstacle, but I think it has a chance to do so. GPCE integrates both people who are interested in generators applications (the Software Engineering side) and people who work on basic mechanisms for generators (the Programming Languages side). GPCE is a research conference: the results that it accepts have to be new contributions to knowledge and not straightforward applications of existing knowledge. Nevertheless, research can be both scientific research (i.e., research based on analysis) and engineering research (i.e., research based on synthesis). Both kinds of work are valuable to GPCE. The hope is that by bringing together the Software Engineering and the Programming Languages part of the community, the result will be a community with both strength in numbers but also a lively, intellectually stimulating exchange of ideas.

Poor Definition. Another symptom suggesting that the generators community could improve is the vagueness of the limits of the community. Most research communities are dynamic, but I get the impression that we are a little more dynamic than the average. The generators conferences get a lot of non-repeat customers. Often, papers are considered relevant under the reasoning of "XYZ could be thought of as a generator". Where do we draw the line? Every community has links to its neighboring communities, but at the same time a community is defined by the specific problems they are primarily interested in or the approach they take to solutions.

Limited Impact. A final, and possibly the most important, symptom of the problems of our community has to do with the impact we have had in practice. There are hundreds of nice domain-specific languages out there. There are several program generation tools. A well-known software engineering researcher recently told me (upon finding out I work on generators) "You guys begin to have impact! I have seen some very nice domain-specific languages for XYZ." I was embarrassed to admit that I could not in good conscience claim any credit. Can we really claim such an impact? Or were all these useful tools developed in complete isolation from research in software generators? If we do claim impact, is it for ideas, for tools, or for methodologies? In the end, when a new generator is designed, domain experts are indispensable. Does the same hold for research results?

One can argue that this symptom is shared with the programming languages research community. Nevertheless, I believe the problem is worse for us. The designers of new general purpose programming languages (e.g., Java, C#, Python, etc.) may not have known the latest related research for every aspect of their design. Nevertheless, they have at least read some of the research results in language design. In contrast, many people develop useful domain-specific languages without ever having read a single research paper on generators.

3.2 Causes?

If we agree that the above observations are indeed symptoms of a problem, then what is the cause of that problem? Put differently, what are the general obstacles to having a fruitful and impactful research community in domain-specific program generation? I believe there are two main causes of many of the difficulties encountered by the generators community.

1. Domain-specificity is inherent to generators: most of the value of a generator is in capturing the domain abstractions. But research is all about transmission of knowledge. If the value is domain-specific, what is there to transmit to others?
2. What is generators work anyway? There is no established common body of knowledge for the area of generators. Consequently, it is not clear what are the big research problems and what should be the next research goals.

In the next two sections, I try to discuss in more detail these two causes. By doing this, I also identify what I consider promising approaches to domain-specific program generation research.

4 Domain-Specificity

4.1 Lessons That Transcend Domains

In generators conferences, one finds several papers that tell a similarly-structured tale: "We made this wonderful generator for domain XYZ. We used these tools." Although this paper structure can certainly be very valuable, it often degenerates into a "here's what I did last summer" paper. A domain-specific implementation may be valuable to other domain experts, but the question is, what is the value to other generators researchers and developers who are not domain experts? Are the authors only providing an example of the success of generators but without offering any real research benefit to others? If so, isn't this really not a research community but a birds-of-a-feather gathering?

Indeed, I believe we need to be very vigilant in judging technical contributions according to the value they offer to other researchers. In doing this, we could establish some guidelines about what we expect to see in a good domain-specific paper. Do we want an explicit "lessons learned" section? Do we want authors to outline what part of their expertise is domain-independent? Do we want an analysis of the difficulties of the domain, in a form that will be useful to future generators' implementors for the same domain? I believe it is worth selecting a few good domain-specific papers and using them as examples of what we would like future authors to address.

4.2 Domain-Independent Research: Infrastructure

In my view, a very promising direction of generators research is the design and development of infrastructure: language abstractions and type system support,

transformation systems, notations for transformations, etc. A lot of generators research, both on the programming languages and the software engineering side, is concerned with generator/meta-programming infrastructure. Infrastructure is the domain-independent part of generators research. As such, it can be claimed to be conceptually general and important to the entire generators community, regardless of domain expertise. I believe that the generators community has the opportunity (and the obligation!) to develop infrastructure that will be essential for developing future generators. In this way, no generator author will be able to afford to be completely unaware of generators research.

Of course, the potential impact of infrastructure work has some boundaries. These are worth discussing because they will help focus our research. The margin of impact for infrastructure is small exactly because domain-specificity is so inherent in generators work—domain knowledge is the quintessence of a generator. I usually think of the range with potential for generator infrastructure as a narrow zone between the vast spaces of the *irrelevant* and the *trivial*. Infrastructure is irrelevant when the domain is important and its abstractions mature. For domain specific languages like Yacc, Perl, SQL, etc., it does not matter what infrastructure one uses for their generators. The value of the domain is so high, that even if one invests twice as much effort in building a generator, the "wasted" effort will be hardly noticed. Similarly, infrastructure is sometimes trivial. A lot of benefit has been obtained for some domains by mere use of text templates. Consider Frame Processors [2]—a trivial transformational infrastructure with significant practical applications. Frame Processors are like low-level lexical macros. A more mature meta-programming technology is certainly possible, but will it matter, when Frame Processors are sufficient for getting most of the benefit in their domain of application?

Despite the inherent limitations of research in generator infrastructure, I believe the potential is high. Although the range between the irrelevant and the trivial is narrow, it is not narrower than the research foci of many other communities. After all, scientific research is the deep study of narrow areas. If the required depth is reached, I am hopeful that practical applications will abound. For example, a convenient, readily available, and well-designed meta-programming infrastructure for a mainstream language is likely to be used by all generator developers using that language.

4.3 Promising Infrastructure Directions

To further examine the promising directions for having real impact on generator development, it is worth asking why generators fail. I would guess (without any statistical basis) that for every 100 generators created, about one will see any popularity. The reasons for failure, I claim, are usually the small benefit (the generator is just a minor convenience), extra dependency (programmers avoid the generator because it introduces an extra dependency), and bad fit of the generator (the code produced does not fit the development needs well). Of these three, "small benefit" is a constant that no amount of research can affect—it is inherent in the domain or the understanding of the domain concepts by

the generator writer. The other two reasons, however, are variables that good generator infrastructure can change. In other words, good infrastructure can result in more successful generators.

Given that our goal is to help generators impose fewer dependencies and fit better with the rest of a program, an interesting question is whether a generator should be regarded as a tool or as a language. To clarify the question, let's characterize the two views of a generator a little more precisely. Viewing a generator as a language means treating it as a closed system, where little or no inspection of the output is expected. Regarding a generator as a tool means to support a quick-and-dirty implementation and shift some responsibility to the user: sometimes the user will need to understand the generated code, ensure good fit with the rest of the application, and even maintain generated code.

The two viewpoints have different advantages and applicability ranges. For example, when the generator user is not a programmer, the only viable option is the generator-as-a-language viewpoint. The generator-as-a-language approach is high-risk, however: it requires buy-in by generator users because it adds the generator as a required link in the dependency chain. At the same time, it implies commitment to the specific capabilities supported by the generator. The interconnectivity and debugging issues are also not trivial. In summary, the generator-as-a-language approach can only be valuable in the case of well-developed generators for mature domains. Unfortunately, this case is almost always in the irrelevant range for generator infrastructure. Research on generator infrastructure will very rarely have any impact on generators that are developed as languages. If such a generator is successful, its preconditions for success are such that they make the choice of infrastructure be irrelevant.

Therefore, I believe the greatest current promise for generator research with impact is on infrastructure for generators that follow the generator-as-a-tool viewpoint. Of course, even this approach has its problems: infrastructure for such generators may be trivial—as, for instance, in the case of the "wizards" in Microsoft tools that generate code skeletons using simple text templates. Nonetheless, the "trivial" case is rare in practice. Most of the time a good generator-as-a-tool needs some sophistication—at the very least to the level of syntactic and simple semantic analysis.

How can good infrastructure help generators succeed? Recall that we want to *reduce dependencies on generator tools* and *increase the fit of generated code to existing code*. Based on these two requirements, I believe that a few good principles for a generator-as-a-tool are the following:

- Unobtrusive annotations: the domain-specific language constructs should be in a separate file or appear as comments in a regular source file. The source file should be independently compilable by an unaware compiler that will just ignore the domain-specific constructs. Although this is overly restrictive for many domains, when it is attainable it is an excellent property to strive for.
- Separate generated code: generated code should be cleanly separated using language-level encapsulation (e.g., classes or modules). A generator should

be a substitute for something the programmer feels they could have written by hand and does not pollute the rest of the application. The slight performance loss due to a compositional encapsulation mechanism should not be a concern. A generator-as-a-tool is foremostly a matter of high-level expressiveness and convenience for the programmer, not a way to apply very low-level optimizations, like inlining.
- Nice generated code: generated code should be well formatted, and natural (idiomatic) for the specific target language. This ensures maintainability.
- Openness and configurability: The generator itself should be written using standard tools and should even be changeable by its users! Code templates and pattern-based transformation languages are essential.

For instance, recall the DiSTiL generator that I mentioned in Section 2. DiSTiL is an extension to the C language. The DiSTiL keywords are obtrusive, however, and the DiSTiL generated code is weaved through the C code of the application for efficiency. I reproduce below the DiSTiL source code fragment shown earlier:

```
struct phonebook_record {...};   // C definition

typeq (phonebook_record, Hash(Tree(Malloc(Transient)))) typeq1;
Container (typeq1, (Hash (phone), Tree (name))) cont1;
Cursor (cont1, name > "Sm" && name < "Sn", ascending(name))
  curs1;                          // DiSTiL definitions

...
foreach(curs1)
  ... ref(curs1, name) ...
// DiSTiL operations mixed with C code
```

If I were to reimplement DiSTiL now, I would introduce all its definitions inside comments. All the generated code would use standard encapsulation features of the target language (e.g., classes in Java or C++) and instead of special traversal operations, like `foreach`, I would use the existing C++ STL idioms for collections. Essentially, DiSTiL would just be a generator for code that the programmer could otherwise write by hand. The dependency would be minimal—the programmer is always in control and can choose to discontinue use of the generator at any time. The source code could look like the following:

```
/*  *@Typeq Hash[phone](Tree[name](Malloc(Transient))) ContType1;
 *  *@Container ContType1 cont1;
 *  *@Cursor Curs1(cont1, name > "Sm" && name < "Sn",
 *                 ascending(name));
 */

struct phonebook_record {...};
for (ContType1::Curs1 curs = cont1.begin();
     curs != cont1.end();
```

```
      curs++)
  ... curs->name ...
// C++ STL collection idiom
```

Clearly, good infrastructure can be invaluable in such a generator implementation. There need to be mature tools for parsing both the original source language code and the structured comments. Creating language extensions inside comments can also be facilitated by a special tool. Debugging of generated code is a thorny practical issue that can be alleviated only with good infrastructure. Making the generator itself be an open tool that its users can configure is essential. This can only be done if the generator is based on a standard meta-programming infrastructure, instead of being a one-time result. All of the above tasks are research and not straightforward development: the right design of all these language tools is very much an open problem.

In short, generators research *can* have impact through infrastructure. Even if the infrastructure is not instantly widely adopted, by designing it correctly we can hope that the generators the *do* adopt our infrastructure end up being the successful ones.

5 What Is Generators Work Anyway?

The second cause of many of the difficulties of the generators research community is the lack of consensus on what is generators research. The community does not have a well-defined background—there is no set of papers or textbooks that we all agree everybody should read before they do research in this area. Of course, the limits of what is considered topical for our research community are inherently vague: some of our research can fit the general programming languages community, while some more fits fine in traditional software engineering outlets. Nevertheless, I do not think this is the real problem. Multiple research communities have significant overlap. In fact, the generators community does have a very distinct research identity. Generators researchers work on very specific problems (e.g., language extensibility, meta-programming, domain-specific language design). Therefore, I believe that the lack of consensus on the core of generators research is a truly solvable problem, which just requires a lot of community vigilance!

It is perhaps interesting to briefly discuss *why* generators researchers do research in this area. What are the elements that attract us to this particular style of research and why do we feel the problems that we work on justify our research existence? I think that there are two main groups of people doing research in this area. The first consists of the people who are enthusiastic about generators as an intellectual challenge. The well-known argument for translation techniques applies perfectly to generators research: "If you are doing Computer Science, you probably think computations are cool. If so, then what can be cooler than computing about computations?" There is a certain amount of excitement when

one observes a program creating another program. Many generators researchers still feel the magic when they come across a self-reproducing program. [2]

The second group of people who do research in generators are those who see a lot of practical potential for simplifying programming tasks through automation. These researchers are primarily interested in the software engineering benefits of generators. Indeed, in my experience, the potential for automating software development is responsible for a steady supply of graduate students who aspire to make significant contributions to generators research. Most of them are quickly discouraged when they realize that automation of programming tasks is a well-studied, but extremely hard problem. Yet others join the ranks of generators researchers for good.

With this understanding of "who we are", I believe we should try to establish a consistent identity as a research area. Again, this requires vigilance. But we can and should have some agreement across the community on issues like:

- What is the standard background in generators research? (This book largely tries to address this question.) What papers or books should most people be aware of?
- What are the elements of a good paper in the area? In the case of domain-specific work, how can we make sure it is valuable to non-domain-experts?
- What are some big open problems in generators? What are promising directions for high impact? (Batory's "The Road to Utopia: A Future for Generative Programming", in this volume, contains more discussion along these lines.)

Finally, we do need a prestigious, reliable publication outlet. Nothing defines a community better than a flagship publication channel. This will allow researchers to be permanently associated with the area, instead of seeking to publish strong results elsewhere.

References

1. William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi, "Transformation in Intentional Programming", in Prem Devanbu and Jeffrey Poulin (eds.), proc. *5th International Conference on Software Reuse (ICSR '98)*, 114-123, IEEE CS Press, 1998.
2. Paul Basset, *Framing Software Reuse: Lessons from the Real World*, Yourdon Press, Prentice Hall, 1997.
3. Don Batory, Bernie Lofaso, and Yannis Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages", in Prem Devanbu and Jeffrey Poulin (eds.), proc. *5th International Conference on Software Reuse (ICSR '98)*, 143-155, IEEE CS Press, 1998.
4. Don Batory, Gang Chen, Eric Robertson, and Tao Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software engineering*, 26(5), 441-452, May 2000.

[2] E.g., `((lambda (x) (list x (list 'quote x))) '(lambda (x) (list x (list 'quote x))))` in Lisp.

5. William Clinger and Jonathan Rees, "Macros that work", *Eighteenth Annual ACM Symposium on Principles of Programming Languages (PoPL '91)*, 155-162, ACM Press, 1991.
6. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Techniques, and Applications*, Addison-Wesley, 2000.
7. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An Overview of AspectJ", in Jørgen Lindskov Knudsen (ed.), proc. *15th European Conference on Object-Oriented Programming (ECOOP '01)*. In Lecture Notes in Computer Science (LNCS) 2072, Springer-Verlag, 2001.
8. Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba, "Hygienic macro expansion", in Richard P. Gabriel (ed.), proc. *ACM SIGPLAN '86 Conference on Lisp and Functional Programming*, 151-161, ACM Press, 1986.
9. Brian McNamara and Yannis Smaragdakis, "Functional programming in C++", in Philip Wadler (ed.), proc. *ACM SIGPLAN 5th International Conference on Functional Programming (ICFP '00)*, 118-129, ACM Press, 2000.
10. Brian McNamara and Yannis Smaragdakis, "Functional Programming with the FC++ Library", *Journal of Functional Programming (JFP)*, Cambridge University Press, to appear.
11. Charles Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference*, 1995.
12. Yannis Smaragdakis and Don Batory, "DiSTiL: a Transformation Library for Data Structures", in J. Christopher Ramming (ed.), *Conference on Domain-Specific Languages (DSL '97)*, 257-269, Usenix Association, 1997.
13. Yannis Smaragdakis and Don Batory, "Implementing Reusable Object-Oriented Components", in Prem Devanbu and Jeffrey Poulin (eds.), proc. *5th International Conference on Software Reuse (ICSR '98)*, 36-45, IEEE CS Press, 1998.
14. Yannis Smaragdakis and Don Batory, "Implementing Layered Designs with Mixin Layers", in Eric Jul (ed.), *12th European Conference on Object-Oriented Programming (ECOOP '98)*, 550-570. In Lecture Notes in Computer Science (LNCS) 1445, Springer-Verlag, 1998.
15. Yannis Smaragdakis and Don Batory, "Application Generators", in J.G. Webster (ed.), *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons 2000.
16. Yannis Smaragdakis and Don Batory, "Scoping Constructs for Program Generators", in Krzysztof Czarnecki and Ulrich Eisenecker (eds.), *First Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, 65-78. In Lecture Notes in Computer Science (LNCS) 1799, Springer-Verlag, 1999.
17. Yannis Smaragdakis and Don Batory, "Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM Trans. Softw. Eng. and Methodology (TOSEM)*, 11(2), 215-255, April 2002.
18. Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis and Marc Fleury, "Aspectizing Server-Side Distribution", in proc. *18th IEEE Automated Software Engineering Conference (ASE'03)*, 130-141, IEEE CS Press, 2003.
19. Todd Veldhuizen, "Scientific Computing in Object-Oriented Languages web page", `http://www.oonumerics.org/`