Nordic Journal of Computing

AML: ATTRIBUTE GRAMMARS IN ML

Sofoklis G. Efremidis*
Department of Computer Science
Cornell University
Ithaca, New York, USA
sofoklis@cs.cornell.edu

Khalid A. Mughal†
Department of Informatics
University of Bergen
N-5020 Bergen, Norway
khalid@ii.uib.no

John H. Reppy
AT&T Labs Research
600 Mountain Ave.
Murray Hill, New Jersey, USA
jhr@research.att.com

Lars Søraas‡
Department of Informatics
University of Bergen
N-5020 Bergen, Norway
larss@ii.uib.no

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.

CR Classification: D.1.1, D.2.6, D.3.4

Key words: attribute grammars, attribute evaluation, functional programming, program generator, programming environments

1. Introduction

Attribute grammars provide a formalism for assigning meaning to parse trees of a context-free language [26]. Because of their syntax-directed form and declarative style, they provide a useful notation for specifying compilers [23] and language-based editors [37]. 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 [28].

*This work was supported, in part, DARPA-ONR Grant N00014–91–J–4123. Current address: Intracom S.A., 190 02 Peania Attica, Greece. E-mail: sefr@intranet.gr

†Support for this work was provided, in part, by the Norwegian Research Council.

‡Current address: NERA Telecommunications ASA, Kokstadveien 12, N-5061 Kokstad, Norway. E-mail: las@nera.no

Received July 1996. Revised December 1996. Accepted January 1997.

`Let:`	e_0	::=	$ID\ e_1\ e_2$	$e_1.env$	=	$e_0.env$
				$e_2.env$	=	$insert(e_0.env,\ (ID,\ e_1.value))$
				$e_0.value$	=	$e_2.value$
`Const:`	e_0	::=	NUM	$e_0.value$	=	NUM
`Use:`	e_0	::=	ID	$e_0.value$	=	$lookup(e_0.env,\ ID)$
`Sum:`	e_0	::=	$e_1\ e_2$	$e_1.env$	=	$e_0.env$
				$e_2.env$	=	$e_0.env$
				$e_0.value$	=	$e_1.value + e_2.value$

Fig. 1: A simple attribute grammar

AML is a spiritual heir to the Synthesizer Generator project at Cornell University [37], which focused on efficient incremental evaluation techniques, and the Pegasus project at AT&T Bell Laboratories [35], 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 [4]. This paper is an updated version of the technical report [5].

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 $p : X_0 ::= X_1 \ldots X_{n_p}$, there are evaluation rules that define the synthesized attributes of X_0 and the inherited attributes of $X_1, \ldots, X_{n_p}$. 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.

Fig. 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`,

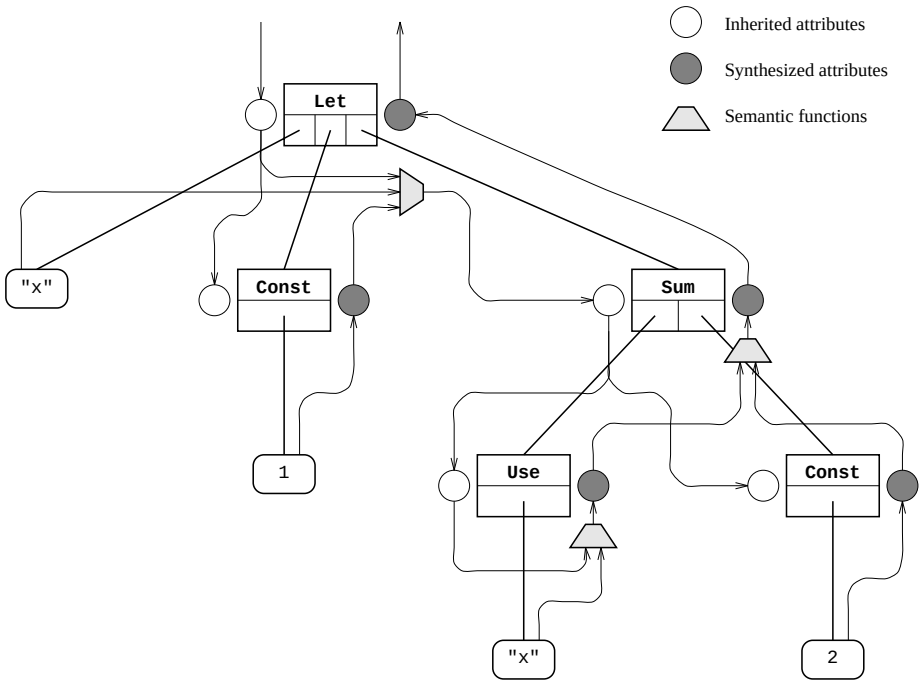

Fig. 2: The attributed tree for the expression "`let x = 1 in x + 2 end`"

`Use`, `Sum`). The symbols *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 the Appendix.

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

```
let x = 1 in x + 2 end
```

using the grammar from Fig. 1. The graph formed from the attribute instances

and the dependencies between them is called the *attribute dependency graph*. Any attribute evaluation strategy employed must respect the dependencies in this graph which embodies the constraints defined by the attribution rules. Seen from the point of view of a constraint solving problem, application of the attribution rules is driven by the dependency graph either explicitly or implicitly depending on the evaluation strategy. Most evaluation strategies require that the dependency graph be acyclic, although there are fixed-point techniques for handling grammars with cyclic dependencies [9, 44, 19].

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 [1]. 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 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) [22]. 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 non-terminal 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$_i$**:** visit a child for the ith time (i.e., transfer control to the child's plan), and

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

Productions

`Let:` $e_0 ::= ID\ e_1\ e_2$	`Const:` $e_0 ::= NUM$
EVAL $e_1.env$	EVAL $e_0.value$
VISIT e_1	SUSP
EVAL $e_2.env$	
VISIT e_2	
EVAL $e_0.value$	
SUSP	

`Use:` $e_0 ::= ID$	`Sum:` $e_0 ::= e_1\ e_2$
EVAL $e_0.value$	EVAL $e_2.env$
SUSP	VISIT e_2
	EVAL $e_1.env$
	VISIT e_1
	EVAL $e_0.value$
	SUSP

Fig. 3: Evaluation plans for the simple attribute grammar.

The VISIT and SUSP instructions are essentially "call" and "return." Kastens gives a number of run-time evaluation techniques for these plans in [22].

The grammar given in Fig. 1 is an OAG. The attributes of e are ordered by $env \prec v$. The evaluation plans for this grammar are given in Fig. 3 (we have omitted the subscripts on the VISIT instructions). Note that the children of `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 [38]. 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 [38, 45, 17, 18]. 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 [11, 37], interactive theorem provers [14] and incremental code generators [29], as well as optimizing compilers [2].

3. The design of AML

We have developed a system, called AML, for supporting attribute evaluation in SML. 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. Some systems, such as the Synthesizer Generator, have one language for specifying the attribute evaluation rules and a different one for the implementation. We believe that this is a mistake, and instead use a subset of SML, which is the target language of our generator, for writing attribution rules. In addition to attribute evaluation, we also use SML for specifying auxiliary types and functions. By unifying the source and target languages, it is much easier to connect attribute evaluation with other parts of an application. Subsequent sections will elaborate on these aspects. In addition, we have followed a number of design guidelines:

- 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 is 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 is designed to support experimentation in all of these areas.

Some familiarity with SML is helpful in the subsequent sections on various aspects of the AML system. Paulson's book [32] is a good source for learning SML.

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.
- `prodtype` 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.

4.1 Syntax

The syntax of the grammar is defined by the `termtype`, `prodtype`, 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, and to specify their representation. For example:

```
termtype int = int
termtype ident = string
```

The `termtype` declaration is essentially an SML type definition, with the restriction that it does not allow type variables.

The nonterminals and productions of a grammar are defined using the `prodtype` declaration. For example:

```
prodtype 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:

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

and stmt_list
  = StmtListNil
  | StmtListCons of (stmt * stmt_list)
```

The `prodtype` declaration is essentially a restricted SML `datatype` declaration. The restrictions are that the constructor argument types can only involve termtypes, prodtypes, and tupling.

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, which are the SML representation of the trees. This is further discussed in Section 7.

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 defines an evaluation function for each root type.

4.2 Attributes

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

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

Attributes can have any monomorphic SML type.[1] To allow for more concise specifications, factored declarations are allowed, which define a collection of attributes for several nonterminals:

```
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:

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

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

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

[1] This restriction is necessary to avoid problems with SML's value polymorphism restriction.

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

is equivalent to the declarations:

```
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 an SML pattern, and evaluation rules for the attributes. For example:

```
attribution 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:

```
attribution 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:

```
attribution e0 : 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
```

which defines the value of the local attribute `error` and uses the syntactic reference `id`. This is also an example of *multiple attribution*; i.e., the definition of multiple attributes in a single semantic rule. In general, the left-hand-side of a semantic rule can be any SML pattern, as long as the bound variables are attributes, local attributes, or syntactic references.

5. Attribute evaluation in ML

A number of issues must be addressed in the design of the tree machine and evaluator:

- 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.
- It should be possible to convert between attributed and unattributed versions of terms. The evaluator is used to attribute an unattributed term, and attributes are discarded when going in the other direction. Mapping between attributed and unattributed representations is also necessary for *higher-order attribute grammars* [43].
- 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 and run-time overhead.

We have chosen a light-weight approach: we use the `datatype` equivalent of the `prodtype` declarations as our tree representation, and store attributes in an auxiliary data structure (e.g., hash table). Positions in the tree are labeled by a path from a root; the path is used to access the node's attributes, and to navigate the tree. We describe one implementation of this design below.

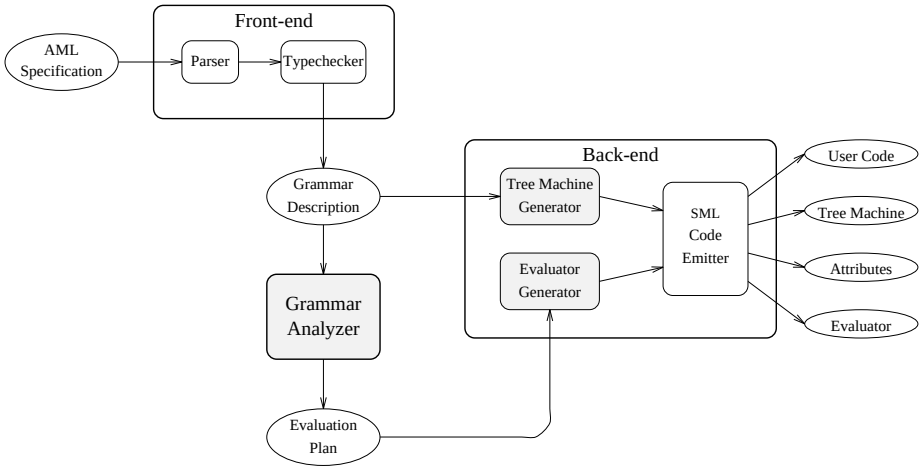

Fig. 4: Organization of the AML compiler

	Lines of code		
Component	**Grammar specific**	**Run-time specific**	**General purpose**
Front-end			3,764
OAG analyzer	718		
Back-end	544	1,166	386
Misc.			73
Total	1,262	1,166	4,223

TABLE I: Compiler size break-down

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. Fig. 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 I 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. Section 8 provides examples of experimentation with the system which bear out this assumption.

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 embedded 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 description. 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 9):

```
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:

```
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$value`), and has two free attribute references (`id`, and `e0$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 specification. 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 [16].

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.

The grammar analyzer applies Kastens' five-step analysis algorithm [22].[2] It produces a static set of *fixed plans*, one per production, for evaluating the grammar. Section 8 discusses various attribute evaluation strategies, based on fixed plans, which have been implemented in the AML system. See [22] or Chapter 12 of [37] 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.

[2] The actual implementation is a mix of Kasten's algorithm and the one presented by Reps and Teitelbaum [37].

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 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. An implementation of attribute evaluation

There are many possible design choices that satisfy the requirements discussed above. In this section, we describe the straightforward scheme for OAG evaluation that we have implemented as a first prototype. Section 8 discusses various other OAG evaluation strategies, which have also been implemented in the AML system. Compiling an AML specification results in four modules: the *user-types* module, the *tree* module, the *attributes* module, and the *evaluator* module. These 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 `termtype` 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. While reporting type errors at this stage would be useful, it would require reimplementing an SML front-end in the AML compiler — not a trivial task.

As mentioned previously the `prodtype` and `termtype` declarations are translated directly into *datatype* and *type* declarations. These and other definitions in the user-types module can be used by other client modules. For the Calculator example in the Appendix, these are available through the `Calc_UserTypes` structure.

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 termtypes 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 Fig. 5. The type `tree` is a union of the nonterminal and terminal types

```
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 defaultProd : symb_kind -> tree
    val nodeName    : tree -> string
  end
```

Fig. 5: The signature of the tree operations

(i.e., a datatype with one constructor per nonterminal and terminal type). For the Calculator example, type generated `tree` type is:

```
datatype tree
  = T_calc of Calc_UserTypes.calc
  | T_exp of Calc_UserTypes.exp
  | T_ident of Calc_UserTypes.ident
  | T_letter of Calc_UserTypes.letter
  | T_digit of Calc_UserTypes.digit
```

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.

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.

defaultProd *sk* returns the *completing tree t* corresponding to the *completing production* for the symbol kind which is a nonterminal; this is used for structural modification and re-attribution of the tree.

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

We use this *interpretive* approach to tree manipulation and navigation, because it allows a generic implementation of the supporting run-time system (e.g., higher-level editing operations, etc.). We discuss an example of this in the next section. Another approach would be to specialize the run-time system to the specific grammar, but we expect that this would result in code bloat with little gain in efficiency.

7.3 Tree navigation and editing

Support for tree navigation and editing is implemented by a functor over the abstract `TREE` signature. The `PATH` signature is the interface to this functor, and is given in Fig. 6. As we mentioned above, we use paths to label nodes in the tree. Paths are represented by the following datatype:

```
datatype path
  = Base of tree (* the tree's root *)
  | Path of int * tree * path
```

Thus `Path(i,t,p)` represents a path where `t` is the current subtree specified by the path, and `t` is its parent's `i`'th child and `p` is the path to `t`'s parent.

There are various navigation operations defined that map paths to new paths (e.g., `up` moves to the node's parent), as well as a subtree replacement operation (`replace`).

7.4 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

```
signature PATH =
  sig
    structure T : TREE
    type path
    exception Down
    exception Up

    val path : T.tree -> path
    val rootOfPath : path -> T.tree
    val treeOf : path -> T.tree

    val isBase : path -> bool
    val isLeaf : path -> bool
    val childOrd : path -> int
    val up : path -> path
    val down : path * int -> path
    val eqPath : path -> path -> bool
    val eqPath : path -> path -> bool
    val left : path -> path
    val leftMostLeaf : path -> path
    val right : path -> path
    val rightMostLeaf : path -> path
    val replace : path * T.tree -> path
  end
```

Fig. 6: The signature of the path operations

that it makes the translation between attributed and unattributed values difficult. Unattributed values may have sharing, which must be broken before attribution [41], 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 [15]). This has the advantage that our abstract syntax trees are represented as plain SML datatypes, and 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 the Appendix produces the datatype declarations shown in Fig. 7. where the `attr` type constructor is defined

```
datatype attr_types
  = ATTR_exp of {
       value        : int attr,
       env          : (string * int) list attr,
       local__attrs : 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
```

Fig. 7: The generated datatypes of the attributes in the Calculator example

as:

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

Each nonterminal (`exp` and `calc` in this example) has a constructor of a record of its attributes. The `local__attrs` 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 in 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. Fig. 8 gives the signature of these functions for the Calculator example in Appendix 9. 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.

7.5 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 corresponding 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.

```
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
```

Fig. 8: The attribute functions for the Calculator example

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. Fig. 9 gives the visit function code for the `Sum` production in the Calculator example in Appendix 9. 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.

7.6 Incremental evaluation

We also have an incremental evaluator for OAGs. This evaluator is based on the algorithm described in Chapter 12 of [37]. Unlike the batch evaluator described above, the incremental evaluator is not compiled. Rather, it is a generic interpreter

```
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.v *)
      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
  | ...
```

Fig. 9: Visit function code for `Sum` production

that executes plans represented as actions, where an action is defined as:

```
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` instruction). When one of these is encountered, the evaluator stops. Again see [22] or Chapter 12 of [37] for more information about incremental OAG evaluation.

7.7 Manipulation of attributed trees

An attributed term, represented as a pair of `attr_tbl` and `path`, can be fed to the attribute evaluator, which utilizes the navigation operations on `path` and attribute storage operations on `attr_tbl`. Client code can build and maintain attributed trees at the low-level from the datatypes corresponding to the prodtypes along the lines described in the previous sections.

At a more intermediate level, a *clipboard*-paradigm providing facilities for structured editing of *consistent* terms can be utilized. Each term comprises of a syntax tree, together with the path of the selected subtree and the associated attribute values. An *abstract* clipboard (called abstract since it only allows manipulation of the abstract syntax) matches the following signature:

```
signature ABS_CLIPBOARD =
  sig
  structure AT : ATTR_TERMS
  type clipboard
  val cut : AT.attr_term -> AT.attr_term * clipboard
  val copy : AT.attr_term  -> clipboard
  val paste : AT.attr_term * clipboard -> AT.attr_term
  end
```

`ATTR_TERMS` is a signature (not shown here) which provides the appropriate bundling for an attributed term and the path of the selected subtree under consideration. The values of type `clipboard` are also attributed terms, which obviously do not have any selected node and therefore do not have any path information associated with them. The standard operations on the clipboard are defined as follows:

cut *t1* $\rightarrow$ **(***t2***,** *cb***):** Cut indicated subterm from the term *t1* to a clipboard *cb*. *t2* is the consistent structured modified version of *t1*.

copy *t* $\rightarrow$ *cb***:** Copy indicated subterm from the term *t* to a clipboard *cb*.

paste (*t1***,** *cb***)** $\rightarrow$ *t2***:** Paste contents of *cb* over indicated subterm of *t1*. *t2* is the consistent structured modified version of *t1*.

Note that the operations `cut` and `paste` require re-attribution of the modified term. During a `cut` operation the selected subtree is replaced by the term constructed from the completing production of the nonterminal of the root of the subtree; re-attribution is thereby performed on a consistent term.

8. Experiences with AML

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 [36]. We have also experimented with connecting the generated evaluator with an incremental pretty-printer being developed for eXene [12, 13], 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. This prototype has also served as the basis for a number of explorations into attribute grammar technology.

- Extensions to the AML specification language.

  Our front-end supports the syntax of higher-order attribute grammars (HAGs) [43, 41]. HAGs blur the distinction between syntax trees and attributes. Attribution rules can compute trees, which then can be grafted into the tree structure and attributed. Likewise, attributed trees can be referenced by the attribution rules. Support for HAGs in our OAG analyzer and back-end, together with attribute evaluation for HAGs based on fixed plans, has been implemented [31]. Our design choice of separating trees from attributes makes HAGs much easier to support, because the representation of syntax trees does not have any extra fields to support attribution. A tree that is computed as an attribute can be grafted into the syntax tree without any special transformation. Likewise, attributed trees can be used as values in attribution rules by just discarding the attributes.

- Experimentation with evaluation strategies.

  Apart from the treewalk evaluators based on execution of fixed plans (both batch and incremental) and visit functions (batch only), other evaluation strategies have also been implemented in our system. The major motivation behind experimentation with other evaluation strategies has been the fact that the visit functions implement a *functional* batch evaluator, which does not easily amend to incremental behavior. A new approach to implementing an incremental functional evaluator, based on *neighbor functions*, is described in [30, 39]. Neighbor functions are comprised of both *visit* and *suspend* functions, and allow re-evaluation to start at the node of subtree replacement rather than at the root of the syntax tree.

  Pennings in his Ph.D. thesis, [33], discusses how incremental evaluators can be generated by memoising visit functions. The only attributes of interest in his algorithms are those of the root of the syntax tree. The algorithms, however, do not lend themselves to interactive systems where all the attributes need to be stored. Incremental attribute evaluation in AML by memoising neighbor functions is explored in [42].

  Zaring in his Ph.D. thesis, [46], describes plan-based algorithms for parallel attribute evaluation in attribute grammar-based systems. Experimentation with implementation of asynchronous algorithms, based on extensions of

neighbor functions, using CML [34] in the AML framework, are discussed in [27].

- Application development using AML system.

  We view language-based editors as the most important application area for the AML technology. As mentioned above, we have already experimented with hooking an evaluator to an incremental pretty-printer based on eX-ene. The pretty printing of abstract syntax in *prodtypes* declarations can be specified using *pptype* declarations. AML compiler translates this external representation into pretty-printing functions used by the editor [6].

  Another promising approach investigated is coupling an evaluator to Emacs as it is a popular and readily available extensible editor [40]. This experiment also demonstrates rudimentary *text* editing facilities (in addition to structured editing) [7]. This has led to the definition of a third specification, that of *concrete syntax*, which will be used to translate textual input so that it can be incorporated into the internal representation of the syntax tree.

These experiments have demonstrated that the modular design of our compiler promotes experimentation with significant code reuse.

9. Future work

The work reported in this paper is a snapshot of an ongoing research effort. While we now have the basic infrastructure in place, and have investigated several aspects, there are many issues that we are exploring and intend to explore in the future. These fall under the categories mentioned in the previous section.

- Further extensions to the AML specification language.

  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 [24], 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 [3, 10, 8, 21]. While many approaches address the problems of partitioning a grammar specification, it can be argued how well they provide the concept of a *module*. We intend to build on previous work in this area, and 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 `prodtype` 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 prodtypes. One possibility is polymorphic `prodtype` declarations, but it is

not clear how attribution rules would work. Prodtypes are interpreted types (i.e., they have associated semantics), thus a mechanism based on functors is probably the right way to support parameterization.

If we generalize the patterns allowed in attribution rules, we would have something very close to Farnum's *attribute patterns* [8]. We plan to explore this similarity.

- Further experimentation with evaluation strategies and storage schemes.

  We plan to use it to explore other evaluation strategies, including demand-driven evaluation [18] and parallel attribute evaluation [20, 25].

  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).

- Application development using AML system.

  We intend to provide facilities for other widget-based user-interfaces to hook into an attribution engine generated by the AML compiler. We also plan to explore editing issues (text vs. structure), and automating the generation of pretty-printers from higher-level specifications.

  One ambitious goal is the implementation of real applications using AML; in particular, a programming environment for SML.

Acknowledgements

We would like to thank Tim Teitelbaum, David Gries, Sanjiva Prasad, Chet Murthy and Aswin van den Berg for stimulating discussions in the early stages of this project.

Many individuals have contributed to the AML project. We would like to acknowledge the contribution of Thomas Yan relating to pretty-printing in AML. Following members of the AML group at the Department of Informatics, University of Bergen, have made significant contributions to this work: Gøran Ellingsen, Roy Oma, Johnny Tvedt and Tord Kolsrud. In addition we would to thank Norvald Espedal for providing the Emacs connection.

Appendix — An example grammar

```
(* Calc.aml : a simple calculator example. *)

grammar Calc =
  struct

  (* Terminals *)
    termtype int = int
    termtype ident = string

  (* Productions *)
    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 declarations *)
    attribute calc, exp with
        synth value : int
      end

    attribute exp with
        inher 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)

  (* Attribution rules *)
    attribution c0 : calc
```

```
      = NullCalc
        with rule c0$value = 0 end
      | Top(e1)
        with
          rule c0$value = e1$value
          rule e1$env = []
        end

    attribution 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

    attribution 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 *)
```

References

[1] ALBLAS, H. 1991. Attribute Evaluation Methods. In *Attribute Grammars, Applications and Systems*, Alblas, Henk and Melichar, Bořivoj, Editors. Springer-Verlag, 48–113.

[2] ALBLAS, H. 1991. Incremental Attribute Evaluation. In *Attribute Grammars, Applications and Systems*, Alblas, Henk and Melichar, Bořivoj, Editors. Springer-Verlag, 215–233.

[3] DUECK, G.D.P. AND CORMACK, G.V. 1988. Modular Attribute Grammars. Tech. Report CS-88-19, Faculty of Mathematics, University of Waterloo, Canada.

[4] EFREMIDIS, S.G., MUGHAL, K.A., AND REPPY, J.H. 1992. Attribute grammars in ML. In *Proceedings of the ACM SIGPLAN'92 Workshop on ML and its Applications*, 194–200.

[5] EFREMIDIS, S.G., MUGHAL, K.A., AND REPPY, J.H. 1993. AML: Attribute grammars in ML. Tech. Report Report No. 89, Department of Informatics, University of Bergen, Norway.

[6] ELLINGSEN, G.S. 1994. *Generating pretty printing facilities in AML system*. Cand. scient. thesis, Department of Computer Science, University of Bergen.

[7] ESPEDAL, N. 1996. *Realisering av brukergrensesnitt for AML i Gnu Emacs*. Siv.ing. thesis, Department of Electronics and Computer Science, Stavanger College.

[8] FARNUM, C. 1992. Pattern-based Tree Attribution. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, 211–222.

[9] FARROW, R. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, 85–98.

[10] FARROW, R., MARLOW, T.J., AND YELLIN, D.M. 1992. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, 223–234.

[11] FISCHER, C.N., JOHNSON, G.F., MAUNEY, J., PAL, A., AND STOCK, D.L. 1984. The POE Language-based Editor Project. In *ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 21–29.

[12] GANSNER, E.R. AND REPPY, J.H. 1991. eXene. In *Third International Workshop on Standard ML*. Carnegie Mellon University.

[13] GANSNER, E.R. AND REPPY, J.H. 1993. *A Multi-threaded Higher-order User Interface Toolkit*, Volume 1 of *Software Trends*. John Wiley & Sons, 61–80.

[14] GRIFFIN, T.G. 1987. An Environment for Formal Systems. Tech. Report 87-846, Department of Computer Science, Cornell University.

[15] HOOD, R. 1985. Efficient Abstractions for the Implementation of Structured Editors. In *Proceedings of the ACM SIGPLAN'85 Symposium on Language Issues in Programming Environments*, 171–178.

[16] HOOVER, R. 1986. Dynamically bypassing copy rule chains in attribute grammars. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, 14–25.

[17] HOOVER, R. 1987. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York 14853.

[18] HUDSON, S.E. 1991. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems 13*, 3, 315–341.

[19] JONES, L.G. 1990. Efficient Evaluation of Circular Attribute Grammars. *ACM TOPLAS 12*, 3 (July), 429–462.

[20] JOURDAN, M. 1991. A Survey of Parallel Evaluation Methods. In *Attribute Grammars, Applications and Systems*, Alblas, Henk and Melichar, Bořivoj, Editors. Springer-Verlag, 234–255.

[21] JOURDAN, M., PARIGOT, D., JULIÉ, C., DURIN, O., AND LE BELLEC, C. 1990. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Conf. on Programming Languages Design and Implementation*. White Plains, NY, 209–222.

[22] KASTENS, U. 1980. Ordered Attribute Grammars. *ACTA Informatica 13*, 3, 229–256.

[23] KASTENS, U., HUTT, B., AND ZIMMERMANN, E. 1982. *GAG: A Practical Compiler Generator*. Volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag.

[24] KASTENS, U. AND WAITE, W.M. 1994. Modularity and Reusability in Attribute Grammars. *Acta Informatica 31*, 601–627.

[25] KLAIBER, A. AND GOKHALE, M. 1992. Parallel Evaluation of Attribute Grammars. *Transactions on Parallel and Distributed Systems 3*, 2 (March), 206–220.
[26] KNUTH, D.E. 1968. Semantics of context-free languages. *Mathematical Systems Theory 2*, 127–145.
[27] KOLSRUD, T.A. 1996. *Inkrementell attributtevaluering i AML med parallelle nabofunksjoner*. Cand. scient. thesis, Department of Computer Science, University of Bergen.
[28] MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass.
[29] MUGHAL, K.A. 1988. *Generation of Runtime Facilities for Program Editors*. PhD thesis, University of Bergen, Norway.
[30] MUGHAL, K.A. AND SØRAAS, L. 1995. Attribute Evaluation using Neighbour Functions. Tech. Report Report No. 112, Department of Informatics, University of Bergen, Norway.
[31] OMA, R.A. 1996. *Høyere Ordens Attributtgrammatikker i AML*. Cand. scient. thesis, Department of Computer Science, University of Bergen.
[32] PAULSON, L.C. 1996. *ML for the Working Programmer*. 2nd Edition. Cambridge University Press, Cambridge, England.
[33] PENNINGS, M.C. 1994. *Generating incremental attribute evaluators*. Ph.D. thesis, Computer Science, Utrecht University.
[34] REPPY, J.H. 1992. *High-Order Concurrency*. Ph.D. thesis, Department of Computer Science, Cornell University.
[35] REPPY, J.H. AND GANSNER, E. R. 1986. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 218–227.
[36] REPS, T. AND ALPERN, B. 1984. Interactive Proof Checking. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, 36–45.
[37] REPS, T. AND TEITELBAUM, T. 1988. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer-Verlag, New York, NY.
[38] REPS, T. W. 1982. *Generating Language-Based Environments*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY.
[39] SØRAAS, L. 1994. *Generering av attribueringsystemer for AML spesifikasjoner*. Cand. scient. thesis, Department of Computer Science, University of Bergen.
[40] STALLMAN, R.M. 1993. *GNU EMACS Manual*. Free Software Foundation, Cambridge, Mass.
[41] TEITELBAUM, T. AND CHAPMAN, R. 1990. Higher-order Attribute Grammars and Editing Environments. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 197–208.
[42] TVEDT, J. 1996. *Inkrementell attributtevaluering i AML ved memoisering av nabofunksjoner*. Cand. scient. thesis, Department of Computer Science, University of Bergen.
[43] VOGT, H.H., SWIERSTRA, S.D., AND KUIPER, M.F. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, 131–145.
[44] WALZ, J.A. AND JOHNSON, G.F. 1988. Incremental Evaluation for a General Class of Circular Attribute Grammars. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 209–221.
[45] YEH, D. 1983. On Incremental Evaluation of Ordered Attributed Grammars. *BIT 23*, 308–320.
[46] ZARING, A.K. 1990. *Parallel Evaluation in Attribute Grammar-based Systems*. Ph.D. thesis, Department of Computer Science, Cornell University.