Generation of Components for Software Renovation Factories from Context-free Grammars

Mark van den Brand, Alex Sellink, Chris Verhoef *

University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

markvdb@wins.uva.nl, alex@wins.uva.nl, x@wins.uva.nl

Abstract

We present an approach for the generation of components for a software renovation factory. These components are generated from a context-free grammar definition that recognizes the code that has to be renovated. We generate analysis and transformation components that can be instantiated with a specific transformation or analysis task. We apply our approach to COBOL and we discuss the construction of realistic software renovation components using our approach.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.2.m [**Software Engineering**]: Miscellaneous—Rapid prototyping

Additional Key Words and Phrases: Reengineering, System renovation, Restructuring, Language migration, COBOL

1 Introduction

Software engineers are faced with serious problems when dealing with the renovation of large amounts of legacy code. Manual approaches are not feasible, in general, both due to the amount of code and since it is usually unreliable to renovate by hand. Nowadays, it is more and more recognized that a factory-like approach to renovate legacy code is a sensible paradigm [11]. Before we can use such a factory we need to construct its components. Since there are many and diverse renovation tasks for a myriad of languages and their dialects, there are as many components imaginable. These components should preferably be reliable, maintainable, reusable, and easy to construct. So, in our opinion, there is a need for a construction technique of components that are necessary in software renovation factories. In this paper we propose a method to generate such components from the context-free grammar that recognizes the code that has to be renovated (in [7] a method is discussed to obtain grammars from legacy code). Due to our generative approach and the presence of a grammar the components are reliable, maintainable, maximally reusable, and their implementation is usually measured in minutes rather than hours. At the time of writing this paper we are in a preliminary phase to design a software renovation factory to restructure legacy code from a large financial company and we are using the methods we report on in this paper to construct its components.

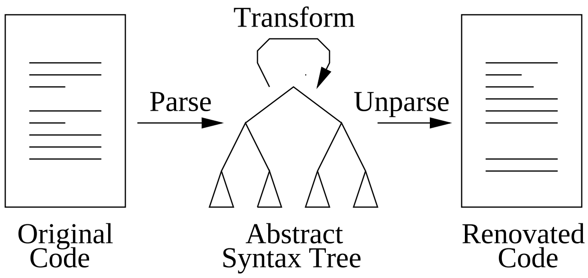

Figure 1: Schematic software renovation factory.

We think of a software renovation factory as the batch oriented transformation of massive amounts of code (see Figure 1). Roughly, such code is processed in three phases. First, the code is translated by a parser into an annotated abstract syntax tree. Then the annotated abstract syntax tree is manipulated, e.g. transformed or restructured, according to the desired renovation strategy. Finally, an unparser translates the abstract syntax tree back to text. So, parsers, transformations, and unparsers form the main parts of a software renovation factory. Parsing and unparsing components can be obtained by powerful techniques: they can be generated from the context-free grammar of the language to be parsed or unparsed. Lex and Yacc [15, 19] are well-known examples of a scanner generator and a parser generator, respectively. For the generation of unparsers from a context-free grammar we refer to [8]. The front-end and back-end of a software renovation factory can thus be generated. We will show that it is also possible to generate the other components of a software renovation factory.

The formalisms ASF and SDF [3, 10, 13] and their support environment the ASF+SDF Meta-Environment [17] have been used to generate programming environments from a context-free grammar. This includes the generation of, parsers, unparsers, structured editors, etc. SDF stands for *syntax definition formalism* and can be used to define a context-free grammar. ASF means *algebraic specification*

*Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

formalism and can be used to define arbitrary behaviour over a given grammar. For example, the syntax and semantics of a programming language can be defined using SDF and ASF, respectively. We will use the same technology to generate components for software renovation factories from a given context-free grammar. The generic techniques we use to do this are similar to the ones that were used to construct an unparser generator, see [8]. For more information on connections between reengineering and generic language technology we refer to [6]. To give the reader an idea why it is possible to generate components for software renovation factories we will give some intuition. For the functions generated by a parser generator the result type is always an abstract syntax tree, for the unparser functions the result type is text, for an occurrence testing component it is a Boolean, for a software metric component it is a (natural) number, for a restructuring component the input type equals the result type, and so on.

We discuss the various ways of constructing components for a software renovation factory. Given a specific renovation problem in a specific language for which a component has to be developed, several approaches are possible to construct the component. We introduce the following abbreviations LP, LS, PP, and PS standing for *language parameterized*, *language specific*, *problem parameterized*, and *problem specific*, respectively. The classical way of constructing a component for a given problem in a given language is handcrafted construction of the component, that is, the LS-PS approach. A serious problem in renovation is that every new customer uses a new dialect so the component has to be re-implemented. A more generative approach can solve that problem. Known generic examples are LP-PS components: parser generators [15, 19] and unparser generators [8], which take the language as a parameter. We are not aware of LS-PP components in the literature. We discuss them in this paper: generic components for COBOL which can be instantiated for a specific problem, e.g., local transformations like scope termination transformations and global transformations like current date transformations (see Section 3). Examples of LP-PP components are generic component generators that can be instantiated with a specific problem in a specific language to obtain a component solving the specific problem. This approach is also new. Next, we will give the reader an idea of this approach.

The LP-PP approach to construct a component is depicted in Figure 2. In this figure we assume the presence of so-called input code in some input language and some given reengineering problem, for instance, an analysis or a transformation task. The figure describes a method to obtain components to perform the reengineering task. First we use a generator that takes the grammar parameter as input. This generator generates a generic component that can be instantiated with the specific problem thus obtaining the component that we need to solve the given problem. We have four grammars: one that recognizes the input code, one that recognizes the output code, one that reconizes both, and one that recognizes the results of an analysis. Recognition of code is expressed with the asymmetric open arrows. A component that transforms input code into output code should understand both grammars, in order to be able to combine transformation components sequentially. This is expressed in Figure 2 by the merge operator. Note that the implementation should be able to merge grammars in a convenient way without having reduce/reduce and shift/reduce conflicts, so Lex+Yacc approaches are not always satisfactory. We do not have these problems, since we use generalized LR parsing that can handle general context-free grammars. Thus, we obtain a merged grammar: the I/O grammar that understands both the input and output code. Given this merged grammar a generator generates a generic code transformer. A component that performs an analysis should have knowledge of the input code and the types in which the data is presented, for instance, a Boolean value or a (natural) number. So, the analyzer generator takes as arguments the input grammar and the result data type to generate the generic analysis functions. Now we come to the PP part: for a specific reengineering task, say a transformation or an analysis, we can instantiate the generic components to obtain a specific component that implements the reengineering task. This instantiation consists of instantiating the non predictable parts of the component. In this way we can generate the various components that are necessary in a software renovation factory.

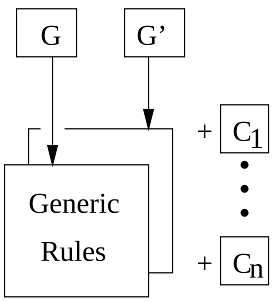

Figure 3: Maximal reuse of actual components.

An, in our opinion, very important consequence of the LP PP approach is that the generated components are maximally reusable and thus easily maintainable. We sketched the situation in Figure 3. In fact, components consist of a shared part that is generated from a grammar G containing the generic rules and usually small handcrafted parts $C_1, \ldots, C_n$ containing problem specific rules. Now suppose that we want to reuse the components $C_1, \ldots, C_n$ in a different dialect G'. Then we usually only have to regenerate the generated part and we do not have to modify any of the handcrafted parts. We will give an extensive example of this in Section 2.3.

We will explain in Section 2 our generic approach, in Section 3 we will apply it to COBOL, in Section 4 we will discuss implementation details, and in Section 5 we draw some conclusions.

1.1 Related Work

In [12] we can find a control-flow normalization tool for COBOL74 developed using the TAMPR system [4] – this is a general purpose program transformation system. With the use of REFINE [20] it is also possible to develop com-

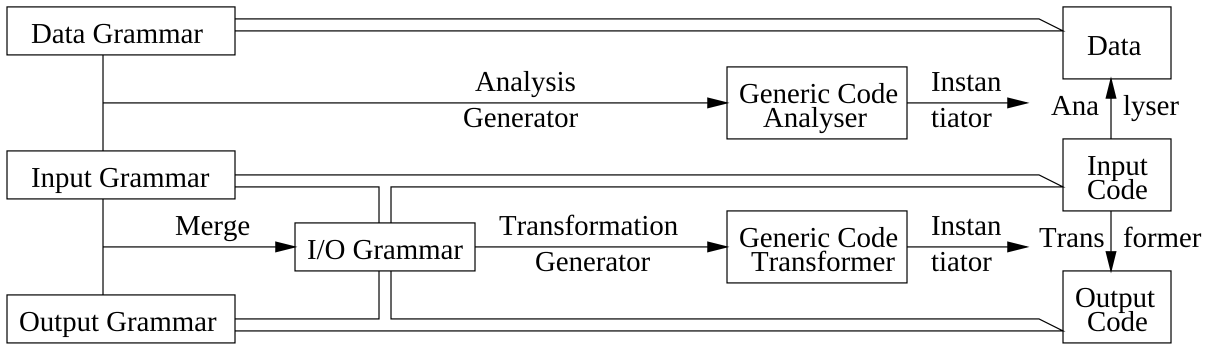

Figure 2: Generation of components for software renovation factories.

ponents for software renovation. With the TXL transformation system [9] it is also possible to construct software renovation components. All these systems share the property that the transformations are coded by hand. We propose a method to generate them from a given context-free grammar. One of the benefits of our approach is that we can reuse components for different dialects whereas in the above cases the components have to be rewritten.

The techniques we use to generate traversal functions to be used by the transformation and analysis components are similar to those to generate unparsers described in [8]. In attribute grammars [1] an implicit tree traversal function is present. It is comparable to our generated traversal functions. Since they are implicit it is not possible to manipulate syntax trees so it is difficult to express program transformations using that technology. Higher order attribute grammars [21] is an extended form of attribute grammars in which the traversal functions are no longer implicit. In principle, we think that it should be possible to implement our generic approach using higher order grammars. We have not seen any publication that discusses the generic construction of components.

2 Generic Components

It is feasible to generate analysis and transformation components, when the result types of the functions which implement these components are known in advance. In this section we describe the generation of these two types of generic components, we describe how to instantiate them, and we pay attention to their reusability.

2.1 Generating Generic Components

As we already announced in the introduction we are able to generate generic components of various types. In Figure 2 we depicted two such generic components: a generic code transformer and a generic code analyzer. Let us make clear that we think of program transformation as a batch oriented process and not as an interactive process. In this section we will give some details on how to generate generic components.

Just as with parser generators and unparser generators it is possible to develop generators for reengineering purposes. Given a grammar in some syntax definition formalism, say SDF [13], we are able to generate a generic code transformer and a generic code analyzer. These generated generic components consist of a collection of functions capable of recursively traversing a syntax tree. In our case they are either geared towards being instantiated with a specific code transformation problem or are precooked to perform a specific analysis task.

Next, we will explain how the generation process works by giving an example of an abstract grammar rule and showing the output of both the transformer and the analyzer generator. Then we will give a more elaborate example.

Suppose we have the following (abstract) context-free grammar rule in SDF-style: `L1 S1 L2 S2 L3 -> S3` Where `L1, L2, L3` stand for literals or keywords like `begin`, `end`, or `if`. `S1, S2, S3` are nonterminals, for example, `declaration`, `statement`, or `program`. In BNF-style the rule would be: `S3 ::= L1 S1 L2 S2 L3`. In the next two paragraphs we give a short characterization of both generic components and their generated output of the above grammar rule.

Generic Code Transformer We generate traversal functions for a given grammar. Their purpose can be language migration, dialect conversion, code restructuring, etc. For each grammar rule we generate one ASF equation (we recall that ASF [3] stands for algebraic specification formalism). We note that functions in algebraic formalisms are defined by means of equations. See Section 4 for more details on the underlying implementation. So we generate for the context-free grammar rule presented above the following ASF equation:

```
T_S3(L1 var_S1 L2 var_S2 L3)^{Attr*} =
  L1 T_S1(var_S1)^{Attr*}
  L2 T_S2(var_S2)^{Attr*}
  L3
```

Where `T_S3` is the top traversal function and `T_S1` and `T_S2` are traversal functions for the body. Moreover, `var_S1` is a variable representing the subtree with root `S1`, `var_S2` a variable for the subtree `S2` and `Attr*` a variable which represents a list of attributes which may contain values needed for specific transformations such as the introduction of new variables. Note that in this situation we leave the keywords (`L1-3`) untouched and that the traversal function traverses the abstract syntax tree. In fact, T_S3 is suggestive syntax: the _S3 is a postfix operator and T is a variable (of type TRANSFORM for which we must substitute some function name. By instantiating this variable we construct the actual components, as we will see in Section 2.2.

Generic Code Analyzer Here we generate slightly different recursive functions. The purpose of these components is to check for something, to perform some software measurement task, etc. We generate for our example context-free grammar rule the following ASF equation:

```
A_S3(Op,Data,L1 var_S1 L2 var_S2 L3)^{Attr*} =
  A_S1(var_S1)^{Attr*} Op A_S2(var_S2)^{Attr*}
```

Here A_S3 is the top analysis function and the others are for the body; Op is an operator variable, like +, or and; Data is a variable representing some default value, like 0 or true; the other symbols are as with the transform function. The difference with the transform function is that the analyze function does not preserve the structure of the context-free rule: the keywords (L1-3) are omitted. Another difference is the presence of the variable Op representing an operator. This operator prescribes how the evaluations of the subtrees S1 and S2 should be combined. Again, A_S3 is suggestive syntax: the _S3 is a postfix operator and A is a variable (of type ANALYSE for which we must substitute some function name. If we instantiate A we construct an actual analyzer.

Concrete Example Now we give a more elaborate example that serves as a running example for the remainder of this section. Consider the following simple production rules representing a language called Cat:

```
lexical syntax
    [ \n\t]     -> LAYOUT
    [A-ZO-9]+   -> ID
    "dog"       -> EXP
context-free syntax
    STAT             -> SERIES
    STAT ";" SERIES -> SERIES
    ID ":=" EXP                                -> STAT
    "if" EXP "then" SERIES "else" SERIES "fi" -> STAT
    "while" EXP "do" SERIES "od"              -> STAT
```

As can be seen from the lexical syntax, Cat contains the layout symbols space, newline, and tab. Identifiers are uppercase characters mixed with numbers. We have just one expression dog in Cat. The generated output of the generic code analyzer is a module called A-Cat containing the following equations:

```
[1] A_ID(Op,Data,id(var_CHAR*))^{Attr*} = Data
[2] A_EXP(Op,Data,exp(var_CHAR*))^{Attr*} = Data
[3] A_SERIES(Op,Data,var_STAT)^{Attr*} =
    A_STAT(Op,Data,var_STAT)^{Attr*}
[4] A_SERIES(Op,Data,var_STAT;var_SERIES)^{Attr*} =
    A_STAT(Op,Data,var_STAT)^{Attr*} Op
    A_SERIES(Op,Data,var_SERIES)^{Attr*}
[5] A_STAT(Op,Data,var_ID := var_EXP)^{Attr*} =
    A_ID(Op,Data,var_ID)^{Attr*} Op
    A_EXP(Op,Data,var_EXP)^{Attr*}
[6] A_STAT(Op,Data,if var_EXP then var_SERIES1
              else var_SERIES2 fi)^{Attr*} =
    A_EXP(Op,Data,var_EXP)^{Attr*} Op
   (A_SERIES(Op,Data,var_SERIES1)^{Attr*} Op
    A_SERIES(Op,Data,var_SERIES2)^{Attr*})
[7] A_STAT(Op,Data,while var_EXP do var_SERIES od)^{Attr*}=
    A_EXP(Op,Data,var_EXP)^{Attr*} Op
    A_SERIES(o,Data,var_SERIES)^{Attr*}
```

We see that seven ASF equations are generated. We generate for each grammar rule one ASF equation, except for the layout production rules since this sort is not in the abstract syntax tree to be analyzed. Consider, for instance equation [5], its left-hand side consists of the function name A_STAT and three arguments, the operator variable, the data variable, and the assignment statement. The nonterminals in the assignment statement are translated to function calls in the right-hand side of the equation. This right-hand side consists of a function applied to the left-hand side of the assignment, an application of the operator Op, and a function applied to the right-hand side of the assignment. The operator Op connects the parts of the right-hand side. In the next section we will show how to obtain an actual component from this generic one.

2.2 Instantiating Generic Components

The construction of components which perform reengineering tasks is rather straightforward when using our approach. It consists of two phases. Modifying the generated functions, and instantiating the variables in the generated functions. We will show this using the example above, and then we will show that the actual component is maximally reusable. Note that in Section 3 we will discuss several useful components which can be integrated in a renovation factory that have been constructed in the same way.

Our target is to construct a component for the language in the above concrete example. It counts the number of assignment statements in the code. This means that we overwrite equation [5], which is responsible for the generic analysis of assignments, by substituting cnt for A and replacing the right hand side of the equation by 1. We import the generic analysis functions of module A-Cat, state that cnt is a term of sort ANALYSE, which implies that cnt will make use of the generic equations of module A-Cat. This results in a component which can count the number of assignments in Cat programs.

```
imports A-Cat
exports
  context-free syntax
    "cnt" -> ANALYSE
equations
[5] cnt_STAT(Op,Data,var_ID := var_EXP)^{Attr*} = 1
```

Of course, this is an artificial example, but it shows exactly what has to be done in order to construct an actual component. The construction of transformation components can be done in a similar way. We treat more examples of analyzers and transformations in Section 3.

2.3 Reusability

Next we will show that our approach is maximally reusable. That is, we can completely reuse the component as long as the grammar of the new language does not affect the part of the original grammar on which the component is working. To show what we mean we define a language Dog, which is a dialect of Cat.

```
lexical syntax
    [ \n]        -> LAYOUT
    [A-Z][a-z]* -> ID
    "cat"        -> EXP
context-free syntax
    STAT "."       -> SENT
    STAT SENT      -> SENT
    ID ":=" EXP                   -> STAT
    "print" EXP                   -> STAT
    "foreach" ID "=" EXP "to" EXP "do" SENT  -> SENT
```

Note that only the production rule `ID ":=" EXP -> STAT` is the same both in Cat and Dog. Although the sorts `ID`, `EXP`, `STAT` occur in both languages, they are not the same. In fact, we have other layout (no tabs allowed), different identifiers (no numbers allowed), disjoint expressions (`cat` instead of `dog`), different statements (no `if`, no `while` but `print` and `foreach`), and no series of statements separated by a semi-colon, but sentences ending in a dot. Since the grammars for Cat and Dog share one production rule we still call Dog a dialect of Cat.

Now we show that we can reuse the assignment counter for the Dog dialect of Cat. This is possible since the grammar of Dog does not affect the part of Cat on which the assignment counter is working. All other parts differ. This is not a problem since all the equations over these parts are automatically generated. The generated output of the generic code analyzer is a module called `A-Dog` containing the equations:

```
[1] A_EXP(Op,Data,exp(var_CHAR*))^{Attr*} = Data
[2] A_ID(Op,Data,id(var_CHAR*))^{Attr*} = Data
[3] A_SENT(Op,Data,var_STAT.)^{Attr*} =
    A_STAT(Op,Data,var_STAT)^{Attr*}
[4] A_SENT(Op,Data,var_STAT var_SENT)^{Attr*} =
    A_STAT(Op,Data,var_STAT)^{Attr*} Op
    A_SENT(Op,Data,var_SENT)^{Attr*}
[5] A_STAT(Op,Data,var_ID := var_EXP)^{Attr*} =
    A_ID(Op,Data,var_ID)^{Attr*} Op
    A_EXP(Op,Data,var_EXP)^{Attr*}
[6] A_STAT(Op,Data,print var_EXP)^{Attr*} =
    A_EXP(Op,Data,var_EXP)^{Attr*}
[7] A_SENT(Op,Data,foreach var_ID = var_EXP1
        to var_EXP2 do var_SENT)^{Attr*} =
    A_ID(Op,Data,var_ID)^{Attr*} Op
   (A_EXP(Op,Data,var_EXP1)^{Attr*} Op
   (A_EXP(Op,Data,var_EXP2)^{Attr*} Op
    A_SENT(Op,Data,var_SENT)^{Attr*}))
```

We only have to tell the component that is should import the generic functions generated for Dog. We do this by changing the import section of the counter:

```
imports A-Dog
exports
  context-free syntax
    "cnt" -> ANALYSE
equations
[5] cnt_STAT(Op,Data,var_ID := var_EXP)^{Attr*} = 1
```

Of course, this example is artificial but it shows that when dealing with substantially different dialects of a language, we can still reuse components that have handcrafted parts on shared constructions.

3 Generating COBOL Reengineering Components

We will apply the approach that we proposed to construct reengineering components from a COBOL grammar. This COBOL grammar is based on real legacy code, see [7] for more details. We will show how to construct COBOL reengineering components from the generated generic components. More precisely, we will discuss a number of real life components that are useful in restructuring COBOL code and we discuss components to migrate COBOL74 code to COBOL85 code. So, we distinguish components to analyze, components to restructure, and components to migrate. We use analysis components to decide whether or not certain migration or restructuring components should be activated, e.g., if the COBOL74 special register `CURRENT-DATE` does not appear in the code, it is not necessary to perform a `CURRENT-DATE` transformation. What all these components have in common is that they are generated and that their specification is straightforward using the methods discussed in Section 2.2. The effort to construct them is usually measured in minutes rather than hours. Moreover, they are maximally reusable.

In Section 3.1 we will discuss the construction of two simple components, in Section 3.2 we will describe a restructuring which inserts missing `END-IF` in COBOL code. In Section 3.3 a migration component which deals with the `CURRENT-DATE` transformation is discussed and in Section 3.4 we compare the effort when constructing components from scratch or using our generic approach.

3.1 Two Simple Components

We will first treat two simple components: a restructuring component and a migration component. In the next subsections we will discuss two more complex restructuring and migration components.

We discuss a component that restructures a `MOVE CORR` statement with more than one receiving field into separate `MOVE CORR` statements containing the OS/VS COBOL statement connector `THEN`. It takes one equation by hand to construct this component. The other functions have default behaviour so the generated part takes care of that.

```
[1] mct_Stat(MOVE CORR B-exp
             TO Data-name Data-name-p)^{Attr*} =
    MOVE CORR B-exp TO Data-name THEN
    mct_Stat(MOVE CORR B-exp TO Data-name-p)^{Attr*}
```

`mct` stands for `MOVE CORR` transformation. `B-exp` stands for basic expression which can be a variable, a string, a constant, etc. `Data-name` stands for a variable name. From here onwards `-p` after a variable means one or more occurrences of it.

Only at the statement level (`Stat`) we want special behavior: a `MOVE CORR` with one or more `Data-names` should be changed into a `MOVE CORR` with statement connectors `THEN`. This is recursively defined in equation `[1]`.

Next we discuss a component that migrates the `THEN` statement connector (COBOL74 specific) into multiple statements. In COBOL/370 (a COBOL85 dialect) `THEN` as a statement connector is obsolete. We have to change three equations by hand:

```
[1] rth_Sent(var_Stat1 THEN var_Stat2 var_Sent1)^{Attr*} =
    var_Stat1 rth_Sent(var_Stat2 var_Sent1)^{Attr*}
[2] rth_Sent(var_Stat1 THEN var_Stat2.)^{Attr*} =
    var_Stat1 rth_Stat(var_Stat2)^{Attr*}.
[3] rth_Stat-p(var_Stat1 THEN
                var_Stat2 var_Cond-body)^{Attr*}=
    var_Stat1 rth_Stat-p(var_Stat2 var_Cond-body)^{Attr*}
```

`rth` stands for remove THEN. Equations [1-3] treat all the different cases of occurrences of THEN as a statement connector: at the beginning of a sentence [1] and at the end of a sentence [2]. Equation [3] takes care of the THEN inside IF-constructions. Note that `rth` does *not* remove the THEN which is part of an IF statement. We mention for the sake of completeness that the context-free grammar rule for THEN on the `Stat` level in SDF is:

```
Stat "THEN" Stat -> Stat {right}
```

Where {right} indicates that the binary operator THEN is right associative. Therefore, it is not necessary to process the `var_Stat1` subtree in the right-hand sides. Recursively, the statements which form a sentence or a list of statements are processed and if a THEN connector is found it is removed.

Example We will show how `mct` and `rth` process the following code fragment and why it is useful to transform and migrate in this way.

```
MOVE CORR A TO B C D. --mct-->

MOVE CORR A TO B THEN           MOVE CORR A TO B
MOVE CORR A TO C THEN --rth--> MOVE CORR A TO C
MOVE CORR A TO D.               MOVE CORR A TO D.
```

First, we see that `mct` transforms the COBOL74 code into COBOL74 code with a statement connector. Then `rth` removes the statement connectors. We use an extra phase because we want to keep components as simple as possible. It is possible to equip `mct` with the extra functionality so it could make from one statement an arbitrary number of statements, which is in this example from one to three, however this would make the component more complicated. In practice, transformations from n to m statements often occur. Therefore, we perform such transformations in two phases: first we use THEN as a statement connector to keep the number of statements constant and in the end we use the `rth` transformation to remove the connectors. Another example of the use of `rth` can be found in Section 3.3; in that example we use `rth` to transform one statement into two.

3.2 A Restructuring Component

Now that we have seen a simple restructuring component we discuss here a more involved one. Transformations that are more complex than the ones we treated thus far are, for instance, implicit to explicit scope terminating transformations in COBOL. In COBOL74 dialects there is, for instance, no way to explicitly terminate an IF statement. A dot (.) terminates all open IF clauses. When we have nested IF statements an ELSE closes a higher IF clause. Note that a dot is not possible in nested IF statements since that would close *all* open IF statements. In COBOL85 dialects an explicit scope terminator END-IF has been introduced. Although the END-IF is optional in COBOL85 dialects, much of the company code that we have seen uses this feature and abandons the implicit termination options since it is error prone.

There are not only END-IF terminators but also, e.g., END-READ, END-STRING, and END-WRITE terminators in COBOL85 dialects. We will discuss the construction of a restructuring component that transforms the implicit scope terminators for conditionals into an END-IF. We mention that other implicit to explicit scope termination components are constructed analogously, but simpler since their grammar is simpler.

First, we discuss a fragment of the COBOL grammar in SDF. These grammar rules recognize COBOL74 programs without scope terminators as well as COBOL85 programs with or without scope terminators (this is an example of an I/O grammar depicted in Figure 2). The grammar of the conditionals is rather complicated. The reason for this are the scope termination rules. We recall those from the ANSI Standard [2]:

The scope of the IF statement may be terminated by any of the following:

a. An END-IF phrase at the same level of nesting.

b. If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting.

c. A separator period.

We have expressed the above rules in the context-free grammar. Let us first explain that conditional expressions in COBOL85 dialects come in three flavors: the good, the bad, and the ugly. The good ones use the explicit scope terminator END-IF (rule a). The bad ones come in nested conditional COBOL constructs and seem, due to the nature of the implicit scope termination rules, to be incomplete (rule b). The expression `IF L-exp Stat1 ELSE Stat2` is an example of a bad conditional. The ugly ones are terminated with a separator period (rule c).

We explain the grammar fragment below. Since also normal statements can occur in the body of an IF statement, we first created a nonterminal called `Cond-body` in which those statements occur, possibly ended by a bad conditional (called `Bad-cond`). Then we are able to define what a `Bad-cond` is: an IF statement that is not explicitly terminated and that can contain such non terminated IF statements as well. Then we say that with the addition of an END-IF we can turn an incomplete IF statement into a complete Statement, called `Stat`. Finally, we express in the last two rules how to complete an IF with the addition of a separator period. Let us recall that a COBOL sentence (`Sent`) is one or more statements followed by a separator period (note that this implies that in COBOL74 dialects only IF sentences exist and not IF statements!).

```
Stat            -> Cond-body
Bad-cond        -> Cond-body
```

```
Stat Cond-body -> Cond-body
"IF" L-exp Cond-body "ELSE" Cond-body -> Bad-cond
Bad-cond "END-IF"             -> Stat
"IF" L-exp Cond-body "END-IF" -> Stat
"IF" L-exp Sent                     -> Sent
"IF" L-exp Cond-body "ELSE" Sent  -> Sent
```

Note that the above context-free grammar rules are only a selection of a complete context-free grammar definition of COBOL in SDF. An example of a code fragment that can be parsed by the above grammar fragment and that contains all three conditional flavors is the `slight-slot` program in Figure 4; we indicated the three possible `IF` statements in there, with good, bad, and ugly tags.

Now we can discuss the transformation component that turns bad and ugly conditionals into good ones; we call it `aei` for "add `END-IF`". It is the component that inserts an `END-IF` at the appropriate places.

We have to adapt three of the generated generic equations. This is not surprising, since we have three places in the grammar where an `END-IF` is missing and an implicit scope terminator takes care of the scope of the `IF` statement. They are the one that defines `Bad-cond` (that one misses an explicit scope terminator) and the two that take care of the separator period (they use a dot as terminator instead of an `END-IF`). Furthermore, we need an auxiliary function that removes a separator period from a sentence; it is called `rsp` which stands for "remove separator period".

```
[1] aei_Cond-body(Bad-cond)^{Attr*} =
      aei_Bad-cond(Bad-cond)^{Attr*} END-IF
[2] aei_Sent(IF L-exp Sent)^{Attr*} =
      IF L-exp rsp(aei_Sent(Sent)^{Attr*}) END-IF.
[3] aei_Sent(IF L-exp Cond-body ELSE Sent)^{Attr*} =
      IF L-exp aei_Cond-body(Cond-body)^{Attr*}
      ELSE rsp(aei_Sent(Sent)^{Attr*}) END-IF.
[4] rsp_Sent(Stat.) = Stat
```

In Figure 4 we have depicted a COBOL program that displays `slight` or `slot` depending on the value of `X`. Next to it we see the result of its transformation with `aei`. Let us briefly discuss how `aei` processes the input program so that the above definition of `aei` becomes clear.

We recall that all the `aei` equations are generated and that only a few of them are overruled. First, the generated function `aei_Program` is applied to the example program. Via the generated traversal functions we will reach the relevant parts of the code for `aei` where it will perform the desired transformation. Suppose we arrived at the ugly `IF` in this way. Then we recognize that this piece of code is a COBOL sentence. It matches with equation `[3]`. This results in removing the separator period by `rsp` and then adding `END-IF` followed by a separator period, see Figure 4. We continue with the remaining construction inside the ugly `IF`. This happens to be a `Cond-body` consisting of a `Stat` and a `Bad-cond`. On a simple `Stat` the function `aei` is the identity. We arrive at the bad conditional: on the `Bad-cond` it is not the identity: we changed that one by hand. According to rule `[1]` it will add an `END-IF` and will continue with the inside of the second `IF`. In the body of that `IF` we find a simple statement and a good conditional, which is also a statement. Now the body of that conditional is examined. Since the body consists of only two simple statements we are done.

```
IDENTIFICATION DIVISION. IDENTIFICATION DIVISION.
PROGRAM-ID. slight-slot. PROGRAM-ID. slight-slot.
DATA DIVISION.           DATA DIVISION.
WORKING-STORAGE SECTION. WORKING-STORAGE SECTION.
    01 X PIC 9 VALUE 1.      01 X PIC 9 VALUE 1.
PROCEDURE DIVISION.      PROCEDURE DIVISION.
par1.                    par1.
DISPLAY 'S'.             DISPLAY 'S'.
DISPLAY 'L'              DISPLAY 'L'
IF X=1             |     IF X=1
 DISPLAY 'I'       |      DISPLAY 'I'
 IF X=2         |  |      IF X=2
   DISPLAY 'N'  |  |        DISPLAY 'N'
 ELSE           |  |      ELSE
   DISPLAY 'G'  |  |u       DISPLAY 'G'
 | IF X=3       |b |g       IF X=3
g|   DISPLAY 'S'|a |l         DISPLAY 'S'
o| ELSE         |d |y       ELSE
o|   DISPLAY 'H'|  |          DISPLAY 'H'
d| END-IF       |  |        END-IF
ELSE               |     END-IF
 DISPLAY 'O'.      |     ELSE
DISPLAY 'T'.              DISPLAY 'O'
                         END-IF.
                         DISPLAY 'T'.
```

Figure 4: A program before and after `aei` transformation.

3.3 A Migration Component

In this section we will discuss a migration component with more advanced functionality than the `rth` component. This migration component transforms the OS/VS COBOL `CURRENT-DATE` special register to the special register `DATE` that is supported by both OS/VS COBOL and COBOL/370. We recall that `CURRENT-DATE` has the 8-byte alphanumeric format `MM/DD/YY` which is a common way of representing month, day, and year. `DATE` has the 6-byte alphanumeric format `YYMMDD` which is more geared towards comparison of dates. `CURRENT-DATE` is allowed as sending field in a `MOVE` statement; `DATE` is only valid as sending field in an `ACCEPT` statement. We have to deal with two issues for this component. First, we have to take care of the incompatible formats of the special registers and, second, we have to take care of the actual transformation of certain `MOVE` statements into `ACCEPT` statements. But before we can migrate, we first need to know if input source needs a `CURRENT-DATE` transformation at all. Therefore, we construct a component that analyzes a program and returns `true` if `CURRENT-DATE` occurs in it. We will first discuss this component and then the transformation component.

The component `cda` (`CURRENT-DATE` analyzer) checks whether or not `CURRENT-DATE` occurs in a program. Next we display the overruled equations for `cda`.

```
[0] cda_Program(var_Program) =
      cda_Program(or,false,var_Program)^{}
[1] cda_Stat(Op,Data,MOVE CURRENT-DATE
      TO var_Data-name-p)^{Attr*} = true
```

Equation `[0]` takes care of the correct initialization of the binary operator `Op` and the default value `Data` (see Section 2.2 for details). We introduce this extra equation to give the `cda` component its default values (it is not necessary to do this, only convenient). Note that the default is `false` so we assume no occurrences of `CURRENT-DATE` by

```
IDENTIFICATION DIVISION.     IDENTIFICATION DIVISION.
PROGRAM-ID. cdt-test.         PROGRAM-ID. cdt-test.
DATA DIVISION.                DATA DIVISION.
WORKING-STORAGE SECTION.      WORKING-STORAGE SECTION.
/ 01 TMP PIC X(8).            01 F-Date.
* 01 TMP PIC X(6).               02 F-YY PIC XX.
01 H-DATE.                       02 F-MM PIC XX.
   02 H-MM   PIC XX.             02 F-DD PIC XX.
   02 FILLER PIC  X.          01 TMP PIC X(8).
   02 H-DD   PIC XX.          01 H-DATE.
   02 FILLER PIC  X.             02 H-MM   PIC XX.
   02 H-YY   PIC XX.             02 FILLER PIC  X.
PROCEDURE DIVISION.              02 H-DD   PIC XX.
/ MOVE CURRENT-DATE TO TMP       02 FILLER PIC  X.
* ACCEPT TMP FROM DATE           02 H-YY   PIC XX.
  MOVE TMP TO H-DATE          PROCEDURE DIVISION.
  DISPLAY 'DAY   = ' H-DD.     ACCEPT F-Date FROM DATE
  DISPLAY 'MONTH = ' H-MM.     STRING F-MM '/' F-DD '/' F-YY
  DISPLAY 'YEAR  = ' H-YY.       DELIMITED SIZE INTO TMP
                               END-STRING
                               MOVE TMP TO H-DATE
                               DISPLAY 'DAY   = ' H-DD.
                               DISPLAY 'MONTH = ' H-MM.
                               DISPLAY 'YEAR  = ' H-YY.
```

Figure 5: A program before and after cdt transformation.

default. The operator or is used to combine the results of the cda analyzer on subtrees. Equation [1] recognizes a MOVE CURRENT-DATE TO expression and returns true.

We will use cda in the definition of the cdt component (this stands for CURRENT-DATE transformer). Before we display its equations, we discuss a way to perform the CURRENT-DATE transformation, we do this using the example program depicted in Figure 5.

The input program is the program on the left in Figure 5 with / omitted. This programs displays the current date as day, month, and year separately. One possible solution, proposed in [14], is to change the PIC string of TMP by making it X(6) and to change MOVE CURRENT-DATE into an ACCEPT statement. This is expressed in the same program by the statements preceded with the comment marker / instead of the original ones marked with * comment markers. This solution breaks down as soon as TMP is used in a context assuming the original format X(8). H-DATE is such a context. The output of today (04/14/97) yields DAY = 41, MONTH = 97, and YEAR = which is wrong: there is no day 41 and the YEAR field is empty. The cdt component returns the above right-hand side program. It introduces a fresh variable F-Date to store DATE in its (fresh) subfields: F-YY, F-MM, F-DD. Subsequently, we simulate the format of CURRENT-DATE by using the STRING statement. In this way the format of TMP is exactly the same as before the migration so the above problem will not arise.

Next, we discuss the construction of cdt. It has four equations.

```
[0] cda_Program(var_Program) = false
    ================================
    cdt_Program(var_Program) = var_Program

[1] cda_Program(var_Program0) = true,
    cdt_Program(var_Program0)^{id(F-Date)
      id(F-YY) id(F-MM) id(F-DD)} = var_Program1
    ==========================================
    cdt_Program(var_Program0) = rth_Program(var_Program1)^{}
```

```
[2] cdt_Data-desc-p(var_Data-desc-p)^{id(var_Id0)
      id(var_Id1) id(var_Id2) id(var_Id3)} =
      01 var_Id0.
          02 var_Id1 PIC XX.
          02 var_Id2 PIC XX.
          02 var_Id3 PIC XX.
      var_Data-desc-p

[3] cdt_Stat(MOVE CURRENT-DATE
             TO var_Data-name-p4)^{id(var_Id0)
      id(var_Id1) id(var_Id2) id(var_Id3)} =
      ACCEPT var_Id0 FROM DATE
      THEN
      STRING var_Id1 '/' var_Id2 '/' var_Id3
        DELIMITED SIZE INTO var_Data-name-p4
      END-STRING
```

Equation [0] defines cdt_Program under the condition that no CURRENT-DATE occurs in var_Program: then cdt_Program just returns the input unchanged. Note that in ASF, conditions are expressed with a double line; the condition is above it and if it is satisfied the rule below the line is applied. Equation [1] defines cdt_Program on var_Program0 to be the rth_Program (remove THEN) of another program var_Program1 under the following conditions. First, there is a CURRENT-DATE, this is checked by cda_Program. Second, the input program is extended with a fresh record, yielding var_Program1. The function id just injects identifiers into attributes, it has no equations. We use the Attr* here for the first time, it is used as a memory so that we can both *declare* the fresh variables in the working storage section and *use* the fresh variables in the PROCEDURE DIVISION. So, we make a change at two different locations in the program. This means that we can do global transformations using the Attr* mechanism. Equation [2] takes care of the insertion of the fresh variables in the WORKING-STORAGE SECTION. It matches the one or more present records in that section with variable var_Data-desc-p and it adds the fresh record F-Date to it. Finally in equation [3], on the statement level, cdt matches the MOVE CURRENT-DATE TO phrase and returns a new statement consisting of the ACCEPT part and the simulation part. Note that the THEN is removed in equation [1]. This is done to keep the number of statements a constant while transforming (see Section 3.1 for details).

Of course, cdt is not flawless: in order to migrate automatically, we have to know that the variables are fresh, so we have to check that with an analyzer. If there are non-fresh variables, we should make them fresh automatically. We will not describe this component since the scope of this paper is not to design a software renovation factory but to show how to generate its components. We are aware of the flaws of cdt and we can construct a cdt that is more sophisticated with the methods described in this paper.

Let us conclude this section with the remark that a component that takes care of the transformation of the OS/VS COBOL TIME-OF-DAY special register to the COBOL/370 TIME special register can be constructed analogously.

3.4 Handcrafted Versus Generated

In this section we will give the reader an indication of the effort it takes to conventionally construct components versus our generative approach. We took the components that we presented in Section 3 as a point of departure. We constructed the components that we generated in this paper also by hand in order to compare our approach with a conventional method. We expressed the amount of effort in the number of equations we had to specify in ASF in order to obtain the component. These figures are presented in Table 1. The total amount of generated equations is the same as the number of production rules in our COBOL definition, in our case we had 435 generated equations for each generic component. In the *Component* column we list the components of Section 3. The column *From Scratch* gives the number of equations that we needed to construct a component from scratch. The column *Adapted* gives the number of equations that have to be adapted or added on top of the generated equations to obtain the component. Components like `mct`, `rth`, and `aei` have a similar functionality in the sense that they only need to process the `PROCEDURE DIVISION`. If we construct a component by hand many equations are necessary to specify the location where the actual transformation should be performed. The column *Tuned* gives the total number of handcrafted equations necessary to obtain a component with a performance similar to a handcrafted one.

Next, we will discuss Table 2. The numbers presented in this table represent the maximum number of equations that can be applied when executing one of the components. In the *From Scratch* column we can see how many equations can maximally be applied to perform the transformation. In the next column we see that many more equations are involved. The functionality of the component is the same as with the handcrafted component. Note that the handcrafted component only looks at the appropriate place and the generic one looks everywhere. By adapting a few of the generated equations this is solved. This can be seen in the last column: the sum of the *used* generated equations and

	From	*435 Generated Equations*	
Component	*Scratch*	*Adapted*	*Tuned*
`mct`	23	3	4
`rth`	23	4	5
`aei`	23	5	6
`rsp`	1	1	1
`cda`	15	2	4
`cdt`	28	4	5

Table 1: Handcrafted equations for some components.

	From		
Component	*Scratch*	*Adapted*	*Tuned*
`mct`	23	435	24
`rth`	23	435	25
`aei`	23	435	26
`rsp`	1	1	1
`cda`	15	435	17
`cdt`	28	435	29

Table 2: Maximum number of used equations.

adapted equations equals the number equations we used to construct them by hand.

Let us discuss some typical figures. Since the `rsp` component only works on sentences, it will never traverse a whole program. So our approach does pays off in case a transformation requires the traversal of a significant part of the grammar. This becomes clear with the `cdt` component where we have to process two divisions. The handcrafted version takes more effort whereas the generic approach takes as much effort as a component that needs to process only one division like `aei`.

4 Implementation

This section discusses the implementation details that play a role in the generated components. We will briefly explain the ASF+SDF Meta-Environment and the two accompanying formalisms. For more details on ASF+SDF we refer to [3, 13] and for a description of the ASF+SDF Meta-Environment to [17].

ASF+SDF is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. It is a combination of two formalisms ASF (Algebraic Specification Formalism [3]), and SDF (Syntax Definition Formalism [13]). The ASF+SDF formalism is supported by an interactive programming environment, the ASF+SDF Meta-environment [17]. This system is called *meta-environment* because it supports the design and development of programming environments.

ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. SDF allows the definition of concrete (i.e., lexical and context-free) syntax. Abstract syntax is automatically derived from the concrete syntax rules.

ASF+SDF has already been used for the formal definition of a variety of (programming) languages and for the specification of software engineering problems in diverse areas. See [5] for details on industrial applications.

ASF+SDF specifications can be executed by interpreting the equations as conditional rewrite rules or by compilation to C. For more information on conditional rewrite systems we refer to [18] and [16]. It is also possible to regard the ASF+SDF specification as a *formal* specification and to implement the described functionality in some programming language.

The generic components that we described in Section 2 are generated ASF+SDF modules. The modular structure of the underlying context-free grammar is clearly visible in the generated components. Each module in the context-free grammar corresponds with a module in the generated components. Each context-free grammar rule corresponds with an equation in a generic component. We generate so-called *default* equations. A default equation is applied when none of the other equations was successful. So in our case as soon as there is no adaptation, the system will use the generated equations, and if we add a rule by hand, it will be used instead of the default equation. The generated equations presented in the example of Section 2.1 should have been marked as default equations, e.g., `[default-1] A_SERIES(...)`. This mechanism enables us to develop components with sophisticated functionality by only specifying a few equations.

5 Conclusions

We have developed a powerful generative approach to construct two types of components for software renovation factories: components to analyze code and ones to transform it. In Section 2 we explained that our approach uses a context-free grammar as input and that it generates from that grammar those generic components. We showed that switching from one dialect to another can be done easily while maximally reusing components constructed using our approach. This is an important feature since in software renovation factories one should be able to use different dialects in a flexible way. Moreover, we have shown that our approach can be applied to large languages like COBOL: we generated components that can play a realistic role in a COBOL renovation factory. We elaborately discussed applications, both simple introductory and more advanced ones. They all served the purpose of migrating or restructuring COBOL code. The components for analyzing and transforming COBOL code were generated and then instantiated by adding and overruling a few of the generated equations. We provided a comparison of the effort it took to construct components using generic techniques and by hand. We hope to have shown that our approach allows the rapid construction of reliable components with advanced functionality. It is our estimation that a software renovation factory is an appropriate paradigm to deal with massive amounts of legacy code, and that the generation of components for such a factory is of economic importance.

Acknowledgements We thank Paul Klint (UvA/CWI) and the referees for their valuable comments.

References

[1] H. Alblas and B. Melichar, editors. *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[2] American National Standards Institute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[3] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[4] J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[5] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[6] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[7] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. Technical Report P9702, University of Amsterdam, Programming Research Group, 1997.

[8] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[9] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[10] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[11] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*, page 14. Gartner Group, 1996.

[12] T. Harmer, P. McParland, and J. Boyle. Using knowledge-based transformations to reverse engineer COBOL programs. In *11th Knowledge-Based Software Engineering Conference*. IEEE-CS-Press, 1996.

[13] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[14] IBM Corporation. *COBOL/370 Migration Guide*, 1 edition, 1992.

[15] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[16] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175–193, 1984.

[17] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[18] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[19] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, unix programmer's supplementary documents, volume 1 (PS1) edition, 1986.

[20] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[21] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*