AML: Attribute Grammars in ML*

Sofoklis G. Efremidis
Khalid A. Mughal
John H. Reppy

TR 93-1401
December 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This report is available as Report No. 89, Reports in Informatics, University of Bergen.
AML: Attribute Grammars in ML*

Sofoklis G. Efremidis
Cornell University

Khalid A. Mughal
University of Bergen

John H. Reppy
AT&T Bell Laboratories

December 1993

*This report is available as Technical Report 93-1401, Department of Computer Science, Cornell University and as Report No. 89, Reports in Informatics, University of Bergen.
AML: Attribute Grammars in ML

Sofoklis G. Efremidis*  
Cornell University  
sofoklis@cs.cornell.edu

Khalid A. Mughal†  
University of Bergen  
khalid@ii.uib.no

John H. Reppy  
AT&T Bell Laboratories  
jhr@research.att.com

December 17, 1993

Abstract

Attribute grammars are a valuable tool for constructing compilers and building user interfaces. This paper reports on a system we are developing, called AML (for Attribution in ML), which is an attribute grammar toolkit for building such applications as language-based programming environments using SML. This system builds on the proven technology of efficient attribute evaluation, while using a higher-level foundation for the implementation of interactive systems. It supports a general and uniform platform for building applications that can manipulate attributed terms and allow access to attribute values. We describe the design of the AML system, its current implementation status, and our plans for the future.

1 Introduction

Attribute grammars provide a formalism for assigning meaning to parse trees of a context-free language [Knu68]. Because of their syntax-directed form and declarative style, they provide a useful notation for specifying compilers [KHZ82] and language-based editors [RT88]. This paper reports on a system we are developing, called AML (for Attribution in ML), which is an attribute grammar toolkit for building applications such as language-based editors using SML [MTH90].

AML is a spiritual heir to the Synthesizer Generator project at Cornell University [RT88], which focused on efficient incremental evaluation techniques, and the Pegasus project at AT&T Bell Laboratories [RG86], which focused on providing a high-level foundation for interactive systems. In our system, we are building on the evaluation technology of the Synthesizer Generator, while using a higher-level foundation for the implementation.

In the next section, we give some background about attribute grammars. Then we describe our specification language for attribute grammars, followed by a description of the internals of our system. Lastly, we discuss the project’s status and future plans. An earlier description of this project was presented in [EMR92].

*This work was supported, in part, DARPA-ONR Grant N00014-91-J-4123.
†Support for this work provided, in part, by the Norwegian Research Council and the University of Bergen. The author would like to express his sincere thanks to the Dept. of Computer Science, Cornell University, for giving him the opportunity of spending his sabbatical (academic year 91-92) at Cornell.
| Let:  | $e_0 ::= ID \ e_1 \ e_2$ | $e_1.env = e_0.env$ |
|       |                           | $e_2.env = \text{insert}(e_0.env, (ID, e_1.value))$ |
|       |                           | $e_0.value = e_2.value$ |
| Const: | $e_0 ::= \text{NUM}$ | $e_0.value = \text{NUM}$ |
| Use:  | $e_0 ::= ID$ | $e_0.value = \text{lookup}(e_0.env, ID)$ |
| Sum:  | $e_0 ::= e_1 \ e_2$ | $e_1.env = e_0.env$ |
|       |                           | $e_2.env = e_0.env$ |
|       |                           | $e_0.value = e_1.value + e_2.value$ |

Figure 1: A simple attribute grammar

2 Attribute grammars

An attribute grammar is a context-free grammar (CFG), together with a set of attributes for each nonterminal and a set of attribute evaluation rules for each production. An attribute is either synthesized or inherited. For each production $p : X_0 ::= X_1 \ldots X_{np}$, there are evaluation rules that define the synthesized attributes of $X_0$ and the inherited attributes of $X_1, \ldots, X_{np}$. These attributes are known as the output attributes of $p$. Each evaluation rule defines the value of an output attribute in terms of other attributes in the production. Systems based on attribute grammars tend to extend this basic model in various ways. Two common extensions, which are supported by AML, are local attributes and syntactic references. Local attributes are attributes that are associated with a specific production. Syntactic references are references to grammar symbols in the attribute evaluation rules.

Figure 1 gives an example of a simple attribute grammar. There is a single nonterminal ($e$) with two attributes ($value$, $env$), and four productions (Let, Const, Use, Sum). The symbols $\text{NUM}$ and $ID$ are terminals, representing integer literals and identifier names. The attribution rules are given to the right of the productions. This grammar computes the value of expressions involving integer constants, addition, and a simple variable scheme. Note the use of references to the terminal values. The environment is passed down the expression tree using the inherited $env$ attribute, and the resulting value is passed up via the synthesized $value$ attribute. An expanded version of this grammar is given as an AML specification in Appendix B.

2.1 Attribute evaluation

Each node of a parse tree in the underlying CFG is labeled with instances of the attributes associated with the nonterminal at the node. The evaluation rules define a set of constraints on the attribute instances, and computing the attribute values can be viewed as a constraint solving problem. An
Figure 2: The attributed tree for the expression "let x = 1 in x + 2 end"

attributed tree is said to be consistent if its attribute values satisfy the constraints defined by the attribution rules. Figure 2 gives the attributed tree for the expression:

\[
\text{let } x = 1 \text{ in } x + 2 \text{ end}
\]

using the grammar from Figure 1. The graph formed from the attribute instances and the dependencies between them is called the attribute dependency graph. Most evaluation strategies require that the dependency graph be acyclic, although there are fixed-point techniques for handling grammars with cyclic dependencies [Far86, WJ88, Jon90].

The simplest way to consistently attribute a tree is to topologically sort the attribute dependency graph, and then to evaluate the semantic rules in topological order. This strategy guarantees that the inputs to a semantic rule will be available when the rule is evaluated. For many grammars, however, more efficient and specialized evaluation strategies can be used. Attribute grammars are classified by their evaluation strategies; for example, the parser generator yacc implements a grammar in which all attributes are synthesized and can be evaluated in a single bottom-up pass.

Techniques for evaluating attribute grammars can be divided into dynamic and static [Alb91a]. Dynamic evaluators use the dependency graph of the specific tree that they are evaluating to order their work, while static evaluators use a static ordering of dependencies between attributes of productions in the grammar. One can also distinguish between batch and incremental evaluation. A batch evaluator takes an unattributed tree and attributes it, whereas an incremental evaluator takes
 Productions

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Let: } e_0 ::= ID \ e_1 \ e_2 & \text{Const: } e_0 ::= NUM & \text{Use: } e_0 ::= ID & \text{Sum: } e_0 ::= e_1 \ e_2 \\
\text{EVAL} \ e_1.env & \text{EVAL} \ e_0.value & \text{EVAL} \ e_0.value & \text{EVAL} \ e_0.env \\
\text{VISIT} \ e_1 & \text{SUSP} & \text{SUSP} & \text{VISIT} \ e_2 \\
\text{EVAL} \ e_2.env & \text{EVAL} \ e_1.env & \text{VISIT} \ e_1 & \text{EVAL} \ e_0.value \\
\text{VISIT} \ e_2 & \text{EVAL} \ e_0.value & \text{SUSP} & \text{SUSP} \\
\text{EVAL} \ e_0.value & \text{VISIT} \ e_1 & \text{EVAL} \ e_0.value & \text{SUSP} \\
\text{SUSP} & \text{SUSP} & \text{SUSP} & \\
\hline
\end{array}
\]

Figure 3: Evaluation plans for the simple attribute grammar.

an already attributed tree plus a modification to the tree, and re-attributes it. Incremental evaluation is useful in optimizing compilers, where optimizations rewrite the parse tree, and is crucial in programming environments, where edits are represented as sub-tree replacements (see Section 2.3).

2.2 Ordered attribute grammars

One important class of attribute grammars is the class of Ordered Attribute Grammars (OAGs) [Kas80]. This class of grammars is interesting because it includes most useful “real-world” grammars, and it is possible to generate efficient evaluators for them.

OAGs have the characteristic that for each symbol in the grammar, there is a partial order over the attributes of that symbol, such that in any context of the symbol, the attributes instances are evaluable in that order. This property allows the construction of fixed-plan evaluators, which consist of statically determined evaluation plans for each production. These plans consist of a sequence of three kinds of instructions:

**EVAL:** evaluate an attribute,

**VISIT:** visit a child for the \(i\)th time (i.e., transfer control to the child’s plan), and

**SUSP:** suspend evaluation (i.e., transfer control back to the parent).

The VISIT and SUSP instructions are essentially “call” and “return.” Kastens gives a number of run-time evaluation techniques for these plans in [Kas80].

The grammar given in Figure 1 is an OAG. The attributes of \(e\) are ordered by \(env < v\). The evaluation plans for this grammar are given in Figure 3. Note that since each node is visited exactly once, the subscripts on the VISIT instructions have been omitted. Also note that the children of \(\text{Sum}\) are visited from right to left; this is just an artifact of the planning algorithm, the left to right order would also be correct.
2.3 Subtree replacement and incremental evaluation

In his seminal thesis, Reps showed that attribute grammars provide a useful formalism for defining language-based editors [Rep82]. Such systems use attributed abstract syntax trees to represent programs being edited and map editing operations to subtree replacements (e.g., a delete operation is implemented by replacement with a null tree). After a subtree replacement operation, the tree will no longer be consistent; Reps and others have described so called “optimal” evaluation algorithms for restoring attribute consistency [Rep82, Yeh83, Hoo87, Hud91]. These algorithms basically work by change propagation; i.e., by starting with a set of possibly inconsistent attributes and propagating changes through the tree (when an attribute value is changed to become consistent, its successors may become inconsistent). Applications of this technology include interactive systems, such as structured editors for programming languages [FJM+84, RT88], interactive theorem provers [Gri87] and incremental code generators [Mug88], as well as optimizing compilers [Alb91b].

3 The design of AML

We have been developing a system, called AML, for supporting attribute evaluation in ML. Our approach to supporting attribute grammars in SML is based on compiler (or generator) technology. Our system takes a specification as input, and generates a collection of SML modules that implement the specification and support code. These are then combined with additional grammar-independent modules to construct a complete system. We chose this approach, rather than trying to embed AML in the interactive SML system, because it provides more flexibility for analysis and optimization. In addition, we have a number of design goals:

- Easy interaction between SML and AML. It should be straightforward to feed attributed trees into SML code, and, likewise, to attribute the results of SML computations.

- Attribute grammars can be large (for example, the specification of a Pascal editor in the Synthesizer Generator system is over 9,000 lines of SSL code), so it is important to support modular specifications.

- The specification language should be a minimal extension of SML; we have tried to avoid unnecessary new keywords, and to follow the syntactic style of SML.

- There are many different classes of attribute grammars, and extensions to attribute grammars, as well as different run-time evaluation techniques. The system should be designed to support experimentation in all of these areas.

4 The AML specification language

An AML specification consists of a collection of related declarations; in many ways, it is similar to an SML structure definition. It has the form
grammar name = struct declarations end

where name is the name of the grammar being specified. There are six basic kinds of declarations in an AML specification:

- termtype declarations define terminal symbols.
- prodttype declarations define nonterminals and their productions.
- root declarations define root nonterminals.
- attribute declarations define attributes.
- attribution declarations define semantic rules.
- SML top-level declarations are used to define auxiliary types and functions.

In the remainder of this section, we describe the first five of these in more detail, and illustrate them with examples. A complete syntax for AML specifications can be found in Appendix A.

4.1 Syntax

The syntax of the grammar is defined by the termtype, prodttype, and root declarations. AML specifications deal with abstract syntax and most terminals, such as keywords and punctuation, are omitted from the grammar. Some terminals, however, carry semantic information, and are included in the grammar. The termtype declaration is used to define these terminals. For example:

```
  termtype int = int
  termtype ident = Name.name
```

In general, any monomorphic SML type expression is allowed on the right-hand-side of a termtype declaration.

The nonterminals and productions of a grammar are defined using the prodttype declaration.

```
prodttype exp
  = Use of ident
  | Const of int
  | Sum of (exp * exp)
  | Diff of (exp * exp)
```

Mutually recursive prodtypes are declared using the keyword and, as in the following example:

```
prodttype stmt
  = Block of stmt_list
  | Assign of (ident * exp)
  | While of (exp * stmt)

and stmt_list
  = StmtListNil
  | StmtListCons of (stmt * stmt_list)
```
In addition to defining the syntax of the grammar, the `termtype` and `prodtype` declarations also result in the generation of equivalent SML `type` and `datatype` declarations.¹

The other syntax declaration is the `root` declaration. This is used to distinguish certain nonterminals as roots of free standing abstract syntax trees. The `root` declaration does not affect other aspects of the specification, but is used in the interface to the outside world. For example, the evaluator generator may define an evaluation function for each root.

4.2 Attributes

Once a nonterminal has been defined, attributes can be declared for it. For example:

```plaintext
attribute exp
  with
    synth value : int
    inher env : (ident * int) list
end
```

Attributes can have any monomorphic SML type. To allow for more concise specifications, factored declarations are allowed, which define a collection of attributes for several nonterminals:

```plaintext
attribute stmt, stmt_list
  with
    inher env : (ident * int) list
end
```

The attributes of a nonterminal can be defined by several different `attribute` declarations.

The local attributes of productions are also defined using `attribute` declarations. For example:

```plaintext
attribute Let, Use
  with
    local error : string option
end
```

defines the local `error` attribute for the `Let` and `Use` productions.

The AML specification language also supports combined `prodtype` and `attribute` declarations as a syntactic extension. The declaration:

```plaintext
prodtype nonterm with attrs end
  = Prod of rhs
  with local-attrs end
```

is equivalent to the declarations:

¹I.e., replace `termtype` with `type`, and `prodtype` with `datatype`.
prodtype nonterm
    = Prod of rhs

attribute nonterm with attrs end

attribute Prod with local-attrs end

4.3 Semantic rules

The semantic rules of a grammar are defined using attribution declarations. In its simplest form, an attribution declaration defines names for the symbols of a production using a SML pattern, and evaluation rules for the attributes. For example:

attribute e0 : exp
    = (Sum(e1, e2))
    with
    rule e1$env = e0$env
    rule e2$env = e0$env
    rule e0$value = e1$value + e2$value
    end

In this declaration, the identifier e0 is bound to the exp nonterminal on the left-hand-side of the Sum production, and the identifiers e1 and e2 are bound to the right-hand-side children. The notation nonterm$attr is used to refer to the attribute attr of the nonterminal referred to by nonterm.

As with the attribute declarations, the semantic rules of a production can be split across several attribution declarations, and factoring is supported to reduce the size of the specification. We use the "or-pattern" notation to support this. For example, the following declaration defines the rules for passing the environment to the children of a binary operator:

attribute e0 : exp
    = (Sum(e1, e2) | Diff(e1, e2))
    with
    rule e1$env = e0$env
    rule e2$env = e0$env
    end

The set of symbol identifiers bound on the right-hand-side must be the same for each production in a factored attribution rule. Furthermore, only those local attributes that are defined for all of the productions may be referenced.

Local attributes and syntactic references are referred to by name in semantic rules. This is illustrated in the following example:

---

2Or-patterns are an extension of SML supported in SML/NJ (version 0.94 and later).


5 Attribute evaluation in ML

A number of issues must be addressed in the implementation of the tree editor and evaluator:

- It should be possible to convert between attributed and non-attributed versions of terms. Attribution of a non-attributed term is done by the evaluator; the converse should also be possible.

- A mechanism for associating attributes with tree nodes must be provided. It is especially useful for this mechanism to support sparse attribution (e.g., because of copy rules). In addition, operations for accessing and setting the values of attributes must be provided.

- Navigation and subtree replacement operations for attributed terms must be supported.

The Synthesizer Generator uses a heavy-weight tree-node representation that relies heavily on mutable fields (attribute instances, parent and child pointers, and other status fields). This representation supports efficient navigation, tree editing and attribute evaluation but does not support sparse attribution, sharing of trees and easy mapping between values computed by user code and abstract syntax trees. Furthermore, the heavy reliance on mutable fields does not map well to ML, since it requires ref cells, which add extra space overhead.

Our approach is light-weight: we use the datatype equivalent of the prototype declarations as our tree representation. We use paths in the tree to label nodes and support navigation, and auxiliary hash tables to hold the attribute values. Subtree replacement operations can be supported by the use of efficient incremental attribute evaluators. These features are described in more detail below.

6 The AML compiler

We have implemented a prototype of the AML compiler in SML. Our compiler is designed in a modular fashion to allow easy experimentation with different classes of grammars and different
evaluation techniques. Figure 4 gives a schematic view of the compiler. The shaded components are specific to particular choices of evaluation strategy and run-time representation; the other components are reusable. Table 1 gives a break-down of the number of source lines (including comments) in each major component for our current implementation. The first column refers to code in modules that are specific to the class of grammar (OAGs in this case); the second column refers to code in modules that are specific to the run-time representations; and the last column refers to code that is common across different versions. Note that almost two thirds of the code is general purpose; thus, we predict that the effort to retarget the system to a different class of grammars, or run-time system should be no more than a couple of weeks.

6.1 The front-end

The front-end of the AML compiler is responsible for translating an AML specification into an abstract grammar description. The abstract grammar description is a compiled representation of
the grammar description. It allows subsequent phases to efficiently extract information about the
grammar. The front-end uses two passes, which are discussed in the remainder of this section.

The first pass parses the specification producing an abstract syntax tree, which contains em-
bedded SML abstract syntax for the SML code in the specification. We do not analyze the fixity
or precedence of SML identifiers, so the SML abstract syntax contains fairly detailed syntactic
information, such as parenthesization information, to allow the back-end to emit the equivalent
code.

The second pass typechecks the abstract syntax tree, producing the abstract grammar de-
scription. For most parts of the specification, the analysis is straightforward. It records the declared
nonterminals, terminals, production identifiers, and attributes. The tricky part is the analysis of
the semantic rules. An identifier in the SML code for a rule might be a syntactic reference, local
attribute or an SML constructor or variable identifier. This requires keeping track of the binding and
scoping of SML identifiers while analyzing a semantic rule. During the analysis, the SML abstract
syntax tree is rewritten, by attribute references and syntactic references being replaced with new
SML variables. The resulting expression is then wrapped up as a function expression. For example,
the attribution rule (taken from the Calculator example in Appendix B):

```plaintext
  attribution e : exp
  = (Use id)
    with
      rule (error, e0$value) = (case (lookUp (id, e0$env))
        of (SOME v) => (NONE, v)
        | NONE => (SOME "<* undeclared identifier *>", 0)
        (* end case *))
    end
```

is translated into the following SML expression:

```plaintext
  fn (s_id_1, a_e0 Env_21) => let
    val (l_error_1, a_e0_value_20) = (    
      case (lookUp (s_id_1, a_e0_env_21))
      of (SOME v) => (NONE, v)
      | NONE => (SOME "<* undeclared identifier *>", 0)
      (* end case *))
    in
      (l_error_1, a_e0_value_20)
    end
```

The semantic rule defines two results (error, and e0$value), and has two free attribute refer-
ences (id, and e0$env). Thus, the resulting function takes a pair of arguments, and returns a pair
of results. We first bind the results and then build the pair, since the left-hand-side pattern might be
more complicated than a simple tuple.

The other complication in analyzing the semantic rules is the factoring allowed in the specifi-
cation. This means that a given semantic rule may be used in different productions, with different
argument types. We represent this sharing in the grammar description by splitting the representation
of a semantic action into two pieces: the semantic function (which is independent of the production), and the semantic rule associated with each production. In the resulting SML code, we bind the function expression that implements the rule to an identifier in the beginning of the evaluator module, and then apply that function in the evaluator steps that use the rule.

The analysis of the semantic actions also detects and identifies "copy rules," which allows subsequent phases to employ copy-rule optimizations [Hoo86].

6.2 The grammar analyzer

The grammar analyzer takes the abstract grammar produced by the front-end, and generates an evaluation strategy for the grammar. The form that the evaluation strategy takes depends on the class of AGs being supported, and the run-time evaluation model. For example, a topological-order batch evaluator requires no additional analysis, while an incremental OAG evaluator must determine a static evaluation plan, as well as change-propagation information.

Our current implementation supports batch evaluation of OAGs. The grammar analyzer applies Kasten's five-step analysis algorithm [Kas80]. It produces a static set of plans, one per production, for evaluating the grammar. See [Kas80] or Chapter 12 of [RT88] for more information about OAG evaluation.

6.3 The back-end

The back-end is responsible for generating the evaluator code and various support modules. In our current implementation, this output consists of four SML modules:

- The user module, which consists of the SML equivalents of the prodtype and termtype declarations, as well as any SML declarations in the AML specification.

- The tree module, which supports a tree machine interface to the abstract syntax trees defined in the specification.

- The attributes module, which supports the storage of attribute values.

- The evaluator module, which implements the tree-walk evaluator.

These are discussed in more detail in Section 7.

The back-end consists of a general-purpose pretty-printer for SML code, a translator for generating the support modules from the abstract grammar description; and a translator for generating the evaluator from the evaluation strategy and grammar description. For the most part, this code is boiler-plate. The one exception is the evaluator generator, which attempts to optimize the evaluation code by caching values. For example, fetching an attribute value might involve computing a

---

3The actual implementation is a mix of Kasten's algorithm and the one presented by Reps and Teitelbaum [RT88].
hash-key and then doing a hash-table look-up; if the attribute is used more than once, then caching
its value can avoid the hash-table overhead. For batch evaluation, this optimization is straight-
forward, but for incremental evaluators, the analysis and resulting program structure can be quite
complicated.

7 A prototype attribution system

There are many possible design choices that will satisfy the requirements discussed in the previous
section. In this section, we describe the straightforward scheme for OAG evaluation that we have
implemented as a first prototype.

Compiling an AML specification results in four modules: the user-types module, the tree
module, the attributes module, and the evaluator module. These modules are discussed in the
sequel.

7.1 User types

The various SML definitions given in the specification, plus the definitions generated from the
prodtype and termype declarations are collected together in the user-types module. The user
definitions are analyzed for syntactic correctness by the AML compiler, but are not typechecked.

7.2 The tree machine

Operations such as navigation and subtree replacement require a uniform interface to the abstract
syntax represented by the prodtypes and termypes of the grammar. The back-end generates a
module, called the tree module, that provides a typesafe collection of basic operations on trees.
This structure matches the abstract signature given in Figure 5. The type tree is a union of the
nonterminal and terminal types (i.e., a datatype with one constructor per nonterminal and terminal
type). The types symb_kind and prod_kind are enumerations of the symbols (terminals and
nonterminals) and productions, respectively. The operations are defined as follows:

symbKind \( t \) returns the symbol kind of the root of the tree.

prodKind \( t \) returns the production kind of the root of the tree.

sameSymb \((t1, t2)\) returns true, if the two trees have the same symbol at their roots.

sameProd \((t1, t2)\) returns true, if the two trees have the same production at their roots.

isTerminal \( sk \) returns true, if the symbol kind \( sk \) is a terminal symbol.

isRoot \( t \) returns true, if the root of the tree \( t \) is the root nonterminal.

isLeaf \( t \) returns true, if the root of the tree \( t \) is a leaf.
signature TREE =
sig
  type tree
  eqtype symb_kind
  eqtype prod_kind

  val symbKind : tree -> symb_kind
  val prodKind : tree -> prod_kind
  val sameSymb : (tree * tree) -> bool
  val sameProd : (tree * tree) -> bool
  val isTerminal : symb_kind -> bool

exception Child

  val isRoot : tree -> bool
  val isLeaf : tree -> bool
  val nChildren : tree -> int
  val children : tree -> tree list
  val nthChild : (tree * int) -> tree
  val replaceNth : (tree * int * tree) -> tree
  val nodeName : tree -> string
end

Figure 5: The signature of the tree operations

nChildren \( t \) returns the number of children of the tree \( t \).

children \( t \) returns a list of the children of the tree \( t \).

nthChild \((t, n)\) returns the \( n \)th child of \( t \); it raises the exception Child if \( n \) is out of range.

replaceNth \((t, n, s)\) replaces the \( n \)th child of \( t \) with \( s \); it raises the exception Child if \( n \) is out of range.

nodeName \( t \) returns a string representation of the \( t \)'s root; this is mainly provided for debugging purposes.

7.3 Storing attribute values

The most common place to store attributes is in the abstract syntax tree nodes. This approach has the advantage that the attributes are immediately accessible, and is used by the Synthesizer Generator. The main disadvantage of this approach is that it makes the translation between attributed and unattributed values difficult. Unattributed values may have sharing, which must be broken before attribution [TC90], and attributed terms may have a different representation than their unattributed counterparts. We have chosen a different approach, which is to store the attributes in an auxiliary structure. We use paths in the tree to define unique names for the nodes (Hood describes a similar
approach in [Hoo85]). This has the advantage that our abstract syntax trees are represented as plain SML datatypes, which means that mapping between attributed and unattributed terms is trivial.

In our current implementation, we use a hash table as the auxiliary attribute storage mechanism. For hash keys, we use a function of the path from the root to the node. The hash keys are represented as a pair of the integer hash value and the path from the root to the node (represented as a list of integers). From the hash key of a node, we can compute the hash key of one of its children in constant time. Thus, as we walk the tree, we can incrementally maintain the key of the current tree node, which we can use to access the current node’s attributes. The values in the hash table are members of a tagged union, which is generated from the specification. For example, the grammar given in Appendix B produces the following datatype declarations:

```plaintext
datatype attr_types
   = ATTR_exp of {
       value : int attr,
       env   : (string * int) list attr,
       localAttrs : locals_of_exp ref
     }
   | ATTR_calc of { value : int attr }

and locals_of_exp
   = LOCAL_Quot of { error : string option attr }
   | LOCAL_Use of { error : string option attr }
   | LOCALS_exp_VOID
```

where the `attr_type` constructor is defined as:

```plaintext
type 'a attr = 'a option ref
```

Each nonterminal (exp and calc in this example) has a constructor of a record of its attributes. The `localAttrs` field is for those productions that have local attributes (Quote and Use in this example).

For each attribute, there are `get` and `put` functions generated, which take an attribute table and a hashed path as arguments. Since an attribute record consists of references to attribute values, functions to fetch the reference for a particular attribute of a nonterminal are also provided. This can be useful when reducing the number of table lookups; for example, in incremental evaluation, where the old and new attribute values need to be compared before storing the new value. Figure 6 gives the signature of these functions for the Calculator example in Appendix B. The types `attr_tbl` and `key` are the attribute table and hash keys, respectively.

The first time an attribute of a production is stored in the table, it is necessary to allocate an attribute-value record for the production. The AML compiler generates default records for every nonterminal and every production that has locals. These default records are passed to the generic lookup routine as part of the implementation of the `attrRef_nt_attr` functions.
val attrRef_calc_value : (attr_tbl * key) -> int attr
val attrRef_exp_env : (attr_tbl * key) -> (string * int) list attr
val attrRef_exp_value : (attr_tbl * key) -> int attr
val attrRef_Quot_error : (attr_tbl * key) -> string option attr
val attrRef_Use_error : (attr_tbl * key) -> string option attr

val get_calc_value : (attr_tbl * key) -> int
val get_exp_env : (attr_tbl * key) -> (string * int) list
val get_exp_value : (attr_tbl * key) -> int
val getLocal_Quot_error : (attr_tbl * key) -> string option
val getLocal_Use_error : (attr_tbl * key) -> string option

val put_calc_value : (attr_tbl * key * int) -> unit
val put_exp_env : (attr_tbl * key * (string * int) list) -> unit
val put_exp_value : (attr_tbl * key * int) -> unit
val putLocal_Quot_error : (attr_tbl * key * string option) -> unit
val putLocal_Use_error : (attr_tbl * key * string option) -> unit

Figure 6: The attribute functions for the Calculator example

7.4 The treewalk evaluator

The treewalk evaluator is implemented as a collection of mutually recursive visit functions. For each visit $i$ down to a non-terminal $X$, there is a function visit $X$ $i$. These functions take three arguments: the tree node being visited, the attribute table, and the hash key for the node being visited. A visit function consists of a pattern match on the productions of the nonterminal.

In general, the evaluation of an attribute involves first fetching the arguments to the semantic rule from the attribute table, then computing the function, and finally storing the result in the attribute table. Fetching and storing the attributes of a node requires the hash key for the node, which must be computed for the child nodes. Likewise, visiting a child requires computing its hash key. As an optimization, the evaluator generator keeps track of already computed values, such as hash keys, so that it can avoid computing the same thing twice. Figure 7 gives the visit function code for the Sum production in the Calculator example in Appendix B. Note that the visits are numbered from 0 in the generated code. The function r_0004 is the generated code for the semantic rule, and the function down computes a child's hash key from its parent's hash key.

7.5 Incremental evaluation

We also have an incremental evaluator for OAGs. This evaluator is based on the algorithm described in Chapter 12 of [RT88]. Unlike the batch evaluator described above, the incremental evaluator is not compiled. Rather, it is a generic interpreter that executes plans represented as actions, where an action is defined as:
fun visit_exp_0 (tbl, lhs_key, Sum(c1, c2)) =
  (* Sum: e₀ := e₁ e₂ *)
  (* EVAL e₂.env *)
  val lhs_env = get_exp_env(tbl, lhs_key)
  val c2_env = (lhs_env)
  val c2_key = down (lhs_key, 2)
  val _ = put_exp_env(tbl, c2_key, c2_env)
  (* VISIT₁ e₂ *)
  val _ = visit_exp_0 (tbl, c2_key, c2)
  (* EVAL e₁.env *)
  val (c1_env) = (lhs_env)
  val c1_key = down (lhs_key, 1)
  val _ = put_exp_env(tbl, c1_key, c1_env)
  (* VISIT₁ e₁ *)
  val _ = visit_exp_0 (tbl, c1_key, c1)
  (* EVAL e₀.env *)
  val c1_value = get_exp_value(tbl, c1_key)
  val c2_value = get_exp_value(tbl, c2_key)
  val (lhs_value) = r_0004 (c1_value, c2_value)
  val _ = put_exp_value(tbl, lhs_key, lhs_value)
  (* SUSP *)
  in () end

Figure 7: Visit function code for Sum production

datatype action
  = Eval of (attr_tbl * path * key) -> key list
  | Visit of (int * int)
  | Suspend of int
  | EndOfPlan

The AML back-end generates the plans in this form, which are then passed to the interpreter by
functor application.

As described in Section 2.3, we model editing as subtree replacement. After a consistent tree
has been edited (i.e., a subtree in it has been replaced with a new subtree), the evaluator is called
with a path to the root of the new subtree as an argument. The evaluator maintains a set of active
productions, called the REACTIVATE set. Initially, the root of the new subtree and its parent are the
only members of REACTIVATE. The Eval instructions in the plan contain an attribute evaluation
function. This function checks to see if the evaluated attribute has changed, and, if so, returns a
list of hash keys of productions that should be added to the REACTIVATE set. The Visit and
Suspend instructions check to see if the node to be visited is in the REACTIVATE set; if not, they
do not move the locus of control. Thus, the number of attributes re-evaluated is \(O(|AFFECTED|)\),
where AFFECTED is the set of attributes that change value.

Each plan is terminated with an EndOfPlan instruction (following the last Suspend instruc-
tion). When one of these is encountered, the evaluator stops.
8 Current status and future work

We have implemented a prototype of our system, consisting of a front-end, OAG plan generator, and back-end for the runtime model described in Section 7. We have tested this prototype on several simple grammars, including a typechecker for a subset of C, and the Hoare-logic proof checker described in [RA84]. We have also experimented with connecting the generated evaluator with an incremental pretty-printer being developed for eXene [GR91, GR93], resulting in a simple structured editor. While the resulting editor is far from being a useful tool, it is a good demonstration of the technology.

The work reported in this paper is a snapshot of an ongoing research effort. While we now have the basic infrastructure in place, there many issues that we intend to explore in the future:

- Design and implement support for modular specifications. We feel that for AML to be a practical tool, modular specifications must be supported. Attribute grammars as such do not provide any means for modularization. As pointed out in [Kas91], an effective strategy is to allow a module for an AG to contain the attribution of one semantic aspect only. Several approaches have been advocated to modularize AGs [DC88, FMY92, Far92]. While these approaches address the problems of partitioning a grammar specification, they do not provide the concept of a module. We hope that this can be done in a way that builds on the SML module system and interacts well with the SML/NJ separate compilation mechanism.

- Lists of items are a common structure in most grammars. Currently, each different kind of list requires a different proctype declaration, and attribution rules. We would like to add a general-purpose mechanism for defining the syntax of lists, and higher-level syntax for defining the attribution of lists. Such a mechanism might build on the polymorphic lists of SML.

- Lists are one example of polymorphic structure, but there may be others. It may be useful to have a general-purpose mechanism for parameterizing prototypes. One possibility is polymorphic proctype declarations, but it is not clear how attribution rules would work. Prototypes are interpreted types (i.e., they have associated semantics), thus a mechanism based on functors is probably the right way to support parameterization.

- Our front-end supports the syntax of higher-order attribute grammars [VSK89, TC90]. We plan to implement support for these in our OAG analyzer and back-end.

- Our compiler is designed for experimentation. We plan to use it to explore other evaluation strategies, including demand-driven evaluation [Hud91] and parallel attribute evaluation [Jou91, KG92].

- Our current technique for storing attributes is simple, but not very space efficient. Also, removing subtrees requires expensive hash table operations, since the attributes of the subtree
must be removed from the table. We have some ideas for other storage schemes for attributes that we would like to experiment with. We would also like to examine techniques for supporting sparse attribution; i.e., only caching a subset of the attributes in the incremental evaluator. The most obvious example of this is so-called copy rules (i.e., semantic functions that are the identity).

- We view language-based editors as the most important application area for the AML technology. We have already experimented with hooking an evaluator to an incremental pretty-printer based on eXene, and we plan to continue this work by exploring editing issues (text vs. structure), and automating the generation of pretty-printers from higher-level specifications.

- If we generalize the patterns allowed in attribution rules, we would have something very close to Farnum's attribute patterns [Far92]. We plan to explore this similarity.

- The implementation of real applications using AML; in particular, a programming environment for SML.

9 Acknowledgements

We would like to thank Tim Teitelbaum, David Gries, Sanjiva Prasad, Chet Murthy and Aswin van den Berg for stimulating discussions pertaining to this project. Lars Søraas implemented much of the AML compiler back-end.
References


A  The collected syntax of AML

The syntax of AML is an extension of the SML syntax (see Chapter 2 and Appendix B of the
Definition [MTH90]). In presenting the grammar, we use an extended-BNF style with the following
conventions:

- grammar nonterminals are given in italic font; e.g., `Grammar`.
- terminal symbols are written in typewriter font; e.g., `val`.
- optional items are enclosed in angle brackets; e.g., `{;}`.
- a sequence of zero or more items is denoted by braces; e.g., `{Dec}`.

A.1  Reserved words

In addition to the those of SML, AML has the following reserved words, which may not be used as
identifiers:

```
attribute attribution grammar prodtype root rule
termttype $
```

A.2  Identifiers

AML requires a number of new identifier classes; all of which are restricted to be `alphanumeric`
identifiers. Some of these classes overlap with the SML identifier classes. The identifier classes are
as follows (overlaps with SML identifier classes are given in parentheses):

- **GramId**: Grammar name
- **NontermId**: Nonterminals  (TyCon)
- **Termld**: Terminals       (TyCon)
- **ProdId**: Production constructors (Con)
- **AttrId**: Attribute names
- **Localld**: Local attribute names (Var)
- **SymblId**: Symbols (nonterminal and terminal) (Var)

In addition, there is the class `AttrInstance` of `attribute instances`, which are written

```
SymblId $ AttrId
```

These can appear in patterns and expressions, as described in the next section.
A.3 SML nonterminals

As mentioned above, the AML grammar is an extension of the SML grammar. We use a small collection of nonterminals to refer to parts of the SML grammar. These are summarized in the following table:

<table>
<thead>
<tr>
<th>AML nonterminal</th>
<th>SML nonterminal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLDec</td>
<td>dec</td>
<td>SML declarations</td>
</tr>
<tr>
<td>SMLType</td>
<td>ty</td>
<td>Monomorphic SML types</td>
</tr>
<tr>
<td>Pat</td>
<td>pat</td>
<td>Extended SML patterns</td>
</tr>
<tr>
<td>Exp</td>
<td>exp</td>
<td>Extended SML expressions</td>
</tr>
</tbody>
</table>

The extended patterns and expressions allow attribute instances (AttrInstance), symbols (SymbId), and local attributes (LocalId), wherever variables (var) are allowed.

A.4 Grammar

```
Grammar ::= grammar GramId = struct Dec { (;) Dec} end { ;}

Dec ::= root NontermId
| prodttype ProdTyDec { and ProdTyDec }
| termttype TermTyDec { and TermTyDec }
| attribute AttributeDec { and AttributeDec }
| attribution SemanticRules { and SemanticRules }
| SMLDec

ProdTyDec ::= NontermId (WithAttrs) = ProdDec { | ProdDec}

ProdDec ::= ProdId { of ProdRHS } (WithLocals)

ProdRHS ::= ( NontermId { * NontermId} )
| NontermId { * NontermId}

TermTyDec ::= Terml = SMLType

AttributeDec ::= NontermId { , NontermId} WithAttrs
| ProdId { , ProdId} WithLocals

WithAttrs ::= with Attribute { ( ; ) Attribute } end

Attribute ::= synth AttrId : SMLType { and AttrId : SMLType }
| inher AttrId : SMLType { and AttrId : SMLType }

WithLocals ::= with LocalAttribute { ( ; ) LocalAttribute } end

LocalAttribute ::= local LocalId : SMLType { and LocalId : SMLType }
```
SemanticRules ::= SymbId : NontermId = Rules { | Rules}

Rules ::= ProdPat with Rule { (;) Rule} end

Rule ::= rule Pat = Exp { and Pat = Exp}

ProdPat ::= ( ProdPat { | ProdPat } )
| ProdId AtomicProdPat
| AtomicProdPat

AtomicProdPat ::= _
| SymbId
| ProdId
| ( ProdPat { , ProdPat } )
B An example grammar

(* Calc.aml
  *
  * A simple calculator example.
  *)

grammar Calc =
  struct

    termtype int = int
    termtype ident = string

  prodtype exp
  = NullExp
    | Sum of (exp * exp)
    | Diff of (exp * exp)
    | Prod of (exp * exp)
    | Quot of (exp * exp)
    | Const of int
    | Let of (ident * exp * exp)
    | Use of ident

  prodtype calc
  = NullCalc
    | Top of exp

root calc

attribute calc, exp with
  synth value : int
end

attribute exp with
  inner env : (string * int) list
end

and Quot, Use with
  local error : string option
end

(* look up an identifier in an environment *)
fun lookUp (id, []) = NONE
  | lookUp (id, (x, v)::r) =
    if (id = x) then (SOME v) else lookUp(id, r)
attrribution c0 : calc
  = NullCalc
    with rule c0$value = 0 end
  | Top(e1)
    with
      rule c0$value = e1$value
      rule e1$env = []
    end

attrribution e0 : exp
  = NullExp
    with rule e0$value = 0 end
  | (Sum(e1, e2))
    with rule e0$value = e1$value + e2$value end
  | (Diff(e1, e2))
    with rule e0$value = e1$value - e2$value end
  | (Prod(e1, e2))
    with rule e0$value = e1$value * e2$value end
  | (Quot(e1, e2))
    with
      rule (error, e0$value) = if (e2$value = 0)
        then (SOME "<* division by zero *>", e1$value)
        else (NONE, e1$value div e2$value)
      end
    | (Const n)
      with rule e0$value = n end
    | (Let(id, e1, e2))
      with rule e0$value = e2$value end
    | (Use id)
      with
        rule (error, e0$value) = (case (lookUp (id, e0$env))
          of (SOME v) => (NONE, v)
            | NONE => (SOME "<* undeclared identifier *>", 0)
              (* end case *))
        end
  end

attrribution e0 : exp
  = (Sum(e1, e2) | Diff(e1, e2) | Prod(e1, e2) | Quot(e1, e2))
    with
      rule e1$env = e0$env
      rule e2$env = e0$env
    end
  | (Let(id, e1, e2))
    with
      rule e1$env = e0$env
      rule e2$env = (id, e1$value) :: e0$env
    end

end (* Calc *)