

Implementing Domain-Specific Languages

Thesis Proposal

Yannis Smaragdakis
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
smaragd@cs.utexas.edu

Abstract

Automatic program generation usually focuses on well-defined and well-understood domains. Domain knowledge can be encoded in specific abstractions and compactly expressed in domain-specific programming languages. There are many ways in which such languages can be implemented: as stand-alone compilers, macro libraries, object libraries, etc. Compiler implementations are usually called *software generators*. Building generators, just like building any compiler, can be a difficult task. One way to facilitate it is to concentrate on transforming domain-specific abstractions into code in a conventional high-level language. *Generation scoping* is a programming facility to support such transformations. It solves the usual scoping problems present in generating programs in high-level languages and provides a convenient way to specify generated programs.

Generation scoping has been implemented as a tool in IP, a transformation system under development at Microsoft Research. It was used to build the DiSTiL software generator for container data structures. This proposal outlines the current features of generation scoping, their use in the implementation of DiSTiL, and a plan for further work on tools and techniques for implementing domain-specific languages. The Appendix describes generation scoping and its potential applications in greater detail.

1 Introduction

Domain-specific languages represent a way of capturing knowledge from a particular field in order to facilitate the development of related applications. Domain-specific primitives provide a higher level of abstraction than regular programming mechanisms (see, for instance, [1]). This contributes to program simplicity, and, consequently, correctness and performance enhancement.

The general problem that domain-specific languages address is one of (or absence of) *software reuse*. Conventional programming languages are not well suited to designing reusable components. Common library components lack flexibility and cannot adapt to the needs of their users. Such components are either overly specific (in which case they are very rarely reused) or very broad and inefficient (since they cannot take advantage of the particular characteristics of a given problem) [9]. Specializing a component manually has been demonstrated to be inefficient for everything but trivial changes [24]. The solution is to make components *intelligent*: they should automatically adapt to their current intended use. Intended use information, however, is not available in regular (low-level) code. Therefore, intelligent components commonly interface with their users through a domain-specific (higher-level) language. The primitives of this language are such as to make explicit the component user's intentions. Based on this intention information, the component can specialize and optimize itself to produce an efficient implementation.

Domain-specific languages can be implemented in a variety of ways. They can be compilers, macro-libraries, template-libraries, object libraries, etc. The Microsoft Foundation Class library, for instance, implements a domain-specific language as a collection of macros and objects. Using naive macros (as MFC does) to write an entire language (often with complicated syntax) can be dangerous. More sophisticated implementations, such as those that rely on optimizing program transformations, would offer better results. A full optimizing compiler can encode a more expressive language, adapt to a variety of uses, even perform domain-specific error checking i.e., detect errors at a higher level of abstraction.

Our research is concerned with the development of tools and techniques to facilitate the implementation of domain-specific languages. This implementation task can be complex and is currently not very well understood. Our thesis is that tools and methodologies can be developed to make it easier. We have begun our work with the common case of writing a compiler for a domain-specific language. Such compilers are called *software generators*. Writing a powerful generator is not easy. Experience in building multiple generators ([1], [2]) suggests that it usually requires an investment of 3-5 man years distributed over a wide time frame, and that only 30% of the effort is actually spent on specifying the way the source code is to be produced or transformed; the vast majority of time is focused on infrastructure development (e.g., tools for code fragment specification, parameterization, and synthesis). This overhead severely limits the possibilities of serious experimentation with generators and renders the generation approach to software development virtually impractical under realistic time and funding constraints.

Although all of our work to this point has been on generators, these are not the only possible implementations of domain-specific languages. Often, we can write powerful collections of domain-specific components using existing programming language constructs. An interesting problem is that of analyzing such collections and constructs. Its importance is twofold: We need to know if there are restrictions of component patterns that can be encoded using existing language mechanisms (inheritance and parameterization appear the most promising). Also, we would like to specify extensions to existing languages so that they can handle a larger class of reusable components.

2 Generation Scoping

The technique of *program generation* consists of creating and assembling code fragments expressed in a high-level language. The majority of software generators are implemented as compilers employing program generation. Creating high-level language programs, however, has its dangers. The “meaning” (*variable binding*) of symbolic identifiers in such programs is usually determined by their position (a policy commonly called *static scoping*). This is not desirable when the program is generated dynamically: a program fragment may be used in places not originally expected (we will present examples in Section 2.1 and in the Appendix [27]). As a result, identifiers in this fragment may be bound to the wrong variables.

Such problems are not specific to software generators but appear in all *meta-programming* applications (programs that write other programs). Meta-programming systems are notoriously ambiguous about the bindings of free identifiers in generated program fragments. The problem has been studied extensively in the context of macro expansion and several solutions have been proposed. *Hygienic macro expansion* [19] solves binding problems for code passed as a macro parameter. Similarly, the technique of *syntactic closures* [5] has been used to resolve the bindings of identifiers in code produced by expanding a macro. Both facilities have been combined to produce *hygienic, lexically scoped* macros [11]. These form the basis of several implementations of macro facilities (see, for instance, [10], [15]) for the Scheme programming language [12].

These solutions, however, are specific to macro expansion mechanisms. To address identifier binding problems in the general context of program generation, we have developed *generation scoping*. Generation

scoping is a programming language facility for conveniently specifying generated program fragments and bindings for their free identifiers. Additionally, it allows the user to easily express generated code specifications in the generator and isolate changes to them. Both these aspects of generation scoping contribute to the simplification of the generator programmer’s task.

In the next sections we give a short, high-level description of the generation scoping facility and its applications. The reader should refer to the Appendix for a more detailed treatment of these issues.

2.1 Background: the Binding Problem

We will use two operators for specifying code templates: `quote` (`'`) and `unquote` (`$`). `quote` designates the beginning of a code template and `unquote` escapes from it to evaluate a code generating expression. Thus, the two operators are similar to the LISP “backquote” and “comma” macro pair or the Scheme `quasiquote` and `comma` primitives. However, an important difference is that `quote` and `unquote` are used to represent expressions in the *language of the generated code* (which might be very different from the *language of the generator code*). So it is perfectly reasonable to write:

```
code = '(+ x 5);
```

(1)

if we have a `quote` operator in an extension of the C language that generates LISP code.

While this distinction seems elementary, in fact it is quite important. Generation scoping is related to hygienic macro expansion and syntactic closures, which are macro expansion techniques used in extensible programming languages. A hallmark of extensible languages is the blurring of the distinction between generator code, generated code, and their namespaces. This blurring is called *reflection*. Generators are typically *not* reflective. `'5` is not the same as `5` and does not evaluate to `5`. That is, `'5` is the code representation of number `5` in the language of the target program (e.g., a LISP list in (1)) while `5` is a constant in the language of the generator program (e.g., a C integer). *Thus, the separation of generator and target program namespaces and languages is the primary distinction between program generation and macro expansion (which can be viewed as a special reflective case).*

We can use code template operators in software generators to manipulate code as data. Code fragments are manufactured and composed to form a complete program. Fragments, however, are created in isolation from each other, thereby making it hard to determine the binding of identifiers to variable instances. Consider the following function:

```
CODE fun(CODE x, CODE y) {
    return '{ int temp = $y;
           printf("%d", temp != 0 ? 0 : $x/temp); } }
```

(2)

`fun` produces a code fragment from two input fragments (`x` and `y`). The idea of `fun`’s code fragment is to evaluate the expression in `y` once, and refer to this result (stored in `temp`) multiple times. Look what happens when `fun` is called by `outer`:

```
CODE outer() {
    return '{ int temp = 4;
           $fun( '6*temp, 'temp ); } }
```

(3)

The generated code fragment is:

```
{ int temp = 4;
  { int temp = temp;
```

```
printf("%d", temp != 0 ? 0 : 6*temp / temp ); } } (4)
```

The result is certainly not what the author of (2) or (3) expected and, furthermore, is clearly wrong (since it uses an undefined value to initialize the inner `temp` variable). The problem is that two different variables have the same identifier, and thus cannot be correctly distinguished. We call this the *binding problem* for generated identifiers.

As we saw with the `quote` and `unquote` operators, their origins can be traced back to extensible languages (mainly variants of LISP). Similarly, the binding problem has been extensively studied in the context of macro expansion. A full solution appeared in the form of hygienic, lexically scoped macros (HLSM, in brief). This solution, however, does not translate to program generation. Recall that macro expansion is essentially program generation where the generator program coincides with the generated program. In general program generation there is no way (as in HLSM) to use the lexical environment where a transformation is specified to guide the bindings of the identifiers it generates. The generator and target lexical environments are totally independent and they may even be in different programming languages.

Another consequence of the separation of generator and target program namespaces is that, generally, we are unable to generate new “generator code” on-the-fly. The reason is that generators usually cannot be extended with new transformations — often they are even compiled into binary format for reasons of source-code hiding and efficiency. This is in contrast to macro systems, where it is common to have macros that expand into the definitions of other macros (since the generator coincides with the target program, specifying new transformations is as easy as generating code). This feature is heavily used in powerful hygienic macro mechanisms in conjunction with primitives that break the hygiene.

2.2 Generation Scoping Operators

In generation scoping the binding problem is solved using the idea of *generation environments*. These are specified through the `environment` construct, a linguistic extension to the programming language of the generator. Its syntax is:

```
environment (<environment-id>) <statement>;
```

If `<statement>` contains a code template, the code is said to be generated in the generation environment identified by the value of the expression `<environment-id>`. Variables declared in code generated in a certain generation environment are only visible to other code fragments generated in the same environment. This way an environment can be viewed as a collection of variables that have the same scope.

To solve the binding problem presented above we can rewrite the code of (2) and (3) as follows:

```
Environment E = new Environment( ); // global variables
Environment F = new Environment( );

CODE fun(CODE x, CODE y) {
    environment(E)
    return `{ int temp = $y;
              printf("%d", temp != 0 ? 0 : $x/temp); } }
CODE outer( ) {
    environment(F)
    return `{ int temp = 4;
              $fun( `6*temp, `temp ); } } (5)
```

Since E and F are different environments (scopes), local variables in the fragment generated by `fun` are hidden from the code generated by `outer`. The result could be visualized easily if we use a “name mangling” policy: all identifiers belonging to the same environment will have a common suffix. Thus, if we assign “mangle” number 0 to E and 1 to F above the code generated will be:

```
{ int temp_0 = 4;
  { int temp_1 = temp_0;
    printf("%d", temp_1 != 0 ? 0 : 6*temp_0 / temp_1 ); } }
```

 (6)

The application of generation environments is not restricted to solving binding problems for generated identifiers. Generation environments can be organized into arbitrary directed acyclic graphs (DAGs) through the `SetParent` operator. `SetParent` can be called with two generation environment arguments (a child and a parent). All parent bindings then become visible to the child. To bind the identifiers of a quoted code fragment we begin at the current generation environment and recursively visit all its parents looking for variables with the same name.

This capability can be straightforwardly used to group related variables together and avoid multiple environment qualifications. Consider the following code fragment:

```
environment(Curs)
*Remove_Code = `{ ... if (container->first == cursor.obj)
                  container->first = cursor->next; ...};`
```

 (7)

(7) has been taken from the DiSTiL software generator and shows part of the code that deletes an element of an ordered linked list. The code template appears oblivious to the fact that there may be more than one `container` and `cursor` variables in the target program, more than one `first` fields in the `container` type, etc. The generation environment for this template (identified by the value of `Curs`) is used to bind these identifiers to the appropriate declarations. In this case, the bindings for `container`, `cursor`, `first`, etc. are not part of the current environment (`Curs`) but each one of them belongs to an environment reachable by `Curs` by following its parent links. The paper “Scoping Constructs for Program Generators” in the Appendix, has a more thorough explanation and additional examples.

2.3 Implementation and Application

Generation scoping was implemented as part of *IP (Intentional Programming)* [25], a general-purpose transformation system under development by Microsoft Research. It was subsequently used to build the DiSTiL software generator as a domain-specific extension to IP. DiSTiL is a generator for container data structures that evolved from P2 [1] and follows the GenVoca design paradigm. Its specification language is an extension of C with declarative data structure primitives. These allow the user to express the data structure specifics at a high level using a composition of basic components and retrieval predicates.

DiSTiL is an example of a sophisticated domain-specific language implementation. In about 10,000 lines of source code it specifies elementary data structure operations on various types of linked lists, arrays, red-black trees, hash tables, as well as a number of components affecting different aspects of a data structure (for instance, allocation, storage, run-time bound checking, etc.). All components are *factored* [9] and can be used in arbitrary combinations (compositions). This way, for instance, we can have a data structure consisting of a hash table together with a doubly linked list, with run-time bounds checking enabled and stored persistently. DiSTiL can validate a composition and the operations performed on it to confirm that they are meaningful (not all components can co-exist and not all operations are supported on all compositions). Additionally, data structure operations are optimized according to the structure on which they are applied. The most important of such optimizations is the selection of a retrieval structure. This identifies the best primitive data structure (for instance, binary tree or linked list) to use for accessing a certain set of data

structure elements.

In the implementation of DiSTiL we found generation scoping to be invaluable. It offered convenient ways to express generated code fragments and dependencies among them. To give some perspective to the value of generation scoping, we note that the P2 software generator, which inspired the creation of DiSTiL, has a similar facility (XP) for generating code. XP, however, is written specifically for the domain of data structures; it had specific keywords for `cursor` and `container`, among many other data-structure-specific concepts. While XP is a specialized language for P2 code generation, a generation scoping facility is domain-independent, and is specialized only for its target language (for instance, we only need one such mechanism for all programs generating C code).

3 Proposed Work

The design of DiSTiL and the generation scoping infrastructure has contributed to our understanding of how to implement a (GenVoca) software generator using a general-purpose transformation system. Our experience will be described in a forthcoming paper. Note that this work was entirely in the area of software generators (i.e., compilers for domain-specific languages). A different research area explores the kind of reusable components that can be expressed using common language mechanisms.

With the above in mind, we propose to work on two different (but related) research directions:

- *Extensions and applications of generation scoping:* Generation scoping and software generators have many important and practical applications. For this reason, we propose to continue work on generation scoping. DiSTiL presently only implements components with a single “realm” parameter. We would like to examine how P2 components with multiple parameters (like `replicate`) could be implemented using generation scoping. The main feature of such components is that they introduce new data structure types (for instance, containers and cursors) in the course of transforming existing ones. (An example is `replicate`: a container and its contents are replicated). Building multi-parameter components in P2 has been horrendously difficult, leading to very obscure code. Being able to handle it elegantly using generation scoping in DiSTiL would offer another strong validation of the mechanism’s capabilities.

Additionally, we would like to consider the interaction of generation scoping and different namespaces in the target program. Right now generation scoping is totally orthogonal to target language scoping — e.g., variables from distinct namespaces may be part of the same generation environment. In some cases a mechanism combining target namespaces and arbitrary sub-environments inside them may make sense.

Extensible languages, in general, are a large part of our ongoing research. There are interesting topics associated with the implementation of *Jakarta*: an extensible version of Java that is targeted to be a language for writing software generators. Jakarta will need to incorporate some kind of hygienic program generation (if not the full generation scoping capabilities). We expect to participate actively in the design of related parts of Jakarta.

- *Existing language support:* We would like to examine the common language facilities that can be used to implement domain-specific languages. This is an issue of particular practical interest that has attracted many researchers (see, for instance, [28], [29]). Implementing collections of powerful components using only standard language features can be extremely valuable. The generality and reusability of such components is hard to match — although the sophistication of their composition mechanism is, expectedly, limited. One of the best known outcomes of this work is the C++ Standard Template Library (STL). STL is a language for data structures implemented using entirely existing language mechanisms (C++ templates).

We expect that our work will reveal the potential of several combinations of language mechanisms for implementing interesting components. Preliminary results seem to indicate that with the combination of inheritance and parameterization (templates) we can get results similar to those obtained through a compositional GenVoca approach (as in P++ [26]), but may have to suffer some performance overhead. Our particular implementation method can also be viewed as a cleaner alternative of the work in [29]. This concentrated on straightforwardly implementing role-based designs using existing language facilities (C++ templates and inheritance).

Hopefully, our results will also help clarify the relationship between several design and implementation models. For instance, we feel that the paradigms of *parameterized programming* [13] and role based design (see, for instance, [7]) are nearly identical and can be unified. Such work would be illuminating for the programming community. In general, we expect to create a “technology road map”: an evaluation of different ways to design and write components using existing language facilities. This should help answer recurring questions of the form “how is X different than Y?”, where X and Y are design or implementation methods (e.g., GenVoca, static Object Oriented programming, etc.), and “when should X be used instead of Y?”

A breakdown of the estimated time to complete these tasks would be as follows:

- *Design and Implementation* (language facilities, generation scoping and namespaces, program generation in Jakarta, DiSTiL paper): approximately six months.
- *Experimentation* (multiple parameter components, test of language facilities for performance/efficiency): approximately three months.
- *Writing*: approximately four months.

4 References and Reading List

- [1] D. Batory and J. Thomas, “P2: A Lightweight DBMS Generator”. Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, June 1995.
- [2] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM Transactions on Software Engineering and Methodology*, October 1992.
- [3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”. *ACM SIGSOFT* 1993.
- [4] D. Batory, “Subjectivity and GenVoca Generators”. *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996.
- [5] A. Bawden and J. Rees, “Syntactic Closures”. In *Proceedings of the SIGPLAN ‘88 ACM Conference on Lisp and Functional Programming*, 86-95.
- [6] I. Baxter, “Design Maintenance Systems”. *CACM* April 1992, 73-89.
- [7] K. Beck and W. Cunningham, “A Laboratory for Teaching Object-Oriented Thinking”. In *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1-6.
- [8] T. Biggerstaff and A. Perlis (editors), *Software Reusability* (2 vols.), ACM Press / Addison-Wesley, 1989.
- [9] T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”. *Third International Conference on Software Reuse*, Rio de Janeiro, 1994.

- [10] S. P. Carl, “Syntactic Exposures — A Lexically-Scoped Macro Facility for Extensible Languages”. M.A. Thesis, University of Texas, 1996. Available through the Internet at `ftp://ftp.cs.utexas.edu/pub/garbage/carl-msthesis.ps`.
- [11] W. Clinger and J. Rees, “Macros that Work”. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991, 155-162.
- [12] W. Clinger and J. Rees (editors), “The Revised⁴ Report on the Algorithmic Language Scheme”. *Lisp Pointers IV(3)*, July-September 1991, 1-55.
- [13] J. Goguen, “Reusing and Interconnecting Software Components”. *Computer*. February 1986.
- [14] D. Gries and D. Volpano, “The Transform: a New Language Construct”. *Structured Programming*, 1992.
- [15] C. Hanson, “A Syntactic Closures Macro Facility”. *Lisp Pointers IV(4)*, Oct.-Dec. 1991, 9-16.
- [16] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”. *OOPSLA 1993*, 411-428.
- [17] I. Holland, “Specifying Reusable Components Using Contracts”. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, 1992, 287-308.
- [18] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, “A Software Engineering Experiment in Software Component Generation”. *Fifth International Conference on Software Engineering*, 1996.
- [19] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, “Hygienic Macro Expansion”. In *Proceedings of the SIGPLAN '86 ACM Conference on Lisp and Functional Programming*, 151-161.
- [20] J. Neighbors, “Draco: A Method for Engineering Reusable Software Components”. In *Software Reusability*, T.J. Biggerstaff and A. Perlis, eds, Addison-Wesley/ACM Press, 1989.
- [21] J. Neighbors, “Software Construction Using Components”. Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.
- [22] G. Novak, “Creation of Views for Reuse of Software with Different Data Representations”. *IEEE Transactions on Software Engineering*, December 1995, 993-1005.
- [23] G. Novak, F. Hill, M. Wan, and B. Sayrs, “Negotiated Interfaces for Software Reuse”, *IEEE Transactions on Software Engineering*, July 1992, 646-653.
- [24] R. Selby, “Empirically Analyzing Software Reuse in a Production Environment”. *Software Reuse — Emerging Technology*, editor Will Tracz, IEEE Computer Society, New York, 1988, 176-189.
- [25] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”. *NATO Science Committee Conference*, 1995.
- [26] V. Singhal, “A Programming Language for Writing Domain-Specific Software System Generators”. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, August 1996.
- [27] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. Technical Report TR-96-37, Department of Computer Sciences, University of Texas at Austin, December 1996.
- [28] A. Stepanov and M. Lee, “The Standard Template Library”. Incorporated in ANSI/ISO Standards Committee C++ Draft.
- [29] M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”. In *Proceedings of the 1996 Conference on Object-Oriented Programming Systems, Languages, and Applications*.
- [30] M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.