

# Scoping Constructs for Program Generators

Yannis Smaragdakis and Don Batory  
Department of Computer Sciences  
The University of Texas at Austin  
Austin, Texas 78712  
{smaragd,dsb}@cs.utexas.edu

## Abstract

*Program generation* is the process of generating code in a high-level language (e.g., C, C++, Java) to implement an abstract specification of a program. Generated programs are created by synthesizing and composing code fragments. Binding identifiers in generated code with their correct variable declarations has been the focus of a lot of research work in the context of macro-expansion (e.g., hygienic macro expansion and syntactic closures mechanisms). The common solutions include automatically maintaining identifier *environments*, which determine the legal bindings for an identifier.

In this paper we present *generation scoping*: an adaptation of hygienic macro-expansion techniques to general-purpose program generation. The conceptual novelty of generation scoping lies in making identifier environments first-class objects and organizing them explicitly into directed graphs. This approach yields significant benefits: We are able to express scoping relationships independently of the structure of the generated program. This way we get a stronger tool for detecting errors in the specification of generated code. Additionally, we are able to simplify the specification by employing powerful implicit qualification. Thus, generation scoping becomes a useful layer of infrastructure for implementing software generators: it both solves binding problems and makes code specification more convenient.

**Keywords:** generation scoping, software generators, program transformations, hygienic macro expansion, syntactic closures

## 1 Introduction

*Program generation* is the process of generating code in a high-level language (e.g., C, C++, Java) to implement an abstract specification of a program. Compilers for many modern programming languages rely on program generation. Prominent examples include *software generators* (compilers for domain-specific languages — e.g., see [4], [6], [13]), parallelizing compilers (where the emphasis is on transformations of high-level abstractions to parallel programs — e.g., see [12]) and translators from one high-level language to another. Macro expansion itself can be viewed as an example of program generation where the target program coincides with the generator.

We believe that program generation, and software generators in particular, will play a key role in future software development. There is already a heavy demand for language facilities to automate rote tasks (indicated, for instance, by the popularity of the Microsoft Foundation Class library). Such generators can contribute to program quality, simplicity, and performance by enabling domain-specific programming at a high level of abstraction.

Writing a generator, however, is not easy. Our experience in building multiple generators (see [1], [4], [16]) 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 code is to be generated; the vast majority of time is focused on infrastructure development (e.g., tools for code fragment specification, parameteriza-

tion, 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. The purpose of our work is to provide better tools for writing generators.

*Generation scoping* is a model aimed at facilitating program generation. Although conceived independently, many of the ideas it incorporates are similar to *syntactic closures* [5] and *hygienic macro expansion* [11] in extensible programming languages. This is not surprising since the same fundamental problem is being addressed: given a fragment of generated code, how can we determine the “meaning” of its identifiers? Normally, the meaning of an identifier (also called its *variable binding*) is determined by its position in the program and the language *scope rules*. In generated code, however, the position of a generated code segment in the target program is not known (typically) until the entire program has been assembled. On the other hand, the generator programmer usually has an intended meaning for generated identifiers and we want to help him/her express it. A central concept in generation scoping is that of a *generation environment* — a modified version of *syntactic environments* in the syntactic closures work. Generation environments are first-class objects — they can freely be assigned to variables and passed as arguments. The operations performed on them, however, are limited. The most important one is that of organizing them hierarchically, in much the same way as lexical scopes are hierarchically related by nesting. With this addition, generation environments do more than solve binding problems. They can be used to express arbitrary binding relationships in generated code, help view related variables as a unit, and restrict the variables that a generated code fragment can access.

In this paper, we explain the binding problem faced in program generation and introduce the generation scoping mechanism. We show how its primitives can be used to solve the binding problem and to render program generation more convenient. Furthermore, we discuss an application of the model in implementing DiSTiL: a software generator for data structures. Finally, we examine which ideas in macro expansion do and do not generalize to program generation, thus showing the differences between our mechanism and existing macro facilities.

## 2 Background

### 2.1 Conventions and Terminology

Generators make a clear distinction between the code that is to be executed (i.e., *the generator code*) and the code that is to be produced (i.e., *the generated code* or *target program*). Their *namespaces* (i.e., constructs that assign meaning to identifiers) are distinct. For instance, a variable  $x$  in generator code is quite different from a variable  $x$  in generated code.

We use two different operators to specify code templates: `quote (‘)` and `unquote ($)`. `quote` designates the beginning of a code template and `unquote` escapes from it to evaluate a code generating expression. For instance, the expression:

```
‘(sqrt($x) * ($x + 9))
```

has the code fragment “value”:

```
sqrt(5) * (5 + 9)
```

when  $x$  has the value `‘5`. Quote expressions hide the underlying representation of source code, which might be parameterized strings (e.g., like in the C preprocessor), generalized lists, parse trees, or abstract syntax trees.

The `quote` and `unquote` operators are similar to the LISP “backquote” and “comma” macro pair or the Scheme `quasiquote` and `unquote` primitives [9]. 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).*

In this paper, we use the convention of presenting quoted expressions (code templates) in boldface to make the distinction between generator code and generated code apparent. The language used in most of our examples (both for the generator and the target program) syntactically resembles C/C++ but we will try to avoid C/C++ specific constructs. Lastly, we use *identifier* to refer to any name representing a program variable. (Note: we will use the term *variable* to represent both program variables and method names). The association of an identifier with a variable is called an identifier *binding*. In the example below, identifier `k` has two different bindings depending on the scope of the corresponding variable declarations:

```
{ int k; ...
  void k() {...}; ...
}
```

The ellipsis (...) symbol denotes an unspecified code segment.

## 2.2 Problems in Program Generation

Generators manipulate code as data (e.g., abstract syntax trees). Code fragments are manufactured and composed to form a complete program. Fragments 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`<sup>1</sup> once, and refer to this result (stored in `temp`) multiple times<sup>2</sup>. The name `temp`, however, could conflict with a similarly named user variable. For instance, when `fun` is called by outer:

---

1. When it is clear that we refer to variables in the target program, we say “the value of `x`” instead of “the value of the expression in `x` at target program run-time”.

```
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. Correctly generated code would make this distinction:

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

 (5)

*Hygienic macro expansion (HME)* was invented to address this problem [11], where `fun` is viewed as a macro. The idea is that each code generation action (macro application) introduces its own namespace for its generated declarations. One possible implementation of HME would “mangle” variable names and produce the code of (5). All identifiers referring to variables introduced during the  $n$ -th generation action (that is, the  $n$ -th time that quoted code has been evaluated during generator execution) would have “mangle” number  $n$ .<sup>3</sup>

The above example is an instance of the *binding problem*: how to bind<sup>4</sup> identifiers to variables in generated code. The binding problem is central to program generation. It can cause errors that are hard to trace and that occur only in specific contexts. Function `fun` of (2) may only exist in binary form in a library. When a user writes `outer` (as in (3)), he/she will discover (belatedly) the unwanted bindings. This is a problem that we want to eliminate.

Binding problems can occur in many diverse circumstances. The example previously presented has to do with parameterizing code fragments but problems may occur even without parameterization. A common case is when identifiers in generated code are bound to the wrong variables just because they happened to be generated in the variables’ scope. One could imagine the fragment of (2) creating code in a program site where `printf` means something different than the standard C library function. In the context of macro expansion, syntactic closures were invented to solve this problem [5]. A *syntactic closure* is a generated code fragment that is “closed” in a *syntactic environment* — a construct that specifies identifier bindings.

The above discussion seems to indicate that all program generation should be performed using lexically scoped units, in analogy to call-by-name procedures. All variables generated in a unit should only be visi-

- 
2. The expression in  $x$  may not be evaluated at all (if the expression in  $y$  evaluates to 0). Hence, the semantics of this code segment are not preserved if we generate it in a function and call it (“by value”) with the expressions in  $x$  and  $y$  as arguments (since  $x$  will then be evaluated at least once). In a language like C that does not support call-by-name argument passing, the code will have to be generated in-line.
  3. Actually, the only requirement is that we assign a unique “mangle” number to each generation action.
  4. Note that “bind” here does not have the usual meaning of associating names with storage locations (naming) but the static meaning of associating names with variables (resolution). Since our discussion does not examine the run-time of generated code at all, no confusion can result.

ble inside it. This is the *hygiene condition* for program generation and, in practice, it is often too strict. Sometimes it is required that a newly generated variable is visible to code generated by other entities (e.g., macros). This constitutes *breaking the hygiene* of program generation. There is a variety of constructs that cannot be implemented as a single hygienic macro (i.e., without breaking the hygiene). For this reason, mechanisms for breaking the hygiene are available in practically every hygienic macro system (see, for instance, [7], which also presents cases where the macro facility can automatically detect that the hygiene needs to be broken).

A common case where the hygiene needs to be broken is when generating an iteration construct. Such a construct can introduce an index variable for the iteration. The index variable has to be visible to the parameters of the entity generating the iterator (e.g., the iteration body) but any other variables introduced must remain hidden. An example of implementing an iterator “macro” will be presented in Section 3.2.

Another case where non-hygienic generation is useful is that of coordinated transformations. Such transformations need to communicate information among them. To do so they must have access to common storage locations — i.e., variables. The scope of such variables extends outside the fragment in which they were generated. For example, consider a set of macros that add a record (structure) type to a programming language:

```
{ DefineRecord (FOO, { float field1; float field2; } );
  DefineInstance (FOO, fool);
  SetField (fool, field1, 6.0);
  printf("%f\n", AccessField(fool, field1)); }
```

 (6)

Code segment (6) uses three macros (`DefineRecord`, `DefineInstance`, `SetField`, `AccessField`) that cooperate. `DefineInstance` introduces variables that both `SetField` and `AccessField` reference. Therefore, it cannot be implemented as an isolated hygienic macro. This case is quite common: Many program generation activities are *global program transformations* — coordinated rewrites of entire programs. Any single rewrite action cannot be considered by itself but only as a part of a larger effect.

### 3 Generation Scoping

*Generation scoping* is a convenient facility for program generation. It emphasizes isolating unrelated generated code fragments but also makes it easy to specify complex dependencies among related ones. It helps solve the problems outlined in Section 2.2 and keeps code generating expressions as simple as possible.

#### 3.1 Generation Environments

A *generation environment* is a set of identifier-variable bindings. We say that a quoted expression is *evaluated* within a generation environment when its identifiers are interpreted using the bindings of that environment. Such evaluations are accomplished using the `environment` construct, a linguistic extension to the programming language of the generator. Its syntax is:

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

where `statement` contains one or more quoted expressions. Consider (7): a generation environment `E` is created and then used to interpret two quoted expressions, both of which generate a declaration of an integer variable:

```

E = new Environment;
...
environment(E) {
  x1 = `(int j);
  x2 = `(int k); }

```

(7)

Generating these declarations causes the variable names ( $j$  and  $k$ ) to be added to  $E$ 's set of bindings. Whenever a quoted expression is subsequently evaluated within environment  $E$  and references identifiers  $j$  or  $k$ , these identifiers will be bound to variables  $j$  and  $k$  defined above<sup>5</sup>.

Evaluating quoted expressions within an environment is equivalent to a consistent identifier renaming policy. Assume a unique “mangle” number is assigned to each generation environment. If that number is 0 for environment  $E$  then the variables appearing as  $j$  and  $k$  in a quoted expression will be identified by  $j_0$  and  $k_0$  in the target program. Thus, evaluating the expression:

```

environment(E)
  y = `(k + j);

```

(8)

assigns  $y$  the value “ $k_0 + j_0$ ”.

Because generation environments are (generator) run-time objects, we can define functions like:

```

CODE genfun(Environment A) {
  environment(A)
  return `{ int j, k; } }

```

Calling `genfun` with different environment arguments will produce distinct declaration instances:

```

E = new Environment;      // has mangle number 0
F = new Environment;     // has mangle number 1
genfun(E);               // generates "int j 0, k 0;"
genfun(F);               // generates "int j 1, k 1;"

```

(9)

In fact, during program generation, a single template is often used many times to produce different code fragments. We will see such examples in Section 4.1.

### 3.2 Problems in Program Generation Revisited

Generation environments easily solve the problems of Section 2.2. Suppose  $E$  and  $F$  are generation environments that are used specifically to rewrite examples (2) and (3):

```

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

```

---

5. Actually, bindings also depend on lexical visibility in the target program. So if we define another variable  $k$  (e.g., `(float k)`) in environment  $E$ , references to identifier  $k$  will be bound according to the scoping rules of the target language (e.g., to the variable defined in the innermost scope).

```

CODE outer( ) {
    Environment F = new Environment;
    environment(F)
    return `{ int temp = 4;
              $fun( `6*temp, `temp ); } }

```

(10)

Since `E` and `F` are different environments, local variables in the fragment generated by `fun` are hidden from the code generated by `outer`. If the mangle number for the environment in `E` is 0 and for that in `F` is 1, then code segment (5) is generated when `outer` is called. Similarly, we can protect against inadvertent bindings of generated identifiers. The reference to `printf` in the code generated by (10) does not depend on the meaning of `printf` in the lexical environment where this fragment will eventually be placed.

Furthermore, recall the case of iterator macros discussed in Section 2.2. The iteration variable needs to be visible to the macro body but any other variables should be hidden. This is easily handled using generation scoping. To see a concrete example consider a macro `ForAllInFile` iterating over the contents of a file. This macro will have to introduce local variables (for instance, the file pointer) in its expansion. These should remain hidden. The macro will also have to introduce a variable representing the current byte read from the file. This variable should remain free for reference inside the macro body. Since there is no (easy) automatic way to tell which variables should be in a private scope and which should be visible, such macros are often written by designating visible (or hidden) variables explicitly. With generation scoping this common case comes at no extra effort. Of course, generation scoping doesn't deal with macros but the analogous unit would be a small, self-contained program generation entity. Thus we could have the equivalent of `ForAllInFile` as a program generating function:

```

CODE CreateForAllInFile (CODE byte, CODE filename, CODE body)
{
    environment(new Environment)
    return `{
        FILE *fp;
        if ((fp = fopen($filename, "r")) == NULL)
            FatalError(FILE_OPEN_ERROR);
        while ( feof(fp) == FALSE)
        {
            int $byte = fgetc(fp);
            $body;
        }
    }
}

```

(11)

Notice how the desired behavior is straightforwardly obtained. The declaration of `fp` is local to the environment used by `CreateForAllInFile`. In contrast, the declaration of variable `byte` has the variable name escaped. Thus the supplied variable name has already been generated in some environment (the generation environment used by the caller of `CreateForAllInFile`). This environment is guaranteed to be different from the environment of `CreateForAllInFile` (since `CreateForAllInFile` creates a new one). This way the iteration variable (`byte`) can be visible to the code in `body` (assuming it has been generated in the same environment) but the new variable (`fp`) introduced by `CreateForAllInFile` is private. A possible call to `CreateForAllInFile` could have the form:

```

environment(E)
    CreateForAllInFile( `buffer, `"tempfile.txt", `putchar(buffer) );

```

This will generate code to print all characters in a text file. Note that the property of generation scoping

that guarantees that the above will work has to do with the representation of generated code. Code templates (i.e., the ``` operator) don't evaluate to simple text but to identifiers closed in an environment (e.g., pairs of environment identifiers and textual names). This way, once a code fragment has been generated it can be freely passed around and used verbatim without binding problems.

We can also use generation environments to solve the problem of (6) in Section 2.2. It is easy to relate generated code fragments that reference each other's variables. Operator `DefineInstance` of (6) can generate variables for the record fields in a generation environment  $G$ . If `SetField` and `AccessField` generate their code in the same environment  $G$ , they will be able to access the record field variables.

The above examples show that generation scoping takes the most straightforward (and low-level) route to specifying scoping of generated code. The entities that determine identifier bindings (generation environments) become first-class and can be directly manipulated by the generator programmer. A reasonable question is whether the same effect could be achieved by just keeping track of where a code fragment will be placed and taking advantage of the scoping constructs in the target language. Take, for instance, example (6) in Section 2.2. We could think of operator `DefineInstance` generating variables encapsulated together with accessor functions for them (for instance, using an object definition in the target language). Then `SetField` and `AccessField` could just expand to method calls for the right object. The object name would serve to distinguish between the results of multiple invocations of `DefineInstance`. Such an approach is not always possible, however. Consider the code in (11): There is no easy way in conventional programming languages to express the fact that variable `fp` should be hidden while the current byte read should be visible.

In fact, all work on hygienic program generation is exactly about overcoming such shortcomings of programming languages. Programming languages have traditionally associated allocation with scoping. Entities that are allocated together (for instance, automatic variables of a procedure, global variables, etc.) usually have the same scope. For instance, automatic variables of a C++ function have the same scope (the function body). Similarly, all data members of an object are visible in object method code. In other words, an object designates both a unit of allocation and a scope. This policy can occasionally lead into problems and many well-known programming language mechanisms were invented to address them. C++ *namespaces* were introduced to make explicit scoping possible for global variables. Call-by-name procedures are a way to introduce variables that have different scoping but retain the allocation properties of their invocation site. All the examples presented in this section represent cases where scoping and allocation units do not coincide. In fact, *generation scoping is all about dissociating scoping from the position where a generated fragment will appear in the target program.*

Making scoping independent from allocation is not always a good practice in programming language design. When programs are generated automatically, however, it makes perfect sense. This observation is exploited in the generation scoping work and should become clear in Section 3.5. Before that, we need to introduce some extra capabilities of generation scoping.

### 3.3 Hierarchical Organizations of Environments

Generation environments can be organized hierarchically. This is accomplished by using the `SetParent` operator with two generation environment arguments (a child and a parent). All parent bindings then become visible to the child. Like traditional scoping mechanisms, variable bindings of the child (inner environment) eclipse bindings of the parent (outer environment). Consider the following:

```

environment(OUTER)           // OUTER has mangle number 3
    x1 = `(int j, k);
environment(INNER)          // INNER has mangle number 4
    x2 = `(float j);
SetParent (INNER, OUTER);
environment(INNER)
    x3 = `(j + k);

```

(12)

The application of `SetParent` will make all bindings of `OUTER` be available to `INNER`. The value of `x1` after the execution of code segment (12) will appear in the target program as “`int j_3, k_3;`”. Similarly, the value of `x2` will appear as “`float j_4;`”, and the value of `x3` will be generated as “`j_4 + k_3`”. In other words, the binding for identifier `k` in `OUTER` will be visible but the binding for `j` in `INNER` will take precedence over that in `OUTER`.<sup>6</sup>

With `SetParent`, generation environments can express complicated interdependencies between generated code fragments. This is a powerful feature with many interesting applications. For instance, related variables cannot always exist together in one environment. Consider two variables `foo` and `bar` that have a many-to-one relationship: For each generated instance of `bar` there are many `foo`s that correspond to it. This precludes entering both `foo` and `bar` in a single generation environment. The variables, however, are related and code that refers to `foo` should be able to straightforwardly refer to the corresponding `bar`. This can easily be accomplished by making the environment of `bar` a parent for all the environments of the different instances of `foo`. The resulting code is greatly simplified: The choice of a `foo` immediately implies the choice of its corresponding `bar`. This will be further discussed in Section 3.5 and Section 4.1.

With `SetParent` we can emulate lexical scopes in the target program. This is most interesting when the target language has little or no support for lexical scoping. This is the case when, for instance, the generated code is C/C++ macros (which are not lexically scoped), assembly language programs, or abstract syntax trees. Generation environments can then represent “scopes” for generated identifiers.

### 3.4 The Alias Operator

Generation environments are mainly used to map identifiers to variables. With a simple extension, identifiers can be aliases for arbitrary code fragments. This is the purpose of operator `alias` which, in effect, turns identifiers into *implicit parameters* of code templates. Consider:

```

environment (D)
    alias ( x, `xArr[ipos] );

```

This makes identifier `x` act as an implicit parameter in code templates expanded in environment `D`. Whenever `x` is encountered in a template expansion, it is replaced by the expression `xArr[ipos]`. Hence, when the above `alias` is in effect, statement:

---

6. `SetParent` can actually be used to organize generation environments in arbitrary directed acyclic graphs (DAGs). The search for bindings of identifiers in a quoted code fragment begins at the current generation environment and recursively visits all its parents looking for variables with the same name. In case there is no unique match, our policy can be described as follows: we say that a binding *b1* hides another binding *b2* iff all parent chains from the current environment to the environment where *b2* was introduced contain the environment where *b1* was introduced. In the case of conflicting bindings, if there exists one that hides all others, it is applied. Otherwise the result is undefined. So care must be taken by generator programmers to ensure the correctness of environment DAGs.

```
environment (D)
    y1 = `(x + y);
```

(13)

is equivalent to the statement:

```
environment (D)
    y1 = `(xArr[ipos] + y);
```

Section 4.2 contains more examples of the use of `alias`.

### 3.5 Grouping Variables Together

Using generation scoping, we can express scoping relationships in generated code fragments regardless of the fragments' locations in the target program. In this section we will examine the benefits of this approach. The obvious one is the standard advantage of scoping: it limits the variables that a program fragment can access so that it becomes easier to understand what the fragment does. Additionally, references to variables outside the allowed scope (for instance, due to misspelling) are reported as errors.

Having a well-defined scope also allows us to employ powerful implicit qualification over all “visible” variables when generating code. Implicit qualification mechanisms let the user select a context and access all its private information without needing to explicitly qualify it. Examples are the Pascal `with` statement for record contexts and the C++ `using` statement for namespace contexts. Once we select a Pascal record through the `with` statement we can access all its fields as if they were visible variables without having to explicitly qualify them with the record name.

Let's have a look at an example of implicit qualification. The example discussed comes from C++ namespaces but similar comments apply to most other such mechanisms. Consider the following fragment of C++ code:

```
namespace RGB_COLOR {
    int red, green, blue;
}

namespace PHYS_POINT {
    using namespace RGB_COLOR;
    int x, y;
    BOOL marked;
}

namespace LOG_POINT {
    using namespace RGB_COLOR;
    float x, y, z;
    BOOL marked;
}
...
void Filter()
{
    using namespace PHYS_POINT;
    draw( x, y, red*C1 + green*C2 + blue*C3 / C4 ); //C1...C4 constants
```

(14)

The interesting part of this example is that the `PHYS_POINT` and `LOG_POINT` namespaces both have access

to the declarations of namespace `RGB_COLOR`. Thus, when a filter function accesses the declarations of namespace `PHYS_POINT`, it immediately gets access to namespace `RGB_COLOR`. This way it can use information pertaining to both the point coordinates ( $x$ ,  $y$ ) and the color of the region the point belongs to. This kind of implicit qualification is quite powerful: by specifying a single “context” (namespace in this case) we can automatically access declarations of a different (“parent”) context as well.

Note that the three namespaces of example (14) cannot be collapsed into one, for then we would have name conflicts between variables of `PHYS_POINT` and `LOG_POINT` (e.g., `marked`). We could access the necessary variables using explicit qualification, which in this example would yield the code fragment:

```
...
draw( PHYS_POINT::x, PHYS_POINT::y,
      RGB_COLOR::red*C1 + RGB_COLOR::green*C2 + RGB_COLOR::blue*C3 / C4 );
```

The advantage of implicit qualification lies in the simplicity of the resulting code (written in its most general form without worrying about binding problems). The disadvantage is that we are weakening the encapsulation provided by the namespaces mechanism (e.g., it is not easy to see what an identifier refers to).

Now imagine that the code fragment in (14) was generated automatically using a software generator that employs generation scoping. We could encode essentially the same relationships between generated declarations by using *generation environments* instead of namespaces. Each generation environment would correspond to one of the namespaces of program (14). We could make all the declarations of an environment visible to another by linking them together through the `SetParent` primitive (corresponding to the `using` statement inside namespaces `PHYS_POINT` and `LOG_POINT`). References to declarations in an environment are implicitly qualified when we generate code in this environment (i.e., the code template is in the scope of a generation scoping environment primitive).

But generation scoping can do much more than what shown in (14). Its “namespaces” (generation environments) can represent any collection of declarations. Variable scoping is completely dissociated from the position of a variable in the program — a single scope (generation environment) may contain global variables, automatic variables, fields of several classes, etc. Being able to implicitly qualify over all of them simplifies the specification of generated code inside the generator.

Since this is hard to show using the generated code specification itself, we will present the analogous concept as an extension of C++ namespaces. In other words, we will try to show what generation environments can do by walking on more familiar ground and presenting an extension of C++ namespaces that could do the same. Our example should demonstrate clearly the kinds of scoping relationships that can be expressed using generation scoping as well as how we can take advantage of them.

Namespaces in C++ can only include global declarations. Let’s extend the semantics of namespaces to include any kind of declarations (e.g., of class/structure fields, automatic variables, etc.). These *extended namespaces* are otherwise similar to the existing C++ namespaces. It is easy to simulate the effect of extended namespaces in an automatically generated program using generation scoping. Each namespace can be mapped straightforwardly to a generation environment and “`using namespace`” clauses correspond to `SetParent` statements. In the following example, one can think of extended namespaces and generation environments as equivalent notions (although generation environments are applicable only when the program in question is created automatically).

Now, consider a program fragment that manages complex data structures — inspired by the DiSTiL software generator, presented in Section 4. All data structures can be modeled as three distinct entities: a con-

tainer, elements, and iterators. The example presented here uses a simpler model, omitting iterators (pointers to elements can be used instead). One such data structure consists simultaneously of a doubly linked list and a singly linked list (and conceptually could be much more complicated). The doubly linked list can be ordered by a certain field of the records stored in the structure (e.g., “salary”), while the singly linked list could be organized by a different field (e.g., “name”). Using extended namespaces this program fragment can be written as:

```
// We need a namespace here to avoid conflicts with other data structures
namespace DS1 {
    struct ELEMENT {
        int salary;
        char *name;
        namespace DLIST1 {
            ELEMENT *next;
            ELEMENT *prev;
        } // These two fields have to do with the doubly-linked list
        namespace SLIST1 {
            ELEMENT *next;
        } // This field has to do with the singly-linked list
    };

    struct CONTAINER {
        namespace DLIST1 {
            ELEMENT *first;
            ELEMENT *last;
        } // This is the same namespace as the one used for the
        // doubly-linked list fields of ELEMENT!
        namespace SLIST1 {
            ELEMENT *first;
        } // This is the same namespace as the one used for the
        // singly-linked list field of ELEMENT!
    };
}

namespace DLIST1 {
    using namespace DS1;
    // Declarations of DS1 (e.g., CONTAINER) are visible from DLIST1
}

namespace SLIST1 {
    using namespace DS1;
    // Declarations of DS1 (e.g., CONTAINER) are visible from SLIST1
}

using namespace DLIST1; // Our implicit qualifier
void delete(CONTAINER cont, ELEMENT *el)
{
    if (el->next != NULL)
        el->next->prev = el->prev;
    if (el->prev != NULL)
        el->prev->next = el->next;
    if (cont->first == el)
        cont->first = el->next;
}
```

```

    if (cont->last == el)
        cont->last = el->prev;
// No explicit qualification needed for CONTAINER, ELEMENT,
// prev, next, first, last
}

```

(15)

What is interesting about this example is that in two cases (DLIST1, SLIST1) we have a single namespace containing some fields from one record type and some from another. The obvious advantage of grouping declarations this way is that it becomes clear which entities can access what data. For instance, `delete` is prevented from accessing any variables introduced in the SLIST1 namespace.

Also, both DLIST1 and SLIST1 have access to all the declarations of another namespace (DS1). As a result, the code of operation `delete` that needs access to all these variables is substantially simplified. The straightforward alternative would be to introduce intermediate structures to hold the fields for doubly linked and singly linked list elements and containers. Then, for instance, the CONTAINER class declaration would become:

```

struct CONTAINER {
    DLIST_CONT dlcont; // DLIST CONT is a new structure containing
                     // fields first and last
    SLIST_CONT slcont; // SLIST CONT contains first
};

```

Any such alternative, however, does not offer the option of implicitly qualifying over all doubly-linked list or singly-linked list declarations. Hence, the code for `delete` would become something like:

```

using namespace DS1; // Our implicit qualifier for CONTAINER, ELEMENT
void delete(CONTAINER cont, ELEMENT *el)
{
    if (el->dlelem.next != NULL)
        el->dlelem.next->prev = el->dlelem.prev;
    if (el->dlelem.prev != NULL)
        el->dlelem.prev->next = el->dlelem.next;
    if (cont->dlcont.first == el)
        cont->dlcont.first = el->dlelem.next;
    if (cont->dlcont.last == el)
        cont->dlcont.last = el->dlelem.prev;
}

```

(16)

This second implementation of `delete` is unnecessarily complicated by explicit qualification. This could become much worse in more complex namespace configurations. Such configurations are commonly encountered in automatically generated programs. Note that, in general, automatically generated programs have greater scoping requirements than hand-written programs. For instance, several slightly different copies of variable collections could exist (possibly generated by running the same generator code more than once with different parameters). Such collections are highly likely to introduce declarations with conflicting names and, thus, should be isolated in private contexts. These contexts may have complex interdependencies.

Powerful implicit qualification (such as that presented in (15)) is dangerous. Unqualified identifiers may be bound to different variables than those intended by the programmer. Also, it is hard to determine the binding of an identifier just by inspecting the code. For automatically generated programs, however, this danger is not present. Note that what happened in program (15) was that the declarations of fields `next` and `prev`

for the element class were grouped together with the declarations of fields `first` and `last` of the container class. This encodes a piece of higher level knowledge, namely that all these fields cooperate to implement a doubly linked list. From the perspective of the generator programmer, such relationships are easy to maintain. Although the four fields may not be close in the target program (e.g., `first` and `last` belong to a different class than `next` and `prev`), they will probably be together in the specification of a doubly linked list inside a generator (and so will be all other declarations that will be generated in the same generation environment/namespace). In the example of (15) this means that the code that generates the implementation of the `delete` operation will probably be close to the code generating the declarations of fields `next`, `prev`, `first`, and `last`. In other words, generation scoping allows scoping to be expressed relative to the position of code in the specification — not in the target program. This way the meaning of an identifier may not be apparent when inspecting the generated code but is more apparent in the generator components (which is what matters).

Observation shows that the structure of generator components is much different than the structure of generated programs (even though generated programs consist of component fragments). This is why it may be quite useful to dissociate scoping of generated code fragments from their location in the target program. Also, what constitutes a good practice from the perspective of the generated program is not necessarily as good from the perspective of the generator. In our particular example, extended namespaces may not be a good idea for general purpose C++ programming (e.g., implicitly qualifying their members can lead to code that is hard to understand). Nevertheless, emulating extended namespaces using syntactic environments in automatically generated programs does not suffer from the same problems.

## 4 Application: DiSTiL

Generation scoping was implemented as part of *IP (Intentional Programming)* [15], 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 GenVoca generator for container data structures that evolved from P2 [4] and follows the GenVoca [2] design paradigm. In the following sections, we examine some interesting applications of generation scoping in the implementation of DiSTiL.

### 4.1 Encapsulating Related Variables

GenVoca generators are a class of sophisticated software generators that synthesize high-performance, customized programs by composing pre-written components called *layers*. Each layer encapsulates the implementation of a primitive feature in a target domain. The P2 and DiSTiL generators are GenVoca generators for the domain of container data structures. Complex container data structures can be synthesized by composing primitive layers, where each layer implements either a primitive data structure (e.g., ordered linked lists, binary trees, etc.) or feature (sequential or random storage, logical element deletion, element encryption, etc.). Code for each operation is generated by having each layer manufacture a code fragment (that is specific to the operation whose code is being generated) and by assembling these fragments into a coherent algorithm.

P2 and DiSTiL data structures are modeled in much the same way as the data structures of Section 3.5. That is, each data structure consists of three distinct entities: a container, elements, and iterators (called *cursors*). The approach outlined in Section 3.5 is followed: Variables are grouped together into a common scope according to the entity to which they are related. For instance, all declarations related to the cursor part of a doubly linked list will belong in a single generation environment. These declarations, however, may vary considerably in their allocation behavior: some may be fields of more than one records, others can be global variables, types, automatic variables, etc. We will now examine in detail the kinds of genera-

tion environments used in DiSTiL.

In general, there is a many-to-one relationship between cursors and containers (i.e., there can be many cursors — each with a different retrieval predicate — per container). So using a single generation environment to encapsulate both cursor *and* container data members is not possible in DiSTiL. Instead, separate environments are defined for every cursor and container. The `ContGeneric` environment encapsulates element data members (because element types are in one-to-one correspondence with container types) and generic container-related variables (including the container identifier, which is subsequently aliased). The `CursGeneric` environment encapsulates generic cursor-related variables (including the cursor identifier, which also is subsequently aliased). By making `ContGeneric` a parent of `CursGeneric`, code for operations on containers (which do not need cursors) can be generated using the `ContGeneric` environment, while code for operations on cursors (which invariably reference container fields) is generated using the `CursGeneric` environment. Figure 1a depicts this relationship.

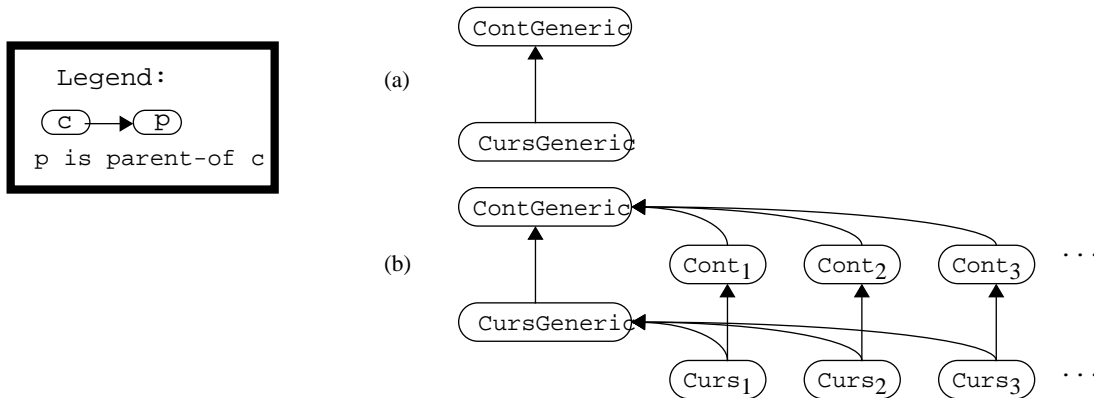


Figure 1: Hierarchical Organizations of Environments in DiSTiL

A hallmark of GenVoca layers is that they encapsulate refinements of multiple classes. Each DiSTiL layer refines cursor, container, and element types by adding layer-specific data members. The data members added to the container and element types by layer  $L_i$  are encapsulated by environment `Conti` which is a child of `ContGeneric`. Similarly, data members added by  $L_i$  to the cursor type are encapsulated by environment `Cursi` which is a child of both `CursGeneric` and `Conti` (because cursors of layer  $L_i$  reference layer-specific container-data members as well as layer-specific cursor data members). Figure 1b shows this hierarchical organization of environments.

An ordered doubly-linked list layer, for example, would refine elements by adding `next` and `prev` fields, and would refine containers by adding `first` and `last` fields. This refinement can be accomplished by a `RefineTypes()` method: `elem_type`, `cont_type`, and `curs_type` are code fragments that respectively define the set of variables (data members) in element, container, and cursor classes. When `RefineTypes()` is called with these code fragments as parameters, the `next`, `prev`, `first`, and `last` fields are added to the element and container types. As these fields are always used together, they are declared within a single environment `Cont` (which is equal to some `Conti` of Figure 1):

```

void RefineTypes( CODE *elem_type, CODE *cont_type, CODE *curs_type,
                 Environment Cont) {
    environment(Cont) {
        *elem_type = '\{ $(*elem_type); element *next, *prev; };
        *cont_type = '\{ $(*cont_type); element *first, *last; };
    }
}
  
```

It is common in a composition of GenVoca layers that a single layer appears multiple times. An example in P2 and DiSTiL would be linking elements of a container onto two (or more) distinct ordered lists, where each list has a unique sort key. Every list layer adds its own fields to the element and container types. Maintaining the distinction among these fields (so that the code for the  $j$ -th list will only reference its own fields  $next_j$ ,  $prev_j$ , etc.) is trivial using generation environments as organized in Figure 1. Each copy of the list layer will have its own generation environments  $Cont_j$  and  $Curs_j$ , and all code generated by that copy would always use these environment variables.

As in example (15) of Section 3.5, we use implicit qualification to simplify the code specification in the generator. For instance, the DiSTiL counterpart of the `delete` operation of (15) is the `Remove` method for ordered doubly-linked lists, appearing below. Let `Remove_Code` be the code that is to be generated for removing an element from a container. The `Remove` method for ordered doubly-linked lists adds its code (to unlink the element) when it is called (the code that actually deletes the element is added by another layer). Thus, given `Remove_Code` and the environment `Curs` (equal to some  $Curs_i$  of Figure 1), `Remove()` adds the unlinking code where the `next`, `prev`, etc. identifiers are bound to their correct variable definitions.

```
void Remove( CODE *Remove_Code, Environment Curs ) {
    environment(Curs) {
        *Remove_Code = `{ $(*Remove_Code);
            if (cursor->next != null)
                cursor->next->prev = cursor->prev;
            if (cursor->prev != null)
                cursor->prev->next = cursor->next;
            if (container->first == cursor.obj)
                container->first = cursor->next;
            if (container->last == cursor.obj)
                container->last = cursor->prev; };
    }
}
```

(17)

Note that, for instance, the bindings of identifiers `cursor`, `container`, and `next` in this template exist in three different generation environments: `container` is in `ContGeneric`, `cursor` in `CursGeneric`, and `next` in  $Cont_i$ . Nevertheless, all of them can be accessed from environment `Curs` (following its parent links), so this is the only environment that needs to be specified.

## 4.2 Aliases in DiSTiL

A very powerful use of `alias` is to replace explicit parameters (using the `$` operator) with implicit parameters<sup>7</sup>. Let's revisit (17), a portion of which is replicated below:

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

This code fragment defines a general relationship between cursors and containers when deleting elements from an ordered list. The identifiers `cursor` and `container` need not be restricted to reference particular cursor and container variables; rather the `cursor` identifier should be able to represent *any* expression that

7. Obviously the effect of `alias` could be simulated by creating macros in the target document but this seems gratuitous for simple expressions. It also relies on the macro processor of the target language which may be very primitive (like the C/C++ pre-processor, which is token-based and observes no scoping rules).

evaluates to a cursor, and the `container` identifier should be able to represent *any* expression that evaluates to the container referenced by that cursor. This can be accomplished by using aliases. The `Curs` and `Cont` environments below bind `cursor` and `container` to their respective expressions:

```
environment (Cont)
  alias( container, `(Container_array[j]) );
environment (Curs) {
  alias( cursor, `(Cursor_array[j]) );
```

When the `Remove()` method of (17) is called (given a null code fragment as input):

```
environment(Curs) {
  Remove_Code = `{}`; // null code fragment
  Remove( Remove_Code, Curs ); // add on unlinking statements
}
```

The value of `Remove_Code` will be:

```
...   if (Container_array[j]->first == Cursor_array[j].obj)
       Container_array[j]->first = Cursor_array[j]->next; ...
```

In building multiple generators, we have found that highly parameterized code fragments, such as (17), are very common. Creating implicit parameters, which is the primary use of `alias`, greatly simplifies the specification of generated code fragments. Such fragments are *substantially* easier to write, to understand, to debug, and to maintain than their explicitly parameterized (i.e., using the `$` operator) counterparts.

Also, in using the `alias` construct, we found that it makes specifications of quoted expressions more flexible. A typical scenario that we encountered in the development of DiSTiL was that an identifier `x` referred to a single variable in the majority of code templates. At some point later in DiSTiL's development, we discovered an exception where `x` needed to represent an entire expression. Instead of rewriting all existing quoted expressions to treat `x` as an explicit parameter, we `aliased` it to the expression in the environment of the special case.

### 4.3 Experience

The P2 generator, which was the inspiration of DiSTiL, had a similar facility (`xP`) to allow writing data structure code in a form as simple as that of (17). `xP` was written specifically for the domain of data structures. Its keywords include `Cursor` and `Container`, and it was aware of relationships between such entities (for instance, it knew that there is one container for each cursor). Generation scoping is a significant advance for generators like P2. It provides a general-purpose, domain-independent way to express generated code fragments and dependencies among them conveniently. The use of generation scoping made DiSTiL *much* easier to write.

## 5 Comparison

Generation environments are similar to syntactic environments in the syntactic closures work. There are, however, significant differences: Syntactic environments can only represent the set of variables that are lexically visible at a specific point in a program<sup>8</sup>. In contrast, generation environments can be arbitrary col-

---

8. In the original syntactic closures work [5] this point was almost always the site of the macro call. Later, syntactic environments were used to represent macro definition sites, as well (see, for instance, [10]).

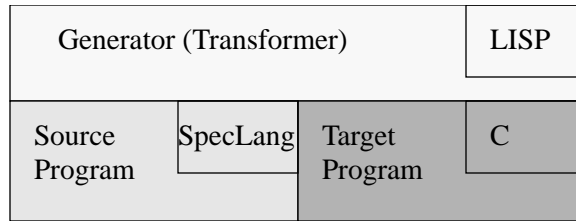


Figure 2: The three separate program spaces involved in program generation

lections of bindings (i.e., smaller sets of lexically visible variables) that can be organized hierarchically using `SetParent`. We saw earlier in Section 3.3 and Section 4.1 that these features enable generation environments to express complex scoping relationships in generated programs and to simplify parameterized code fragment specification by using implicit qualification on groups of variables.

Technically, generation scoping departs from the syntactic closures approach of [5] in many ways. Variable references are not represented as text which is then “closed” in the right environment. Instead they are generated from the beginning as “closed” references (for instance, as a name-environment pair). This technique is similar to syntactic closures that implement hygienic, *lexically scoped macros* [8]. Such implementations (see, for instance, [14]) “close” in a syntactic environment only the *generated* code fragments — not the user-supplied parameters to a macro. The implementation of generation scoping is very similar to that described in [14], and thus provides generation environments the power to emulate the hygienic macros of [11]<sup>9</sup>. The difference is that in generation scoping the environments of [14] become visible to the user of the mechanism (first-class objects). They can then be explicitly organized hierarchically using `SetParent`.

Lexically scoped macros have proven to be particularly useful in solving binding problems. Their behavior is best illustrated by an example:

```

{ int foo = 4;
  #define bar foo/2
  { int foo = 8;
    y = bar; } }

```

(18)

If C macros were lexically scoped (assuming they were part of the language and not handled by a preprocessor) the code fragment in (18) would set `y` to the value 2. In other words, macro `bar` would always refer to the variable `foo` that is visible where `bar` was defined — not where it is used.

What is interesting about lexically scoped macros is that they present *another* way to determine the bindings of *generated* identifiers (for instance, `foo` in the second line of (18)). The variables that can bind such identifiers, however, are those lexically visible *at the macro definition site* (like `foo` in the first line of (18)). In other words, static information in the program is used to determine bindings automatically. This is a desirable feature that we would like to exploit in general-purpose program generation.

Unfortunately, this seems unlikely. Program generation, in general, deals with three distinct program entities: the generator, the source program (specification), and the target program (generated code). Each can have its own namespace as depicted in Figure 2. Generators compile domain-specific specifications into target programs coded in familiar high-level languages. For instance, the P2 generator [4] compiles its specification (C extended with declarative data structure primitives) into pure C code. Figure 2 also sug-

---

9. As explained in [8], syntactic closures cannot be used to implement hygienic macro expansion.

gests implementation languages for the three programs to emphasize their independence at source level. In the case of Scheme and other languages whose extensibility is based on macros, these three namespaces collapse into one: there is a single namespace that contains macro definitions (transformer), macro invocations (source program), and macro expansion code (target code).

Lexically-scoped macros use the variables visible *in the generator* to determine bindings of code *in the target program*. If, however, the three namespaces are separate, this is not feasible — the generator writer will have to specify the bindings of generated identifiers explicitly. Consider example:

```
{ int foo = 4;
  #define bar foo/2
  { Generate `(/ bar 2); } }
```

 (19)

There does not seem to be any good reason to equate the two `bar` identifiers in (19). The generated identifier `bar` is a variable in a code fragment written in a LISP variant. The other `bar` is a C macro in the generator. The two are obviously distinct. In general, it is not likely that we can ever solve binding problems for generated identifiers by using static information from the source program or the generator.

The separation of namespaces gives rise to other basic differences between program generation and macro expansion. For instance, generator code is normally compiled into a binary format for reasons of source-code hiding and efficiency. This precludes generating new “generator code” on-the-fly. (Again, this is because generators, unlike macro systems, are not reflective). In macro systems, however, 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).

## 6 Conclusions

Program generation is a valuable technique for software development that will become progressively more important in the future. In this paper we have argued for dissociating the scoping of generated variables from their position in the target program. Instead, variables are grouped together intentionally — that is, according to their roles in the generated program. We have presented generation scoping as a general-purpose, domain-independent mechanism to address all scoping needs of generated programs. This approach can make writing software generators easier and more convenient. These capabilities were proven in the design and implementation of DiSTiL, a GenVoca generator for container data structures. We used Microsoft’s IP as the platform for our development; generation scoping is now an integral part of the IP toolset.

There are many interesting issues that can be explored further. First, the facility presented here is not currently suited for incremental program generation. Generators typically do not produce code incrementally: if a small part of a specification changes, the entire target application will be re-generated. In many cases this is not necessary. The problem can be solved by adding a mechanism to persistently store the generation environment structures for a given target document. These can later be retrieved and guide the incremental generation process. Another interesting issue is that of implementing a stand-alone version of generation scoping, perhaps in conjunction with a powerful macro facility. This should provide easy ways to extend an existing language. Second, generation scoping was developed in the context of the C-language (or IP’s version of C). Integration of generation scoping with object-oriented languages appears to offer enhanced opportunities of programming convenience.

We believe that generation scoping is a useful mechanism for program generation. It is safe, simple, and efficient. We hope the ideas incorporated in its design will find their way in other similar tools (for

instance, macro expansion facilities).

**Acknowledgments.** We gratefully acknowledge funding from Microsoft Research that supported this work.

## 7 References

- [1] D. Batory, L. Coglianesi, M. Goodwin, and S. Shafer, “Creating Reference Architectures: An Example From Avionics”, *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.
- [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 and J. Thomas. “P2: A Lightweight DBMS Generator”, Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, June 1995.
- [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] 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`.
- [8] 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.
- [9] W. Clinger, J. Rees (editors). “The Revised<sup>4</sup> Report on the Algorithmic Language Scheme”. *Lisp Pointers IV(3)*, July-September 1991, 1-55.
- [10] C. Hanson. “A Syntactic Closures Macro Facility”. *Lisp Pointers IV(4)*, October-December 1991, 9-16.
- [11] 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.
- [12] C. Lin and L. Snyder, “ZPL: An Array Sublanguage”. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds, 1993, 96-114.
- [13] 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.
- [14] J. Rees. “The Scheme of Things: Implementing Lexically Scoped Macros”. *Lisp Pointers VI(1)*, January-March 1993.
- [15] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference*, 1995.
- [16] M. Sirkin, D. Batory, and V. Singhal. “Software Components in a Data Structure Precompiler”. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.