RESEARCH ARTICLE OPEN ACCESS

# Scoping Constructs for Software Generators

Bandita Das, Sambit Sahoo,
*Gandhi Institute of Excellent Technocrats, Bhubaneswar,India*
*Vignan Institute of Technology and Management,Berhampur, Ganjam, Odisha, India*

**ABSTRACT.** A well-known problem in program generation is scoping. When iden-tifiers (i.e., symbolic names) are used to refer to variables, types, or functions,program generators must ensure that generated identifiers are bound to theirintended declarations. This is the standard scoping issue in programming lan-guages, only automatically generated programs can quickly become too com-plexandmaintainingbindingsmanuallyishard.Inthispaperwepresentgeneration scoping: a language mechanism to facilitate the handling of scopingconcerns. Generation scoping offers control over identifier scoping beyond thescoping mechanism of the target programming language (i.e., the language inwhich the generator output is expressed). Generation scoping was originallyimplemented as an extension of the code template operators in the IntentionalProgrammingplatform,underdevelopmentbyMicrosoftResearch.Subse-quently, generation scoping has also been integrated in the JTS language exten-sibility tools. The capabilities of generation scoping were invaluable in theimplementation of two actual software generators: DiSTiL (implemented usingthe Intentional Programming system), and P3 (implemented using JTS).
**Keywords**:softwaregenerators,programtransformations,generationscoping,hygienicmacro expansion

## I. INTRODUCTION

Programgenerationistheprocessofgeneratingcodeinahigh-levelprogramminglanguage. A well-known problem with program generation has to do with the resolu-tion of names used to refer to various entities (e.g., variables, types, and functions) inthe generated program. This is the standard scoping issue of programming languagesbutscopingproblemsareexacerbatedwhenprogramsaregeneratedautomatically.Forinstance, often the same macro or template is used to create multiple code fragments,whichallexistinthesamescopeofthegeneratedprogram.Inthatcase,careshouldbetaken so that the generated fragments do not contain declarations that conflict (e.g.,variableswith the same name in the samelexical scope).
Avoidingscopingproblemsinprogramgenerationcanbedonemanually:Lisppro-grammers are familiar with the gensymfunction for creating new symbols. Usinggensymto create unique names for generated variable declarations is one of the com-monly recommended practices for Lisp programmers. Unfortunately, this practice istedious; it complicates program generation and makes the generator code harder toread and maintain. Mechanisms have been invented to relieve the programmer of theobligation to keep track of declared variables and generate new symbols for theirnames.Thesemechanismsfallunderthegeneralheadingofhygienicmacro-expansion(e.g.,[7],[8],[10])andaddressthescopingproblemformacros:self-containedtrans-formationsthatarebothspecifiedandappliedinthesameprogram.Adesirableprop-ertyinthissettingisreferentialtransparency:identifiersintroducedbyatransformationrefertodeclarationslexicallyvisibleatthesitewherethetransforma-tionisdefined—notwhereitisapplied.Inthispaperweadapttheideasofhygienicmacro-expansiontoamoregeneralprogramgenerationsetting,wherereferentialtransparencyisnotmeaningful.Ourmechanismcanbeusedforsoftwaregenerators,whichareessentiallystand-alonecompilers.Thedefinitionoftransformationsinsoft-waregeneratorshasnolexicalconnectiontotheprogramgeneratedbythesetransfor-mations(forinstance,thegeneratorprogramandthegeneratedprogrammaybeindifferentprogramminglanguages).Ourmechanismiscalledgenerationscopingandgivesthegeneratorprogrammerexplicitandconvenientcontroloverthescopingofthegeneratedcode.(Infact,thegenerationscopingideawasinventedindependentlyofhygienicmacro-expansiontechniques,butintheprocessweessentiallyre-inventedtheprinciplesthatarecommontobothgenerationscopingandhygienicmacroexpansion.)Generation scopinghasbeenimplementedontwolanguageextensibilityplat-forms:MicrosoftResearch'sIntentionalProgrammingsystem[13]andtheJakartaToolSuite(JTS)[1].Twocomponent-basedsoftwaregenerators,DiSTiL[14]andP3[1],werebuiltusinggenerationscoping.Inbothcases,generationscopingprovedinvaluable,asitsimplifiedthegenerat

orcodeandaccentuatedthedistinctionbetween executedandgeneratedcode.

**Background:ScopingforGeneratedPrograms**
Foraquickillustrationofsomeofthescopingissuesinprogramgeneration,we will use an (imaginary[1]) extension of the C language with code template operators.Weintroducetwosuchoperators:quote(abbreviatedas‘)andunquote(abbreviatedas$).quotedesignatesthebeginningofacodetemplateandunquoteescapesfrom it to evaluate a code generating expression.[2] Consider generating code to iterate over atextfileandperformsomeactionsonitsdata.Apossible implementationinourexam-plelanguageis shownbelow,with thequoted codeappearingin bold:

```
CODE CreateForAllInFile (CODE filename, CODE actions)
{return ‘{FILE *fp;
if ((fp = fopen($filename, “r”)) == NULL)FatalError(FILE_OPEN_ERROR);
while ( feof(fp) == FALSE) {intbyte = fgetc(fp);
$actions;
}
}
}
```

The first scoping issue in the above code has to do with the scope used to bind thereferences in the generated code fragment. That is, the generated code fragment onlyhas meaning in a lexical environment where FILE, FatalError, fopen, etc., aredefined.Wewilldisregardthisissuefornowandconcentrateonthescopeofgenerateddeclarations.

In the above example, two declarations are generated (these are underlined in thecode). The scope of these declarations should be quite different. The first is the decla-ration of file pointer fp. This variable should be invisible to user code—the code frag-mentrepresented by actions should not be able to refer to fp. This is the rule ofhygienicprogramgenerationanditensuresthatnoaccidentalcaptureofreferencescanoccur:thecodefragment representedbyactions maycontainareferencetosomefp, but this will never be confused with the fpgenerated by the code above. Obvi-ously, this is a good property to guarantee. The fpvariable is just an implementationdetail and its name should be protected from accidental clashes with other names thatmay be in use.

The generated declaration of variable byte, on the other hand, demonstrates theneed for breaking the hygiene. Variable byte represents the current character beingread from the text file. The code represented by actions should be able to accessbyte—infact,byteistheonlyinterfaceforexploitingthefunctionalityoftraversingthetext file.
To illustrate the above points, consider an example use of the CreateForAllIn-Filefunction.Aprogram can haveafile pointer,fp,thatpoints toatext file.Wemaywanttogeneratecodethatdetermineswhetherafileisaprefixofthefilepointedtobyfp:

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

The fpidentifier above is not the same as the fpintroduced accidentally by theCreateForAllInFilefunction in (1). Nevertheless, a naive generation process willresultintofp(above) accidentally referring to the internal variable ofCreateForAl-lInFile. This is a scoping problem that we want to avoid, so that the client of Cre-ateForAllInFilecan be oblivious to the choice of name used for the internal filepointer variable. On the other hand, the reference to byte should refer to the variablewhose declaration is generated in (1). Clearly, it is hard to satisfy both requirementswith code fragment (1), as the two declarations are never differentiated. We now dis-cuss two existing approaches to scoping and why they are not sufficient for our pur-poses.

**First Approach: Generating Unique Symbols Manually.** The simplest way to sat-isfy this dual requirement is manually. We can generate a unique symbol for all decla-rationsthatshouldbehiddenfromothercode.Thisis,for instance,acommonpracticefor Lisp programmers, who can use the gensymfunction to create unused, uniquenamesingeneratedcode.Withourexamplelanguageandthecodefragmentin(1),weget:

```
CODE CreateForAllInFile (CODE filename, CODE actions)
{ CODEmfp = gensym();return‘{
FILE*$mfp;
if (($mfp= fopen($filename, “r”)) == NULL)FatalError(FILE_OPEN_ERROR);
while ( feof($mfp) == FALSE) {intbyte = fgetc($mfp);
$actions;
}
}
}          (2)
```

For typical software generators, where many code fragments are created and com-posed, this solution is clearly unsatisfactory. The code becomes immediately harder toread and maintain, with many alternations between generated (quoted) and evaluated(unquoted)code.Theintentionthatthemfp(formeta-file-pointer)variableholdsasin-gle variable name (and not an entire expression) is not enforced at the language level.Furthermore, understanding the code generated by code fragment (2) requires under-standingthecontrol flow of (2) (e.g., to ensure that the value ofmfpnever changes).

Themostimportantdisadvantageofthe“manual”creationofuniqueidentifiers, however, is that the generator programmer has to anticipate which identifiers maycause name clashes and need to be hidden. The most likely problem with code frag-ment (2) is that the generated code will be used in a lexical environment where anidentifier like FILE, FatalError, etc., does not have the meaning intended by theauthor of (2). The only way to avoid this problem is to use unique symbol names forall definitions. Then the new names will have to be passed around in the generatorcode so that only their legitimate clients have access to them. For instance, one can imaginethattheactualnameforprocedureFatalErrorwillneedtobeanew,uniquesymbol (to avoid accidental capture), which is then passed as a parameter to Create-ForAllInFile,resulting in a more complicated code fragment:

```
CODE CreateForAllInFile (CODE mFatalError, CODE filename, CODEactions)
{ CODEmfp = gensym();return‘{
FILE*$mfp;
if(($mfp=fopen($filename, “r”))== NULL)
$mFatalError(FILE_OPEN_ERROR);while(
feof($mfp) ==FALSE) {
intbyte= fgetc($mfp);
$actions;
}
}
}          (3)
```

Ifwetakethisapproachtoanextreme(e.g.,doingthesameforFILE_OPEN_ERROR, FALSE, and all other generated variables), the code will becomecompletelyunreadableandtheprogrammerwillhaveanobligationtokeepclosetrackof all generated declarations as well as their clients.

**Second Approach: Hygienic Macros.** Another way to satisfy the scoping require-ments for the two generated variables, is through a hygienic mechanism, such as thoseproposed in the work on hygienic macro expansion (e.g., [5], [7], [8], [10], [11]).Hygienicmechanismsworkbymakinggenerateddeclarationsbydefaultinvisibleout-side the pattern or template (e.g., macro) that introduced them. In the example of (1),this would mean that both the declaration of fpand that of byte will be invisible tocode in actions. Since this is not desirable in the case of byte, the hygiene must beexplicitlybroken.Inthehygienicmacroswork,thiscaseisconsideredtobearare exception.[3] Carl’s hygienic mechanism [5] even attempts to automatically detect com-monpatternsthatrequirebreakingthehygiene.Additionally,lexically-scopedhygienic macros [7][8] use the lexical environment of the generation site as the lexicalenvironmentofthegeneratedcode(apropertycalledreferentialtransparency).

The problem with using this approach in software generators is that it is not possi-ble to reliably deduce the scope of a variable from the lexical location of the code thatgenerates its declaration. In particular there are two important differences betweenmacrosand software generators:

1. Macros are (more or less) self-contained units. There is a clear distinction betweenthe macro code and the code that is passed as a parameter to the macro. This is not thecase with software generators. The code generating a declaration is not, in general, incloselexical proximity of the code generating a referenceto that declaration.

2. The lexical environment of a program-generating code fragment cannot be identi-fied with the lexical environment of the generated code in software generators. (Inhygienic 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 scopedmacrosusethelexicalenvironmentofthemacrodefinitiontodeterminethebindingofall references generated by the macro.

Thefirstpointisaresultofobservation.Thetransformationsinmostsoftwaregen-erators 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 generatorthat will be identified with a scope in the generated program.

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

```
CODE CreateForAllInFile (CODE filename, CODE actions)
{return ‘{
FILE*fp;
if ((fp = fopen($filename, “r”)) == NULL)FatalError(FILE_OPEN_ERROR);
while ( feof(fp) == FALSE) {
intbyte = fgetc(fp);
$actions;
}
}
}
```

CreateForAllInFilehasseveraldependenciestoothergeneratedcode(e.g.,theFILEtypeidentifier,theFatalErrorfunction,theFALSEconstant,etc.).Inthecaseoflexi cally-scopedmacrossuchdependenciesareresolvedatthesiteofthemacrodef-inition. This would be equivalent to trying to find bindings for FILE, FatalError,etc.,intheprogramsitewhereCreateForAllInFileisdefined.Thisapproachisnotvalid for software generators. For instance, the FatalErrorroutine may not bedeclared as a routine in the generator or a standard library, but instead exist only inthe generated program. Hence, the declaration of FatalErrormust be non-hygienicso that the code fragment generated by CreateForAllInFilecan access it.

**Generation Scoping**
**GenerationEnvironments**

Because of the differences between macros and software generators, we cannothopetoachievethesamedegreeofautomationforsoftwaregeneratorsaswithhygienic lexically-scoped macros. Nevertheless, we can still do better than manuallygenerating new symbols, as in example (3) of Section 2. This is the purpose of genera-tion scoping. Generation scoping is a mechanism that represents lexical environmentsinthegeneratedprogramasfirst-classentities.Inthisway,thegeneratorhascontrolofthe scoping of the generated program, beyond that offered by the target programminglanguage.

To support lexical environments as first-class entities, generation scoping adds anew keyword, environment, to the language in which the program generator is writ-ten. Its syntax is:

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

wherestatement contains one or more quoted expressions. The generation-environment is an expression that yields a value of type ENV. ENV is a type used torepresent environments and only has a constructor and equality function defined (i.e.,we can only create newvaluesoftype ENVandcomparethemwithexistingones).Theconstructor for environments, new_env, can take an arbitrary number of argumentswhose values are other environments. These environments become the parents of thenewly created environment (the child). All variable declarations in a parent becomevisible to the child environment. Like traditional scoping mechanisms, variable bind-ings of the child eclipse bindings with the same name in the parent.

An example use of environment in code implementing our example text file tra-versalfollows 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 fpisgenerated in a new environment—which becomes a child of an environment passedinto the function). At the same time, if the variable represented by mtbyteis gener-ated in the same environment as actions, they are visible to each other. This is thecasewith most straightforward uses of this function. Forinstance:

environment(e)result = CreateForAllInFile(global_env,**‘byte**,**‘(“file.txt”)**, **‘putchar(byte)**); (5)

Comparing code fragments (4) and (3), we can see why using environments ismoreconvenientthanmanuallyhandlingvariablesby creatingnewsymbols.Inpartic-ular,there are severalimportant 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 envi-ronmentstatement will be automatically added to the corresponding environment.Additionally,thegeneratorprogrammer doesnotneedtoexplicitlyretrievethebindingfor 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 acode fragment is generated in the intended environment, it can later be used withoutproblemsinalocalcontext,evenifthelocalcontextcontainsdifferentbindingsforthesame identifiers. For example, in code fragment (5), above, if global_envhas theintended declaration for, e.g., FILE, it will not subsequently matter if the generatedcode fragment is output in the middle of a function where FILE means something dif-ferent. The reference will always be to the FILE type variable defined in the environ-mentrepresented by global_env.

**2.** The alternation between executed and generated code is avoided. There is no needto unquote code just to supply a unique symbol name.

**3.** Declarationsaretreatedasagroup,insteadofi ndividually.Intheaboveexample there is only one variable declared, so this is not really an advantage. In quoted codewith several generated declarations, however, handling environments is easier thanhandling all new symbols individually. Of course, the same grouping effect could beachieved by using a mapping data structure in the generator code. The advantage ofgeneration scoping is that the data structure is now integrated in the language andinsertions and lookups are implicit (i.e., the programmer never has to specify them—seethe first point above).

**Implementation Issues**

Itisperhapsworthstressingagainthatthemain advantageofgenerationscopingisthat the generator programmer is relieved of the responsibility of adding declarations to environments and looking up identifier bindings in those environments. That is, theimplementation of quote will determine whether a generated identifier is actually adeclaration(ofavariable,function,type,etc.)orarefer encetoanexistingentity.Eachenvironment has a symbol table and a collection of pointers to the parent environ-ments. In case an identifier represents a declared entity, it is added to the current envi-ronment's symbol table together with a corresponding generated unique name for thedeclared entity. When a generated identifier is a reference, it will be looked up in theappropriateenvironment'stableand,ifitisnotthere, intheparentenvironmentsrecur-sively.[4] The result of the identifier lookup is the unique generated name for the match-ing declaration. In this way, no accidental reference to the wrong variable, type,function,etc.,canoccur,aslongas theenvironmentsaresetupproperly.

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

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

Inmostprogrammingenvironments,[5]itisimpossibleto tellbeforethecodeisgen-eratedwhetherthegeneratedcodedeclaresnewvarorref erstoanexistingvariableofthe same name. If the parameter type holds the type specifier**'int**, then newvarisbeing declared. If, on the other hand, it holds the operator **'***, it is not. This problemhas been studied extensively in the hygienic macro community and the commonapproach is to employ a "painting" algorithm that marks each identifier with the envi-ronmentwhereitwascreated.Itiseasytoadaptthisappro achtogenerationscoping:

After all the code has been generated, the marked declarations can be matched tomarked references (assuming they came from the same environment). Remaining ref-erencescanthenbejustunmarked,sothattheybecomefr eereferencesandcanrefertoexternally declared symbols. A more thorough discussion on implementing a "paint-ing" algorithm for program generation can be found in [11].

**GenerationScopinginDiSTiL**

Generationscopingwasimplementedaspart ofIP(IntentionalProgramming)[13],ageneral purposetransformationsystemunderdevelopmentby MicrosoftResearch.Itwas subsequently used to build the DiSTiL software generator [14] as a domain-spe-cific extension to IP. DiSTiL is a generator that follows the GenVoca [3] design para-digm.GenVocageneratorsareaclassofsophisticateds oftwaregeneratorsthatsynthesize high-performance, customized programs by composing pre-written compo-nents called layers. Each layer encapsulates the implementation of a primitive featurein a target domain. The DiSTiL generator is essentially a compiler for the domain ofcontainer data structures. Complex container data structures are synthesized by com-posing 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 opera-tion is generated by having each layer manufacture a code fragment (that is specific totheoperationwhosecodeisbeinggenerated)andbyas semblingthesefragmentsintoacoherent algorithm.

Generationscopingwasindispensableintheimplemen tationofDiSTiL.Evenrela-tively short DiSTiL specifications (around 10-20 lines) could generate thousands oflines of optimized code. Due to the complexity of the generated code, as well as theflexibility of parameterization (a layer could be composed with a wide variety of otherlayers), maintaining correct scoping for generated code would have been a nightmare without generation scoping. In fact, initially we had attempted to implement DiSTiLwith manual resolution of generated references (by generating unique symbols, as incode fragment (3)). The sheer difficulty of this task was what motivated generationscopingin the first place.

Generation scoping is used in DiSTiL not only to ensure the correctness of refer-ences to global declarations (e.g., library functions) but also to overcome the scopinglimitations of the target language (C). With generation scoping, DiSTiL effectivelymanages different namespaces for every layer in a composition. In this way, there areno clashes between identically named variables introduced by different layers (or dif-ferent instances of the same layer). At the same time, the code is simplified by havingnamespaces connected appropriately so that generated code can access all the requireddeclarationswithout explicit qualification.

DiSTiL data structures consist of three distinct entities: a container, elements, anditerators (called cursors). Generated variables are grouped together into a commonenvironmentaccordingtotheentitytowhicht heyarerelated.Forinstance,alldeclara-tions related to the cursor part of a doubly linked list will belong in a single generationenvironment.Thesevariablesneednotbelo ngtoasinglelexicalcontext.Forexample,variablesina nenvironmentmaybeglobal,orlocal,orfieldsofarecor dtype.Thus, variablesofanenvironmentcouldbelongtoslicesofma nydifferentlexicalcontextsinthe generated program. In this way, the environment acts as a generator-managednamespacemechanism for the target language.

Consider the following organization used in DiSTiL (and, in fact, also in P3). Ingeneral, there is a many-to-one relationship between cursors and containers (i.e., therecan be many cursors—each with a different retrieval predicate—per container). Sousing a single generation environment to encapsulate both cursor and container datamembers is not possible. Instead, separate environments are defined for every cursorand container. The ContGenericenvironment encapsulates element data members(because element types are in one-to-one correspondence with container types) andgeneric container-related variables (including the container identifier). The Curs-Generic environment encapsulates generic cursor-related variables (including thecursor identifier). By making ContGenerica parent of CursGeneric, code foroperations on containers (which do not need cursors) can be generated using theContGenericenvironment, while code for operations on cursors (which also refer-ence container fields) is generated using the CursGenericenvironment. Figure 1(a)depicts this relationship.

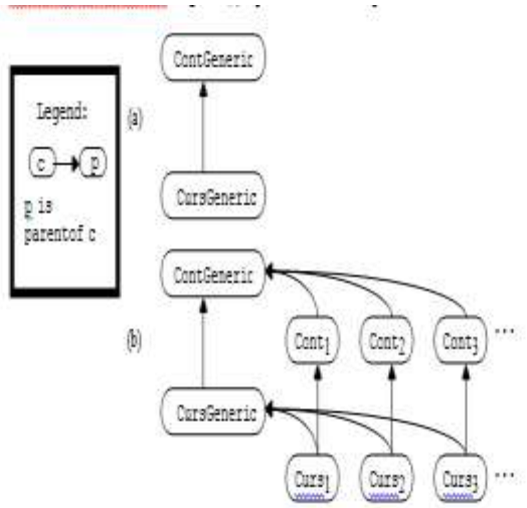


Figure1:HierarchicalOrganizationsofEnvironmentsi nDiSTiL

Asmentionedearlier,ahallmarkofGenVocal ayersisthattheyencapsulaterefine-ments of multiple classes. Each DiSTiL layer refines cursor, container, and elementtypesbyaddinglayer-specificdatamembers.Thedatamembersaddedtothec ontainer and element types by layer $L_i$ are encapsulated by environment Cont$_i$ which is a childofContGeneric.Similarly,datamembersaddedb y$L_i$tothecursortypeareencapsu-latedbyenvironmentCurs$_i$whichisachildofbothCurs GenericandCont$_i$(because cursors of layer $L_i$ reference layer-specific container-data members as well aslayer-specific cursor data members). Figure 1(b) shows this hierarchical organizationofenvironments.

To illustrate these ideas, consider an ordered doubly-linked list layer. This layerwouldrefineelementsbyaddingnextandprevfiel ds,andwouldrefinecontainersbyaddingfirstandlastfiel ds.ThisrefinementcanbeaccomplishedbyaRefine-Types()method:elem_type,cont_type,andcurs_typea recodefragmentsthatrespectively define the set of variables (data members) in element, container, and cur-sorclasses.WhenRefineTypes()iscalledwiththesecodefr agmentsasparameters,thenext,prev,first, andlastfields areadded totheelement andcontainer types.Asthesefieldsarealwaysusedtogether,theyared eclaredwithinasingleenvironmentCont(which is equal to some Cont$_i$of Figure 1):

voidRefineTypes( CODE *elem_type, CODE *cont_type, ENV Cont) {environment(Cont) {
*elem_type=‘{$(*elem_type)**;element *next, *prev; }**;
*cont_type=‘{$(*cont_type)**;element *first, *last;**

```
};
}
}
```

ItiscommoninacompositionofGenVocalayersthatasinglelayerappearsmulti-ple 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 layeradds its own fields to the element and container types. Maintaining the distinctionamong these fields (so that the code for the j-th list will only reference its own fieldsnext$_j$, prev$_j$, etc.) is simple using generation environments as organized in Figure 1.EachcopyofthelistlayerwillhaveitsowngenerationenvironmentsCont$_j$andCurs$_j$,andallcodegeneratedbythatcopywouldalwaysusetheseenvironmentvariables. For an example, consider the Remove method for ordered doubly-linked lists,appearing below. Let Remove_Codebe the code that is to be generated for removinganelementfromacontainer.TheRemovemethodforordereddoubly-linkedlistsaddsitscode(tounlinktheelement)whenitis called(thecodethatactuallydeletestheele-mentisaddedbyanotherlayer).Thus,givenRemove_CodeandtheenvironmentCurs(equal to some Curs$_i$of Figure 1), Remove() adds the unlinking code where thenext,prev,etc. identifiersarebound totheircorrect variabledefinitions.

```
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; };
}
}
```

Notethatthebindingsofidentifierscursor,container,andnextinthistem-plateexistinthreedifferentgenerationenvironments:containerisinContGen-eric,cursorinCursGeneric,andnextinCont$_i$.Nevertheless,allofthemcanbe accessed from environment Curs (following its parent links), so this is the only envi-ronment that needs to be specified. Note also that there are two generated temporarydeclarationsinthiscodefragment,whichare completelyprotectedfromaccidentalref-erence.

This example is convenient for demonstrating the benefits of generation scoping.Weattempttoshowthesebenefitsbyspeculatingonthealternatives.Clearlytheabovecode fragment has many external generated references, so default hygiene is not reallyan option. The generator writer has to explicitly create new symbols (as in code frag-ment (3)) for the declarations of container, cursor, etc. (not shown). Instead ofmanaging all the new symbols individually, the generator writer could set up a datastructure in the generator (unquoted) code to maintain the mappings of identifiers tovariables. 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 refer-encesneedtobelookedupexplicitly,thecodewouldbe muchmorecomplicated.Onecan add some syntactic sugar to make the code more appealing. For instance, we canuse $$(ds, id) to mean "unquote and lookup identifier id in bindings data structureds". Similarly, we can use $%(ds, id) to mean "unquote and add variable id inbindingsdata structure ds".Even then, thecode would bepractically 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); };
}
```

Asoutlinedearlier,generationscopingimprovesoverthiscodeinthreeways: First,noexplicitdatastructureinsertions/lookupsneedtobeperformed(e.g.,therearenoc$$and $%operators). Second,noexplicitescapesareintroduced—thereisnoalter-nationbetweenquotedandunquotedcode.Third,thegroupingofvariablesisimplicit—thereis no need to repeatedly refer to a data structure likeds.

**Related Work**

Givenourpriordiscussionofhygienicmacros,herewe willonlytouchuponafewotherpieces of related work. Theenvironmentsusedingenerationscopingaresimila rtosyntacticenvironments inthesyntacticclosureswork[4][9].Insyntacticclosures,environmentsarefirst-classentitiesandcodefragmentscanbeexplicitly“closed”inalexicalenvironment.Never-theless, there are significant differences between the two approaches: Syntactic clo-suresenvironmentscanonlycapturethesetofvariablesthatarelexicallyvisibleata specific point in a program.[6] In contrast, our environments can be arbitrary collectionsof bindings (i.e., smaller sets of lexically visible variables) and can be organized hier-archically. More importantly, however, declarations are added to generation scopingenvironments implicitly by generating (quoting) code that declares new variables.Thus, our approach is much more automated than syntactic closures and is ideallysuited to software generators (where the lexical environment is being built while codeis generated). Also, generation scoping can be used to implement the hygienic, lexi-cally-scoped macros of [7], unlike syntactic closures, which cannot be used to imple-menthygienic macro expansion, as explained in [7].

Generation scoping is concerned only with maintaining correct scoping for gener-ated code fragments. Other pieces of work deal with various other correctness proper-ties of composed code fragments. Selectively, we mention some work on the problemofensuringtypecorrectnessforgeneratedprograms,bothfortwo-stagecode[12](i.e.,generator and generated code) and multi-stage code [15] (i.e., code generating codethat generates other code, etc.).

## II. CONCLUSIONS

Program generation is a valuable technique for software development that willbecome progressively more important in the future. In this paper we have shown howtoaddressthescopingissuesthatariseinsoftwaregenerators.Wehavepresentedgen-eration scoping: a general-purpose, domain-independent mechanism to address allscopingneedsofgeneratedprograms.Generationscopingcanmakewritingandmain-taining software generators easier. Its capabilities were proven in the implementationofthe 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 infuture software development. We consider generation scoping to be a valuable lan-guage mechanism for generator writers and hope that it will be adopted in even moreextensiblelanguages and transformation systems in the future.

## REFERENCES

[1] D.Batory,G.Chen,E.Robertson,andT.Wang,“WebAdvertisedGeneratorsandDesignWizards”,InternationalConference onSoftware Reuse(ICSR), 1998.

[2] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-SpecificLanguages”,International ConferenceonSoftware Reuse(ICSR), 1998.

[3] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical SoftwareSystems with Reusable Components”, ACM Transactions on Software Engineering andMethodology,October 1992.

[4] A. Bawden and J. Rees, “Syntactic Closures”. In Proceedings of the SIGPLAN ‘88 ACMConferenceon Lispand Functional Programming,86-95.

[5] S. P. Carl, “Syntactic Exposures—A Lexically-Scoped Macro Facility for ExtensibleLanguages”. M.A. Thesis, University of Texas, 1996. Available through the Internet atftp://ftp.cs.utexas.edu/pub/garbage/carl-msthesis.ps.

[6] W.Clinger,J.Rees(editors),“TheRevised$^4$ReportontheAlgorithmicLanguageScheme”.Lisp Pointers IV(3),July-September 1991, 1-55.

[7] W. Clinger and J. Rees, “Macros that Work”. inConference Record of the EighteenthAnnual 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 Lispand Symbolic Computation, 5(4), December 1993, 83-110.

[9] C.Hanson,“ASyntacticClosuresMacroFacility”,LispPointersIV(4),October-December 1991, 9-16.

[10] E. Kohlbecker, D.P. Friedman, M. Felleisen, and B. Duba, “Hygienic Macro Expansion”,inProceedingsoftheSIGPLAN‘86ACMConferenceonLispandFunctionalProgramming,151-161.

[11] J. Rees, “The Scheme of Things: Implementing Lexically Scoped Macros”, Lisp PointersVI(1), January-March 1993.

[12] T. Sheard and N. Nelson, “Type Safe Abstractions Using Program Generators”, OregonGraduateInstitute Tech. Report 95-013.

[13] C. Simonyi,“TheDeath of ComputerLanguages,theBirth

ofIntentionalProgramming”, NATOScienceCommitteeConference,1995.

[14] Y.SmaragdakisandD.Batory,“DiSTiL:aTransformationLibraryforDataStructures”,USENIXConferenceonDomain-SpecificLanguages(DSL),1997.

[15] W. Taha and T. Sheard, Multi-stage programming with explicit annotations, ACM Symp.PartialEvaluationandSemantics-BasedProgramManipulation(PEPM‘97),1997.