

# Scoping Constructs for Software 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.** A well-known problem in program generation is *scoping*. When identifiers (i.e., symbolic names) are used to refer to variables, types, or functions, program generators must ensure that generated identifiers are *bound* to their intended *declarations*. This is the standard scoping issue in programming languages, only automatically generated programs can quickly become too complex and maintaining bindings manually is hard. In this paper we present *generation scoping*: a language mechanism to facilitate the handling of scoping concerns. Generation scoping offers control over identifier scoping beyond the scoping mechanism of the target programming language (i.e., the language in which the generator output is expressed). Generation scoping was originally implemented as an extension of the code template operators in the Intentional Programming platform, under development by Microsoft Research. Subsequently, generation scoping has also been integrated in the JTS language extensibility tools. The capabilities of generation scoping were invaluable in the implementation of two actual software generators: DiSTiL (implemented using the Intentional Programming system), and P3 (implemented using JTS).

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

## 1 Introduction

*Program generation* is the process of generating code in a high-level programming language. A well-known problem with program generation has to do with the resolution of names used to refer to various entities (e.g., variables, types, and functions) in the generated program. This is the standard scoping issue of programming languages but scoping problems are exacerbated when programs are generated automatically. For instance, often the same macro or template is used to create multiple code fragments, which all exist in the same scope of the generated program. In that case, care should be taken so that the generated fragments do not contain declarations that conflict (e.g., variables with the same name in the same lexical scope).

Avoiding scoping problems in program generation can be done manually: Lisp programmers are familiar with the `gensym` function for creating new symbols. Using `gensym` to create unique names for generated variable declarations is one of the commonly recommended practices for Lisp programmers. Unfortunately, this practice is tedious; it complicates program generation and makes the generator code harder to read and maintain. Mechanisms have been invented to relieve the programmer of the obligation to keep track of declared variables and generate new symbols for their names. These mechanisms fall under the general heading of *hygienic macro-expansion* (e.g., [7], [8], [10]) and address the scoping problem for macros: self-contained trans-

formations that are both specified and applied in the *same* program. A desirable property in this setting is *referential transparency*: identifiers introduced by a transformation refer to declarations lexically visible at the site where the transformation is defined—not where it is applied. In this paper we adapt the ideas of hygienic macro-expansion to a more general program generation setting, where referential transparency is not meaningful. Our mechanism can be used for *software generators*, which are essentially stand-alone compilers. The definition of transformations in software generators has no lexical connection to the program generated by these transformations (for instance, the generator program and the generated program may be in different programming languages). Our mechanism is called *generation scoping* and gives the generator programmer explicit and convenient control over the scoping of the generated code. (In fact, the generation scoping idea was invented independently of hygienic macro-expansion techniques, but in the process we essentially re-invented the principles that are common to both generation scoping and hygienic macro expansion.)

Generation scoping has been implemented on two language extensibility platforms: Microsoft Research’s Intentional Programming system [13] and the Jakarta Tool Suite (JTS) [1]. Two component-based software generators, DiSTiL [14] and P3 [1], were built using generation scoping. In both cases, generation scoping proved invaluable, as it simplified the generator code and accentuated the distinction between executed and generated code.

## 2 Background: Scoping for Generated Programs

For a quick illustration of some of the scoping issues in program generation, we will use an (imaginary<sup>1</sup>) extension of the C language with *code template operators*. We introduce two such operators: `quote` (abbreviated as ```) and `unquote` (abbreviated as `$`). `quote` designates the beginning of a code template and `unquote` escapes from it to evaluate a code generating expression.<sup>2</sup> Consider generating code to iterate over a text file and perform some actions on its data. A possible implementation in our example language is shown below, with the quoted code appearing in bold:

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

(1)

<sup>1</sup> Actually, this extension of C with meta-programming constructs corresponds closely to the state of the Intentional Programming system in 1995, when generation scoping was implemented.

<sup>2</sup> These operators are analogous to the LISP “backquote” and “comma” macro pair or the Scheme `quasiquote` and `unquote` primitives [6].

The first scoping issue in the above code has to do with the scope used to bind the references in the generated code fragment. That is, the generated code fragment only has meaning in a lexical environment where `FILE`, `FatalError`, `fopen`, etc., are defined. We will disregard this issue for now and concentrate on the scope of generated *declarations*.

In the above example, two declarations are generated (these are underlined in the code). The scope of these declarations should be quite different. The first is the declaration of file pointer `fp`. This variable should be invisible to user code—the code fragment represented by `actions` should not be able to refer to `fp`. This is the rule of *hygienic* program generation and it ensures that no accidental capture of references can occur: the code fragment represented by `actions` may contain a reference to some `fp`, but this will never be confused with the `fp` generated by the code above. Obviously, this is a good property to guarantee. The `fp` variable is just an implementation detail and its name should be protected from accidental clashes with other names that may be in use.

The generated declaration of variable `byte`, on the other hand, demonstrates the need for *breaking the hygiene*. Variable `byte` represents the current character being read from the text file. The code represented by `actions` should be able to access `byte`—in fact, `byte` is the only interface for exploiting the functionality of traversing the text file.

To illustrate the above points, consider an example use of the `CreateForAllInFile` function. A program can have a file pointer, `fp`, that points to a text file. We may want to generate code that determines whether a file is a prefix of the file pointed to by `fp`:

```
CreateForAllInFile(`("prefix.txt"),  
                  `{if (byte != fgetc(fp)) return -1;}`);
```

The `fp` identifier above is *not* the same as the `fp` introduced accidentally by the `CreateForAllInFile` function in (1). Nevertheless, a naive generation process will result into `fp` (above) accidentally referring to the internal variable of `CreateForAllInFile`. This is a scoping problem that we want to avoid, so that the client of `CreateForAllInFile` can be oblivious to the choice of name used for the internal file pointer variable. On the other hand, the reference to `byte` *should* refer to the variable whose declaration is generated in (1). Clearly, it is hard to satisfy both requirements with code fragment (1), as the two declarations are never differentiated. We now discuss two existing approaches to scoping and why they are not sufficient for our purposes.

**First Approach: Generating Unique Symbols Manually.** The simplest way to satisfy this dual requirement is manually. We can generate a unique symbol for all declarations that should be hidden from other code. This is, for instance, a common practice for Lisp programmers, who can use the `gensym` function to create unused, unique names in generated code. With our example language and the code fragment in (1), we get:

```

CODE CreateForAllInFile (CODE filename, CODE actions)
{ CODE mfp = gensym();
  return `{
    FILE *$mfp;
    if (($mfp = fopen($filename, "r")) == NULL)
      FatalError(FILE_OPEN_ERROR);
    while ( feof($mfp) == FALSE) {
      int byte = fgetc($mfp);
      $actions;
    }
  }
}

```

(2)

For typical software generators, where many code fragments are created and composed, this solution is clearly unsatisfactory. The code becomes immediately harder to read and maintain, with many alternations between generated (quoted) and evaluated (unquoted) code. The intention that the `mfp` (for meta-file-pointer) variable holds a single variable name (and not an entire expression) is not enforced at the language level. Furthermore, understanding the code generated by code fragment (2) requires understanding the control flow of (2) (e.g., to ensure that the value of `mfp` never changes).

The most important disadvantage of the “manual” creation of unique identifiers, however, is that the generator programmer has to *anticipate* which identifiers may cause name clashes and need to be hidden. The most likely problem with code fragment (2) is that the generated code will be used in a lexical environment where an identifier like `FILE`, `FatalError`, etc., does not have the meaning intended by the author of (2). The only way to avoid this problem is to use unique symbol names for *all* definitions. Then the new names will have to be passed around in the generator code so that only their legitimate clients have access to them. For instance, one can imagine that the actual name for procedure `FatalError` will need to be a new, unique symbol (to avoid accidental capture), which is then passed as a parameter to `CreateForAllInFile`, resulting in a more complicated code fragment:

```

CODE CreateForAllInFile (CODE mFatalError, CODE filename, CODE
actions)
{ CODE mfp = gensym();
  return `{
    FILE *$mfp;
    if (($mfp = fopen($filename, "r")) == NULL)
      $mFatalError(FILE_OPEN_ERROR);
    while ( feof($mfp) == FALSE) {
      int byte = fgetc($mfp);
      $actions;
    }
  }
}

```

(3)

If we take this approach to an extreme (e.g., doing the same for `FILE_OPEN_ERROR`, `FALSE`, and all other generated variables), the code will become completely unreadable and the programmer will have an obligation to keep close track of all generated declarations as well as their clients.

**Second Approach: Hygienic Macros.** Another way to satisfy the scoping requirements for the two generated variables, is through a hygienic mechanism, such as those proposed in the work on hygienic macro expansion (e.g., [5], [7], [8], [10], [11]). Hygienic mechanisms work by making generated declarations *by default invisible* outside the pattern or template (e.g., macro) that introduced them. In the example of (1), this would mean that both the declaration of `fp` and that of `byte` will be invisible to code in `actions`. Since this is not desirable in the case of `byte`, the hygiene must be explicitly broken. In the hygienic macros work, this case is considered to be a rare exception.<sup>3</sup> Carl’s hygienic mechanism [5] even attempts to automatically detect common patterns that require breaking the hygiene. Additionally, lexically-scoped hygienic macros [7][8] use the lexical environment of the generation site as the lexical environment of the generated code (a property called *referential transparency*).

The problem with using this approach in software generators is that it is not possible to reliably deduce the scope of a variable from the lexical location of the code that generates its declaration. In particular there are two important differences between macros and software generators:

1. Macros are (more or less) self-contained units. There is a clear distinction between the macro code and the code that is passed as a parameter to the macro. This is not the case with software generators. The code generating a declaration is not, in general, in close lexical proximity of the code generating a reference to that declaration.
2. The lexical environment of a program-generating code fragment cannot be identified with the lexical environment of the generated code in software generators. (In hygienic macro terminology: referential transparency is not meaningful.) For instance, we could even have the generator be in a different language than the generated code (e.g., unquoted code could be in Java, quoted code in C). In contrast, lexically scoped macros use the lexical environment of the macro definition to determine the binding of all references generated by the macro.

The first point is a result of observation. The transformations in most software generators interleave generating code with arbitrary computation more often than macros. In this way, it is hard to identify a self-contained program fragment *in the generator* that will be identified with a scope in the generated program.

To see the second point, consider again code fragment (1), reproduced below for easy reference.

```
CODE CreateForAllInFile (CODE filename, CODE actions)
{ return `{
    FILE *fp;
    if ((fp = fopen($filename, "r")) == NULL)
        FatalError(FILE_OPEN_ERROR);
    while ( feof(fp) == FALSE) {
```

---

<sup>3</sup> For instance, we read in [7]: “We here ignore the occasional need to escape from hygiene.”

```

        int byte = fgetc(fp);
        $actions;
    }
}

```

`CreateForAllInFile` has several dependencies to other generated code (e.g., the `FILE` type identifier, the `FatalError` function, the `FALSE` constant, etc.). In the case of lexically-scoped macros such dependencies are resolved at the site of the macro definition. This would be equivalent to trying to find bindings for `FILE`, `FatalError`, etc., in the program site where `CreateForAllInFile` is defined. This approach is not valid for software generators. *For instance, the `FatalError` routine may not be declared as a routine in the generator or a standard library, but instead exist only in the generated program.* Hence, the declaration of `FatalError` must be non-hygienic so that the code fragment generated by `CreateForAllInFile` can access it.

### 3 Generation Scoping

#### 3.1 Generation Environments

Because of the differences between macros and software generators, we cannot hope to achieve the same degree of automation for software generators as with hygienic lexically-scoped macros. Nevertheless, we can still do better than manually generating new symbols, as in example (3) of Section 2. This is the purpose of generation scoping. Generation scoping is a mechanism that represents lexical environments in the generated program as first-class entities. In this way, the generator has control of the scoping of the generated program, beyond that offered by the target programming language.

To support lexical environments as first-class entities, generation scoping adds a new keyword, `environment`, to the language in which the program generator is written. Its syntax is:

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

where `statement` contains one or more quoted expressions. The `generation-environment` is an expression that yields a value of type `ENV`. `ENV` is a type used to represent environments and only has a constructor and equality function defined (i.e., we can only create new values of type `ENV` and compare them with existing ones). The constructor for environments, `new_env`, can take an arbitrary number of arguments whose values are other environments. These environments become the *parents* of the newly created environment (the *child*). All variable declarations in a parent become visible to the child environment. Like traditional scoping mechanisms, variable bindings of the child eclipse bindings with the same name in the parent.

An example use of `environment` in code implementing our example text file traversal follows below:

```

CODE CreateForAllInFile (ENV p, CODE mtbyte, CODE filename,
                        CODE actions)
{
  environment(new_env(p))
  return `{
    FILE *fp;
    if ((fp = fopen($filename, "r")) == NULL)
      FatalError(FILE_OPEN_ERROR);
    while ( feof(fp) == FALSE) {
      int $mtbyte = fgetc(fp);
      $actions;
    }
  }
}

```

(4)

To generate code using the `quote` operator, an environment needs to be specified. In this way, the code represented by `actions` can never access variable `fp` (as `fp` is generated in a new environment—which becomes a child of an environment passed into the function). At the same time, if the variable represented by `mtbyte` is generated in the same environment as `actions`, they are visible to each other. This is the case with most straightforward uses of this function. For instance:

```

environment(e)
result =
  CreateForAllInFile(global_env, `byte, `("file.txt"),
                    `putchar(byte) );

```

(5)

Comparing code fragments (4) and (3), we can see why using environments is more convenient than manually handling variables by creating new symbols. In particular, there are several important advantages:

1. The generator programmer does not need to explicitly state which variables get “closed” in the right lexical environment. *All* declarations generated under an `environment` statement will be automatically added to the corresponding environment. Additionally, the generator programmer does not need to explicitly retrieve the binding for a certain identifier. *All* references (e.g., to `fp`, but also to `FILE`, `FatalError`, `fopen`, etc., above) are interpreted relative to that environment. *This means that, if a code fragment is generated in the intended environment, it can later be used without problems in a local context, even if the local context contains different bindings for the same identifiers.* For example, in code fragment (5), above, if `global_env` has the intended declaration for, e.g., `FILE`, it will not subsequently matter if the generated code fragment is output in the middle of a function where `FILE` means something different. The reference will always be to the `FILE` type variable defined in the environment represented by `global_env`.
2. The alternation between executed and generated code is avoided. There is no need to unquote code just to supply a unique symbol name.
3. Declarations are treated as a group, instead of individually. In the above example

there is only one variable declared, so this is not really an advantage. In quoted code with several generated declarations, however, handling environments is easier than handling all new symbols individually. Of course, the same grouping effect could be achieved by using a mapping data structure in the generator code. The advantage of generation scoping is that the data structure is now integrated in the language and insertions and lookups are implicit (i.e., the programmer never has to specify them—see the first point above).

### 3.2 Implementation Issues

It is perhaps worth stressing again that the main advantage of generation scoping is that the generator programmer is relieved of the responsibility of adding declarations to environments and looking up identifier bindings in those environments. That is, the implementation of `quote` will determine whether a generated identifier is actually a declaration (of a variable, function, type, etc.) or a reference to an existing entity. Each environment has a symbol table and a collection of pointers to the parent environments. In case an identifier represents a declared entity, it is added to the current environment's symbol table together with a corresponding generated unique name for the declared entity. When a generated identifier is a reference, it will be looked up in the appropriate environment's table and, if it is not there, in the parent environments recursively.<sup>4</sup> The result of the identifier lookup is the unique generated name for the matching declaration. In this way, no accidental reference to the wrong variable, type, function, etc., can occur, as long as the environments are set up properly.

As is well-documented in the work on hygienic macros [7][10], determining the syntactic role of an identifier (i.e., whether it is a declaration or a reference) is hard when the entire program has not yet been generated. For instance, consider the program-generating function:

```
CODE CreateDclOrRef (CODE type) {
  return `{ $type newvar = 10 };
```

```
 }
```

In most programming environments,<sup>5</sup> it is impossible to tell before the code is generated whether the generated code declares `newvar` or refers to an existing variable of the same name. If the parameter type holds the type specifier `int`, then `newvar` is being declared. If, on the other hand, it holds the operator `*`, it is not. This problem has been studied extensively in the hygienic macro community and the common approach is to employ a “painting” algorithm that marks each identifier with the environment where it was created. It is easy to adapt this approach to generation scoping:

---

<sup>4</sup> In case a matching declaration is found in multiple parent environments, the unique name returned is determined by a depth-first search of the parent tree, based on the order parents were specified in the `new_env` constructor. This is, however, an arbitrary default and not fundamental to the system's operation.

<sup>5</sup> This is not true for the Intentional Programming system, where the most mature version of generation scoping was implemented. The system fundamentally distinguishes (at the editor level, even) between declarations and references, so that a single code fragment cannot be used to create both.

After all the code has been generated, the marked declarations can be matched to marked references (assuming they came from the same environment). Remaining references can then be just unmarked, so that they become free references and can refer to externally declared symbols. A more thorough discussion on implementing a “painting” algorithm for program generation can be found in [11].

## 4 Generation Scoping in DiSTiL

Generation scoping was implemented as part of IP (Intentional Programming) [13], a general-purpose transformation system under development by Microsoft Research. It was subsequently used to build the DiSTiL software generator [14] as a domain-specific extension to IP. DiSTiL is a generator that follows the GenVoca [3] design paradigm. 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 DiSTiL generator is essentially a compiler for the domain of container data structures. Complex container data structures are 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 data structure 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.

Generation scoping was indispensable in the implementation of DiSTiL. Even relatively short DiSTiL specifications (around 10-20 lines) could generate thousands of lines of optimized code. Due to the complexity of the generated code, as well as the flexibility of parameterization (a layer could be composed with a wide variety of other layers), maintaining correct scoping for generated code would have been a nightmare without generation scoping. In fact, initially we had attempted to implement DiSTiL with manual resolution of generated references (by generating unique symbols, as in code fragment (3)). *The sheer difficulty of this task was what motivated generation scoping in the first place.*

Generation scoping is used in DiSTiL not only to ensure the correctness of references to global declarations (e.g., library functions) but also to overcome the scoping limitations of the target language (C). With generation scoping, DiSTiL effectively manages different namespaces for every layer in a composition. In this way, there are no clashes between identically named variables introduced by different layers (or different instances of the same layer). At the same time, the code is simplified by having namespaces connected appropriately so that generated code can access all the required declarations without explicit qualification.

DiSTiL data structures consist of three distinct entities: a container, elements, and iterators (called *cursors*). Generated variables are grouped together into a common environment 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 variables need *not* belong to a single lexical context. For example, variables in an environment may be global, or local, or fields of a record type. Thus,

variables of an environment could belong to slices of many different lexical contexts in the generated program. In this way, the environment acts as a generator-managed namespace mechanism for the target language.

Consider the following organization used in DiSTiL (and, in fact, also in P3). 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. 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). The `CursGeneric` environment encapsulates generic cursor-related variables (including the cursor identifier). 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 also reference container fields) is generated using the `CursGeneric` environment. Figure 1(a) depicts this relationship.

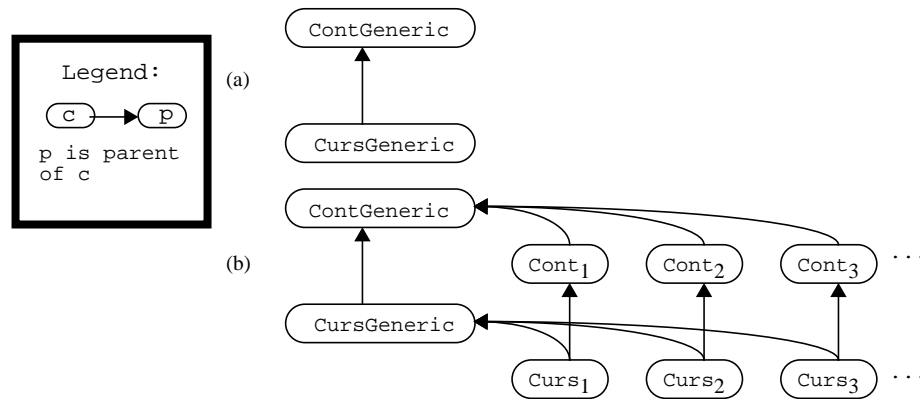


Figure 1: Hierarchical Organizations of Environments in DiSTiL

As mentioned earlier, 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 1(b) shows this hierarchical organization of environments.

To illustrate these ideas, consider an ordered doubly-linked list layer. This layer 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 `Refine-`

Types() 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, ENV 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 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 `nextj`, `prevj`, etc.) is simple using generation environments as organized in Figure 1. Each copy of the list layer will have its own generation environments `Contj` and `Cursj`, and all code generated by that copy would always use these environment variables.

For an example, consider 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 `Cursi` 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, ENV Curs ) {
    environment(Curs) {
        *Remove_Code = `{ Element *next_el = cursor->next;
                          Element *prev_el = cursor->prev;
                          $( *Remove_Code );
                          if (next_el != null)
                              next_el->prev = prev_el;
                          if (prev_el != null)
                              prev_el->next = next_el;
                          if (container->first == cursor.obj)
                              container->first = next_el;
                          if (container->last == cursor.obj)
                              container->last = prev_el; };
    }
}
```

Note that 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 `Contj`. 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. Note also that there are two generated temporary declarations in this code fragment, which are completely protected from accidental reference.

This example is convenient for demonstrating the benefits of generation scoping. We attempt to show these benefits by speculating on the alternatives. Clearly the above code fragment has many external generated references, so default hygiene is not really an option. The generator writer has to explicitly create new symbols (as in code fragment (3)) for the declarations of `container`, `cursor`, etc. (not shown). Instead of managing all the new symbols individually, the generator writer could set up a data structure *in the generator (unquoted) code* to maintain the mappings of identifiers to variables. Then the writer could use explicit unquotes to introduce the right bindings. Given that declarations need to be inserted in the data structure explicitly and references need to be looked up explicitly, the code would be much more complicated. One can add some syntactic sugar to make the code more appealing. For instance, we can use `$(ds, id)` to mean “unquote and lookup identifier `id` in bindings data structure `ds`”. Similarly, we can use `$(ds, id)` to mean “unquote and add variable `id` in bindings data structure `ds`”. Even then, the code would be practically unreadable:

```
void Remove( CODE *Remove_Code, BindingDS ds ) {
    *Remove_Code =
        `{ $(ds, Element) *$(ds, next_el) =
           $(ds, cursor)->$(ds, next);
           $(ds, Element) *$(ds, prev_el) =
           $(ds, cursor)->$(ds, prev);
           $(*Remove_Code);
           if ($(ds, next_el) != null)
               $(ds, next_el)->$(ds, prev) = $(ds, prev_el);
           if ($(ds, prev_el) != null)
               $(ds, prev_el)->$(ds, next) = $(ds, next_el);
           if ($(ds, container)->$(ds, first) ==
               $(ds, cursor).$(ds, obj))
               $(ds, container)->$(ds, first) = $(ds, next_el);
           if ($(ds, container)->$(ds, last) ==
               $(ds, cursor).$(ds, obj))
               $(ds, container)->$(ds, last) = $(ds, prev_el); };
}
```

As outlined earlier, generation scoping improves over this code in three ways: First, no explicit data structure insertions/lookups need to be performed (e.g., there are no `$$` and `$(` operators). Second, no explicit escapes are introduced—there is no alternation between quoted and unquoted code. Third, the grouping of variables is implicit—there is no need to repeatedly refer to a data structure like `ds`.

## 5 Related Work

Given our prior discussion of hygienic macros, here we will only touch upon a few other pieces of related work.

The environments used in generation scoping are similar to syntactic environments

in the *syntactic closures* work [4][9]. In syntactic closures, environments are first-class entities and code fragments can be explicitly “closed” in a lexical environment. Nevertheless, there are significant differences between the two approaches: Syntactic closures environments can only capture the set of variables that are lexically visible at a specific point in a program.<sup>6</sup> In contrast, our environments can be arbitrary collections of bindings (i.e., smaller sets of lexically visible variables) and can be organized hierarchically. More importantly, however, declarations are added to generation scoping environments implicitly by generating (quoting) code that declares new variables. Thus, our approach is much more automated than syntactic closures and is ideally suited to software generators (where the lexical environment is being built while code is generated). Also, generation scoping can be used to implement the hygienic, lexically-scoped macros of [7], unlike syntactic closures, which cannot be used to implement hygienic macro expansion, as explained in [7].

Generation scoping is concerned only with maintaining correct scoping for generated code fragments. Other pieces of work deal with various other correctness properties of composed code fragments. Selectively, we mention some work on the problem of ensuring type correctness for generated programs, both for two-stage code [12] (i.e., generator and generated code) and multi-stage code [15] (i.e., code generating code that generates other code, etc.).

## 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 shown how to address the scoping issues that arise in software generators. We have presented generation scoping: a general-purpose, domain-independent mechanism to address all scoping needs of generated programs. Generation scoping can make writing and maintaining software generators easier. Its capabilities were proven in the implementation of the DiSTiL [14] and P3 [1] generators.

The future of software engineering lies in the automated development of well-understood software. Program generators will play an increasingly important role in future software development. We consider generation scoping to be a valuable language mechanism for generator writers and hope that it will be adopted in even more extensible languages and transformation systems in the future.

## Acknowledgments

Support for this work was provided by Microsoft Research, and the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226). We would like to thank an anonymous referee for his/her useful suggestions.

---

<sup>6</sup> In the original syntactic closures work [4] 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, [9]).

## References

- [1] D. Batory, G. Chen, E. Robertson, and T. Wang, “Web-Advertised Generators and Design Wizards”, *International Conference on Software Reuse (ICSR)*, 1998.
- [2] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, *International Conference on Software Reuse (ICSR)*, 1998.
- [3] 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.
- [4] A. Bawden and J. Rees, “Syntactic Closures”. In *Proceedings of the SIGPLAN ‘88 ACM Conference on Lisp and Functional Programming*, 86-95.
- [5] 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>.
- [6] 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.
- [7] 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.
- [8] R.K. Dybvig, R. Hieb, and C. Bruggeman, “Syntactic Abstraction in Scheme”, in *Lisp and Symbolic Computation*, 5(4), December 1993, 83-110.
- [9] C. Hanson, “A Syntactic Closures Macro Facility”, *Lisp Pointers IV(4)*, October-December 1991, 9-16.
- [10] 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.
- [11] J. Rees, “The Scheme of Things: Implementing Lexically Scoped Macros”, *Lisp Pointers VI(1)*, January-March 1993.
- [12] T. Sheard and N. Nelson, “Type Safe Abstractions Using Program Generators”, Oregon Graduate Institute Tech. Report 95-013.
- [13] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference*, 1995.
- [14] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conference on Domain-Specific Languages (DSL)*, 1997.
- [15] W. Taha and T. Sheard, Multi-stage programming with explicit annotations, *ACM Symp. Partial Evaluation and Semantics-Based Program Manipulation (PEPM ‘97)*, 1997.