The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

DiSTiL: a Transformation Library for Data Structures

Yannis Smaragdakis and Don Batory
The University of Texas, Austin

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

DiSTiL: a Transformation Library for Data Structures

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

Abstract

DiSTiL is a software generator that implements a declarative domain-specific language (DSL) for container data structures. DiSTiL is a representative of a new approach to domain-specific language implementation. Instead of being the usual one-of-a-kind standalone compiler, DiSTiL is an extension library for the *Intentional Programming* (IP) transformation system (currently under development by Microsoft Research). DiSTiL relies on several reusable, general-purpose infrastructure tools offered by IP that substantially simplify DSL implementation.

1 Introduction

In the past few years, the popularity of domain-specific languages has steadily increased. Such languages offer concise ways of expressing complex, domain-specific concepts and applications, which in turn can offer substantially reduced maintenance costs, more evolvable software, and significant increases in software productivity [Kie96, Bat97b, Due97, Die97]. Our research is in the design and implementation of software generators. Generators are compilers for domain-specific languages. Our particular research emphasis, which largely has distinguished our work from others in the generator community, is on generators that synthesize implementations of declarative specifications of domain-specific constructs through component composition. Thus, an integral part of our methodology, called *GenVoca* [Bat92], is to identify the fundamental building blocks of software construction for a target domain. GenVoca components actually define sophisticated program transformations that convert domain-specific language constructs into their host language implementations. In this way, a domain-specific program is reduced (transformed) into an executable host language program by a series of transformations, where each transformation corresponds to a component in a domain-specific transform library. The advantage of this approach is *scalability*: a small number of GenVoca components can be composed in vast numbers of ways to yield huge families of distinct implementations for domain-specific constructs [Bat93].

From our experience, only 30% of the effort in building GenVoca generators is actually spent on coding components (i.e., writing program transformations). The majority of the time is spent on infrastructure development (i.e., developing tools for representing programs as data, writing and composing components, validating component compositions, etc.). This overhead has substantially hindered the development of GenVoca generators under realistic time and funding constraints. Tools are needed both to reduce the effort in building generators and to promulgate their use.

In this paper, we present DiSTiL — a software generator for the domain of container data structures. The language of DiSTiL extends the C programming language with domain-specific constructs for specifying complex data structures declaratively. When a DiSTiL program is "compiled", the declarative data structure specifications are replaced by their C implementation, which is specified by a composition of DiSTiL components. In the following, when no confusion can arise, we will use the name DiSTiL to also mean the domain-specific language that the generator implements.

The overall design of DiSTiL is similar to that of the previously built GenVoca generators P1 and P2 [Sir93, Bat92, Bat97b]. However, its implementation is radically different. Instead of being a one-of-a-kind generator that is totally specific to the data-structure domain, DiSTiL is implemented as a *transformation library* for the *Intentional Programming (IP)* system [Sim95], which is currently under development by Microsoft Research. IP provides a domain-independent implementation substrate and tools that have substantially simplified the implementation of DiSTiL.

The novelty of DiSTiL is that it is the first truly complicated domain-specific generator that was built using IP. We found that IP, by itself, lacked certain features that were required to simplify DiSTiL's development. In this paper, we describe IP, our general-purpose extensions to

it (called *generation scoping* — a general-purpose hygienic code generation facility [Sma96]), and DiSTiL — the language and its implementation. We argue that IP's infrastructure is well-suited for building compilers for DSLs, and that it substantially reduces the effort needed for their construction.

2 Microsoft's Intentional Programming (IP) System

Many domain-specific languages can be implemented as domain-specific extensions to existing programming languages. Up to now this approach had been examined mostly in the context of functional languages. LISP [Ste90] and its variants (e.g., Scheme [Cli91b]) have powerful language extension mechanisms (under the rather misleading name "macros") that are well-suited for DSLs. Unfortunately, using LISP as a software generator infrastructure has undesirable consequences for many applications: LISP's powerful meta-programming system is trapped inside a hard to optimize functional language. The syntax is strange, and many operations impose an unnecessary performance overhead to the unsuspecting user. Furthermore, every program has to pay the cost of garbage collection.

Nevertheless, there is no fundamental limitation preventing the application of the extension approach to other programming languages. The essential elements are a language extensibility mechanism and a powerful meta-programming system (i.e., constructs for representing programs as data). We note that the issue of extending imperative languages has been addressed before (e.g., [Wei93]). Microsoft's IP, however, is the first integrated programming environment specifically designed with language independence and language extensibility in mind. The next two sections describe the Intentional Programming infrastructure and the machinery used to build DiSTiL as a language extension.

2.1 The Intentional Programming Environment

IP [Sim95] is a language-independent programming environment. Language independence is achieved in IP by representing all source code (in whatever language) as an *abstract syntax tree (AST)*. Nodes of an AST are called *intentions* and correspond to semantic constructs of a language. Examples of intentions include if-statements, for-loops, type declarations, assignment-statements, etc. Thus, libraries of intentions can be created for representing programs in various programming languages. Many intentions are themselves language-independent; i.e., their semantic meaning (but not their syntax) is shared in many languages [Vil97]. The if-statement, for example, with a general form of an `if` operator and a 3-tuple argument <`boolean-expression`, `then-statement`, `else-statement`>, is a standard "intention" in virtually all programming languages.

The syntax (or external representation) of an intention is user-controlled. (For example, the syntax of an if-statement in C is different than in Pascal). This variability is captured by *unparsing* methods that are associated with intentions. Unparsing is the process of displaying an AST to the user for direct manipulation. In IP, unparsing is more than just pretty-printing — it is two-dimensional and fully graphical. That is, an intention may be represented as a complex image and can be positioned accordingly. This offers the possibility of developing special, non-ASCII notation for domain-specific languages (e.g., mathematical symbols). For instance, it is relatively straightforward to make the combinatorial intention `choose(n,m)` to unparse as $\binom{n}{m}$, in the usual mathematical notation.

The extensibility of IP lies in its ability to define new intentions and to define enzymes. New intentions express domain-specific programming constructs. In effect, adding intentions is equivalent to extending the grammar of the host language. *Enzymes* are transformations on ASTs, i.e., functions that replace an AST with another AST. Using enzymes, new intentions can be transformed into existing ones, effectively extending the language. The programming interface for transformations is procedural — pattern-based extensions are also provided as a higher-level concept. Although the interface for transformations is not yet final, it includes operations to traverse and create ASTs as well as operations to manipulate semantic information (for instance, variable and expression types). The semantic information elevates the interface above the usual syntactic macros (as, for instance, in LISP). A distinguishing factor between IP and existing transformation systems (e.g., TXL [Cor91], Refine/Dialect [Rea86, Rea89]) is the power of the transformation engine itself. The goal of IP is to have complete, industrial-strength languages implemented entirely as collections of enzymes. A complex transformation infrastructure is in place to accommodate this requirement. For example, sophisticated scheduling of transformations according to their dependencies is performed. Thus, transformations that were designed independently can be applied in such a way that they will not interfere with each other. Additionally, transformations may have non-local effects following a

strictly defined protocol of permissions and information passing.

IP uses parsers for importing programs already written in conventional programming languages. This conversion is one-way, however. After a program is expressed as an AST it can be edited directly. IP provides a powerful structure editor for this purpose. Users edit unparsed versions of a program, but all text-like editing commands directly manipulate the underlying AST. This enables enzymes to be applied at editing time. For instance, it is possible to use a standard syntactic rewrite like a DeMorgan transform of boolean expressions both as an editing enzyme and as a compilation enzyme. A user can select/highlight a boolean expression during editing and invoke the DeMorgan enzyme (for instance, to turn an OR expression into an AND expression for readability). The same enzyme could be automatically applied by IP during compilation (for instance, as an optimization).

The IP environment is fully configurable. Opening a source code document that refers to domain-specific intentions can cause new commands to be added in the environment window menus, new buttons and toolbars to appear, etc. In this way, the author of a domain-specific language implementation can also customize the development environment for language users.

It should be clear from the above that IP is an appropriate platform for implementing domain-specific languages. It allowed us to concentrate on the task of devising powerful data structure abstractions without worrying about infrastructure support. At the same time, DiSTiL is an example of the applications that IP was intended to support. It is a powerful domain-specific language that can be transparently integrated into the system as an extension and take full advantage of its transformational capabilities.

2.2 Generation Scoping

As a transformation library, DiSTiL deals extensively with manipulating code fragments. To facilitate our work, we designed the *generation scoping* mechanism: a meta-programming system for IP in which DiSTiL components are expressed. The system consists of *code template operators*, similar to the `backquote` and `comma` operators of the LISP language. Generation scoping is a general-purpose facility oriented towards large-scale code generation and was not designed to support only DiSTiL. This section reviews its essential features and applicability. A more complete discussion is in [Sma96][1].

Meta-programming systems are notorious for introducing ambiguities regarding the environment in which generated variable references are resolved. Programming languages usually determine the meaning of identifiers using their position in a program. Generated programs, however, are usually composed from small fragments. In this case we are usually unaware of the final position/scope of a fragment in the generated code. Thus, it becomes a bad practice to let the position of identifiers in the final program determine their meaning — erroneous references can easily be introduced.

This problem has been studied extensively in the context of macro expansion and systems that address it are called *hygienic* (e.g., [Koh86], [Cli91a], [Wal97]). A complete solution comes in the form of hygienic, lexically-scoped macros (see [Cli91a]). As we explain in [Sma96] the standard macro-expansion methods are not directly applicable to software generators. Instead we had to develop the generation scoping system which is in many ways similar to the lexically scoped macros machinery of [Cli91a] but is better suited for generator development. The difference is that the environments which determine identifier bindings become first-class objects and can be manipulated directly. Most importantly, environments can be organized hierarchically into directed graphs with every environment having access to all others reachable in the graph. This adds significant power: instead of a hygienic mechanism for small, self-contained units (macros) we get a method that can handle complex scoping in the generated program, independently of the target language [Sma96].

Generation scoping includes standard operators to designate code templates and escapes from them: `quote`, written `‘`, and `unquote`, written `$`. Also it allows explicitly closing a code fragment when it is generated in an environment where identifiers have specific meanings. This is done using the `environment` operator around one or more code templates (quoted code fragments). For instance, the code fragment:

```
environment(E)
  Output(‘int i = 0;);
```

will generate code that declares variable `i` as an integer and initializes it. The code is generated in environment `E` (assumed to have been declared before). The system can detect that `i` is being declared in this fragment.

---

1 Generation scoping is actually yet another example of an embedded domain-specific language. In this case, it is a language to express and compose lexically-scoped code fragments.

Operator	Description
`'<code_template>`	***Quote operator.*** Generates a code fragment according to `code_template`. Similar to LISP `backquote`.
$`<code>`	***Escape operator.*** Executes `code` inside a quoted fragment.
environment `(<Env_id>) <code>`	Evaluates quoted code in a given "environment". Variable references in quoted code will be resolved relative to this environment.
SetParent `(<Env_id1>,<Env_id2>)`	Organizes environments hierarchically. Quoted code in the child environment will be able to view variables in parent environment as well.
alias `(<tree_expr>, <variable>)`	Sets the value of identifier `variable` to `tree_expr`. Every time `variable` appears in quoted code (in the same environment as the **alias** command) it will be replaced by `tree_expr`.

Figure 1: Generation Scoping operators

Therefore, `i` now becomes a declared variable in environment `E` and future occurrences of identifier `i` in the same environment will refer to the variable declared above. By explicitly choosing the environment (scope) of a generated code fragment we can completely dissociate variable scoping from the variable's position in the generated program. This ability is used in DiSTiL to ensure that identifiers are bound to the correct variables.

Other important generation scoping operators include `SetParent` and `alias`. `SetParent` is used to organize environments hierarchically, in much the same way lexical scopes are hierarchically organized by nesting. That is, a "child" environment has access to variables introduced in a "parent" environment but variables with the same name in the "child" eclipse variables in the "parent". `alias` is used to introduce symbolic names for complex generated expressions. A table of the main generation scoping operators is shown in Figure 1. Examples of the use of generation scoping can be found in [Sma96].

3 DiSTiL

3.1 Motivation

A central problem in software development is the creation, maintenance, and evolution of data structures. Initially, with a partial understanding of the system requirements, a programmer invents data/storage structures to address a perceived need. These data structures are then either implemented by hand (a tedious process) or taken from a component library (e.g., STL [Ste95], or the Booch components [Boo87]) . It is quite rare, however, that the projected requirements are accurately reflected in the first design (and even if they are they may change in time). Altering a data structure is often costly; interfaces to different data structures can vary widely, and thus may require extensive source code modifications, leading to yet another (expensive) round of coding and debugging.

We believe that data structures should not have ad hoc interfaces. Instead they should provide a stable, well-designed interface that insulates applications from changes to data structure implementations. This, incidentally, is also the premise behind the C++ Standard Template Library — STL [Ste95]. STL, however, does not take this idea to its logical conclusion. Compatibility is limited to a specific level, while different kinds of STL data structures (e.g., sequences and associative containers) still have different interfaces. This significantly restricts the interchangeability of data structures. Moreover, STL only offers rather elementary data structures. Complex data structures must be implemented by hand. For example, if elements of a container are to be simultaneously linked onto two key-ordered lists, or a key-ordered list (for sequential accessing) and a hash-table (for fast key accessing), STL users have to either (a) write their own, customized STL component to accomplish the task or (b) devise ways of integrating existing STL components manually, and write source code that maintains the correctness of these structures when element keys are updated. Both approaches are unpleasant and preclude the ease of evolving data structure implementations.

We believe a different approach is needed — one based on a declarative language that is specific to the domain of data structures, rather than using typical component libraries (link libraries, macro/template libraries, binary components, etc.). Our language, called DiSTiL, extends the C programming language with declarative

goto_first: Set cursor to first legal position
goto_next/**goto_prev**: move cursor forward/back
goto_nth : move cursor to n-th ordered position
is_legal : is the cursor in a legal position?
foreach : iterate over all elements in cursor range
insert/**delete**: insert/delete current element
getrec/**ref**: return current record or single field
update : change value of field in current record

DiSTiL cursor operations

open_cont : open/initialize the container
close_cont: close the container
size : return the total number of elements
is_full : is the container full?

DiSTiL container operations

Figure 2: Set of DiSTiL operations

statements/operations on data structures. These statements isolate the actual data structure implementation from the application itself, thereby allowing radically different implementations of data structures to be evaluated without requiring modifications to the application's source code [Bat93-95a]. It is the responsibility of the compiler to ensure efficiency. As an added benefit, the ability to reason about programs is greatly enhanced, often allowing for automatic design checking mechanisms and high-level optimizations [Bat97a].

3.2 The DiSTiL Programming Language

All data structures in DiSTiL are modeled using *containers* and *cursors*. Essentially, we view all data structures in our universe as pairs of containers and cursors (iterators). These two facets explicitly decouple the notion of element storage from that of element access. The cursor-container pair provides the only interface the user has to a data-structure. When viewing a data structure as a collection of elements, the most important operation that can be defined is that of a *selection*. A selection gives the user a way to define a subset of a collection according to a certain *selection criterion*. In the case of DiSTiL, the effect is achieved by assigning *selection predicates* to cursors. Such predicates may express an arbitrary relation on the values of the fields of stored elements. A cursor is guaranteed to only access elements satisfying its selection predicate. Additionally the user may specify the order of retrieval as an ordering relation on element fields. *The mechanisms of order and predicate specification are the only way for the user to control element access*. The system is otherwise free to implement data structure operations in any semantically correct way.

An abbreviated example of container-cursor specifications in DiSTiL is given below.

```
typedef struct {
   char[8] phone;
   char[31] name;
} phonebook_record;
 // C struct declaration

Container (phonebook_record) cont1;
 // abbreviated container declaration

Cursor (cont1, phone == "4783487")
   curs1;
 // cursor declaration
Cursor (cont1,
        name > "Sm" && name < "Sn",
        ascending(name)) curs2;
 // another cursor declaration
```

This example presents a phone-book data structure. We begin by declaring the record type for the elements as a C type. Then a container of elements and two different cursors on that container are declared. The first ranges over all elements (probably a single one) with phone number "4783487". The second selects all records in the data structure with a name that begins with "Sm", in ascending order.

DiSTiL offers a standard set of operations on containers and cursors, regardless of the actual data structure implementation. The code fragment below illustrates the `foreach` construct, which is used to iterate over elements selected by a cursor. The element in the current cursor position can be examined, updated, or deleted using standard cursor operations (a summary of all DiSTiL operations appears in Figure 2).

```
foreach (curs1) {
       // for each selected entry
   printf("%s", ref(curs1, name));
       // print name
   update(curs1, phone, "4718731");
       // change phone number
}
```

The interface to DiSTiL data structures does not depend on the actual data structure used. This way DiSTiL programs can stay the same for different data structure implementations. For example, the phone-book could be implemented as an ordered linked list, a binary-tree, a hash-table, or any other structure or combinations of structures. Nevertheless, the above program fragment would remain the same across all different implementations.

It is worth noting that DiSTiL cursors and containers can be composed arbitrarily. Thus we can have a data structure storing cursors, or containers, or containers of containers, etc. This can yield interesting data structure configurations in their own right (i.e., we can explicitly create complex indexes to data structures using containers of cursors).

In effect, we are giving a relational front-end to container data structures [Bat93]. Using relational abstractions to hide data structure details is not new (e.g., see [Coh89, Coh93]), but our ability to couple relational abstractions with component technologies to generate vast families of efficient implementations is novel. In fact, readers might note that our work parallels recent advances in *object-oriented databases (OODBs)* to make them more extensible. Extensible DBMS technologies were developed in the mid-1980s, and one of the original projects was Genesis. Genesis was the first (our first) GenVoca generator [Bat88a-b]: it was also the first technology for assembling relational database systems from components.[2] Combinations of Genesis components produced different relational DBMSs with vastly different implementations. Our work on P3 [Bat97b], and now DiSTiL, can be viewed as a continuing evolution of the Genesis work. It exposes the relationships between domain-specific languages, component-based generators, and the synthesis of high-performance domain-specific software. We could add many more features that would put DiSTiL on par with the embedded capabilities of object-oriented databases, but this has not been our research emphasis or interest.

3.3 Implementation Specification Using Component Compositions

DiSTiL applications can define the features of their data structures and declare how their implementations are to be generated. At the top of a DiSTiL program is a specification of how DiSTiL constructs are to be implemented. This specification, called a *type equation*, is a named composition of DiSTiL components. Each DiSTiL component implements a sophisticated program transformation that encapsulates a primitive building block of container data structures.

As an example, we will show how the phone-book of Section 3.2 can be implemented as a hash table (DiSTiL `Hash` component) in conjunction with a red-black tree (`Tree`) with elements that are allocated when needed (`Malloc`) from main memory (`Transient`). The corresponding type equation appears below. To alter or evolve the data structure merely requires altering the container's type equation and re-compiling; no other source code modifications are needed.

```
typeq (phonebook_record,
       Hash(Tree(Malloc(Transient))))
   typeq1;
 // type equation specification
```

The actual container that will hold the elements is declared below. At container declaration time it is specified that the hash table is organized by phone number (for fast lookups by phone) while the red-black tree has the name field as its key (for fast retrievals of alphabetically ordered names). In database terminology, we are organizing our data with a red-black tree index on the name field and a hash table index on the phone.

```
Container (typeq1,
           (Hash(phone), Tree(name)))
   cont1;
 // container declaration
```

Using component compositions to express complex entities (see [Nei80]) is a hallmark of the scalability of GenVoca. Customized software systems implement *m* features out of a possible *n* features. Rather than building an exponential number of monolithic systems that offer unique sets of features, one should build systems by composing primitive components that encapsulate individual features. Thus, by making feature combinatorics explicit, it is possible to describe vast families of systems with a relatively small number of components. The set of components that implement the same interface is called a *realm*. A realm is, in effect, a library of plug-compatible and interchangeable components. A summary of DiSTiL realms and components can be found in Appendix A.

---

2 A similar approach to relational database system extensibility was later (and independently) developed and deployed in IBM's Starburst project [Haa90].

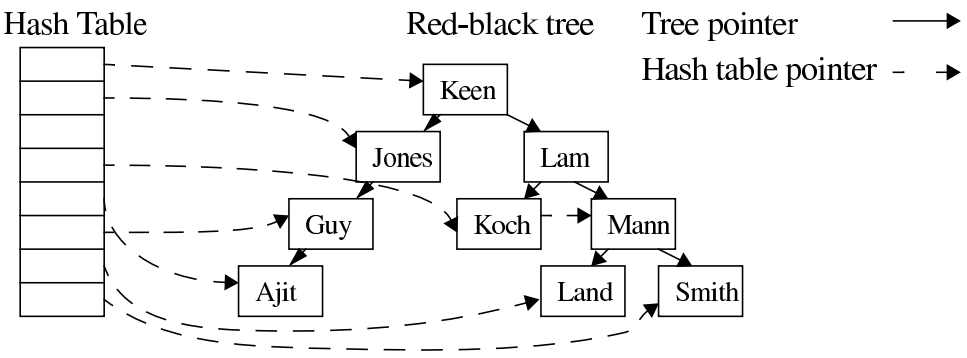

Figure 3: Phone book example

3.4 The DiSTiL Generator

For a given type expression, it is the responsibility of the DiSTiL generator to apply the transformations prescribed by each of the participating components. The generator will thus replace all DiSTiL operations by their corresponding C implementation. The resulting C program can then be compiled and executed. As a higher level language, DiSTiL offers significant leverage to its compiler, allowing it to perform powerful optimizations and error checking.

We illustrate DiSTiL's actions with a concrete example. The cursor definitions of Section 3.2 are replicated below for quick reference.

```
Cursor (cont1, phone == "4783487")
   curs1;
Cursor (cont1,
        name > "Sm" && name < "Sn",
        ascending(name))
   curs2;
```

Consider any of the DiSTiL cursor operations on `curs1` when the container uses the type expression of Section 3.3 (hash table and red-black tree indexes). The most efficient way to retrieve elements satisfying the predicate on `curs1` is to use the hash index on the phone number. DiSTiL can *statically* determine this by analyzing the predicate, estimating the cost of the retrieval using each available index and selecting the data structure that offers the lowest cost. For example, a `goto_first` DiSTiL operation[3] on `curs1` is implemented using code that hashes the given phone number and follows the hash table links until it finds the first element satisfying the desired predicate. Similarly, operations on `curs2` should be transformed only in terms of the red-black tree structure, since this is the most straightforward way to locate records lexicographically by name. If the composition describing our data structure changes (for instance, if we decide we want a second red-black tree in place of the hash table), DiSTiL will re-evaluate the cursor predicate and choose an appropriate way to implement its operations in the new layout without requiring any programmer intervention.

Figure 3 shows a possible run-time configuration of the data structure. Keeping it as a guide, let's see what happens if we change the phone number in one of our records (for instance, "Keen"). The change is one that would regularly be encountered in a realistic setting — people's phone numbers change. The most straightforward way to update the structure is to remove the affected element and re-insert it after the change has been performed. DiSTiL, however, can do better than this: Since only the phone number changes, the record can stay in its original place in the tree. The only structure that needs to be updated is the hash table, which is organized by phone number. DiSTiL detects this by analyzing the `update` operation according to the field being updated. Subsequently, the operation is transformed into code that performs the corresponding changes only to the primitive structures that actually depend on the changed field. In our case, the "Keen" record will remain at the root of the tree but its phone number will be re-hashed in the table possibly making the record to be linked in a different place. Again, the mechanism is straightforward. All DiSTiL components that support the `update` operation implement it according to the pseudo-code of Figure 4. The element is removed and re-inserted only if the update actually

3 The first six cursor operations of Figure 2 are retrieval operations (that is, they are implemented using the most efficient data structure available) while the last five will propagate through all components in a type expression.

```
CODE update(CODE field, CODE new_value)
{
   if (field == key_field)
      return `(
         {  $unlink();
            $update_code(field, new_value);
            $link();  });
   else return update_code(field, new_value));
}
```

Figure 4: Update operation (implementation dependent)

affects the key for this particular component. This optimization is a typical example of *partial evaluation*: the operation is specialized at compile time by exploiting a restriction on its (implicit or explicit) parameters. This is just one of the many cases where partial evaluation (in the form of specialization) is used in DiSTiL.

Additionally, the DiSTiL specification offers itself for easy checking of design consistency. A composition and the operations performed on it can be validated to ensure that they are meaningful. This is the role of the *design-rule checker* of DiSTiL. DiSTiL components come with higher-level knowledge about their properties (explicitly encoded using boolean attributes). This information is used to express domain-specific properties like "this component does not leave the cursor in a valid position after deletion" or "this component keeps track of the data structure size". Compositions are checked in two ways: The system ensures that all their components can co-exist and that they support all operations performed on the particular composition. In other words, compositions may be correct relative to a certain set of operations but not to another (for instance, a certain data structure may not support deletions). It is of interest to examine the results of the design rule checker application. The attributes associated with components are at a much higher level than regular static information (types) in programming languages. As a consequence, the checking mechanism can provide more informative, comprehensive and accurate error messages [Bat97a].

The previous examples are indicative of the flexibility afforded by DiSTiL through use of meta-level reasoning about the code it produces. DiSTiL components are "intelligent": They exchange information to coordinate their actions. This interaction can make code easy to evolve (DiSTiL handles the changes automatically) even though the actual model of operation (for instance, the structures actually used to implement a retrieval) has changed. The ability to process predicates and reason about the best way to implement the associated operations is what sets DiSTiL apart from usual data structure libraries. In short, cursor actions in DiSTiL are specified *intentionally* (declaratively) instead of operationally. The system transforms specifications into efficient code using its knowledge of the characteristics of the given data structure. Such knowledge may include the number and kind of indexes on the stored elements and information specific to each of the data structure components (for instance, that a binary tree is an ordered structure). Additionally, the domain-specific knowledge encoded in DiSTiL components enables higher-level error checking of component compositions. With DiSTiL the level of programming is effectively raised, resulting in code that is much easier to understand and that can straightforwardly evolve to match changing needs.

3.5 Generator Implementation

The DiSTiL generator is a 10Kloc (thousands of lines of code) library that operates on top of the Intentional Programming system. The way to interface with user programs is through the use of new intentions (like the `container` and `cursor` types and the `insert`, `goto_first`, etc. operations). In other words, DiSTiL keywords are introduced as linguistic extensions to IP. Each DiSTiL component implements all the interface operations of Figure 2. The implementations are integrated in a top-down fashion to produce the final transformation for the given operation.

Consider again the example of Section 3.3 (structure with hash table and red-black tree indexes). When an `insert` operation is transformed, the hash table component contributes code of the form:

```
`( if (Container->bucket[i] == 0)
     {  Container->bucket[i] =
              Cursor.obj;
        Cursor.obj.next = NULL;  }
   else ... )
```

Similarly, all other components contribute their own insertion code. For instance, the red-black tree component creates the insertion code fragment

```
`( ELE_TYPE* y = &Container->header;
```

```
ELE_TYPE* x =
     Container->header.parent;
while (x != Container->NIL)
    ...
... )
```

The hash table component (being first in the composition order) determines how the two code fragments are composed. In this case the composition is as simple as adding the hash table code right after the red-black tree insertion code.

The above code expressions are generated in distinct generation environments (see Section 2.2). This way, for instance, the expression `Container->bucket` is only legal in the context of hash table operations. Isolating the generation environment for each instance of a component allows us to specify component code in an abstract form. If a certain composition contained three different containers, each of which is organized using two different red-black trees, the expression `Container->header` would still be unambiguous. All the context information is captured in the generation environment structure (which only has to be set up once). The interested reader is referred to [Sma96] for a more complete example.

In the IP framework no parser is needed for the DiSTiL language — instead we have added support for editing source code with DiSTiL operators directly (in abstract syntax tree format using the IP graphical structure editor). The new primitives are given the right properties to be correctly displayed on screen. The unparsing (display) in this case is straightforward (for instance, we have to make sure that the `foreach` primitive is displayed as an iterator with its last argument being a statement instead of an expression). The graphical unparsing capabilities of IP are used in several other cases as well. For example, code templates (quoted code fragments) in DiSTiL components can be displayed in special styles or colors, etc.

The result of compiling DiSTiL in the IP system is a DLL (dynamic link library) with system extensions. To create a DiSTiL program, the user includes a DiSTiL interface file as a library. This causes the DLL to be loaded. The new intentions can now freely be used in an IP source file — the file becomes a DiSTiL specification. Every time such a file is transformed (compiled) the compiler will dispatch to the DLL to handle DiSTiL-specific intentions. DiSTiL will validate the compositions used, type-check the arguments, and possibly report error-messages using standard IP interfaces. If no errors are detected, it will compose abstract syntax tree fragments to create an implementation for the given operation expressed using only core IP primitives. The result is a program generated through transformation and (possibly) linking to a static library. The static library implements component aspects orthogonal to DiSTiL (like hash functions and persistent storage routines).

The efficiency of the compilation process is obviously a major concern in any transformation system. Since IP is still under development, it is too early to judge the speed of compilation for code with transformation extensions. In our experience, however, transformation from DiSTiL primitives to C code accounted for only a small part of the time spent in compiling.

Detecting specification errors is the responsibility of the DiSTiL design-rule checker. Each DiSTiL component has two logical propositions and two boolean functions associated with it. The first two express the conditions that the component expects the layers "above" it and "below" it to satisfy. The notions of "above" and "below" refer to the ordering of components in a composition (see Section 3.3). The two boolean functions specify the properties of the component in terms of the properties of the layers above and below it. Using these logical attributes the correctness of a composition can be checked with a single traversal of the composition structure. Additionally, DiSTiL operations can impose a condition on the properties of the entire composition. This can ensure that a certain operation is supported in the given type expressions. The mechanics of this validation are rather straightforward and the interested reader can find more details in [Bat97a].

4 Lessons Learned

The design and implementation of DiSTiL gave rise to several interesting observations of wider applicability in the area of domain-specific languages. Below, we discuss some of the lessons we have learned.

Domain-Specific Language Design. Abstracting away the details of an implementation leads to simpler and more powerful specifications. This is the principle behind higher-level languages and it is exploited fully in DiSTiL. So the main goal of a DSL design should be a significant rise in the level of abstraction that the user is exposed to. Thus, one can radically alter the implementation of a specification without needing to modify the application source. Ideally we would like to have implementation-independent declarative specifications. Designating *what* needs to be done and not *how* it should be done is the first step in the abstraction process. No DSL,

however, is usable unless it offers acceptable performance. Otherwise an abstract specification can only serve as a design guideline and the actual implementation will have to result from manual refinement of the specification. The challenge for the DSL designer is to discover a level of abstraction that is amenable to automated reasoning for error-checking and optimization. In this way, the advantages of both abstraction and efficiency can be obtained.

These observations are clear in DiSTiL. All DiSTiL data structure components have a common interface and operations on them are specified through a mix of operational and declarative primitives. The DiSTiL generator can automatically perform the right operation to the right data structure by analyzing the predicate associated with the current cursor. Thus, the two main benefits of domain-specific languages are obtained: The design goal of simplicity is satisfied by the abstract interface and the practical goal of efficiency is satisfied by the generator optimizations. Many of the same optimizations have been applied in the past in the P2 lightweight DBMS generator with impressive results [Bat97b]. It should be noted that the optimizations are applied on top of quite efficient individual component implementations. Our red-black tree component, for instance, without any optimizations, will produce code almost identical to the most common implementation of the STL tree component (the, publicly available, Hewlett Packard implementation of STL).

Domain-Specific Language Implementation. In writing DiSTiL, we found it valuable to have an extensible system (IP, in our case) in which a DSL can be expressed as incremental changes. IP relieves the generator writer from having to parse source code constructs and to maintain source information (for instance, scope). At the same time it offers an easy mechanism for specifying transformations, reporting errors to the user, etc. Furthermore, the extensibility of the system can be exploited during generator implementation. Any machinery commonly used in generators can be expressed as an extension and reused exactly as if it was a part of the original language. In other words, during the implementation of DiSTiL we created domain-specific extensions for the domain of generators itself (e.g., generation scoping of Section 2.2). Generation scoping is a powerful hygienic meta-programming system — a valuable layer of infrastructure for all generators transforming their primitives into code in a high-level language.

To realize the flexibility afforded by the DiSTiL design, consider the P2 generator. P2 used its own parser, data-definition language, component specification language and back-end [Bat97b]. All these elements were specific to data-structures and changes to the system could not be easily isolated. For instance, it was not uncommon that limitations of the component specification language required changing the back-end to add functionality needed for new kinds of components.

DiSTiL relies on IP for obtaining an abstract syntax tree representation of the specification, so it does not need a specialized parser. IP also provided a set of intentions implementing the C language on which DiSTiL was based. Additionally, the system relieved us from a series of low-level chores associated with managing source code. Another big gain was in the area of target code generation. The P2 component specification language (XP) had a generation scoping facility similar to that described in Section 2.2. XP, however, was severely limited in capabilities (being a token-based macro processor) and highly specific to the data structure domain (with built-in keywords like `container` and `cursor`). In contrast, generation scoping presents a domain-independent way to express generated code fragments and dependencies among them conveniently.

DiSTiL is an excellent demonstration of the benefits of using IP. DiSTiL's net development time was in the order of 9 man-months (excluding the development time for the generation scoping facility). The principal developer had not implemented a GenVoca generator before, so it is unlikely that this productivity was based on experience. A comparison with the P2 system may be useful, even though it is hard to compare directly the two generators (there are DiSTiL features that have no P2 counterpart and vice versa). We estimate that the P2 system required more than 3 man-years of work before it reached a level of functionality comparable to that of DiSTiL. The difference is reflected in the relative source sizes of the two generators. DiSTiL is about 10Kloc, with more than half of it (5.2Kloc) being components. P2, on the other hand, is not that much richer in components (17.1Kloc) but has an over 112Kloc infrastructure.

Domain-Specific Languages in Software Engineering. Domain-specific languages emphasize interface abstraction. This way, they can incorporate components that are uniformly treated and independently composable. Thus, the potential for software evolution becomes much greater. In DiSTiL this effect is obvious. Not only can the implementation of a primitive data structure change without affecting user code, but the specification of any user-defined data structure can also change without affecting the application. This is a characteristic of GenVoca generators. In the greater scheme of software

engineering methods, GenVoca generators are compact representations of exponential-size libraries.

Programming in Domain-Specific Languages. In the evolution of languages, the development environments and supporting tools are at least as important as the merits of a language design. New languages require powerful compilers, good editing support, convenient debuggers, and useful libraries of components. This need is acknowledged in the design of IP. The system facilitates incremental changes both to the language and to the development environment. Thus, the cost of providing supporting tools for development in a new DSL is lowered.

Technical Advantages of Domain-Specific Languages. Domain-specific languages, as a means of software reuse, offer concrete technical advantages over standard static libraries (procedure libraries, object libraries, or binary component libraries). These mainly fall under the three categories of *simplicity*, *efficiency*, and *error detection*. A DSL can provide a uniform interface for different operations, thus simplifying its specification. Also, a DSL can often be more efficient than a static library: The flexibility of the DSL approach allows developers to experiment with different ways to express their requirements. Since a DSL implementation has access to high-level information, its optimizations are large scale (i.e., global program transformations) and often result in better code than that produced by a typical programmer. These benefits are well documented in the literature: The controlled-environment study of [Kie96] indicates productivity gain ratios of about 2.9 resulting from the use of generator technology. Our experience with P2 [Bat97b] showed performance gains of at least an order of magnitude in re-engineering a complex application.

Another advantage of the DSL approach is that it enables error checking that is more thorough and at a much higher level than that of general purpose languages. This enables error reporting that is both extensive and accurate [Bat97a], as in the case of DiSTiL. In contrast, traditional static libraries rely on the type checking facilities of their host language. Such mechanisms may be limited and, sometimes, almost non-existent: A well known disadvantage of STL is that C++ offers no mechanism to constrain a parameterization. As a result, invalid compositions are only detected well after composition time, yielding inaccurate and, depending on the implementation, possibly misplaced error messages.

Disadvantages of Domain-Specific Languages. The DSL approach has disadvantages. First of all, there is the non-trivial cost of designing and implementing a DSL. Assuming technical problems can be overcome and a DSL has been specified and implemented, a more serious obstacle emerges. The education cost associated with having developers use a new tool can be significant. This includes the reluctance of developers to abandon their favorite environments and learn to use new mechanisms (language, interpreter/compiler, debugger, etc.). If the DSL is simple and well supported, the problem is alleviated. The decision of adopting a DSL is based on the trade-off between the education cost and the expected benefits.

In the case of DiSTiL, a new element comes into play. The IP system helps reduce the education cost by letting the entire development environment adapt to a DSL. Making small, incremental changes to the programming environment (compiler, debugger) results in lower education costs than implementing a whole new system. There is a different price, however, that the user will have to pay. This is the one-time cost associated with learning to use the IP environment in the first place. The process involves a departure from text-based programming and the use of a structure editor, so the cost is not negligible. It will be interesting to see if the benefits of domain-specific extensions can offset this cost, making IP successful.

5 Conclusions and Further Research

DiSTiL is a software generator for the domain of data structures. It implements a powerful domain-specific language in a novel way: As an extension to the IP transformation system and using generation scoping — a general purpose meta-programming tool. We are interested in further exploring ways to facilitate implementing domain-specific languages. We assert that the existence of an extensible system in which new transformations can easily be specified is a very promising approach to designing domain-specific software. We believe that writing software generators as language extensions is a significant advancement that enables generator programmers to concentrate on the complexities of their task and not tedious infrastructure development. Hopefully, the future will bring more and more generators implemented as simple extensions and leveraging off common infrastructure.

Directions for future research include further development of a common ground for generator implementation — that is, a set of mechanisms widely applicable in generator writing. Such an effort will

require careful modeling of generator activities and, particularly in the case of GenVoca, the interconnections between generator components. We expect this work to culminate to a collection of language primitives ideal for writing generators, in other words, a domain-specific language for implementing domain-specific languages.

6 References

[Bat88a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C.Twichell, T.E. Wise, "GENESIS: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, Vol. 14 #11 (November 1988), 1711-1730.

[Bat88b] D.S. Batory, "Concepts for a Database System Synthesizer", *ACM Principles Of Database Systems Conference* 1988, 184-192.

[Bat92] 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.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat95a] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

[Bat97a] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997.

[Bat97b] D. Batory and J. Thomas, "P2: A Lightweight DBMS Generator", Accepted for publication in the *Journal of Intelligent Information Systems*, 1997.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[Cli91a] W. Clinger and J. Rees. "Macros that Work". *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991, 155-162.

[Cli91b] W. Clinger and J. Rees (editors), "The Revised^4 Report on the Algorithmic Language Scheme". *Lisp Pointers IV(3)*, July-September 1991, 1-55.

[Coh89] D. Cohen, *AP5 Training Manual*. USC Information Sciences Institute 1989.

[Coh93] D.Cohen and N.Campbell, "Automating Relational Operations on Data Structures". *IEEE Software,* 10(3):53-60, May 1993.

[Cor91] J. Cordy, C. Halpern-Hamu and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Languages* 16,1(Jan. 1991):97-107, also in *Proc. IEEE 1988 Intl. Conf. on Computer Languages*, 280-285.

[Die97] P. Dietz, C. Jervis, M. Kogan, and T. Weigert, "Automated Generation of Marshalling Code from High-Level Specifications", RNSG Research, Motorola, Schaumburg, Illinois, 1997./

[Due97] A. Van Duersen and P. Klint, "Little Languages: Little Maintenance?", *Proc. First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris 1997.

[Haa90] L. Haas, et al., "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, March 1990, 143-151.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, "A Software Engineering Experiment in Software Component Generation", *Fifth International Conference on Software Engineering*, 1996.

[Koh86] 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.

[Nei80] J. Neighbors, "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

[Per97] G. Jimenez-Perez and D. Batory, "Memory Simulators and Software Generators", *1997 Symposium on Software Reuse*, 136-145.

[Rea86] Reasoning Systems Incorporated, "REFINE User's Guide", Palo Alto, 1986.

[Rea89] Reasoning Systems Incorporated, "Dialect User's Guide", Palo Alto, 1989.

[Sim95] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference,* 1995.

[Sim97] C. Simonyi, personal communication.

[Sir93] 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.

[Sma96] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. Technical Report TR-96-37, Department of Computer Sciences, University of Texas at Austin, December 1996.

[Ste90] G. Steele Jr., *Common Lisp: The Language*. Digital Press, 1990.

[Ste95] A. Stepanov and M. Lee, “The Standard Template Library”. Incorporated in ANSI/ ISO Standards Committee C++ Draft.

[Vil97] E.E. Villarreal and D. Batory, “Rosetta: A Generator of Data Language Compilers”, *1997 Symposium on Software Reuse*, 146-156.

[Wei93] D. Weise and R. Crew, “Programmable Syntax Macros”. In *Programming Language Design and Implementation*, 1993, 156-165.

Appendix A: DiSTiL Realms and Components

Realm	Component	Description
Data Structures	Array	Random access array. Multi-dimensional in its general form but automatically specialized to fit the current specification without imposing run-time overhead.
	Dlist	Simple doubly-linked list.
	Hash	Hash table — re-implementation of the corresponding P2 layer.
	Odlist	Ordered doubly-linked list.
	Tree	Red-black tree (self-balancing binary tree). Code written to closely match the HP STL [HPSTL] implementation, more efficient than the AVL trees of P2.
Storage	Malloc	Elements are allocated on demand.
	Bounded	A maximum number of elements is pre-allocated. For cases when the maximum number of allocated elements is bounded.
	Persistent	Elements are stored persistently (on disk).
	Transient	Elements are stored in main memory (data structures don’t outlive the process that created them).
Architectural	Functional	Forces code to be generated in functions to avoid source code bloating due to inlining.
DS Supplements	Delflag	Implements element deletion by marking them as deleted instead of de-allocating them.
	Sizeof	Keeps track of the data structure size.
	Check	Adds run-time bound checks.
	Inbetween	Ensures that a cursor always points to a valid element after deletions.
Hidden	Order	Factors out common code from all ordered layers. Inverts operations when the retrieval order is “decreasing” instead of “increasing”.
	Outofbounds	Factors out common code from ordered layers. If a retrieval finds an element outside the range of a predicate, the search is complete.
	Predicate	Factors out common predicate handling code.
Special Purpose	Lrutree	A layer developed for a special-purpose application (LRU memory policy simulation). Implements a queue with special characteristics (quick position computation, move at top of queue).