APPLICATION GENERATORS

INTRODUCTION

When a programming activity is well-understood, it can be automated. Automation transforms software development from activities like rote coding and tedious debugging to that of specification, where the “what” of an application is declared and the “how” is left to a complex, but automatable mapping. Programs that perform such mappings are *application generators* (or just *generators)*. In the technical sense, application generators are compilers for domain-specific programming languages *(DSLs)*. A domain-specific language is a special-purpose programming language for a particular software domain. A “domain” can be defined either by its technical structure (e.g., the domain of reactive real-time programs, the domain of LALR language parsers) or by real-world applications (e.g., the database domain, the telephony domain, etc.). The purpose of restricting attention to specific domains is to exploit the domain features and knowledge to increase automation.

If we view generators as compilers for DSLs, we should ask whether they differ substantially from compilers for general-purpose languages. Indeed, although there is a continuum, the research and practice of application generators is quite different from that of traditional compilers. A general-purpose language compiler implements a stable, separately defined specification and can take several man-years to develop. In contrast, a generator is typically co-designed with the DSL that it implements. The effort of implementing a generator is typically small—comparable to the effort of implementing a software library for the domain. This is largely the result of leveraging the high-level language (commonly called the *object language*) in which the generated programs are expressed. The above technical realities affect the problems that are of main interest to application generators. For instance, a lot of the emphasis in general-purpose compilers is on analyzing a program to infer its properties. In contrast, in generators the emphasis is on designing the DSL so that domain-specific properties are clearly exposed, and on having the generator exploit them with as little effort as possible. To leverage the high-level features of the object language, generators often focus on issues such as language extensibility and program transformations.

Before we delve further into generator specifics, it is worth addressing the following question: why are generators needed? Is it not sufficient to employ other programming tools (e.g., traditional software libraries)?

Libraries/APIs can themselves be thought of as crude domain-specific languages. They have their own simplistic syntax: only function call syntax is allowed and the syntax checking is limited to checking the number of arguments. They have their own semantic restrictions: arguments need to satisfy some preconditions and calls may affect the state of the system, thus needing to occur in specific ordering patterns. Limited static error checking takes place by type checking the function calls in the host language. Libraries/APIs even have their own simple optimization: they typically offer multiple hand-specialized versions of operations for different kinds of arguments, such as special-purpose multiplication operators for sparse arrays in a scientific computing library. The user needs to pick the appropriate operations and to ensure their safety. Users of standard libraries/APIs are often constrained more by the library semantics than by the semantics of the host programming language. This is a common sentiment among users of large parallel processing and scientific computing libraries (like MPI or LAPACK). It is also often expressed by programmers in large projects where each part of the code needs to support the conventions of other parts. In this case, the “domain” is the project itself.

The software engineering benefit of domain-specific languages relative to function libraries/APIs is exactly in addressing the above deficiencies of syntax, safety, and performance. A domain-specific language can offer more concise syntax, increasing the ease of development and maintenance; it can perform static error checking to detect common violations of the library semantics; and it can offer better performance through domain-specific optimizations.

There are many dimensions of variability among generators, despite their common goal. Some generators implement specification languages that have a sound theoretical basis (e.g., [42, 46]) and thus have been used extensively to implement formal specifications. More typically, full axiomatic theories simply do not exist and generator design is based on an informal understanding of a domain.

Another important variability deals with implementation technology. Most generators are self-sufficient, standalone translators (in much the same way as compilers for general-purpose languages). Yet others take on a very different form, such as when generators are implemented using program transformation systems (e.g., [29, 49]). A *program transformation system* (or just *transformation system*) is a platform for expressing and executing program transformations—that is, mappings from programs to programs. Sets of transformations define the automatable mappings of a particular domain. In this case, a generator would be merely a set of transformations that may not even be encapsulated in a single module.

In general, the field of application generators is a collage of ideas from various areas of computer science, such as programming languages, compiler technology, and software engineering, to name a few. An examination of the field reveals a few common principles and many distinct generator “camps,” each promoting a different philosophy of what generators represent and how they should be built. Hence, a representative overview of generators needs to include both basic background and a sampling of approaches. This article cannot be fully comprehensive, however. For further reading, our references emphasize recent work which can serve as a starting point. Partsch and Steinbrueggen [35] provide a good survey of past work on transformation-based systems. Jones and Glenstrup [18] offer a survey of program generation as it pertains to partial evaluation.

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 2007 John Wiley & Sons, Inc.

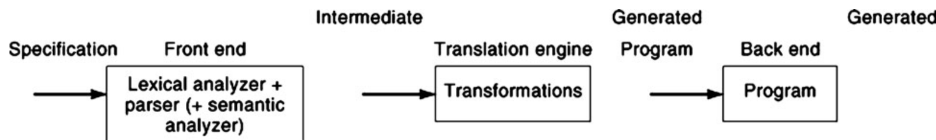

Figure 1. A generator is similar to a conventional compiler, with a front end, translation engine, and back end.

ARCHITECTURE OF A GENERATOR

Application generators have the standard internal form of a compiler with a front-end, translation engine, and back-end component (see Fig. 1). The front-end is responsible for the one-to-one mapping of the input form to an equivalent but more convenient internal representation, such as flow graphs, or abstract syntax trees, possibly annotated with data-flow and control-flow information. Typical input specifications are in text format, in which case the front-end consists of a conventional lexical analyzer and a parser. Other specification formats (e.g., graphical representations) may map straightforwardly to the intermediate representation, thus simplifying the front-end. It is interesting to note that generator writers have often tried to keep the cost of implementing front-ends low by employing ideas from extensible programming languages. Many generators are implemented as extensions of the Lisp language [19] or its variants. Lisp has explicit syntax (mapping directly to parse trees) and a very powerful extension facility (Lisp macros). In other cases, tools that generate parsers from modular grammar specifications (e.g., [4,48,55]) have been used. Such tools can effectively extend a language by adding new language constructs.

The translation engine implements transformations on the intermediate representation. Usually transformations are expected to satisfy some correctness property: the transformed program should have the same semantics as the original, if not for all inputs, at least under well-defined input conditions. Translation engines and transformations are the core of generators and are discussed in detail in the next section.

The result of applying transformations to the intermediate representation is a concrete executable program. The concrete program, however, is still represented as a flow graph or an abstract syntax tree. Mapping from the intermediate representation to program text is straightforward and is the role of a generator's back-end. Generated code is usually in a high-level programming language. Several generators and transformation systems (e.g., [1,20,30,46]) offer multiple back-ends, thus producing code in more than one language. Once again, this can be a straightforward process, if the generator does not rely on unique features of any specific language.

TRANSFORMATIONS IN GENERATORS

Translation engines and the transformations they support are in the heart of all generators. We next classify the most common kinds of transformations with respect to two criteria. Section 3.1 describes transformations from a technical standpoint. This answers the question of how translation engines express and apply transformations. Section 3.2 discusses how transformations differ relative to whether they intend to refine a high-level specification or to optimize at a single level of abstraction.

Transformation Machinery

There are several degrees of variability in the capabilities of translation engines (and, consequently, in the transformations they support). A fairly comprehensive classification of translation engines can be derived by answering the following questions:

- How are transformations expressed? (E.g., procedurally, using syntactic patterns, or using data-flow patterns.)
- How powerful are they? (E.g., can they change the global outline of a program or only local properties?)
- When are they applicable? (E.g., can they depend on complex data-flow properties? How are such properties expressed? What is the machinery to check them?)
- If multiple transformations are applicable, what is the order of their application? (E.g., are transformations always applied in a fixed order? If not, is the order determined automatically or manually?)
- To which extent are transformations automated? (E.g., does the user need to explicitly match program elements to transformation elements, or is the matching automatic?)
- Is the set of transformations fixed or extensible? (E.g., can the user add new transformations? Can the translation engine combine existing transformations to form new ones?)

Instead of answering each question individually, we identify four common axes of variation in transformation engines:

1. **Stand-alone generators vs. general transformation systems:** Generators can be packaged either as stand-alone tools, in much the same way as regular compilers, or as collections of transformations under a general transformation system. Expressing a generator as a collection of transformations has the disadvantage of making the generator dependent on a complicated piece of infrastructure (the transformation system). On the other hand, transformation systems (e.g., [5,29,49,54]) offer support for expressing and applying a variety of transformations in a general, domain-independent way. In other words, both the language in which the transformation is expressed, as well as the mechanism that applies transformations are determined by the transformation system and are the same for every domain. In theory, generators expressed as a collection of transformations are easily extensible, simply by adding more transformations. (In practice, this is may not be the

case. There are often subtle interdependencies among individual transformations that make transformation additions and substitutions hard and error-prone.) Additionally, because transformation systems are domain-independent, they typically allow for a higher degree of sophistication in the translation engine. Thus, general transformation systems commonly support specifying transformations declaratively instead of operationally. Given the declarative specifications of transformations, the translation engine may be able to deduce the appropriate order of transformation application when multiple transformations are applicable. Also, optimizations in the transformation application may be possible, by combining transformations to form new ones.

2. **Programmatic vs. pattern-based transformations:** As mentioned earlier, a transformation is a mapping from an input program to an output program. Such mappings can be expressed in a variety of ways. Transformations are commonly classified as *programmatic* or *pattern-based.* Programmatic transformations are arbitrary programs that manipulate code representations (also known as *meta-programs*). Pattern-based transformations, on the other hand, are written in a special pattern language that repeatedly searches a program representation for instances of patterns. When a pattern is found, the transformation is applicable and may be triggered, resulting in a different pattern being replaced for the original one. (Such a transformation is also known as a *rewrite rule.*) For instance, a simple rewrite rule could have the form:

$$\text{while}\,(\$\text{cond}\$)\,\$\text{stmt}\$$$
$$-\,>\text{L}:\ \text{if}\,(\$\text{cond}\$)\,\{\$\text{stmt}\$;\ \text{goto L};\}$$

The left hand side of the above rule is the pattern to be matched, and the right hand side is the pattern to be replaced. (Patterns are written in self-explanatory syntax resembling that of the C language, with pattern variables explicitly designated.)

Both pattern-based and programmatic transformations offer distinct benefits. Pattern-based transformations are generally simple and easy to understand. At the same time, their declarative nature allows for sophisticated automatic manipulation. In theory, pattern-based transformations are as expressive as any computer program (equivalent to Markov systems, e.g., see [24], p.263-264). Practically, however, pattern-based languages are inconvenient for applying complex transformations that rely on complex properties or contextual information. Programmatic transformations overcome this restriction. Overall, programmatic transformations are usually employed in ad hoc generator systems (i.e., stand-alone compilers for a specific domain) or for expressing global program transformations. Pattern-based transformations are in wide use in general program transformation systems. It is also possible to mix pattern-based and programmatic transformations. For instance, a transformation may be triggered by a certain pattern but the actions executed at this point may be specified programmatically. Similarly, a transformation may be programmatic but use patterns to describe newly created code.

Many interesting languages for expressing transformations are hard to characterize as strictly pattern-based or programmatic. For instance, a large number of transformation systems (e.g., [53, 37]) rely on attribute grammars [22] for expressing transformations. Briefly, attribute grammars are context free grammars extended with syntax-directed functional (i.e., side-effect-free) computations of "attribute" values, which are associated with symbols in the grammar. Thus, transformations expressed as rules in attribute grammars are triggered by parsing (essentially, pattern-matching) the program representation. Nevertheless, the actual action performed when a rule matches is expressed in a limited programmatic form. Limiting the attribute computation to be functional allows the translation engine to determine automatically the order of transformation applications, based on the dependencies among attributes.

3. **Syntax-directed vs. flow-directed transformations:** As in standard compiler-based transformations, the translation engine of a generator could be operating on intermediate representations that reflect syntax (e.g., abstract syntax trees) or control/data flow (e.g., flow graphs). Syntax-based representations have the advantage of being simpler, easier to obtain, and directly reflecting the hierarchical nature of the program to be transformed (e.g., a while-statement is represented as a tree with the while operator at its root). Control flow-based representations have the advantage of providing a normal form for representing control information (e.g., all kinds of loops have the same form in a flow graph).

The vast majority of realistic transformations are only applicable under certain guarantees about the context of a transformation application site. For instance, the following two transformations are context-dependent:

$$\text{x evaluates to a number, has no side} - \text{effects}$$
$$= > (\$x\$ + 0 - > \$x\$)\ \text{cond is guaranteed true,}$$
$$\text{has no side} - \text{effects} = >$$
$$(\text{if}\,(\$\text{cond}\$)\,\$\text{thenbody}\$\ \text{else}\ \$\text{elsebody}\$$$
$$- > \$\text{thenbody}\$)$$

The transformations are read as follows: if the current context implies the property on the left side of the "=>" symbol (called the enabling condition), then the rewrite rule on the right side of "=>" is applicable. To enable context-dependent transformations to be applied automatically, the generator must perform extensive program analysis. This analysis is easier with a program representation that makes the program's control and data flow explicit (e.g., a flow-graph). (A discussion of program analysis techniques is beyond the scope of this article, and can be found in textbooks on optimizing compilers—e.g., [27].)

In practice, unlike general-purpose compilers, few generators use intermediate forms that explicitly reflect control flow. Notable exceptions are stand-alone generators for domains that are best exploited by traditional compiler analysis tools (e.g., matrix algebra [25]).

Only few general transformation systems (e.g.,[5]) use a control flow-based representation, but almost all support the annotation of abstract syntax with information derived from program analysis. The motivation behind this widespread practice is partly its simplicity, but also the fact that generators usually transform generated programs and not arbitrary programs that an end-user has written. That is, control/data flow analysis is less meaningful at the level of the input of a generator. Most generators have input languages that are highly declarative, with very little operational information. When a generator transforms the input specification, it can produce at each step both the transformed code and automatically derived properties of this code, which can be attached as annotations (e.g., see [7]). In this way, one transformation step can supply all necessary contextual information to the steps following it, thus avoiding the need for program analysis. For this approach to be successful, the generator writer has to identify in advance a few high-level properties that are fundamental for the produced implementation (e.g., the property "the expression has no side effects" for transformations (2) and (3), above).

Based on the above observation, it is not surprising that the emphasis in generators (beyond program synthesis) is not on program analysis (deriving program properties) but on expressing program properties and inferring other properties from them. Thus, generators and transformation systems often offer powerful inference capabilities, in the form of specialized theorem-provers (e.g., [44]).

4. **Degree of structure in the transformation process:** The spectrum of translation engines found in generators is very wide. A good heuristic rule for classifying generators is to compute the average number of transformations that are potentially applicable at every step in the transformation process (i.e., how many options the system has when it makes a transformation decision). For stand-alone generators, whose input is a rather concrete specification (e.g., [51,15,25,40), this number is typically small (at most around 10). Furthermore, the transformation process in simple generators may be *confluent:* different orders of transformation application can produce different intermediate results but further transformation will reduce them all into the same normal form. More ambitious generators, translating more abstract specifications (e.g., [8, 42]) usually have to choose among many tens or hundreds of transformations at every step. In other words, generators of the first kind act more like conventional compilers, while generators of the second kind apply more intelligence in the transformation process, using heuristic knowledge to make complex decisions.

   Some of the latter generators (e.g., 42) are based on an equational rewrite paradigm. That is, transformations may be specified only implicitly using a set of axioms in an equational logic. The generator can then use these axioms to derive equational properties (theorems). Each of these equations can be viewed as a pair of transformations: either the left hand side can be matched and the right hand side be replaced, or the converse. In this case, it is not easy to guarantee that the transformation process will always terminate. A naive transformation engine may even repeatedly perform a transformation and its reverse, as they are both derived from the same equation. There has been significant work on deriving (from a set of equality axioms) a set of transformations that are guaranteed to terminate, regardless of their application order. Most work is based on the well-known Knuth-Bendix completion algorithm [23] and a relatively recent comprehensive survey of rewrite systems can be found in [12].

   Although a sophisticated transformation process is desirable, it can also be highly complicated. "Traditional" transformations are rewrite rules that work on a small fragment of code, such as (1)-(3) above. Given a set of such rules, automatically determining the next rule(s) to apply may be very difficult, and hence it is not uncommon for transformation systems with such rules to require periodic guidance/inputs from its users. The degree of interaction becomes more involved as programs become more complicated: Transforming a declarative specification into an optimized program may require many thousands of such rewrites. To address this complexity, many modern generators (e.g., [51,15,40]) encapsulate several small transformations in large components and apply them in a consistent manner (i.e., the generator decides to apply either all the transformations in a component or none). This approach, known as "consistent refinement", is quite beneficial in the domains for which it is applicable (typically such domains are well-structured and well-understood). For example, suppose one is transforming a declarative specification of a program that uses a data structure. At one point in the translation, a concrete implementation must be chosen for the data structure. A large number of small transformations may make a common assumption (e.g., the data structure is a list), and all of them need to be applied consistently.

Transformations and Level of Abstraction

Transformations can usually be classified based on the relative level of abstraction of their input and output. A *refinement* adds implementation detail to an abstract specification. For instance, an abstract data type, such as a set, may be refined to be implemented using a specific data structure, such as a binary tree (e.g., [8]). Refinements can occur at many levels and may fundamentally affect program structure and performance. *Restructuring* transformations reorganize a program, typically in order to improve performance, but maintain the same conceptual level of abstraction.

Next we discuss some common kinds of refinement and restructuring transformations. Our presentation is selective. A valuable further reference is Partsch's textbook [35], which contains a large number of example transformations for many common tasks.

Refinement Transformations. The presence of refinement transformations is the single most striking difference be-

tween generators and compilers for general-purpose languages. We discuss two common types of refinements below:

1. **Algorithm Derivation:** The most important kind of refinement for generators is that of transforming a declarative specification into an operational procedure that produces values satisfying the specification. Common algorithm derivation transformations include mapping operators from the declarative specification into heuristic-guided search procedures. For instance, an existential quantification (i.e., a specification of the form "there exists an element satisfying property P") can be mapped into a search procedure that iterates over elements until one is found to satisfy property P. The challenge is to exploit the structure of property P and use it to derive efficient implementations that do not exhaustively search the space of possible solutions. For instance, P could be a property that admits efficient filtering (i.e., if there is an element satisfying it, then a larger group of elements will satisfy another property Q, which can be used to filter out non-solutions). Excellent starting points for exploring the wealth of research work in general algorithm derivation are Chapter 5 of [36], and [42], [43].

   Deriving algorithms from highly abstract specifications is still a research challenge, however. In practice, most actual generator systems are less ambitious. Stand-alone generators (e.g., [7,17,25,32,40,51]) usually perform algorithmic refinement by using *algorithm schemas:* generic algorithm templates that allow limited specialization for particular data representations and special-purpose operations. For instance, an algorithm schema could provide the skeleton of a global search procedure. This procedure can then be specialized by adding the actual conditions for terminating the search. Local optimizations can be performed, but the overall structure of the search process will be the same for every search procedure generated, regardless of data structure or searched element. Clearly, this approach can only produce efficient code for highly structured domains, but this is sufficient for most generators that cater only to specific programming needs.
2. **Data Type Refinement:** A complementary refinement to algorithm derivation is that of selecting an implementation for data types in a specification. Different data structures offer good performance for different operations (e.g., retrieval of elements with key values in a range, vs. retrieval of elements with a single key value). Additionally, often data structures need to be combined, effectively creating indexes that support the efficient retrieval of groups of elements. Just like in the case of algorithm derivation, the approaches taken by different systems vary with respect to their sophistication. Systems that take input in a declarative language often use a set-theoretic abstraction for specifications. Sets can later be mapped into efficient data structures automatically (see Chapter 9 of [36], and [38, 42]). The choice of data structure depends on the kind of operations commonly performed (e.g., exhaustive searches vs. searches that can be efficiently indexed). At the same time, the guarantees offered by the data structure (e.g., always fully sorted vs. partial priority queue ordering) influence the way algorithms are derived. For instance, the decision to choose a fully sorted data structure may influence the subsequent choice of an algorithm that manipulates data structure elements. The interplay of algorithm derivation and data type refinement provides interesting research challenges.

   Many generators (e.g., 31, 40) employ more realistic approaches to data type refinement, by allowing the user to specify either the desired data structure, or the desired algorithms, and optimizing one choice based on the other. An example of this approach is discussed in a later Section (P2).

Restructuring Transformations. Restructuring transformations are typically used to implement performance optimizations in generators. Compared to compilers for general-purpose languages, generators offer more opportunities for restructuring transformations: automatically generated code is usually highly formulaic. Domain-specific knowledge (which the generator incorporates) can be used to exploit this structure to realize fairly sophisticated optimizations—often improving performance by several orders of magnitude.

Restructuring transformations in generators partly borrow from conventional compiler technology. Nevertheless, several kinds of optimizing restructurings have been developed much more extensively in the transformational programming community than in general-purpose compilers. Restructuring transformations in generators fall mainly into three categories:

1. **Partial Evaluation:** *Partial evaluation* (e.g., see 6, 18) refers to the specialization of a code fragment under the assumption that its (implicit or explicit) parameters satisfy certain conditions. It is probably the most common kind of optimization in application generators (for instance, transformations (2), and (3), shown earlier, represent cases of partial evaluation). This is expected: partial evaluation is a general technique for specializing general pieces of code for use in concrete contexts. Partial evaluation can be effected through pattern-based transformations but the most complex cases are usually treated programmatically. Two special cases of partial evaluation are *function specialization* (producing a new function by fixing some of the arguments of an existing one) and *constant folding* (performing computations on constants at compile time).
2. **Incrementality Optimizations:** Another class of valuable restructuring transformations rely on techniques that perform complex computations incrementally. This is particularly interesting in the context of generators, since, when composing abstract algorithms, a generator often has knowledge of the update patterns for the data used by each algorithm. Thus, it is not surprising that incrementality optimization techniques have been explored extensively in the generator community. One such technique is known by the name *finite dif-*

ferencing or *formal differentiation* [34,33,39]. Finite differencing substitutes expensive computations that occur in a specific pattern (e.g., in a loop) with an incremental update of the result of the previous computation in the pattern. The origins of finite differencing can be traced in the well-known strength reduction optimization in compilers. Continuing work in transformational programming has yielded new results in a more general setting (a good starting point for exploring such research is [26]).

Finite differencing is best applicable when there are strong static guarantees on how data are updated. Other incrementality optimizations can be used even when a strong pattern is not statically known, but run-time uniformities are expected. This is the case with standard caching or memoization optimizations in the context of application generators (e.g., see Chapter 6 of [36]). These techniques store values produced by a computation at run-time so that they can be used by subsequent operations (possibly for incrementally computing other values). The algorithms used need to be modified to take advantage of cached values when these are available.

3. **Traditional Optimizing Restructurings:** Most traditional compiler optimizations can be also applied in the context of generators. These include dead code elimination, loop unrolling, loop invariant code motion, loop fusion, etc. (see 27). The applicability of such optimizations can either be inferred from the code or established by previous refinements, so that expensive program analysis infrastructure is not required.

CASE STUDIES OF CONTRASTING APPROACHES

As indicated earlier, there is significant variability among generators: generators are being used for everything from trivially automatable specifications to formal languages that cannot be transformed without human input. Additionally, generators are built using widely different techniques. In this section, we look at the approaches taken in two generators that are, in many respects, at opposite ends of the spectrum. (Many more (older) systems are discussed in [35].) Each of the two generators that we have selected are among the best-known representatives of a distinct and wide class of successful systems. At the same time, each promotes a distinct philosophy on the principles upon which generators should be based. We end this section with a comparison of these approaches.

KIDS

The Kestrel Interactive Development System (KIDS) [42] is a semi-automatic generator applied to the problems of automatic programming. Although it is hard to strictly define what "automatic programming" is, the name is usually reserved for the most ambitious software production techniques, i.e., those trying to automate most of the software development process. Even though automatic programming has been a moving target (the first compilers were touted as "automatic programming" systems), a consensus on the fundamental elements of the field has evolved in the past three decades (sadly reflecting our failure to advance the "automation" target significantly during this period). Two main approaches to automatic programming are usually identified: the knowledge-based approach and the formal-model-based approach. KIDS is one of the primary representatives of the formal-method-based approach. More importantly, in addition to its ambitious goals, KIDS has seen several practical applications and has tested the limits of common generator optimizations.

The domain of KIDS is that of algorithm design and implementation. The system superficially departs from the usual generator model since several high-level transformation decisions are specified interactively by the user. Nevertheless, it is fundamentally a generator that refines and optimizes a formal specification. The input of KIDS is a functional specification of a problem (i.e., a function characterizing the possible outputs for each input) expressed using first-order logic operators and set-theoretic data types. As a simple example, the notion of an injective sequence of integers (sequences can be viewed as functions with a domain 1...n) can be expressed as:

$$\text{function}\; injective(\mathrm{M} : seq(integer), S : set(integer)) :$$
$$boolean = range(M) \subseteq S \wedge \forall(i, j)(i \in domain(M)$$
$$\wedge\; j \in domain(M)) \Rightarrow (i \neq j \Rightarrow M(i) \neq M(j))$$

That is, a sequence M is injective into a set S if all elements of M are in S and no two elements of M are the same. Distributive laws are common in KIDS specifications, essentially specifying a structural induction phase: the meaning of the combination of two operators is defined in terms of the meanings of "simpler" combinations. An example distributive law for the injective predicate is:

$$\forall(W, a, S)(injective(append(W, a), S))$$
$$= (injective(W, S) \wedge a \in S \wedge a \notin range(W))$$

KIDS gets additional input interactively from a human user. The user can make strategic decisions, such as "design a divide-and-conquer algorithm for this specification" or "simplify this algorithm by applying finite differencing on this value." The system contains a powerful inference engine [44] that applies pattern-based transformations derived from theorems of first-order logic. To schedule these transformations, the engine uses a combination of heuristic measures, such as the number of logical "weakening" rules that it has applied. KIDS encodes knowledge of a few general algorithmic search procedures (such as "global search") in the form of program templates. The result of the inference procedure is a correct specialization of such templates, thus yielding a complete abstract algorithm.

At that point, standard refinement and optimization techniques can be applied to the output. KIDS provides several rewrite rules for either context-independent (i.e., without enabling conditions) or context dependent simplifications. The powerful inference infrastructure collects context information and decides whether an expression can be simplified. Other optimizations (different forms of partial evaluation and finite differencing) can also be applied under user guidance. Finite differencing, in particular, is especially valuable because of the set-theoretic nature of

KIDS specifications. Sets can easily be specified incrementally and most KIDS algorithms reference complex predicates on sets. Refinements are also essential in KIDS to implement abstract data types (such as sets, maps, and sequences) as efficient data structures (e.g., arrays, trees, and lists).

KIDS is a representative of a formal approach to the specification of a domain. Assessing its applicability is hard—there is no general algorithm for satisfying specifications in first-order logic. Thus, we can only judge the practical value of the KIDS approach in empirical terms. In these terms, the system has been successful. Its best known application has been in deriving very fast and accurate transportation schedulers for use by the U.S. Transportation Command [41]. Excellent discussions on the application of KIDS to other (simpler and more easily understood) domains, together with complete examples of program derivations, can be found in [42] and [45].

Many other generator and transformation systems efforts are directly related to KIDS. The system is built on top of the Refine [49] transformation system (later marketed under the name *Reasoning5*). In fact, the input specification language of KIDS (logic-based with set-theoretic types) is part of the standard Refine infrastructure. Refine also offers a front-end tool [48] for the creation of modular parsers and a back-end (unparser) tool. Internally, programs are represented as abstract syntax trees, dataflow graphs, or control-flow graphs, depending on the most convenient level for each manipulation. Finally, many of the ideas introduced in KIDS relative to specifying search theories formally are more systematically explored in the SPECWARE system [46]. SPEC WARE is mainly concerned with modeling domains using algebraic specifications and composing specifications using techniques motivated by category theory.

P2

P2 is a component-based generator for the domain of container data structures. Component-based generators (e.g.,[10,15,17,28,40]) are a common class of generators whose transformations are represented as reusable and interchangeable components. Users declaratively specify their target application (in this case, a container data structure) and use compositions of components to tell the generator how to transform these declarations into efficient code. By using different compositions of components, P2 generates a completely different implementation of the same declarative specification. A key distinction between a P2 component and a KIDS transformation is one of scale: a P2 component encapsulates complex refinements and optimizations of multiple data types and operations on these types, which are presented as a "monolithic" transformation to a user. As each P2 component has a simple interpretation (e.g., there are different components for red-black trees, ordered lists, etc.) and the P2 component library is quite small (a characteristic of all component-based generators), the number of components (or transformations) that have to be composed to specify even complex applications (e.g., data structures) is modest (around 5-15).

P2 imposes relational abstractions on container data structures: data structures implement *containers* of elements and individual elements are accessible through *cursors*. Common data structures—arrays, binary trees, ordered lists—implement the container abstraction and are encapsulated as individual P2 components. P2 components implement protocols by which a component can query other components about what properties they support, what optimizations they can perform, what is the expected complexity of the code they generate, which other components they are compatible with, etc. Such knowledge is needed when generating efficient application source code, as well as when checking the consistency of component compositions.

The P2 language is a superset of the C language, where C is extended with cursor and container declarations and operations on their instances. For example, consider a phone-book data structure and the following declarations:

$$\text{Container} < \text{phonebook} - \text{record} > \text{phonebook}; /* \text{ abbreviated container decl.} */$$
$$\text{Cursor} < \text{phonebook} > \text{where}\ '\$.\text{phone} == 4783487'\ \text{joe}; /* \text{ cursor declaration} */$$
$$\text{Cursor} < \text{phonebook} > \text{where}\ '\$.\text{name} > 'S' \&\& \$.\text{name} < 'T''\ \text{all}/_\text{s}; /* \text{ cursor declaration} */$$

Assuming that elements are instances of the `phonebook_record` record type, the first line above declares a container (phonebook) for such elements. (Actually, the container declaration is abbreviated from the usual P2 syntax since it does not specify the components that implement the data structure—see below.) The subsequent lines declare two different cursors ranging over selected elements of the phonebook container. For example, the `joe` cursor ranges over all elements of phonebook where the phone attribute equals 4783487. Predicates need not be this simple; P2 can handle arbitrarily complicated predicates. In addition, P2 offers the standard operations on containers and cursors. For instance, the foreach operation is used below to iterate over all elements accessible by cursor all_s, and for those selected elements, the name of the element is printed:

$$\text{foreach}\,(\text{all_s})\{\text{printf}('\%\text{s/n},'\ \text{all_s.name});\}$$

Container implementation decisions are controlled by the P2 user by composing components from the P2 library. This is achieved with a typeq (type equation) declaration:

$$\text{typeq}\{\text{simple_typeq} = \text{top2ds}[\text{qualify}[\text{hash}[\text{phone}, \text{odlist}[\text{name}, \text{malloc}[\text{transient}]]]]];\}$$

`simple_typeq` is a composition of six P2 components, where each component encapsulates a consistent data and operation refinement of the cursor-container abstraction and is responsible for generating the code for this refinement. The `top2ds` layer, for example, translates `foreach` statements into `while` loops and primitive cursor operations; `qualify` translates qualified retrieval operations into `if` tests and unqualified retrieval operations; `hash` stores all elements in a hash structure where attribute `phone` is hashed; `odlist` connects all elements of a container onto a doubly-linked list that is ordered on ascend-

ing name values; `malloc` allocates space for elements from a heap; and `transient` allocates heap space from transient memory. The complete container declaration for the `phonebook` container is shown below; it declares the type equation that determines how the container is to be implemented.

```
Container < phonebook_record > using simple_typeq
    phonebook;
```

The type equation determines how elements are to be stored and which fields are to be indexed (e.g., attribute `phone` is hashed and elements are arranged on a list in ascending name order). The P2 generator is responsible for implementing all operations on cursors and containers efficiently using information that it can infer statically from cursor selection predicates and the container type equation. For instance, P2 infers for the `joe` cursor (above) that the fastest way to find elements that satisfy the predicate is to use the hash table on the `phone` field. Similarly, P2 infers for the `all_s` cursor that the fastest way to find elements that satisfy the `all_s` predicate is to traverse the name-ordered list. The techniques that P2 uses to evaluate the cost of each retrieval method are motivated by query optimization in database systems.

In essence, type equations relieve P2 of the burden of making high-level refinement decisions. P2 does not attempt to automate data structure (or type equation) selection, but rather offers a friendlier interface to the user and facilitates program modification when requirements change. This was demonstrated, for example, when P2 was used to re-engineer a hand-coded, highly-tuned container data structures used in a production system compiler (LEAPS). As a result, P2 reduced the code size by a factor of three and offered significant performance benefits (up to several orders of magnitude in some cases) [51].

P2 covers a well-known domain and, hence, is ideal for demonstrating the benefits of component-based generators over traditional software libraries. P2 components capture features that are not easy to compose in their concrete form. Components such as a hash table and a linked list data structure will have very different interfaces if encoded as concrete library components. This is, for instance, true in the C++ Standard Template Library (STL) [47] where sequences and associative containers have different interfaces (and, thus, are not interchangeable). In contrast, P2 raises the level of abstraction up to the point where all data structures have the same interface. At the same time, the specification language (i.e., the selection predicates, discussed previously) supplies enough information to the generator so that the full functionality of individual components (e.g., the fast random access capabilities of a hash table) can be utilized. This way implementation efficiency is regained automatically by the generator, even though an abstract language is used for operation specification.

Comparison

Consider KIDS and P2 both from a technical and from an end-user perspective.

- KIDS is built on top of a general transformation system (Refine), whereas P2 is a stand-alone compiler.
- KIDS is a semi-automatic development system that takes general declarative specifications (first-order logic formulas) as input. P2 is highly specialized for a single domain (data structure programming), and its input contains significant (albeit compact) implementation guidance in the form of a type equation.
- KIDS is based on an equational rewrite engine and uses a complex inference engine to guide the transformation process. P2 has a straightforward translation engine, based on a combination of programmatic transformations and pattern-based macro expansion. In a typical transformation step, KIDS has a wide space of possible choices for the next transformation, whereas P2 has no more than a handful.
- Context information in KIDS is expressed in a rich language and can be combined to derive new properties. P2 only uses a small set of predefined context properties that guide the transformation process.
- KIDS has a sophisticated model for deriving new algorithms, while P2 can only specialize existing algorithm templates. Accordingly, the KIDS refinement process may require significant user interaction, while P2 is fully automatic.

The sharp contrast between our two generator case studies illustrates the heterogeneity of the area, despite the occasional technological similarities. Generators vary as much as the different domains of software, both in depth and in breadth. Generator technology can be quite practical and immediately applicable, as long as the domain of the generator is narrow, well-structured, and well-understood. At the same time, generator technology can be ambitious, tackling domains that have little structure and challenge the limits of our capabilities.

APPLICATION GENERATORS NOW AND IN THE FUTURE

Generators in Practice

Application generators represent a significant software production technology. The breadth of the application generators field allows it to claim successes in many practical settings. Bassett's frames [3] are a generative technique for adapting code text through pure lexical manipulation. Despite their simplicity, frames have been used with great success to create programs of significant size (e.g., million-line) in the information systems domain. Also, many programs that produce code skeletons by composing code templates are primitive generators (e.g., the *wizards* supplied with Microsoft compilers). Similarly many language tools for mature domains are clearly generators (for instance, the *yacc* parser generator or the LaTeX set of typesetting macros). Nevertheless, these are rarely considered examples of what we will call the *generator approach* to software development. The reason is that the above tools do not need sophisticated transformation machinery. For instance, typically such tools do not have to choose which

transformations to apply, either because their domain is so well-structured, or because their job is simply to concatenate code text. Hence, the approaches that we examine here are among those that employ transformation technology of the kind discussed in this article.

We selectively discuss two representative industrial projects employing application generators in the construction of complex software.

- The SciNapse system [2] (formerly called Sinapse [20]) is a generator for mathematical modeling software. SciNapse uses both programmatic and pattern-based transformations and performs algorithmic refinement by using algorithm schemas, which are later specialized extensively. The specializations typically are numerical approximations for discrete representations of the continuous specifications of variables. SciNapse also includes transformations for data structure refinement and optimizations oriented towards scientific computing. The transformation process can be either automatic or interactive, with the user being able to override the system's choices at key points. The system is implemented in the Mathematica programming environment and uses Mathematica's algebraic manipulation capabilities. The system has multiple back-ends, generating code in Fortran 77, CM Fortran, or C.
  SciNapse was used originally to generate programs that solve partial differential equations for sonic wave modeling. These programs have multiple applications in exploring seismic wave propagation between oil wells, measuring the transit time of sonic waves in a moving fluid, exploring the 3D effects in complex geological formations, etc. More recently, the system was applied to financial modeling. SciNapse generates 200-4000 lines of code programs from compact (around 50 lines) specifications. The generated programs exhibit performance often comparable to hand-coded versions and are commonly used with only small manual modifications.
- Mousetrap is a transformation system developed at Motorola, which has been applied to the derivation of efficient real-time code for the company's subscriber radio products [13]. Mousetrap operates on an abstract syntax tree intermediate form with fine-grained pattern-based transformations (tens to hundreds of thousands of transformations may be applied in the derivation of a complex system). The system performs algorithm selection based on algorithm schemas—e.g., translating a finite-state machine specification into code containing nested loops and conditionals. Multiple optimizations are applied in the generated code—for instance, loop invariant code motion, as well as machine-architectural optimizations like grouping bit-operations together and applying them at a machine-word level.
  The primary application of Mousetrap has been in generating *marshalling* code for subscriber radio protocols. The role of such marshalling code is to convert data from an in-memory representation (optimized for fast access) to the representation needed for wireless transmission (optimized for size). A set of Mousetrap transformations implement a generator for a domain-specific language that is used to describe the general structure of protocol packets. Because of optional information, many configurations of protocol packets can exist (all with the same general structure), and the transformation rules ensure that efficient code is created in every occasion. Many of the optimizing transformations employed in this process are domain-independent and part of the general Mousetrap infrastructure.
  The result of generating marshalling code using Mousetrap has been "a tremendous success" [13]. The process was estimated to result in a reduction of the development cycle for marshalling code by a factor of four. Benefits in the maintainability and ease of code evolution were also observed.

Outlook

Generators are gaining momentum in the software engineering community. In the past few decades, software construction has not seen any radical improvements with respect to increased productivity and reliability. The proponents of the generator approach consider generators to offer the greatest promise among emerging technologies for the future of software development. In particular, advocates of generators consider them to be the right tool every time a software product is designed to be reused, or every time a domain exhibits significant systematic variability. This view promotes generators as a substitute for most, if not all, of the existing software libraries for appropriate domains.

There are certainly serious challenges in trying to move generators to the forefront of software development. After all, generators are by nature domain-specific. Envisioning them as primary tools in the software construction process seems somewhat paradoxical. Furthermore, generators are often considered undisciplined and error-prone: reasoning about a generator is much harder than reasoning about a concrete library, as the properties of the generator output may crucially depend on its (unknown) input. Therefore, recent work has focused on transformation systems that offer support for determining the correctness of generators expressed in them.

There are several levels of correctness properties we may need to express and prove for generated code. The first level concerns the syntactic well-formedness of generated code. This is not a difficult property to establish. Most transformation systems (e.g., [4,5,54,56]) are *structured* meta-programming systems, and operate on syntax objects (e.g., abstract syntax trees) instead of text strings. In this case, tree operations can be constrained to only allow creating syntactically well-formed objects. A second level of correctness concerns the well-formedness of generated programs with respect to the target language's type-checker. That is, we want to ensure that the generated program does not suffer from errors typically detected by a conventional compiler, such as type mismatch errors, references to undeclared variables, duplicate variable definitions, etc. We

call this property of a generator *static legality* and discuss it next in detail. A third level of correctness concerns the correct *semantics* of the generated code—i.e., the generator can certify that the generated code satisfies domain-specific correctness properties during its execution.

Static Legality. Static legality is a hard property to ensure. Consider a generator that produces first a declaration of a variable and later a reference to it. (We use a "quote" syntax—' [. . .]—for generated code.)

```
if (pred1) {emit'[int i; ]; }
...
if (pred2){emit '[i + +; ]; }
```

The generator produces a declaration for i, of type `int`, if `pred1` evaluates to true. It then generates a reference of `i` if `pred2` is true. Ensuring that the generated code never refers to an undefined variable `i` is equivalent to ensuring that whenever `pred2` is true, `pred1` is also true. This is an undecidable program analysis property. In general, we can take any hard program analysis property (e.g., any control- or data-flow property) and map it to an equivalent problem of static legality for a generator.

Therefore, to guarantee static legality automatically, our program transformation infrastructure typically needs to limit the kinds of generators that are expressible—either by conservatively rejecting some generators when their static legality cannot be proven, or by restricting the language so that some generators cannot be written. An interesting restrictive kind of program transformation infrastructure that guarantees the static legality of generators is multi-stage languages, such as MetaML [50] and MetaOCaml [9]. Type-checking a generator expressed in a multi-stage language ensures that the generated code is always type-correct. Nevertheless, such languages only allow expressing program specialization: the generated program is a specialization of the original program with some parts computed statically. For example, the following is a simple multi-stage program for efficient exponentiation—we use again the "quote" construct but allow parts of the quoted expression that are designated with an "unquote" operator (#) to be evaluated instead of generated:

```
exp(n, a) =
if(a == 0)'[1]
else '[#[n]  * #[exp(n,  a − 1)]]
```

This function takes in a number `n` for exponentiation, and a number `a` as the exponent. If `a` is 0, the generated program is the constant 1. Otherwise, the function generates a piece of code representing the multiplication of the value of n, and the code returned by calling exp on n and `a-1`. Thus, `exp(3,4)` would return us a piece of code, `3*3*3*3*1`. The property of multi-stage languages that makes them suitable only for code specialization is the *erasure* property: a multi-stage program is still a valid program if all quote and unquote constructs are removed. For instance, we can remove quotes and unquotes from the above exponentiation program to get the following legal program:

```
exp (n, a) =
if (a == 0)1
else n  * exp (n,  a − 1)
```

In this program, `exp(3,4)` is no longer staged—it is now a regular function call, and would return 81.

In practice, most generators are not just program specializers. Not only can generators produce variable declarations and references independently (as in our earlier example) but also generators often need to produce variables whose names are not known until run-time. For instance, a generator can be used to *reflect* over the functions or fields of an existing program and produce a new set of functions with similar or identical names, which may call the original functions as part of their execution. This kind of *reflective* generation is quite common in practice. Its advantage is not performance but productivity. Reflective generation can capture common interfacing conventions with external code and relieve the programmer of the tedious task of producing conforming code. For instance, the GOTECH [52] generator accepts as input arbitrary Java classes and produces isomorphic wrapper classes and interfaces to enable the original classes to interoperate with a specific runtime system (a J2EE Application Server). This approach has since become widespread and program generation is now a common technique in server side computing applications.

There is ongoing work on ensuring static legality automatically for richer transformation languages. SafeGen [16] is a proposal for a reflective transformation language that offers static legality guarantees for any generator expressible in it. The language allows defining iterators over existing programs. The iterators can range over reflective entities such as all fields of a class, all arguments of a method, all classes in a package, etc. All program generation is predicated on an iterator: copies of the quoted code will be generated for each iteration. For example, we could have a code generation expression such as:

```
#for[f in Field(c), '[#[Type(f)] #[Name(f)]  ; ]]
```

(The `#for` primitive is part of the SafeGen syntax. `Field`, `Type`, and Name are iterator functions.) In this example, definitions for several variables are being generated. The generated variable names are not statically known but they depend on existing names of fields in class c. Thus, static checking can be done, based on the assumption that the input class c is legal. For instance, the above code fragment can never generate a duplicate definition: the generated names are in a one-to-one mapping with fields of input class c, which are guaranteed to be uniquely named. Similarly, when generating references, the system can use the iterators to match them to generated definitions. For instance, one can refer to the variables generated by the above fragment in code such as:

```
#for[f in Field(c), '[insert(#[Name(f)])]]
```

We know that the emitted code refers to valid variable names because the iterator f ranges over the same values (all fields of class c) when generating both definitions and references.

SafeGen uses a first-order logic based type-system. All facts about the input program and generated program are represented as first order logic sentences. All properties of legal programs are also encoded as logic axioms. The sys-

tem then constructs an implication of the form *"facts =>" property"*, where property represents the static correctness property of the output program. SafeGen then uses an automatic theorem prover to prove the *validity* of this sentence for all possible values of input variables. If the sentence is valid, then any generated program is legal under the static checks of the target language, or equivalently the generator is statically legal.

Guaranteeing Semantic Correctness. The final level of correctness for generated applications is semantic correctness. We would like to ensure that generated programs satisfy standard software correctness properties. Nevertheless, the semantic correctness of a program is a domain-specific property. Thus, there does not seem to be a general way for transformation infrastructure to help ensure the semantic correctness of multiple generators automatically. Some approaches of general value have been proposed, however. One idea is to avoid establishing the correctness of every possible generator output (which is a hard property) and instead have the generator emit, together with the generated application, a proof of its correctness. A verifier program will then check the proof and certify that the specific generator output satisfies the required semantic property. Consider an axiomatic framework for program verification, such as a Hoare-style invariant framework. A generator can be extended so that, along with the output program, it also generates annotations that indicate formal properties (preconditions, postconditions, invariants) at different points of the generated program. The properties are then automatically checked for consistency (e.g., a statement precondition is ensured to imply its postcondition under Hoare logic) and the desired final correctness property should directly follow from the individual properties. This approach is exemplified by the AutoBayes generator [11, 14] used at NASA to produce statistical data analysis applications.

BIBLIOGRAPHY

1. William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in Intentional Programming. In *Proc. 5th International Conference on Software Reuse*. IEEE Press, June 1998.
2. Robert L. Akers, Elaine Kant, Curtis J. Randall, Stanly Steinberg, and Robert L. Young. SciNapse: A problem-solving environment for partial differential equations. *IEEE Comput. Sci. Eng.*, **4**(3): 32–42, 1997.
3. Paul G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
4. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. Ira D. Baxter. Design maintenance systems. *Commun. ACM*, **35**(4): 73–89, 1992.
6. Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *Computer*, **23**(12): 25–37, 1990.
7. T. J. Biggerstaff. Anticipatory optimization in domain specific translation. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 124, Washington, DC, USA, 1998. IEEE Computer Society.
8. L. Blaine and A. Goldberg. DTRE - a semi-automatic transformation system. In B. Moller, editor, *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 165–204. North-Holland, Amsterdam, 1991.
9. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2830,pages 57–76. Springer, 2003.
10. L. Coglianese and R. Szymanski. DSSA-ADAGE: an environment for architecture-based avionics development. In *Proceedings of Advisory Group for Aeronautical Research and Development (AGARD)*, 1993.
11. E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In D. Cork. Basin andM. Rusinowitch, editors, *Proceedings of Second International Joint Conference on Automated Reasoning (IJCAR 2004)*, pages 198–212. Lecture Notes in Artificial Intelligence(3097).
12. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.
13. Paul Dietz, Thomas Weigert, and Frank Weil. Formal techniques for automatically generating marshalling code from high-level specifications. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 40, Washington, DC, USA, 1998. IEEE Computer Society.
14. Bernd Fischer and Johann Schumann. Generating data analysis programs from statistical models. *Journal of Functional Programming*, **13**(3): 483–508, 2003.
15. John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Trans. Comput. Syst.*, **12**(1): 58–89, 1994.
16. Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In *Generative Programming and Component Engineering (GPCE)*, pages 309–326, 2005.
17. Norman C. Hutchinson and Larry L. Peterson. The X-Kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, **17**(1): 64–76, 1991.
18. Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2487, pages 1–31. Springer, 2002.
19. Guy L. Steele Jr., *Common LISP: the language* (2nd ed.). Digital Press, Newton, MA, USA, 1990.
20. Elaine Kant. Synthesis of mathematical-modeling software. *IEEE Softw.*, **10**(3): 30–41, 1993.
21. Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society.
22. Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2): 127–145, 1968.

23. Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras.In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.
24. Harry R. Lewis, Christos H. Papadimitriou, and Christos Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
25. Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 96–114, London, UK, 1994. Springer-Verlag.
26. Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, **24**(1): 1–39, 1995.
27. Steven S. Muchnick.*Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
28. James M. Neighbors. *Software construction using components*. PhD thesis, University of California, Irvine, 1980.
29. James M. Neighbors. *Draco 1.2 Users Manual*. Department of Information and Computer Science, University of California at Irvine, June 1983.
30. James M. Neighbors. Draco: a method for engineering reusable software systems. pages 295–319, 1989.
31. Gordon S. Novak. GLISP: A Lisp-based programming system with data abstraction. *AI Magazine*, **4**(3): 37–47, 53, 1983.
32. Models Gordon Novak. Generating programs from connections of physical models. In *Proceedings of the 10th Conf. Artificial Intelligence for Applications*, pages 224–230. IEEE CS Press, 1994.
33. Bob Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 58–71, New York, NY, USA, 1977. ACM Press.
34. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, **4**(3): 402–454, 1982.
35. H. Partsch and R. Steinbriiggen. Program transformation systems. *ACM Comput. Surv.*, **15**(3): 199–236, 1983.
36. Helmut A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
37. Thomas W. Reps and Tim Teitelbaum. *The synthesizer generator: a system for constructing language-based editors*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.
38. Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst.*, **3**(2): 126–143, 1981.
39. Micha Sharir. Some observations concerning formal differentiation of set theoretic expressions.*ACM Trans. Program. Lang. Syst.*, **4**(2): 196–225, 1982.
40. Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.
41. D. Smith and E. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 60–68. IEEE Computer Society Press, 1993.
42. D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, **16**(9): 1024–1043, 1990.
43. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Sci. Comput. Program.*, **14**(2–3): 305–321, 1990.
44. Douglas R. Smith. Derived preconditions and their use in program synthesis. In *Proceedings of the 6th Conference on Automated Deduction*, pages 172–193, London, UK, 1982. Springer-Verlag.
45. Douglas R. Smith. KIDS: A knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.
46. Yellamraju V. Srinivas and Richard Jullig. Specware: Formal support for composing software. In *MPC '95: Mathematics of Program Construction*, pages 399–422, London, UK, 1995. Springer-Verlag.
47. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94–0095, WG21/N0482, 1994.
48. Reasoning Systems. *Dialect user's Guide*. Palo Alto, California, 1990.
49. Reasoning Systems. *Refine 3.0 user's Guide*. Palo Alto, California, 1990.
50. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.
51. D. Thomas. *P2: A Lightweight DBMS Generator*. PhD thesis, University of Austin, Texas, 1998.
52. Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.
53. E. Zimmermann U. Kastens, B. Hutt. GAG: A practical compiler generator. page 141, 1982.
54. Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. InC. Lengauer, D. Batory, C. Consel, andM. Odersky, editors, *Domain-Specific Program Generation*, pages 216–238. Springer-Verlag, 2004. LNCS 3016.
55. David Wile. POPART: Producer of parsers and related tools. USC/ISI.
56. David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with meta-AspectJ. In *Generative Programming and Component Engineering (GPCE)*, pages 1–18. Springer-Verlag, October 2004.

YANNIS SMARAGDAKIS
SHAN SHAN HUANG
Dept. of Computer and Info. Sci.
University of Oregon
College of Computing Georgia
Institute of Technology