Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation

R. Farrow, Declarative Systems, Inc., farrow@ernie.berkeley.edu
T. J. Marlowe, Seton Hall University, marlowe@cs.rutgers.edu
D. M. Yellin, IBM T. J. Watson Research Center, dmy@watson.ibm.com

Abstract

This paper introduces Composable Attribute Grammars (CAGs), a formalism that extends classical attribute grammars to allow for the modular composition of translation specifications and of translators. CAGs bring to complex translator writing systems the same benefits of modularity found in modern programming languages, including comprehensibility, reusability, and incremental meta-compilation.

A CAG is built from several smaller *component AGs*, each of which solves a particular subproblem, such as name analysis or register allocation. A component AG is based upon a simplified phrase-structure that reflects the properties of its subproblem rather than the phrase-structure of the source language. Different component phrase-structures for various subproblems are combined by mapping them into a phrase-structure for the source language. Both input and *output* attributes can be associated with the terminal symbols of a component AG. Output attributes enable the results of solving a subproblem to be distributed back to anywhere that originally contributed part of the subproblem, e.g. transparently distributing the results of global name analysis back to every symbolic reference in the source program.

After introducing CAGs by way of an example, we provide a formal definition of CAGs and their semantics. We describe a subclass of CAGs, called separable CAGs, that have favorable implementation properties. We discuss the novel aspects of CAGs, compare them to other proposals for inserting modularity into attribute grammars, and relate our experience using CAGs in the Linguist translator-writing system.

1 Introduction

Modern programming practice recognizes the importance of modularity and composability in the description, implementation, and execution of large programs. A programming application should be designed and implemented as the combination of several *components,* each physically and conceptually separate from the others. The conceptual separateness of components allows most of the information embodied in a component to be hidden behind narrow, precisely-defined interfaces. The benefits of such an organization are well-known and widely discussed in the literature; they include: (1) ease of specification, (2) clear description, (3) aid in verification, (4) interchangeability between different "plug-compatible" components, (5) reuse of components across applications, (6) separate analysis/compilation of components, and (7) incremental or parallel evaluation. Many programming language constructs have been introduced to support program components and their conceptual separateness, including: procedures and functions, modules and classes, typed imports and exports, and parameterized instantiation of components.

Classic AGs [20] are themselves a modular formalism: the components are the *productions*, the interface to a production consists of the symbols incident on the production together with the attributes associated with those, and each production hides from the rest of the AG the semantic rules that define the attributes of its interface.

This form of modularity is organized around a phrase-structure for the source language and it works well for some applications. However, experience [6, 10] suggests that for realistic applications it is inadequate. Symptoms of the inadequacy of this form of modularity include large, complex interfaces whose external description is nearly as complicated as the internal implementation that they are designed to abstract. In concrete terms, realistic AGs have too many attributes, inter-related in such complicated ways that one must read the whole AG to understand their relationship. This is a classic indication of a mismatch between the components of a modularization and the problem at hand.

We herein introduce a new approach, *Composable Attribute Grammars (CAGs)*, that provides support for modularity in attribute grammars. This makes it easier to use AGs to specify programming languages semantics and to design compilers and other translators. A CAG is built from several smaller *component AGs*, each of which solves a particular subproblem, such as name analysis, checking type-conformance, register allocation and assignment, etc. A component AG is based upon a simplified phrase-structure that reflects the properties of its subproblem rather than the phrase-structure of the source language. Different component phrase-structures for various subproblems are combined by mapping them into a phrase-structure for the source language via a mechanism that looks like simple semantic rules attached to productions of the source language phrase-structure. Both input and *output* attributes can be associated with the terminal symbols of a component AG. Output attributes are the most significant and novel feature of CAGs. They enable the results of solving a subproblem to be distributed back to anywhere that originally contributed part of that subproblem, e.g. the results of global name analysis can be transparently distributed back to every symbolic reference in the source program.

There have been several other approaches for modularizing AGs, including extended AGs [21, 26], attribute-coupled grammars (ACGs) [12, 13], higher-order AGs (HOAGs) [25], cascaded evaluation [10], and modular AGs [6]. Each of these provides some modularity, and we have adopted ideas and notation from several of them, but none provides a uniform and general mechanism for attaining the flavor and degree of modularity provided by CAGs.

We introduce CAGs through a brief and familiar example—a simple compiler for Pascal. After the major features of CAGs have been informally presented, we give a formal definition and a simple semantics. We describe two different implementation techniques and define subclasses of CAGs on which those techniques can be used. Next we review how CAGs support modularity and composability, and compare CAGs with other proposals (mentioned above) for supporting modularity in AGs. We have implemented one of our subclasses of CAGs in an AG-based translator-writing-system, and we briefly discuss that implementation and relate our experience so far in designing and implementing CAGs. Finally, we present some issues we have not yet resolved and describe opportunities for further research.

2 An Example

2.1 Component grammars

In a compiler, semantic processing often involves several phases, each of which performs a distinct task. For example, consider the phase commonly called *identifier resolution*. This phase identifies the distinct scopes in a program, builds a symbol table containing the declarations of the program, and associates with each identifier reference in the program a symbol table entry. An AG A_{Res} for identifier resolution in Pascal is given in Figure 1. [1]

Certain points are apparent. Perhaps the most significant is the simplicity of the underlying CFG and of the AG rules. There are only three sorts of entities: declarations, new scopes, and identifier references. This allows one to use an extremely sparse abstract syntax for the grammar. Second, the AG expresses the semantics of identifier resolution not just for Pascal, but for a number of different languages. Finally, alternative semantics for identifier resolution, for example, the semantics of Algol or Ada, could be implemented with different semantic rules for the same abstract grammar, since it does not obscure the semantics of identifier resolution with the syntactic or other semantic details of a particular language.

Another distinct aspect of semantic processing is type checking. Once again, we can isolate the semantics of type-checking into a small AG, A_{Type}, given in Figure 2. For ease of presentation, and to save space, we provide this grammar only for a very restricted set of Pascal.[2] The resulting grammar is comparatively small, natural to the task at hand, and unobscured by irrelevant syntax/semantics. Moreover, it can be extended to handle most Pascal types and operators without losing its clarity or simplicity. It can even be used for any language whose (abstract) type system and operator type signatures are compatible with Pascal's—since the grammar uses an abstract syntax, the symbols used for operators, and even operator associativity and precedence, are immaterial.

Given an expression in which the types of the leaves are known, A_{Type} will determine the type of the expression, as well as the type of each overloaded operator, and generate error messages where appropriate. If a particular type is expected for the expression (for ex-

[1]The notation is a slight extension of ordinary AG notation. Productions of a CFG have the (slightly modified) BNF form: `<label> : X0 ::= X1 ... Xk` where label is a symbolic name for the production. Semantic (attribute grammar) rules are attached to productions in the usual way. Attributes for grammar symbols are declared using the following syntax: `<symbol_name> : <symbol_type> [ : <attribute_decls> ]` where symbol_name is a context-free grammar symbol and symbol_type is either t (terminal) or n (non-terminal). Each attribute declaration for a symbol has the form `<attrib_name>: <direction>: <type>` where attrib_name and type are the name and the type of the symbol's attributes. The direction of non-terminal symbol attribute is either inherited (h) or synthesized (s); the direction of a terminal symbol attribute is either in or out. It is useful to think of the former as input parameters supplied to the grammar and the latter as output parameters exported by the grammar. The semantic rules resemble assignment statements that may define more than one target attribute

[2]Types are real, integer, or boolean. Operators include only the relational operators (<, >, ...), arithmetic operators (+,-,*, ...), and the boolean operators AND, OR.

```
-- grammar symbols

    envREF   : t  NAME:in:tp_NTX,          OBJ:out:entry,     MSGS:out:tp_msgs;
    envDCL   : t  NAME:in:tp_NTX,          OBJ:in:entry;
    envgoal  : n;
    envitems : n  ENVin:h:symbol_table,    ENVout:s:symbol_table;

-- productions
```

```
env_goal  : envGoal ::= envitems.
    envitems.ENVin = emptyENV;

env_empty : envitems ::= .
    envitems.ENVout = envitems.ENVin;

env_list  : envitems ::= envitems1 envitems2.
    envitems1.ENVin = envitems.ENVin,
    envitems2.ENVin = envitems1.ENVout,
    envitems.ENVout = envitems2.ENVout;

-- this production introduces a new scope
env_nest  : envitems ::= envitems1.
    envitems1.ENVin = envitems.ENVin,
    envitems.ENVout = envitems.ENVin;
```

```
-- this introduces a variable reference
env_ref   : envitems ::= envREF.
    envREF.OBJ = envLookup(envitems.ENVin,
                 envREF.NAME),
    envREF.MSGS = if envREF.OBJ = nullObj
                then "invalid reference"
                else "" fi,
    envitems.ENVout = envitems.ENVin;

-- this introduces a variable declaration
env_dcl   : envitems ::= envDCL.
    envitems.ENVout =
          envUpdate(envitems.ENVin,
                 envDCL.NAME, envDCL.OBJ);
```

Figure 1: Specification of A_{Res}

ample, for the right side of an assignment statement), the expected type is compared to the actual type, and appropriate error messages generated.

Many other semantic aspects can be isolated and expressed in concise abstract grammars. These include overloaded operator resolution, memory allocation and address assignment, register allocation, instruction selection, as well as local optimization and data flow analysis. One could build libraries of reusable component AGs if there were only a way to take such grammars and embed them in the semantic specification of a compiler for a particular language. In the next section, we show how this can be done.

2.2 The Glue Grammar

In the grammars given above:

- The context-free phrase structures are different, and neither is closely related to (and both are simpler than) the phrase structure for Pascal.
- Each grammar requires inputs and produces outputs. These outputs may be needed for other component grammars. For example, information from A_{Res} is used in A_{Type}.

Thus, to be able to use a set of these small component AGs as modules, we need to provide a means to:

- Build the phrase structure for each component AG from the phrase structure in Pascal.
- Provide inputs to each component AG, and direct (and possibly transform) the outputs of a component AG to appropriate destinations, typically to the inputs of other components.

Instead of writing code to handle these requirements, we can embed these actions in a "master" AG. We call this AG the *glue*, since it pastes together individual component AGs into a unified semantic specification.

An Example Consider the Pascal production corresponding to an assignment statement, `variable ASGN expression`. In A_{Res}, this involves updating the environment with the references in `expression` plus a reference to `variable`. In A_{Type}, this involves checking that the result type of `expression` is compatible with the type of `variable` retrieved from the environment. The glue rules for this production are given in Figure 3.

The semantic rules of the glue production specify parts of the phrase structure for each component, as well as values for in attributes of component terminals in this phrase structure. The phrase structure for each

```
-- grammar symbols
    typRESULT: t  MSGS:out:tp_msgs, TYPEIN:in:tp_type  TYPEOUT:out:tp_type;
    typOP    : t  MSGS:out:tp_msgs, TYPE:out:tp_type,  OP:in:operator_class;
    typARG   : t                    TYPE: in:tp_type;
    typegoal : n;
    typexp   : n                    TYPE: s :tp_type;
-- productions
type_check : typgoal ::= typRESULT typexp.     -- check expected type with computed type
  typRESULT.TYPEOUT = typexp.TYPE,
  typRESULT.MSGS = if typRESULT.TYPEIN != typRESULT.TYPEOUT
                     and NotCoercible(typRESULT.TYPEOUT, typRESULT.TYPEIN)
                     and typRESULT.TYPEIN != univ_type and typRESULT.TYPEOUT != univ_type
                   then "Cannot assign incompatible types" else "" fi;

typ_leaf : typexp ::= typARG.                   -- leaf of expression tree
  typexp.TYPE = typARG.TYPE;

typ_relop : typexp ::= typexp1 typOP typexp2.   -- operator is relational
    . . .   -- semantics elided for brevity
 ;

typ_sign : typexp ::= typOP typexp1.            -- unary arithmetic operator
    . . .   -- semantics elided for brevity
 ;

typ_bool : typexp ::= typexp1 typOP typexp2.    -- a boolean binary operator
    . . .   -- semantics elided for brevity
 ;

typ_arith : typexp ::= typexp1 typOP typexp2.   -- an arithmetic binary operator
  typexp.TYPE = typOP.TYPE,
  typOP.MSGS, typOP.TYPE =
    if TypeIsArithmetic(typexp1.TYPE) and TypeIsArithmetic(typexp2.TYPE)
    then "", CoerceToHigherType(typexp1.TYPE, typexp2.TYPE)
    else "operands to arithmetic operator must be arithmetic", univ_type   fi;
```

Figure 2: Specification of A_{Type}

component is obtained by identifying attributes in the glue with nonterminals in the component and pasting these *syntactic* attributes together (with some terminal symbols as well) using *production constructors.*

For instance, the glue nonterminal asgn_stmt has attributes envitems and typegoal corresponding to nonterminals in the resolution and typechecking grammars, respectively. The constructor env_list builds a production in A_{Res} whose left-hand side (LHS) is an envitems nonterminal, and whose right-hand side (RHS) is two envitems nonterminals.[3] Similarly, the constructor type_check builds a production in A_{Type} whose LHS is the typegoal nonterminal and whose RHS is a typRESULT terminal followed by a typEXP nonterminal.

[3]Note that the first nonterminal envitems is produced by applying the env_ref constructor to an envREF terminal.

The REF terminal symbol used in the env_ref constructor is introduced by a local declaration in the glue that declares REF to be of type envREF and also defines the only in attribute required for REF, defining REF.name to have the same value as variable.name. Also, in A_{Res}, every envREF terminal produces an out attribute OBJ, the symbol table entry for the variable referred to by envREF. The value of REF.OBJ is used in the glue production to define the input attribute RESULT.TYPEIN in A_{Type}.

This illustrates some of the power of CAGs. In the glue, we need only build the phrase structure for A_{Res} and supply the name of each variable referenced, and we automatically get back symbol table entries for these variables. We can then use these outputs as

```
asgn_stmt ::= variable ASGN expression  -- Pascal grammar production

-- rules for constructing the phrase structure for id-resolution
  asgn_stmt.envitems = env_list(env_ref(REF),expression.envitems),
  REF:envREF = {NAME => variable.name},

-- rules for constructing the phrase structure for typechecking
  asgn_stmt.typegoal = type_check(RESULT,expression.typexp),
  RESULT:typRESULT = { TYPEIN => TypeOf(REF.OBJ) },

  asgn_stmt.MSGS = concat(REF.MSGS, concat(RESULT.MSGS, expression.MSGS));
```

Figure 3: Component Phrase Structures for Assignment Statement

inputs to other component grammars.

Although A_{Res} and A_{Type} interact via input and output attributes of terminals, the grammars do not need to "know" of one another, as all their interactions occur via the glue. Hence, we preserve modularity. Indeed, we could change the data structures used by the identifier resolution grammar, or even its semantics, without changing A_{Type}. If the phrase structure and interface of the changed component remain the same, we would not even need to change the glue. Thus we could have interchangeable component AGs having the same phrase structure but expressing different identifier resolution rules, e.g., those of Pascal versus those of Algol.

3 Composable AGs and Their Semantics

3.1 A formal definition of CAGs

A composable attribute grammar consists of a set of components, a glue, and an interface between them.[4] Component grammars are classical attribute grammars [20] enriched by allowing *input* and *output* attributes for terminals. (In general, terminals in abstract grammars will exist only as carriers for input or output attributes.) The glue grammar is a classical AG enriched with syntactic attributes and production constructors. The interface establishes the correspondence between the glue and the components. More formally:

A Composable Attribute Grammar *component* consists of

1. A context-free grammar given by the 4-tuple $G_i = \langle N_i, T_i, S_i, P_i \rangle$, where each production has a unique label.
2. An attribute grammar A_i associating a set of inherited and/or synthesized attributes to each nonterminal, and a set of input and/or output attributes to each terminal, and a set of attribute-defining rules to each production. Each output attribute, but no input attributes, are defined in A_i. The *interface* of a component C_i consists of its context-free grammar G_i and its association of typed inputs and outputs to terminals.

[4]There exist many possible denotations for CAGs. For the purposes of this paper, interfaces are implicit within the component and glue grammars. However, it is likely that implementations will find it convenient to explicitly declare interfaces, especially in the glue grammar.

A *glue attribute grammar* for a Composable Attribute Grammar is an attribute grammar A_{gl} with underlying context-free grammar G_{gl}, where

- The glue uses two special sets of types: *nonterminal* and *terminal*. Each terminal type has a set of associated typed input and output parameters. A nonterminal type may also be designated as a *root*.
- The attributes of the glue include *syntactic* attributes, each of a particular nonterminal type. Syntactic attributes are defined either by copy rules or by *production constructors*, also referred to as *syntactic rules*. A production constructor takes as parameters nonterminal and terminal types and returns a particular nonterminal type. The actual argument for a nonterminal parameter in a production constructor can either be a syntactic attribute (of the appropriate type) or another production constructor (returning the appropriate type). The actual argument supplied for a terminal parameter is a constant of that type.
- If a terminal constant is used in a syntactic rule of a glue production, then it must define the input parameters of that terminal. Other rules of the glue production can reference the output parameters of that terminal.
- Definitions of syntactic attributes must obey the *single syntactic use requirement:* each instance of

a syntactic attribute, except those whose nonterminal type is designated as a root, is used exactly once in defining some other syntactic attribute [13]. A syntactic attribute whose nonterminal type is designated as a root is not used to define any other syntactic attribute. (Thus the glue constructs parse trees in the components rather than DAGs, and there are no dangling trees—no trees are rooted at a symbol other than a component root.)

A Composable Attribute Grammar consists of a set C of components C_i, and a glue attribute grammar, A_{gl}, with a consistent interface. Namely,

1. The glue has an implicit interface for each component it uses, determined by the nonterminal and terminal types, the designated root type, the production constructors, and the typed input/output parameters associated with terminals for that component.
2. For each component, there must be a 1-1 mapping between nonterminal/terminal types and production constructors in the glue and terminals/nonterminals and productions in the component. These mappings must be consistent in the usual way; e.g., if the glue production constructor p maps to the component production q, then the nonterminal type returned by p must map to the nonterminal on the left-hand side of q. The designated root must map to the start nonterminal in the component.
3. Additionally, the input/output parameters for each terminal used in the glue must have the same types as the input/output attributes for the corresponding terminal of the component.

In this paper, the mapping between the glue and component interfaces is made explicit by using the same names for matching items; e.g., production constructors in the glue are given the same name as production labels in the components, and so on.

3.2 The Semantics of CAGs

We give a semantics for CAGs by defining a transformation that turns any CAG into a classical monolithic AG. The semantics of a CAG is the semantics of its induced monolithic AG, if the latter is non-circular; if the induced AG is circular, then the CAG has no well-defined semantics.

The transformation we use is an extension of a technique called *descriptional composition*, first proposed by Giegerich and Ganzinger [12]. In descriptional composition, each syntactic attribute of the glue corresponding to a non-terminal X of a component is replaced by a collection of new *induced* glue attributes corresponding to the attributes of X in the component grammar. These new attributes are defined by copy rules, or by semantic rules "pulled back" from the component, or by compositions of such rules, depending upon whether the syntactic attribute was defined by a copy rule, a production constructor, or a composition of production constructors. We extend descriptional composition so that we "pull back" the semantics from each component into the glue simultaneously. Secondly, we include semantic rules for computing input/output attributes. We illustrate descriptional composition in our setting by way of an example.

Consider the assignment statement given in the previous section for a glue grammar (Figure 3). In Figure 4 we show what this production would look like after performing the transformation. Since `asgn_stmt` has an associated syntactic attribute from A_{Res}, namely `envitems`, the transformed grammar induces two new attributes for `asgn_stmt`, namely `ENVin` and `ENVout`. The same is true of the `expression` nonterminal of the glue. Similarly, `expression` acquires the attribute `TYPE` from A_{Type}. When the semantic rules for production constructors `env_list`, `env_ref` and `type_check` are "pulled backed" into the transformed grammar, we need to introduce local variables into the production to hold values that, in the untransformed grammar, were held by input/output attributes of component grammar terminal symbols. Hence the local variables `REFname`, `REFobj`, and `REFmsgs` replace the input/output attributes `REF.NAME`, `REF.OBJ`, and `REF.MSGS`. Composition of production constructors can also introduce local variables for missing attributes associated with implicit nonterminals. Hence the local variables `envitems1ENVin` and `envitems1ENVout` replace A_{Res} attributes `envitems1.ENVin` and `envitems1.ENVout`. Of course, many of these temporaries can be automatically optimized away by the AG meta-compiler.

3.3 Modular Analysis and Evaluation

One evaluation strategy for CAGs is to transform a CAG into a monolithic AG by using descriptional composition and then to apply classical AG evaluation techniques [16, 18]. However, for reasons discussed below, this approach has some drawbacks. Thus the bulk of this section is devoted to presenting another evaluations strategy for CAGs, *separated evaluation*, which avoids descriptional composition. This approach can only be used for a restricted class of CAGs, but it permits a greater degree of modularity when it is applicable.

Applying descriptional composition and then building an AG evaluator does not support the separate analysis and meta-compilation of AG components and glue such as that supported by separate compila-

```
asgn_stmt ::= variable ASGN expression -- Pascal grammar production
    envitems1ENVin:symbol_table = asgn_stmt.ENVin,
    envitems1ENVout:symbol_table = envitems1ENVin,
    expression.ENVin = envitems1ENVout,
    asgn_stmt.ENVout = expression.ENVout,
    REFname:tp_NTX = variable.name,
    REFobj:entry = envLookup(envitems1ENVin, REFname),
    REFmsgs:MSGS = if REFobj = nullObj then "invalid reference" else "" fi,
    RESULTtypein:tp_type = TypeOf(REFobj),
    RESULTtypeout:tp_type = expression.TYPE,
    RESULTmsgs:MSGS = if RESULTtypein != RESULTtypeout
                       and NotCoercible(RESULTtypeout, RESULTtypein)
                       and RESULTtypein != univ_type and RESULTtypeout != univ_type
                  then "Cannot assign incompatible types" else "" fi,
    asgn_stmt.MSGS = concat(REFmsgs, concat(RESULTmsgs, expression.MSGS));
```

Figure 4: An Example of Descriptional Composition

tion features of programming languages. Furthermore, one would like to be able to reason about the well-formedness of CAGs by reasoning only about the components and the glue individually. In general, this is not possible, as stated by the following theorem:

Theorem 1 *Let G be a CAG. Even if each component and the glue of G are noncircular, the induced monolithic AG constructed from G may still be circular.*

What this means is that for arbitrary CAGs, analysis and meta-compilation must take into account the entire CAG, since local properties on the component and glue AGs do not translate into global properties on the CAG.

To overcome these problems, we have formulated a restricted class of CAGs, called *separable* CAGs. Separable CAGs bound the potential indirect interaction among different components so that each particular component instance can either (indirectly) contribute information to another component instance, or (indirectly) receive information from that component instance, but not both. The restrictions are stated solely in terms of information flow in the glue AG, and they induce a partial order on the component instances constructed by the glue AG.

Definition 1 *A CAG G is separable if (1) each component AG is noncircular, and (2) the glue is noncircular even under the assumption that every output parameter depends upon every input parameter in any component tree constructed in the glue.*

In contrast to Theorem 1, separable CAGs are guaranteed to induce only a well-defined monolithic AG:

Theorem 2 *If G is a separable CAG then the induced monolithic AG constructed from G is noncircular.*

The definition of separability depends solely on local properties of the glue and component AGs. It requires that each component be noncircular and that the glue, transformed to meet the requirements of the definition, also be noncircular. The purpose of this transformation is to guarantee that any circularity that may arise in the induced monolithic AG, when the components and glue by themselves are noncircular, can be detected by analyzing the glue alone. The transformation forces the glue to assume that all input attributes of a component are used to define each output attribute of that component. Hence any transitive dependences that may arise when instantiating the CAG with a particular component will have already been taken into account in the analysis of the glue.

This glue transformation can be done by replacing each syntactic attribute with a pair of "dummy" attributes, one inherited and one synthesized. The synthesized attributes of these pairs are then made to depend on one another, and the inherited attributes are made to depend on each other both based on the original dependencies of their corresponding syntactic attributes. The result is to cause every input attribute to be a dependency of a "dummy" attribute of the component AG's goal symbol through a chain of "dummy" synthesized attributes, and to cause every output attribute to depend on this goal symbol's "dummy" attribute through a chain of "dummy" inherited attributes. The details of this construction are a little more involved than this, but they are straightforward; for the sake of brevity they are left to the imagination of the reader.

Separable CAGs are useful for detecting circularity by testing each component and glue grammar separately. More importantly, if a CAG is separable, we can build a static evaluators modularly, building evaluators for each component and for the glue separately. As we discuss in Section 7, we have actually constructed a system for evaluating separable CAGs.

One example of a static modular evaluation strategy for CAGs is an adaptation of the ordered evaluation strategy for classical AGs [16]. We say that a CAG G is *ordered separable* if (1) G is is separable, (2) each component grammar is ordered , and (3) $transform(gl)$ is ordered, where gl is the glue of the CAG and $transform$ is the glue transformation described above. If a CAG is ordered separable than an "ordered" evaluator can be built for the glue and for each component separately.

An ordered evaluator for an AG associates, at meta-compilation time, a sequence of instructions for each production in the AG. Each instruction is either an **EVAL X.b** instruction, indicating the evaluation of the attribute $X.b$ of the production, or a **VISIT k** instruction, indicating a descent into the k^{th} $(k > 0)$ child of X or an ascent to the parent of X $(k = 0)$. An ordered separable evaluator can have one additional instruction: **CALL X.root**, where root is a syntactic attribute of nonterminal type X representing the root of a component parse tree. This instruction passes the tree rooted at **X.root** (with instantiated input attributes) to the evaluator for the component grammar and returns the tree with the instantiated output values. [5] One can show that:

Theorem 3 *If a CAG is ordered separable, then the translation produced by an ordered separable evaluator (for a syntactically legal input string) is correct.*

By correct we mean that the results of the translation—the synthesized attributes of the root of the semantic tree—are the same as specified by the induced monolithic AG.

The chief advantage of separable evaluation strategies (such as ordered separable) over building an evaluator for the monolithic AG induced by the CAG is *compilational modularity.* Unlike classical AG systems [8, 17, 22], where a single change can render the entire generated evaluator invalid, in a separable evaluator, a change to a single component or glue AG only renders the evaluator for that component invalid. Other advantages to separable evaluation are likely as well, such as speedier evaluators (since each component evaluates over a typically smaller tree), more storage-efficient evaluators (since once a component evaluator finishes it can discard its semantic tree), and more flexible evaluators (since each component AG evaluator can use a different evaluation strategy if desired).

[5] Recall that if a CAG is separable, then there is an evaluation order that computes all input attributes of a component before any of its output attributes are referenced. Thus the **CALL** instruction can be scheduled after all input attributes have been computed but before any output attributes are referenced.

We have described two strategies for implementing CAGs: descriptional composition and separated evaluation. Each of these determines a (sub-)class of CAGs: all well-defined CAGs and separable CAGs, respectively. These strategies/subclasses can be viewed as the endpoints of a continuum. We believe that there other intermediate points on this continuum which will prove valuable for their combination of descriptional power and efficient implementation. We have informally identified one such class, which we call the *k-separable* CAGs. These are CAGs whose components can be evaluated in **k** different "passes" over their structure-trees, where output attributes of one pass can be used to define input attributes of a later pass.

4 Properties of CAGs

The previous section presented a formal definition of CAGs and a simple semantics for them, descriptional composition, as well as two implementation strategies, descriptional composition at meta-compile time and separated evaluation. This section highlights what we consider the most important features and properties of CAGs and discusses how they support modularity and composability in AGs. The next section further analyzes features of CAGs in the process of comparing them with other proposals for supporting modularity in AGs.

Modularity, Abstraction, and Information Hiding Each component grammar of a CAG describes a separate subproblem and its solution. The component production-constructors used in the rules of the glue AG serve to *abstract* a component subproblem from the original, larger problem described by the glue AG. The details of solving this subproblem are hidden behind the interface of the component. Its solution can be derived, analyzed, and verified without regard for the particular context in which the subproblem was originally embedded. Conversely, the glue AG can be designed, understood, and verified without having to know how component subproblems are solved. The glue AG is responsible only for abstracting a relevant and well-formed instance of the component subproblem, and it can then depend on the accuracy of the solution to that subproblem.

Output Attributes The most important and novel aspect of CAGs is the ability to associate output attributes with terminal symbols of a component AG. This feature allows the semantics of the component AG to specify the outputs of a component with the same granularity as inputs were specified to it. Without output attributes, component phrase structure is used only for construction of the component tree and initialization of input attributes; with output attributes, it is also used to anticipate outputs.

In the sample Pascal front-end of Section 2, the output attribute envREF.OBJ (of Figure 1) is crucial to the brevity of the component AG for identifier resolution, as well as to the succinct description of its instantiation in the glue AG.

Our example Pascal CAG shows another example of the utility of output attributes in the way error messages are generated. The order of error messages in Figure 3 agrees with code order, even though the messages are created in the component AGs, not the glue AG. If error messages were not returned via output attributes this would be much more complicated; e.g., if identifier errors and type errors were collected separately, and then had to be interleaved.

The usual alternative to output attributes is to produce a single, complexly-structured (root) value as the unique output value of a component and then decompose that structured value into its constituents via semantic rules in the glue AG. This expands the interface between the glue AG and a component to include both the structure of the output aggregate and the rules for decomposing it into constituents. It also expands the attribution rules of the glue AG to include appropriate instances of those rules for decomposing the component's output aggregate value. Output attributes allow us to move this work into the component AG, or just eliminate it altogether, thus substantially narrowing the interface between component and the glue AG.

Hierarchical Structure We have so far described a CAG as a "bush": a glue AG and a set of sibling components. However a component AG can itself be a nested CAG consisting of a glue AG and a number of nested components. Thus, an arbitrarily deep hierarchy of CAGs can be assembled in which every component AG except the leaves are glue AGs with their own components. Such interior component AGs would have input and output attributes for communicating with their parents in the hierarchy, and would define/reference the input/output attributes of their children.

For example, one component of a CAG for Pascal might describe code generation using nested component AGs that separately specify optimization, storage allocation, register assignment, and instruction selection. The optimization component could be implemented by further expanding it into several data-flow analysis components, a local optimization component, and a global optimization component.

Our descriptional composition semantics (see Section 3) for hierarchical CAGs is valid only so long as a CAG may not (even indirectly) instantiate another instance of itself. A more sophisticated semantic model is required if we allow such *recursive* CAGs. Such a model is beyond the scope of this paper; this is an area in which we are continuing research.

Reusability of Components One of the most promising benefits of CAGs is in reusing component grammars in several different translators. For instance, a component AG such as A_{Type}, if parameterized by standard types and operators and rules for implicit coercion, could be instantiated in different glue AGs for Pascal and C. Such a "standard" CAG for type-conformance would capture the common traits of the two languages—that overloaded operator disambiguation and operator/operand conformance is determined by examining the actual types of operands and the expected types of operators in a single bottom-up pass over an expression tree. Such differences between the two languages as statement syntax, operator precedence and associativity, and visibility rules are irrelevant to type-conformance and would be handled within the separate glue AGs or other component AGs. At the end of section 6 we discuss some preliminary experience in this direction and propose using hierarchical components to delineate common aspects of translations.

This opens the door to creating libraries of components, both for standard language semantics and for common compiler tasks like identifier resolution or overloaded operator identification. We envision combining such standard components with a specialized glue AG and a few special-purpose components to obtain complete translators for a particular language. Libraries of components for different language semantics, code generation schemes, and optimization strategies would enable a much higher level of *experimental* research in the corresponding fields of programming language and compiler design. Such libraries could also be powerful tools for teaching, e.g., compiler design and comparative programming languages.

5 Related Work

There have been a number of prior attempts to introduce modularity in attribute grammar specification. Of those, this work is most closely related to Attribute Coupled Grammars [12, 13], Cascaded Evaluation [10], Higher-order Attribute Grammars [25], and Modular Attribute Grammars [6].

Attribute Coupled Grammars decompose complex translations into a *sequence* of steps, each of which (conceptually) constructs parse trees for the next step via syntactic attributes and rules. However, (1) the flow is strictly linear and unidirectional, (2) as a consequence, flow to non-immediate successors must be passed through a sequence of copy rules, or components must be combined, (3) as another consequence, it is only meaningful to construct a single tree in the target, and (4) each component has to know the syntax and interface of its successor, severely limiting component reuse.

Cascaded Evaluation also passes syntactic and semantic information between grammars: a stream of terminals, with initial attribute values, is produced in one grammar, which, when consumed in the other, produces a parse and a subtranslation. In Cascaded Evaluation information is returned only at the root of a subtree. Because only the yield of the parse in the target grammar is produced, Cascaded Evaluation cannot use ambiguous component grammars; also, it is recursively unsolvable to determine at (meta-)compile-time whether an AG expressed via Cascaded Evaluation is well-formed.

Higher-order Attribute Grammars [25] allow for construction of multiple instances of "component" parse trees during evaluation of a global AG. However, there is no descriptional modularity in the syntax: the "glue" grammar must be enriched with the productions and grammar symbols of each "component." Like cascaded evaluation, information computed by HOAGs is available only at the root of the component tree. This mechanism does not support modularity particularly well, and that was not its main goal.

With Modular Attribute Grammars (MAGs) one can create AGs from a central AG and components, similar to the monolithic AGs built by descriptional composition from a glue AG and set of component AGs. However, MAG "components" are instantiated through pattern matching and templates rather than explicitly. Multiple instances of a given component may be built at different positions in the original tree. All attributes of "matched" symbols are available to a component instance for reference in or definition by a semantic rule of the component. This mechanism allows the "outputs" of a component to be distributed around the tree in a manner similar to the facility provided by CAG output attributes. However, MAGs provide no well-defined separation of components from main grammar; there is no hard interface analogous to the component phrase-structure of a CAG. Thus, a component only has meaning, and can only be understood, as a part of the larger monolithic AG.

Other general approaches including descriptional modularity [15, 21, 23, 24, 26] and pattern matching [7] are orthogonal to our own and could easily be included in a CAG system.

Approaches to decomposition and structured communication, some reminiscent of features in CAGs, have been included in *ad hoc* AG systems for parallel or incremental compilers or AG evaluators [1, 2, 11, 19]. Such approaches to parallelization and incrementality are mostly orthogonal to CAGs and we expect that they can be used with CAGs without major modification.

6 Implementation and Experience

We have been experimenting with CAGs for several months now and have some preliminary results in two areas: (1) the use of CAGs for design and documentation of particular programming languages and compilers for them, and (2) an implementation of separated evaluation and its applicability.

Although we have not emphasized it in this abstract, CAGs are just as useful in designing and documenting programming languages and other translations as they are in implementing compilers and translators. Recently one of the authors faced the problem of making major changes to an AG-based compiler for the hardware description language VHDL [10], a large and complex language that is an extension of Ada. In particular, the identifier resolution mechanism for this compiler was to be changed to make it more efficient, but we sought to preserve the correctness of the existing implementation. Our strategy was to write a component AG for identifier resolution in VHDL and to then modify the existing monolithic AG for VHDL to make it a glue AG that instantiates an instance of this component.

This strategy worked without any difficulty and the improved identifier resolution implementation was encapsulated entirely within the component AG. Furthermore, the component AG is quite useful as a training tool and as documentation of how identifiers are resolved in this language. It is **much** briefer and more precise than the VHDL Language Reference Manual [27].

We have implemented separated evaluation of CAGs within the Linguist translator-writing-system [4]. We used this separately-evaluating version of Linguist to generate a Pascal compiler out of the glue and component AGs that were described in Section 2. This experience taught us several lessons.

First, the component AG for identifier resolution in Pascal is not separable within our Pascal glue AG. On learning this we carefully analyzed the corresponding component for VHDL and found that it was also not separable. Briefly put, the reason for this is that some visible declarations (e.g. a variable) depend on other visible declarations (e.g. a type). Thus, the result of looking up a type's name, an output of the component AG, is used to build the dictionary entry for a variable which, because it is the object visible under the variables' name, must be an input of the component AG. However, for a separable CAG, all inputs must be available before any outputs are available.

Our Pascal compiler's other component AGs, e.g.

for type analysis and PCODE generation, were separable and the separately-evaluating Linguist successfully generated a compiler from them. The identifier resolution component had to be evaluated via descriptional composition, which was implemented (somewhat awkwardly) using macros.

Another lesson we have learned is that for a component AG to be reusable across several different CAGs, its phrase structure must be as general as possible. The instantiation of a component AG via constructors used in the rules of the glue AG is essentially a mapping of a portion of the glue AG's phrase-structure onto the phrase-structure of the component. If many different phrase-structures are to map easily onto a given component's phrase-structure, then the latter should impose a minimum of restrictions. A good example of this is how lists are expanded. We have found that lists of elements should be described through a tree-structured derivation rather than as left-recursive or right-recursive list derivations. Use `L ::= L L | E.` rather than either `L ::= E | L E` or `L ::= E | E L`. The example makes strong use of the possible ambiguity of component phrase structure; this is acceptable because the semantics in the component are resolved by the induced phrase structure, and are not derived from a parse by the component AG.

We have most recently begun to design components that specify semantic processing for the C and C++ programming languages. This has given us some insight into the reusability of component AGs and how to design CAGs for reuse. Our preliminary conclusions are that CAGs that are directly instantiated in a glue AG for language A typically can not be directly reused in language B unless A and B are closely related. However, subproblems such as name resolution, type conformance, and disambiguating overloaded functions/operators do have very deep similarities that can be exploited.

The trick is to set up a (shallow) hierarchy of component AGs and to reuse the leaves of that hierarchy. Consider name resolution in C and Pascal. If a single CAG for name resolution were directly instantiated in glue AGs for both languages then either the component would have to be complicated and contorted, or the semantic rules in the glue AGs would have to specify much of the name resolution semantics, or both. Nonetheless the name resolution semantics of both languages do have much in common, such as: nested block structure, inheritance of visibility from one block to another nested within it, the ability to override a global declaration with a local one, etc. Such broad similarities can be captured in a single, reusable CAG which can then be instantiated in separate CAGs that describe the peculiarities of name resolution in C or Pascal.

7 Conclusions and Directions for Further Research

We have developed CAGs as an approach to modularity in attribute grammars. CAGs are fully general, allow nearly arbitrary combinations of components and exchanges of information, and express many standard attribute grammar problems elegantly and concisely. Moreover, CAGs possess descriptional and organizational simplicity: they are easy to understand, and facilitate specification and explanation of complicated semantic and translation tasks. Finally, and most importantly, CAGs are highly modular, allowing reuse or modification of components, and providing for modular meta-compilation.

CAGs appear extremely fruitful for further research. Among the issues and opportunities are:

- Is there a natural *modular* way to define CAG semantics? Is there a reasonable way to provide a (fixed-point?) semantics for self-referential CAGs?

- Are there other useful evaluation strategies for CAGs?

- Instances of component AGs whose interface attributes do not depend on one another offer clear opportunities for parallel evaluation. There is potential here for some "coarse-grained" parallelism among different component AG instances as opposed to the "fine-grained" parallelism among individual semantic rules that AGs have traditionally offered. How can useful opportunities be detected? What are the best strategies for exploiting these opportunities? How much speedup in translation will result?

- Similarly, CAGs should expose opportunities for "coarse-grained" *incremental* evaluation that would complement the traditional "fine-grained" attribute-by-attribute incremental evaluation strategies. We envision a scheme in which individual instances of component AGs are re-evaluated or not depending on whether any of their input attributes have changed.

- Can one construct libraries of separately compilable component AGs to facilitate the reuse of translator design?

Acknowledgement

We thank John Field and Roger Hoover for reading a preliminary version of this paper and suggesting improvements for its presentation.

References

[1] H. Alblas. Incremental simple multi-pass attribute evaluation. In *Proceedings of the NGI/SION 1986 Symposium*, pages 319–342, 1986.

[2] H.-J. Boehm and W. Zwanenpoel. Parallel attribute grammar evaluation. Technical Report TR-87-55, Rice University, Houston, TX, June 1987.

[3] B. Courcelle. *Attribute grammars: definitions, analysis of dependencies, proof methods*, pages 81–102. 1984.

[4] *The Linguist translator-writing system User's Manual, Version 6.3*, Declarative Systems, Inc. Palo Alto, CA, February 1991.

[5] P. Deransart, M. Jourdan, and B. Lorho, editors. *Attribute Grammars: Definitions, Systems and Bibliography.* Number 323 in Lecture Notes in Computer Science. Springer-Verlag, May 1988.

[6] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Computer Journal*, 33(3):164–172, 1990.

[7] C. Farnum. *Prototyping optimizing compilers.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, December 1990.

[8] R. Farrow. LINGUIST-86: Yet another translator writing system based on attribute grammars. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982. Published as SIGPLAN Notices, 17 (6).

[9] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, 1986. Published as SIGPLAN Notices, 21 (7).

[10] R. Farrow and A. Stanculescu. A VHDL compiler based on attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 120–130, 1989. Published as SIGPLAN Notices, 24 (7).

[11] N. M. Gafter. Parallel incremental compilation. Technical Report 349, University of Rochester, June 1990.

[12] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 172–184, 1984. Published as SIGPLAN Notices, 19 (6).

[13] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–424, 1988.

[14] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, July 1990.

[15] R. K. Jullig and F. DeRemer. Regular right-part attribute grammars. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 171–178, June 1984. Published as SIGPLAN Notices, 19 (6).

[16] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[17] U. Kastens, B. Hutt, and E. Zimmerman. *GAG: A practical compiler generator.* Number 141 in Lecture Notes in Computer Science. Springer-Verlag, 1982.

[18] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 32–49. Association for Computing Machinery—SIGPLAN, 1976.

[19] E. A. Klein. Attribute evaluation in parallel. In *Proceedings of the Workshop of Parallel Compilation*, page 8, Kingston, Ontario, May 6–8 1990.

[20] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, February 1968. Correction, 5 (1), 95–96, March 1971.

[21] O. L. Madsen. *On defining semantics by means of extended attribute grammars*, pages 259–299. Number 94 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[22] T. Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, MA, 1984. ACM Distinguished Dissertation Award.

[23] T. Reps. and T. Teitelbaum *The Synthesizer Generator reference Manual, Third Edition.* Springer-Verlag, New York, NY, 1989. Texts and Monographs in Computer Science.

[24] M. Tieman. Removing redundancy in attribute grammars. Technical Report ACA-239-87, MCC, July 1987.

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

[26] D. A. Watt and O. L. Madsen. Extended attribute grammars. *Computer Journal*, 26(2):142–153, May 1983.

[27] *IEEE Standard VHDL Reference Manual.* IEEE Std 1076-1087. The Institute of Electrical and Electronic Engineers, Inc. New York, NY, March 31, 1988.