A Context-Based Incremental Evaluator for Hierarchical Attribute Grammars

Alan Carle
CITI/CRPC
Rice University
P. O. Box 1892
Houston, Texas 77251
carle@cs.rice.edu

Lori Pollock
Dept. of Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@udel.edu

Abstract

The recent emergence of *hierarchical* attribute grammar dialects including Higher Order Attribute Grammars, Attribute Coupled Grammars, the Synthesizer Generator's SSL, and Modular Attribute Grammars makes it finally possible to specify large complex transformation and analysis systems naturally and modularly via the attribute grammar formalism. Unfortunately, efficient incremental evaluators for these hierarchical specifications cannot be built by simply applying optimal incremental evaluators for non-hierarchical attribute grammars in a recursive manner. This paper presents a conceptually simple tree-walking incremental evaluation algorithm which we show can evaluate entire hierarchical attribute grammar specifications optimally under standard assumptions. The key insight is that attribute evaluation must be intermingled with the identification of previously computed attribute values, so that the newly computed values can be used in selecting which old attribute values to reuse. In addition, this paper compares the new tree-walking approach with previous matching-based approaches and applicative approaches based on function caching. As presented, the new incremental evaluator can be applied to hierarchical attribute grammar specifications based on ordered attribute grammar components, however, the technique can immediately be applied to convert any batch evaluator based on visit-sequences into an efficient incremental evaluator.

Keywords: attribute grammars, incremental evaluation, modular specification.

1 Introduction

A noted weakness of attribute grammars has been their failure to provide any means for constructing specifications of complete hierarchical computations, such as whole compilers, optimizers, and automatic parallelizers. Several recently developed "dialects" of attribute grammars address this limitation by their ability to naturally describe complex computations through the composition of attribute grammar specified sub-computations within the context of the original attribute grammar formalism. We refer to these new dialects as *hierarchical attribute grammars*. Examples of

such dialects are Attribute Coupled Grammars of Ganzinger and Giegerich [1, 2], the specification language SSL of the Synthesizer Generator of Reps and Teitelbaum [3], Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper [4, 5], and Modular Attribute Grammars of Carle and Pollock [6, 7].[1] When coupled with efficient evaluators, hierarchical attribute grammars provide a practical means for implementing complex transformation and analysis tools.

Although efficient "batch" evaluators can be constructed easily by recursively applying an efficient batch evaluator for non-hierarchical attribute grammars to each of the modules in a hierarchical specification, straightforward application of an efficient incremental evaluator will not result in an efficient incremental evaluator for the whole hierarchical specification. Several algorithms for incremental evaluation of hierarchical specifications have been proposed recently, including tree-walking approaches based on a syntactic matching operation [9, 6], and applicative approaches based on function caching [5, 10]. In this paper, we describe a new context-based, tree-walking incremental evaluation algorithm for hierarchical attribute grammars. The new incremental evaluator can be directly applied to hierarchical attribute grammars based on ordered attribute grammar components [11], however, the technique is easily extended to convert any batch evaluator based on visit-sequences into an efficient incremental evaluator. The new evaluator intermingles evaluation of attributes with the selection and reuse of previously computed attribute values, so that newly computed attribute values can be used to guide the reuse of old attributed subtrees. By taking attribute context into account in its selection of old attributed subtrees, the context-based evaluator is able to incrementally evaluate entire hierarchical specifications optimally.

We begin with some background on attribute grammars, SSL's notion of abstract syntax, hierarchical attribute grammars, the incremental evaluation problem in the context of hierarchical specifications, and matching-based incremental evaluators. We then describe the context-based incremental evaluator in detail and prove its correctness and optimality. The paper concludes by comparing the context-based evaluator to two purely applicative incremental evaluators based on function caching that are in many ways quite similar to the context-based evaluator.

2 Background

2.1 Attribute Grammars

An attribute grammar consists of a context free grammar G and an attribute system AS. AS associates sets of storage cells known as *attributes* with each symbol in G, and sets of equations

[1] The Modular Attribute Grammars of Carle and Pollock are unrelated to the Modular Attribute Grammars defined by Dueck and Cormack [8].

known as *semantic rules* with each production in G. Each semantic rule contains a *semantic function* that defines the value of an attribute at a syntax tree node in terms of the values of a subset of the attributes at adjacent nodes in the syntax tree. An attribute system describes a mapping from a syntax tree of a sentence in a context free language into a computation graph. Each vertex in the computation graph corresponds precisely to an attribute associated with a node in the syntax tree. The meaning of a syntax tree defined by an attribute grammar is the value that results from explicitly constructing the computation graph for the syntax tree according to the attribute grammar specification, and then evaluating the computation graph.

2.2 SSL's Notion of Abstract Syntax

As we describe in the next section, the ability for semantic functions in attribute grammar specifications to construct structured values is key to the usefulness of the hierarchical dialects. It is, therefore, useful to define a consistent notation for describing abstract syntax and for constructing new structured values. To meet this need, we adopt the terminology of the specification language SSL of the Synthesizer Generator [3]. As in SSL, a phylum/operator grammar definition consists of a set of phylum definitions and operator definitions, where *phyla* represent the nonterminals of a context free grammar, and *operators* name the alternative productions that can be derived from each phylum.

A *term* is a structured value constructed by applying a k-ary operator to k terms having the appropriate phylum. Hence, operators are term constructors. Each node in a term constructed using the phylum/operator mechanism is labeled with the operator applied at that node. A term is a representation of a rooted, labeled tree structure, but may be implemented as a directed acyclic graph with sharing of components. A component of a term will be referred to as a *subterm*.

2.3 Hierarchical Attribute Grammars

Hierarchical attribute grammar dialects make it possible to describe complex systems as a collection of sub-computations each defined by their own attribute grammar. For a detailed description of the members of the class of hierarchical attribute grammars, with an emphasis on Modular Attribute Grammars, we refer the reader to [6]. In this paper, we briefly outline Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper [4], and our hierarchical dialect, Modular Attribute Grammars.

Higher Order Attribute Grammars

Higher Order Attribute Grammars are distinguished from traditional attribute grammars in that they allow the tree being evaluated to be expanded during evaluation. In essence, certain subtrees of the syntax tree are not known, a priori, and only become known lazily during the evaluation process. An initial syntax tree consists of a collection of syntax tree nodes, some of which contain unexpanded *nonterminal attributes*. During evaluation, syntactic terms will be constructed and their values assigned to nonterminal attributes to extend the syntax tree. As the syntax tree is expanded, the evaluation process continues, computing values of attributes appearing throughout the old and new components of the syntax tree. Although not addressed in [4], syntactic values constructed during evaluation of semantic functions may be dag-valued or may share components with other syntactic values. To remedy this problem, assume that whenever a syntactic term is installed into a nonterminal attribute, a new tree representing that term will be constructed and installed instead. Attribute evaluation will, therefore, always be performed over a tree. By viewing the evaluation over the base syntax tree and each of the trees assigned to nonterminal attributes as distinct modules, we can consider Higher Order Attribute Grammars to be a hierarchical attribute grammar dialect.

Ordered Higher Order Attribute Grammars, the equivalent of ordered attribute grammars for Higher Order Attribute Grammars, can be evaluated by a visit-sequence interpreter as described in [11] that has been extended to permit assignments of terms to nonterminal attributes [4].

Modular Attribute Grammars

A Modular Attribute Grammar consists of a set of independently-specified modules, each of which describes a mapping from input terms to output terms – where the output term is generated by applying an attribute grammar evaluator to the specification of a module and an input term. Each module in a Modular Attribute Grammar is defined by its own attribute grammar specification which associates attributes and semantic rules with the syntax tree nodes of a common syntax. Each invocation of a module takes the form of a semantic function invocation. Modules are deliberately restricted to have a "single-visit" procedural interface in order to ensure the independence of modules. Although this interface is significantly different than the "multi-visit" interface between modules of a Higher Order Attribute Grammar, the incremental evaluation algorithm presented in this paper is directly applicable to Higher Order Attribute Grammars, as well.

Evaluation of a Modular Attribute Grammar begins with the evaluation of a specified "root" module. During evaluation, a set of *module instances* will be invoked. A module instance is a run-time instantiation of a module. We define MOD-INSTS to be the set of module instances that are invoked during evaluation of an input tree. The module instances in MOD-INSTS are naturally

organized as a tree since each module instance, other than the root module instance, is invoked during the evaluation of some other module instance in MOD-INSTS. It is important to understand that the set of module instances invoked by a module instance M depends upon the structure of M's syntactic input and upon computed semantic information, since modules are invoked as the result of evaluating semantic functions associated with particular operators in the input tree and since modules may be invoked conditionally.

Since a Modular Attribute Grammar is comprised of a set of independently specified attribute grammar specifications, it is natural to extend traditional attribute grammar classification schemes into the hierarchical context. For instance, the algorithms in this paper are designed to evaluate Ordered Modular Attribute Grammars, the class of Modular Attribute Grammars whose modules are ordered attribute grammar specifications.

2.4 Setting Up the Incremental Evaluation Problem

Incremental evaluation algorithms for non-hierarchical attribute grammars have seen extensive use in the context of language-based editors, as pioneered by the work of Reps and Teitelbaum in the Synthesizer Generator [12]. In the language-based editing framework, a user explicitly applies editing operations to a syntax tree and the incremental evaluator is responsible for quickly computing values of attributes indicating various kinds of derived information which are then displayed directly to the user. The editing framework maintains a single *primary* syntax tree and a set of *secondary* syntax subtrees that were previously part of the primary syntax tree, but have become detached as a result of some editing operation by the user. The secondary trees and their attributes, are reserved for possible use at a later time.

The incremental evaluator will be invoked to update attributes associated with the primary syntax tree after a sequence of editing operations transforms a primary tree T into a new tree T'. In order to unify incremental evaluation of non-hierarchical attribute grammars and hierarchical attribute grammars, it is useful to separate syntax tree editing from attribute value reuse by introducing a structure referred to as the *attribute tree* whose nodes will contain the attribute storage cells that are typically considered to be attached to nodes in a syntax tree.

An attribute is *consistent* if its value is that computed by applying its defining semantic function to its arguments, otherwise it is *inconsistent*. Similarly, a region of an attributed tree or a computation graph is consistent if its attributes are consistent. An attributed tree is *consistently attributed* if its attributes are consistent. Prior to the creation of T' from T, the primary syntax tree T will be consistently attributed and each secondary syntax tree will be consistently attributed, except at its root, since inherited attribute values are unavailable at that location.

Reuse of previously computed attribute values traditionally occurs as the result of *initialization*,

an operation that constructs the new (probably) inconsistent attribute tree from the old consistent attribute tree given the editing operations that created the new syntax tree from the old syntax tree. After initialization has been performed to build the attribute tree, a process known as *change propagation* is invoked to reevaluate attributes in the initial inconsistent tree in order to restore it to a consistent state.

The subset of attributes that require new values after editing is called AFFECTED. We define the set RETAINED to be the set of attributes that do not require new values after editing. The traditional goal for incremental evaluators for a non-hierarchical attribute grammar is to limit the total amount of work required to update the inconsistent attribute tree built by initialization to be O(|AFFECTED|), for a fixed attribute grammar, given the assumption that all semantic functions in an attribute grammar have roughly the same cost.

Unfortunately, an incremental evaluator for hierarchical attribute grammars cannot be constructed by simply applying an optimal incremental evaluator to each module of the hierarchical specification. As a result of the user's application of an editing operation to the syntax tree to be evaluated by the root module, the root module instance will be invoked with an attribute tree and an explicit edit description to be used in constructing the new inconsistent attribute tree from the previous consistent attribute tree. Other module instances will be invoked with an attribute tree and an input term generated during evaluation of the calling module instance.

Furthermore, it is somewhat difficult to even define the set AFFECTED in an intuitive fashion in the hierarchical context, since the contents of AFFECTED so strongly depend on how initialization is performed. In Section 5.4, we attempt to provide an intuitive definition of AFFECTED.

There have been only a few solutions proposed for the incremental evaluation of hierarchical attribute grammars. We categorize the previous approaches to this problem as either matching-based approaches or applicative approaches. We briefly describe the matching-based approach here and save the description of the applicative approaches for later.

2.5 Matching-Based Incremental Evaluators

One approach to the problem of the lack of explicit edit inputs to the intermediate module instances during incremental evaluation of a hierarchical attribute grammar is to require that the new input term for a module instance be explicitly compared with that module instance's previous input term to determine which subtrees from the old, consistent attributed syntax tree should be used in constructing the new, inconsistent attribute tree for that module instance. We refer to this process as *matching* since it is responsible for identifying subtrees for the old attribute tree that can be reused to represent regions of the new attribute tree. A matching-based evaluator is one that performs this matching step, and then invokes a traditional change propagation algorithm to

make the new attribute tree consistent.

The first incremental matching-based evaluator for hierarchical attribute grammars is Teitelbaum and Chapman's algorithm for incremental evaluation of Higher Order Attribute Grammars [9]. Teitelbaum and Chapman address the issue of what happens if the user makes a change to any portion of the syntax tree and the change subsequently causes nonterminal attributes to become inconsistent. Their algorithm enables reuse of previously computed values within the term assigned to the nonterminal attribute.

In Teitelbaum and Chapman's formulation of the matching problem, assignment of a term d to a nonterminal attribute D first requires that the term be "copied" to create an attribute tree d_{copy} that is suitable for evaluation. If a nonterminal attribute D with a value of d is assigned a term, say d', then their algorithm attempts to determine the relationship between d and d', and d_{copy} and d'_{copy} so that components from d_{copy} can be used in constructing d'_{copy}. The tree d'_{copy} is updated by invoking the optimal propagation algorithm for non-hierarchical attribute grammars [13].

Figure 1 shows example dags d, d', and the tree d_{copy}. Each tree node in d_{copy} has a label of the form X/Y, where X is the unique id representing the tree node itself, and Y is the unique id associated with a dag node in d represented by that tree node. *Matching* is the process of constructing the new attribute tree representing d' from the old attribute tree representing a term d. A "correct match" constructs an attribute tree that properly represents the term to be attributed.

Teitelbaum and Chapman claim that their algorithm performs the matching process in time proportional to the number of storage cell deallocations needed to eliminate nodes in d_{copy} that were not used in constructing d'_{copy} and the number of storage cell allocations needed to build the nodes in d'_{copy} which were not available in d_{copy}. Unfortunately, this number of allocations and deallocations depends on the match that is actually determined by the matching process, and Teitelbaum and Chapman's algorithms makes no attempt to construct the match that minimizes the number of allocations and deallocations.

Matching-based incremental evaluation algorithms for Modular Attribute Grammars are also

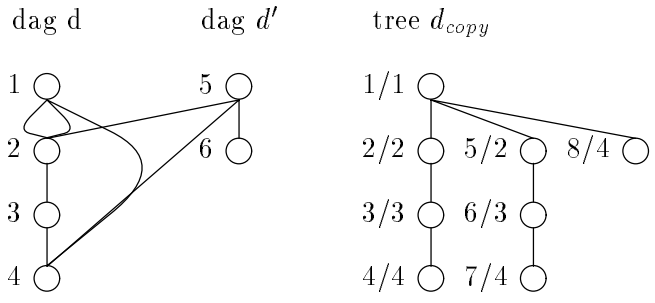

Figure 1 The dags d and d', and the tree d_{copy}

described by Carle and Pollock [6, 7]. In the context of Modular Attribute Grammars, the syntax tree itself is not modified by the evaluation process, but, instead, modules are explicitly invoked with syntactic terms to be evaluated according to the attribute grammar specification for that module. Incremental evaluation is performed in response to the application of a destructive editing operation that maps the base syntax term B into the new base syntax term B', and maps the attribute tree B_{copy} into the new attribute tree B'_{copy}. Given the initial attribute tree B'_{copy}, incremental evaluation begins by applying a version of Reps's optimal incremental evaluator for ordered attribute grammars [13, 14]. When a module instance that was last invoked to evaluate a term d is invoked by a module evaluation function responsible for evaluating a term d', the matching algorithm constructs the new attribute tree d'_{copy} from components of the attribute tree d_{copy} representing the dag d, and a set of new attribute tree nodes.

Carle and Pollock investigated a family of four matching algorithms. The family is based on a pair of heuristics that are intended to improve the utilization of previously computed attribute values over the Teitelbaum and Chapman algorithm. In particular, it was observed that minimizing the total number of allocations and deallocations requires constructing d'_{copy} by reusing maximal subtrees from d_{copy}. Thus, the first heuristic is the *Maximal Reuse Heuristic*, which requires reusing maximal subtrees from the old attribute tree d_{copy} to construct the new attribute tree d'_{copy}. The *Retention Heuristic* requires that subtrees of the old attribute tree d_{copy} that are not used in constructing the new tree d'_{copy} be retained for use by later matching operations. The matching operation must, therefore, be able to construct the new attribute tree d'_{copy} from subtrees taken from a forest of retained attribute trees.

Although we believe that several of the algorithms in the family of incremental matching algorithms presented in [6, 7] will be effective in practice, none of them are optimal if we take as our goal to minimize the total time required to update a set of module instances in response to a change to the input of the root module instance. Optimal evaluation of a particular module instance requires minimizing the sum of the cost of matching and the cost of propagation for that module instance. Through its selection of old attribute subtrees for reuse in the new attribute tree, matching not only determines the number of allocations and deallocations, but also determines which attributes in the new attribute tree are in AFFECTED. Thus, a matching algorithm that is only concerned with discovering a match that uses maximal subtrees from d_{copy} will be unable to minimize the cost of evaluating a module instance, since the algorithm does not attempt to minimize the size of the set of reused, AFFECTED attributes. An optimal matching algorithm must somehow take attribute context into account during the selection of subtrees from d_{copy} for use in d'_{copy} to limit the size of this set. This implies that an optimal evaluator must perform matching and propagation in an interleaved fashion to make attribute context available to the matching algorithm. This is the approach taken by the new context-based incremental evaluation method.

3 The Context-Based Incremental Evaluation Method

Our context-based incremental evaluator, which we call *CBOE* for Context-Based Ordered Evaluator, will perform incremental evaluation of a module of a hierarchical attribute grammar when applied to an input term by evaluating attributes associated with an attribute tree consisting of a crown of new attribute tree nodes and a tentatively chosen set of retained attribute subtrees. A crown consists of a contiguous set of nodes including the root of the attribute tree. During evaluation, the crown of new attribute tree nodes will be extended and the choices of retained attribute subtrees will be revised based on computed attribute values. The context-based evaluator is based on a surprisingly simple modification to the visit-sequence interpreter for Kastens's non-incremental ordered attribute grammar evaluator [11]. Retained attribute subtrees will be selected in a manner that permits the context-based evaluator to skip visits to those subtrees in order to minimize the cost of evaluating module instances.

It is worth emphasizing that the CBOE algorithm is responsible for incrementally updating a single module instance of a hierarchical specification. Later we will argue that recursive application of the CBOE algorithm to each of the module instances that need to be updated in response to a user's editing operation will result in an efficient incremental update of the entire collection of module instances.

We first describe Kastens's visit-sequence interpreter for the non-incremental ordered attribute grammar evaluator, and then the details of the CBOE algorithm.

3.1 The Visit-Sequence Interpreter

The tree-walking approach to ordered attribute grammar evaluation as described by Kastens is a "distributed-control" evaluator in the sense that a very simple "machine" is associated with each node in the tree to be evaluated. Initially, the machine at the root of the tree is active. The ordered attribute grammar evaluator, presented in Figure 2, spends no time making runtime decisions about which attributes to evaluate next — all of that information is encapsulated in a table of "visit-sequences" or "plans" generated prior to runtime. The visit-sequences are transition tables for the machines and govern the evaluation of attributes and the transfer of control from node to node in the tree. Each visit-sequence consists of a series of the following instructions:

- **EVAL(i,a)** - evaluate attribute a of the ith child of the current node
- **VISIT(i,r)** - perform the rth visit to the ith child of the current node
- **VISIT-PARENT(r)** - perform the rth visit to the parent of the current node

The VISIT and VISIT-PARENT instructions transfer control to the ith child or the parent of the currently active machine and make the machine for the child or parent become the active

```
Function Evaluate(TreeRoot)
  TreeNode = TreeRoot
  index = 1
  forever do
    if plan[TreeNode.op][index] is EVAL(i,a)
      args = ArgList(TreeNode,i,a)
      TreeNode.kids[i].a = apply semfunc[TreeNode.op,i,a] to args
      index = index + 1
    elseif plan[TreeNode.op][index] is VISIT(i,r)
      index = MapVisitToIndex(TreeNode.kids[i].op,0,r)
      TreeNode = TreeNode.kids[i]
    elseif plan[TreeNode.op][index] is VISIT-PARENT(r)
      if TreeNode = TreeRoot
        return ResultAttr(TreeRoot)
      index = MapVisitToIndex(TreeNode.parent.op,TreeNode.son,r)
      TreeNode = TreeNode.parent
    fi
  od
end
```

Figure 2 Visit-Sequence Evaluator for an Ordered Attribute Grammar (Kasten's Algorithm)

machine. Once a VISIT instruction to the ith child of a node N has been executed, control remains within the subtree rooted by the ith child of N until a VISIT-PARENT instruction is executed at the child to return control to N. A node labeled with operator *op* corresponding to the production $p : X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$ will be visited $NumVisits(X_0)$ times by its parent, and will visit its ith child, with $1 \leq i \leq n_p$, $NumVisits(X_i)$ times.

In addition to passing control from node to node, VISIT and VISIT-PARENT instructions can be logically viewed as communicating sets of newly computed attribute values to the visited node. With respect to node N, a VISIT from a node N to its ith child N_i passes the values of a set of inherited attributes associated with N_i to the machine associated with N_i so that node N_i can compute values for a subset of its synthesized attributes and then pass those values back to N with the execution of a VISIT-PARENT instruction. In actuality, there is no need to pass the values of the sets of attributes when VISIT and VISIT-PARENT instructions are executed since those attributes can be accessed directly from the visited node. In addition, the members of the subsets of available inherited and synthesized attributes that become available with each visit are implicit in the construction of visit-sequences, i.e., the kth visit to a node labeled with nonterminal n from its ith child always provides values for the same set of synthesized attribute, and the kth visit to the ith child of a node labeled with nonterminal n always provides values for the same set of inherited attributes. The sets of attributes whose values are provided by visits to nodes can be represented by a pair of tables, $AvailInhAttrs$ and $AvailSynAttrs$. The table $AvailInhAttrs$ maps each nonterminal n and visit number $visitNum$, for $1 \leq visitNum \leq NumVisits(n)$, to the ordered

sequence of inherited attributes whose values are available immediately before the *visitNum*th visit to a node labeled with a nonterminal n. The table *AvailSynAttrs* maps each nonterminal n and visit number *visitNum* to the ordered sequence of synthesized attributes whose values are available immediately after the *visitNum*th visit from a node labeled with a nonterminal n to its parent.

Figure 3 depicts three productions from a simple attribute grammar. Figure 4 depicts the visit-sequences for the three operators $OP1$, $OP2$ and $OP3$. Throughout the remainder of this paper, we refer to this visit-sequence evaluator interpreter as the batch ordered evaluator.

3.2 Applicative Reuse of Retained Attribute Subtrees

The CBOE algorithm is designed to access reused attribute subtrees only in a "read-only" manner, so that attribute subtree reuse can be implemented applicatively rather than destructively. To reuse an attribute subtree, the context-based evaluator will simply place a pointer to the subtree into a child field of a new attribute tree node. Applicative reuse of attribute subtrees significantly simplifies the matching component of the context-based evaluator by permitting attribute subtrees to be selected for reuse without concern for any needs for attribute subtrees that might arise later during the evaluation process, i.e., the reuse of an attribute subtree s will never prevent

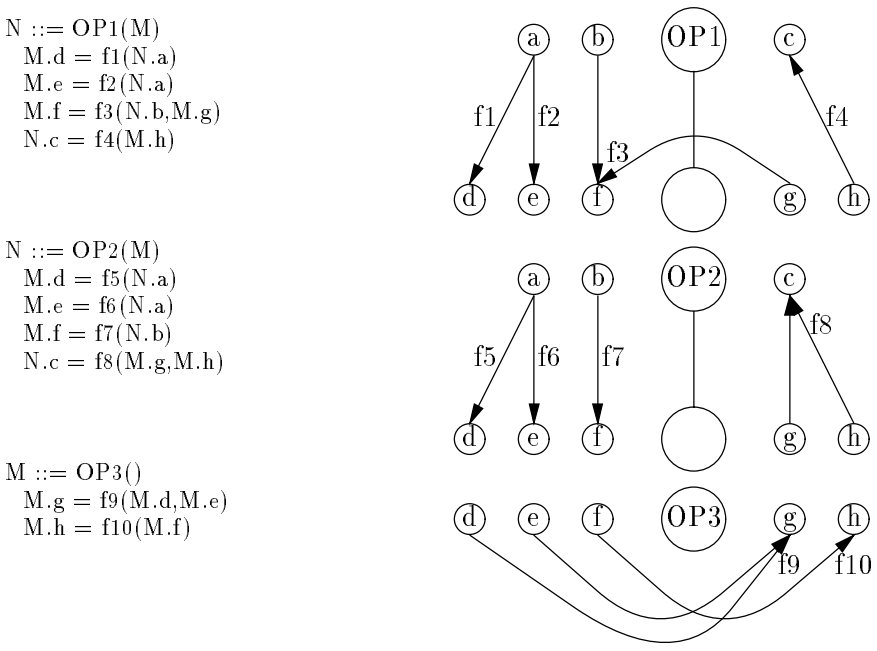

Figure 3 An Attribute Grammar Fragment

```
Visit-Sequence(OP1) = Visit-Sequence(OP2) =
[
                          -- values of N.a and N.b are available
     EVAL(1,d)            -- evaluate M.d
     EVAL(1,e)            -- evaluate M.e
     VISIT(1,1)           -- first visit computes M.g
     EVAL(1,f)            -- evaluate M.f
     VISIT(1,2)           -- second visit computes M.h
     EVAL(0,c)            -- evaluate N.c
     VISIT-PARENT(1)      -- final visit to parent with N.c computed
]

Visit-Sequence(OP3) =
[
                          -- values of M.d and M.e are available
     EVAL(0,g)            -- evaluate M.g
     VISIT-PARENT(1)      -- first visit to parent with M.g computed
                          -- values of M.d, M.e and M.f are available
     EVAL(0,h)            -- evaluate M.h
     VISIT-PARENT(2)      -- final visit to parent with M.h computed
]
```

Figure 4 Visit-Sequences for the Attribute Grammar Fragment

a larger attribute subtree t, which contains subtree s, from being reused elsewhere in the new attribute tree. Applicative reuse also ensures that evaluations of module instances in a complete hierarchical system cannot interfere in any manner. The goal of the CBOE algorithm is to construct a consistent attribute tree for a module instance. Although applicative reuse of old attribute subtrees implies that attribute trees will not necessarily be tree-structured, we will continue to use the term "attribute tree" to refer to these structures.

3.3 Identification of Appropriate Subtrees

The CBOE algorithm relies on the ability to map a "context" described by a triple of the form $\langle n,\ visitNum,\ valList \rangle$ into an "appropriate" subtree t such that $t.dagnode = n$, where $t.dagnode$ is the syntax tree node represented by the root of t, and the values of the inherited attributes in the sequence $AvailInhAttrs(phylum(n),\ visitNum)$ of $root(t)$ are identical to the values in *valList*. In the triple, n is the root of a syntax term, *visitNum* is an integer in the range 1 to $NumVisits(phylum(n))$, and *valList* is a sequence of inherited attribute values having a length of $|AvailInhAttrs(phylum(n), visitNum)|$.

An association map containing entries of the form

$$Context \rightarrow AttrNode$$

will be used to locate an attribute subtree that is appropriate for a given context. All access to retained attribute subtrees will be provided by this association map. Entries will be inserted into the association map by invoking the procedure *Enter* with a map, a context, and an appropriate attribute subtree for that context. The function *Lookup* will be used to retrieve appropriate attribute subtrees when invoked with a map and a context.

Instead of maintaining an association map for each module instance, we will maintain an association map for each module, thereby making all of the attribute subtrees attributed by an instance of a module be available for use during incremental evaluation of any instance of that module. Since old attribute subtrees are reused applicatively, increasing the size of the database of retained attribute trees by maintaining a single database for each module serves to decrease the time required to update module instances throughout a hierarchical system by increasing the chances of locating appropriate subtrees. Since each module of a Modular Attribute Grammar is defined by a different set of attributes and semantic rules, there is nothing to be gained by combining the association maps for each of the modules into a single global association map. The notation *Map(Module)* will refer to the association map for a module *Module*.

3.4 Matching with Evaluation

The context-based evaluator constructs an initial attribute tree representing an input term d' consisting of a set of *new* attribute tree nodes, a set of *interface* nodes, and a set of retained attribute subtrees. The new attribute tree nodes will form a crown of the initial attribute tree. The fringe of the new crown is a set of interface nodes, each of which contains fields typically associated with an attribute tree node, including the fields *parent*, *dagnode*, and *operator*, and, in addition, a field *AppropSubtree* which will reference an attribute subtree that has been identified as being appropriate by using the association map for the invoked module. The *parent* field of attribute tree nodes and interface nodes will only be accessed at times when the *parent* field is actually meaningful, i.e., when it can be guaranteed that the attribute tree node is only accessible as the child of a single attribute tree node.

Each interface node contains a set of inherited attributes, but synthesized attributes will not be represented explicitly. An attempt to retrieve the value of synthesized attribute a of interface node f will, instead, return the value of synthesized attribute a of the root of the subtree referenced by $f.AppropSubtree$.

The initial attribute tree will be constructed during a depth-first traversal over a crown of d'. At each node n in d', the function *Lookup* will be invoked to locate a retained attribute subtree that is appropriate for the context $\langle n,\ 0,\ \{\}\rangle$. An attribute subtree t is appropriate for the context $\langle n,\ 0,\ \{\}\rangle$ if $t.dagnode = n$, i.e., if the syntax subterm represented by t is rooted by node n. If the *Lookup* operation returns the value NULL, then a new attribute tree node is inserted into the crown of the new attribute tree. If the *Lookup* operation returns an attribute subtree t, then an interface node whose *ApppropSubtree* field is initialized to t is inserted into the crown of the new attribute tree.

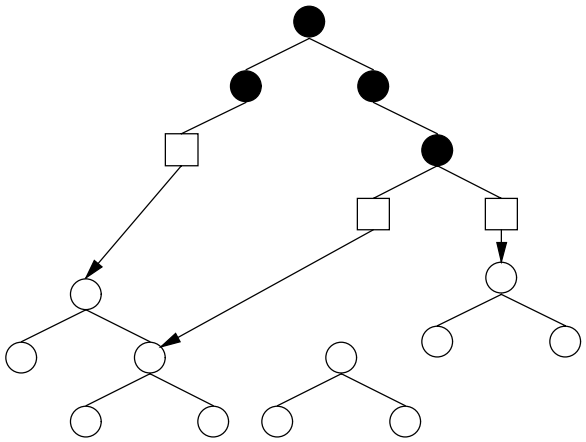

Figure 5 An Initial Attribute Tree

Figure 5 depicts an initial attribute tree for an unspecified input term and an unspecified module. Filled circles represent new attribute tree nodes, boxes represent interface nodes, and unfilled circles represent old attribute tree nodes.

The visit-sequence interpreter for the context-based evaluator will be invoked after the initial attribute tree has been constructed. The interpreter will execute EVAL(i,a) instructions exactly as in the batch ordered evaluator. VISIT(i,r) instructions will be performed exactly as before if the node to be visited is a new attribute tree node. If the node to be visited by a VISIT(i,r) instruction is an interface node f, then an attempt will be made to identify an attribute subtree that is appropriate for the rth visit to f by invoking the function *Lookup* with a context $\langle f.dagnode,\ r,\ valList\rangle$, where *valList* contains the values of the inherited attributes in *AvailInhAttrs(phylum(f),r)* of interface node f. If an appropriate subtree t can be found for the rth visit to f then the visit will be skipped, and t will be attached to the new attribute tree. If an appropriate attribute subtree cannot be located for the rth visit to f, then interface node f will be expanded into a new attribute tree node by replacing f with a new attribute tree node that has interface nodes for each of its children, and the new attribute tree node will be visited. When the kth visit to any new attribute tree node, say *AttrNode*, completes, as directed by a VISIT-PARENT(k) instruction, an entry will

be inserted into the association map for context $\langle AttrNode.dagnode,\ k,\ valList \rangle$, with $valList = AvailInhAttrs(phylum(AttrNode),k)$. As soon as this new map entry is inserted into the association map, the attribute subtree rooted by *AttrNode* becomes available for use as a retained attribute subtree that is appropriate for the kth visit to an interface node. Visit-sequence interpretation terminates when the final VISIT-PARENT(r) instruction is executed at the root of the attribute tree.

4 The CBOE Algorithm

Function *HagEvaluate*, shown in Figure 6, invokes function *Evaluate*, also shown in Figure 6, to apply the root module to an input term. Function *Evaluate* is invoked with the name of a module and the syntax term d'. *Evaluate* first retrieves the association map for *Module* and then invokes the function *Lookup* in an attempt to find an old attribute subtree that represents the entire term d'. If an old attribute subtree is successfully located, then the result attribute associated with the root of the attributed tree is returned as the result of evaluating the term d'. If an old attribute subtree cannot be located by *Lookup*, then *Evaluate* invokes function *BuildInitial* to construct the initial attribute tree representing d' and then function *MatchEval* to evaluate the new tree. On completion of *MatchEval*, the value of the result attribute associated with the root of the attributed tree is returned as the result of evaluating d'.

Function *BuildInitial*, shown in Figure 7, builds the initial attribute tree during a depth-first traversal of a crown of d'. First, *BuildInitial* allocates a new attribute tree node. It then invokes the recursive procedure *BuildInitialRec* for each of the children of the root of d'. Whenever *Buil-*

```
Function HagEvaluate(d')
  return Evaluate(Root,d')
end

Function Evaluate(Module, d')
  Map = MapFor(Module)

  AttrTree = Lookup(Map,0,{})
  if AttrTree != NULL
    return ResultAttr(root(AttrTree))

  AttrTree = BuildInitial(d',Map)
  MatchEval(root(AttrTree),Map)
  return ResultAttr(root(AttrTree))
end
```

Figure 6 HagEvaluate and Evaluate – CBOE Algorithm

```
Function BuildInitial(d', Map)
  NewAttrRoot = CreateAttrNode(root(d'))
  newAttrRoot.new = true
  for each child i of root(d')
    BuildInitialRec(root(d').kids[i],NewAttrRoot,i,Map)
  return NewAttrRoot
end

Procedure BuildInitialRec(dagnode,ParentAttrNode,i,Map)
  AppropSubtree = Lookup(Map,0,{})
  if AppropSubtree = NULL
  then
    - insert a new attribute tree node
    newAttrNode = CreateAttrNode(dagnode)
    ParentAttrNode.kids[i] = newAttrNode
    newAttrNode.parent = ParentAttrNode
    newAttrNode.which_son = i
    newAttrNode.new = true
    - - recurse
    for each child i of dagnode
      BuildInitialRec(dagnode.kids[i],newAttrNode,i,Map)
  else
    - - found an appropriate subtree
    newInterfaceNode = CreateInterfaceNode(dagnode)
    ParentAttrNode.kids[i] = newInterfaceNode
    newInterfaceNode.parent = ParentAttrNode
    newInterfaceNode.which_son = i
    newInterfaceNode.AppropSubtree = AppropSubtree
  fi
end
```

Figure 7 BuildInitial and BuildInitialRec – CBOE Algorithm

dInitialRec, also shown in Figure 7, is unable to locate an appropriate subtree for a node n in d', it allocates a new attribute tree node, inserts the new node into the crown of the initial attribute tree, and then recursively invokes itself with each of the children of n. Whenever *BuildInitialRec* is able to locate an appropriate subtree for a syntax node n in d', it allocates an interface node, inserts it into the crown of the initial attribute tree, and then assigns the subtree to the *AppropSubtree* field of the interface node.

```
Procedure MatchEval(AttrRoot,Map)
  AttrNode = AttrRoot
  index = 1
  forever do
    if plan[AttrNode.op][index] is EVAL(i,a)
      args = ArgList(AttrNode,i,a)
      AttrNode.kids[i].a = apply semfunc[AttrNode.op,i,a] to args
      index = index + 1
    else if plan[AttrNode.op][index] is VISIT(i,r)
      if not(isInterface(AttrNode.kids[i]))
      then -- we are visiting an attribute tree node in the crown of d'_copy
        index = MapVisitToIndex(AttrNode.kids[i].op,0,r)
        AttrNode = AttrNode.kids[i]
      else -- attempt to locate an appropriate subtree for this interface node
        AttrVals = retrieve AvailInhAttrs(phylum(Attrnode.kids[i]),r) from AttrNode.kids[i]
        AppropSubtree = Lookup(Map,⟨AttrNode.kids[i].dagnode,r,AttrVals⟩)

        if AppropSubtree != NULL
        then
          Install(AppropSubtree,AttrNode,i,r)
          -- skip the visit
          index = index + 1
        else -- no appropriate subtree found, so expand the tree and do the visit
          Expand(AttrNode.kids[i],r)
          index = MapVisitToIndex(AttrNode.kids[i].op,0,r)
          AttrNode = AttrNode.kids[i]
        fi
      fi
    elseif plan[AttrNode.op][index] is VISIT-PARENT(r)
      if AttrNode = AttrRoot
        return

      -- extend map for this module
      AttrVals = retrieve AvailInhAttrs(phylum(AttrNode),r) from AttrNode
      Enter(Map,⟨AttrNode.dagnode,r,AttrVals⟩, AttrNode)

      index = MapVisitToIndex(AttrNode.parent.op,AttrNode.which_son,r)
      AttrNode = AttrNode.parent
    fi
  od
end
```

Figure 8 MatchEval – CBOE Algorithm

Procedure *MatchEval*, shown in Figure 8, is the visit-sequence interpreter for the context-based evaluator. *MatchEval* restricts its traversal to the crown of new nodes in the new attribute tree.

The crown of new nodes expands as *MatchEval* interprets visit-sequences. *MatchEval* executes EVAL, VISIT and VISIT-PARENT instructions in the visit-sequence for an attribute tree node *AttrNode* using the following procedures:

- Each EVAL(i,a) instruction is executed exactly as in the batch ordered evaluator.
- Each VISIT(i,r) instruction whose target is an attribute tree node in the crown of the new attribute tree is executed unconditionally. If the target of the VISIT instruction is an interface node, the inherited attribute values that are available for the rth visit to the interface node will be retrieved from the interface node and function *Lookup* will be invoked in an attempt to identify an old attribute subtree that is appropriate for the rth visit to the interface node. If an appropriate attribute subtree is located by *Lookup*, then procedure *Install* will be invoked to attach the subtree to the new attribute tree, and then the visit will be skipped. If an appropriate subtree cannot be located by *Lookup*, then the interface node will be expanded into a new attribute tree with interface nodes for its children by invoking *Expand*. The new attribute tree node is then visited.
- Each VISIT-PARENT(r) instruction causes an entry to be inserted into the association map for the current module and then transfers control to the parent of the current attribute tree node. The inserted entry indicates that the subtree rooted at the current attribute tree node, say *AttrNode*, is appropriate for the rth visit to any interface node f with $f.dagnode = AttrNode.dagnode$.

```
Procedure Install(AppropSubtree,AttrNode,i,r)
  -- AttrNode.kids[i] is an interface node
  if r != NumVisits(phylum(AttrNode.kids[i]))
  then
    AttrNode.kids[i].AppropSubtree = AppropSubtree
  else
    -- remove the attribute tree node from the tree, since AppropSubtree
    -- fits exactly into the new attribute tree
    Dealloc(AttrNode.kids[i])
    AttrNode.kids[i] = AppropSubtree
  fi
end
```

Figure 9 Install – CBOE Algorithm

Procedure *Install*, shown in Figure 9, is invoked with a retained attribute subtree *AppropSubtree*, an attribute tree node *AttrNode*, a child index i, and a visit number r, and is responsible for attaching the subtree *AppropSubtree* to the new attribute tree. Whenever *Install* is invoked, *AttrNode.kids[i]* contains a reference to an interface node. If the rth visit to *AttrNode.kids[i]* is not

the final visit to the interface node, then *AppropSubtree* will be assigned to the *AppropSubtree* field of the interface node. If the rth visit is the final visit to the interface node, then the interface node will be deallocated, and *AppropSubtree* will be inserted in its place. In this manner, a retained attribute subtree that is appropriate for the last visit to an interface node becomes attached to a new attribute tree node without an intervening interface node.

```
Procedure Expand(Interface,r)
  AttrNode = CreateAttrNode(Interface.dagnode)
  AttrNode.new = true
  AttrNode.parent = Interface.parent
  AttrNode.which_son = Interface.which_son

  copy the inherited attributes of Interface into AttrNode
  copy the synthesized attributes of Interface.AppropSubtree into AttrNode

  for each child i of AttrNode
    newInterface = CreateInterface(AttrNode.dagnode.kids[i])
    AttrNode.kids[i] = newInterface
    newInterface.parent = AttrNode
    newInterface.which_son = i
    copy the inherited attributes of Interface.AppropSubtree.kids[i] to newInterface
    newInterface.AppropSubtree = Interface.AppropSubtree.kids[i]

  Dealloc(Interface)
end
```

Figure 10 Expand – CBOE Algorithm

Procedure *Expand*, shown in Figure 10, is invoked whenever an appropriate attribute subtree cannot be located for an interface node. It is called with an interface node *Interface* and a visit number r and is responsible for allocating a new attribute tree node containing interface nodes for each of its children. After allocating the new attribute tree node, *Expand* copies the inherited attributes from *Interface* and the synthesized attributes of *Interface.AppropSubtree* into the new attribute tree node. *Expand* then allocates a new interface node for each child of the new attribute tree node, inserts each of the new interface nodes into the new attribute tree node, and then copies the inherited attributes of the ith child of *Interface.AppropSubtree* into the ith new interface node. In addition, *Expand* assigns the ith child of *Interface.AppropSubtree* into the *AppropSubtree* field of the ith new interface node. Finally, *Expand* deallocates *Interface*.

4.1 Application of the CBOE Algorithm to Modular Attribute Grammars

To evaluate a Modular Attribute Grammar, the CBOE algorithm is applied to the root module instance of the Modular Attribute Grammar and a modified input term to be evaluated. The CBOE algorithm is also applied whenever a module is invoked as a semantic function. The CBOE

algorithm is invoked by calling function *Evaluate* with the name of the module to be executed and a term to be attributed according to that module's specification.

4.2 Application of the CBOE Algorithm to Higher Order Attribute Grammars

To evaluate a Higher Order Attribute Grammar, the CBOE algorithm is applied to the Higher Order Attribute Grammar specification and a modified input term to be evaluated. As new syntactic terms are assigned to non-terminal attributes and the base syntax tree is extended, there is no need to recursively invoke the CBOE algorithm, since the proper updating steps will be taken directly by the modified visit-sequence interpreter.

5 Analysis of the CBOE Algorithm

We now prove that the CBOE algorithm correctly updates a complete hierarchical specification and does so optimally with respect to any evaluation algorithm that explicitly constructs a consistent attribute tree for each module instance and requires that the attribute tree be traversed from its root by following parent and child links between adjacent attribute tree nodes.

5.1 Correctness

The proof of correctness of the CBOE algorithm requires that we be able to make claims about the values of attributes throughout the attribute tree being constructed and evaluated by the context-based evaluator. An abstract view of attribute trees must be adopted, though, since the actual nodes in the attribute tree may change as the result of assigning a retained attribute subtree to the *ApproprSubtree* field of an interface node or as a result of expanding an interface node into a new attribute tree node having new interface nodes for each of their children. Each attribute, therefore, will be considered to be a pair $\langle Path, Attr \rangle$ where *Path* is a sequence of child indices leading from the root of the attribute tree to a specific attribute tree node, and *Attr* is an attribute associated with that node. An interface node f and the root of the subtree referenced by $f.ApproprSubtree$ will be abstractly treated as a single attribute tree node, i.e., if the node reached via *Path* is an interface node f and *Attr* is an inherited attribute, then the actual value of the attribute can be found in f, otherwise, if *Attr* is a synthesized attribute, then the actual value of the attribute can be found in the root of $f.ApproprSubtree$.

Instead of referring to the operations EVAL, VISIT and VISIT-PARENT as instructions, it is useful to define an instruction to be a pair $\langle Path,\ index \rangle$, where *Path* identifies an attribute tree node in the new attribute tree, and *index* is the index of an operation in the visit sequence for

the attribute tree node identified by *Path*. The function *PathTo* will be used below to convert an attribute tree node into a path to that node.

To prove that the CBOE algorithm is correct we will prove that *Evaluate* correctly updates each module in a hierarchical specification.

Theorem 1: Function *Evaluate* of the CBOE algorithm will correctly compute the result of applying a module $Module$ to a term d'.

Proof: Let $ConsBatch(Inst)$ be the set of attributes in an attribute tree being evaluated by the batch evaluator that are guaranteed to be consistent just before executing $Inst$, and let $ConsInc(Inst)$ be the set of attributes in an attribute tree being evaluated by the context-based evaluator that are guaranteed to be consistent just before executing $Inst$.

To prove Theorem 1, we will demonstrate that the following invariant is satisfied after each EVAL, VISIT and VISIT-PARENT operation is executed by the context-based evaluator:

> If the attribute set $ConsBatch(Inst)$ is identical to the attribute set $ConsInc(Inst)$, and if instruction $Inst'$ is the instruction that will be evaluated by the context-based evaluator after executing instruction $Inst$, then $ConsBatch(Inst')$ will be identical to $ConsInc(Inst')$.

Satisfaction of this invariant implies that the result attribute associated with the root of the new attribute tree will be assigned a consistent value, and, therefore, *Evaluate* will return the correct value. Notice that the instruction that will be executed by the batch evaluator after executing instruction $Inst$ will not necessarily be the same as the instruction that will be executed by the incremental evaluator after executing instruction $Inst$, since the incremental evaluator may skip entire visits to attribute subtrees.

We now examine the processing of operations by the batch evaluator and CBOE for an instruction $Inst = \langle PathTo(AttrNode), index \rangle$ assuming that $ConsBatch(Inst) = ConsInc(Inst)$.

- If $Inst$ = EVAL(i,a).

  The instruction to be executed after $Inst$ by both evaluators will be

  $$Inst' = \langle \text{PathTo(AttrNode),index+1} \rangle.$$

  Both the batch evaluator and the context-based incremental evaluator will evaluate attribute a of *AttrNode.kids[i]*, hence

  $$ConsBatch(Inst') = ConsBatch(Inst) \cup \langle PathTo(AttrNode.kids[i]), a \rangle$$

  and

  $$ConsInc(Inst') = ConsInc(Inst) \cup \langle PathTo(AttrNode.kids[i]), a \rangle.$$

Therefore,

$$ConsBatch(Inst') = ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if *AttrNode.kids[i]* is a new attribute tree node, then the instruction to be executed after $Inst$ by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode.kids[i]), MapVisitToIndex(AttrNode.kids[i].op, 0, r)\rangle.$$

  No attributes are evaluated by either evaluator when executing a VISIT(i,r) operation. Therefore,

  $$ConsBatch(Inst') = ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if *AttrNode.kids[i]* is an interface node for which an appropriate retained subtree can be found, then the visit will be skipped, and, therefore, the instruction to be executed after $Inst$ by the context-based evaluator will be

  $$Inst' = \langle PathTo(AttrNode.kids[i]), index+1\rangle.$$

  Let *oldRoot* be the root of the attribute subtree that was last referenced by the *ApppropSubtree* field of *AttrNode.kids[i]* and which must, therefore, be appropriate for visit $r-1$ to the interface node *AttrNode.kids[i]*, and let *newRoot* be the new attribute subtree that is appropriate for the rth visit to *AttrNode.kids[i]*. *AttrNode.kids[i].dagnode*, *oldRoot.dagnode* and *newRoot.dagnode* must have the same value, say *dagnode*.

  By the time that the batch evaluator reaches instruction $Inst'$, it will have computed consistent values for every attribute in *EvalWithVisit(dagnode,r)*, the set of attributes in any attribute subtree representing the syntax subterm rooted by node *dagnode* that will be assigned consistent values during the rth visit to its root. Therefore,

  $$ConsBatch(Inst') = ConsBatch(Inst) \cup EvalWithVisit(dagnode, r).$$

  Since subtree *oldRoot* was appropriate for visit $r-1$ to *AttrNode.kids[i]*, every attribute of the subtree rooted by *oldRoot* in $\bigcup_{1\leq j\leq r-1} EvalWithVisit(dagnode, j)$ must be consistent, and since subtree *newRoot* is appropriate for the rth visit to *AttrNode.kids[i]*, every attribute in $\bigcup_{1\leq j\leq r} EvalWithVisit(dagnode, j)$ of the subtree rooted by *newRoot* must be consistent. Therefore, as a result of replacing *oldApprop* with *newApprop*,

  $$ConsInc(Inst') \;=\; (ConsInc(Inst) - \bigcup_{1\leq j\leq r-1} EvalWithVisit(dagnode, j))$$

$$\cup \bigcup_{1 \le j \le r} EvalWithVisit(dagnode, j),$$

which can be simplified to

$$ConsInc(Inst') = ConsInc(Inst) \cup EvalWithVisit(dagnode, r),$$

and, therefore,

$$ConsBatch(Inst') = ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if *AttrNode.kids[i]* is an interface node for which an appropriate retained subtree cannot be found, the instruction to be executed by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode.kids[i]), MapVisitToIndex(AttrNode.kids[i].op, 0, r) \rangle.$$

  The context-based evaluator, however, will first expand the interface node *AttrNode* into a new attribute tree node with interface nodes for each of its children by invoking the procedure *Expand*. No attributes are modified by the batch evaluator or the context-based evaluator. Therefore,

  $$ConsBatch(Inst') = ConsInc(Inst').$$

- If $Inst$ = VISIT-PARENT(r) and *AttrNode* is not the root of the new attribute tree, then the next instruction to be executed by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode.parent),$$

  $$MapVisitToIndex(AttrNode.parent.op,\ AttrNode.son,\ r) \rangle.$$

  No attributes are evaluated by either evaluator during the execution of a VISIT-PARENT(r) operation. Therefore,

  $$ConsBatch(Inst') = ConsInc(Inst').$$

- If $Inst$ = VISIT-PARENT(r) and if *AttrNode* is the root of the new attribute tree, then both evaluators are done, and there is no next instruction $Inst'$.

Since, each of the attributes in the new attribute tree constructed by the context-based evaluator will have consistent values once the final VISIT-PARENT instruction has been executed at the root of the new attribute tree, the attribute result returned by function *Evaluate* will be correct. □

5.2 Efficiency

The CBOE algorithm constructs consistent attribute trees for each module instance invoked during evaluation of a modified input term. Under the assumption that any incremental evaluator for a module of a hierarchical specification must traverse the crown of new attribute tree nodes reaching from the root of the new attribute tree to any node containing an attribute whose value must be recomputed in order to construct a consistent attribute tree for that module, we now prove optimality of the algorithm.

Theorem 2: The CBOE algorithm evaluates Modular Attribute Grammars optimally.

Proof:

The standard accounting procedures for incremental attribute grammar evaluators abstract away from the details of semantic functions by assuming that all attributes can be computed by performing the same amount of work to evaluate their defining semantic rules. In the hierarchical context, this is clearly a bad assumption, since some attributes will be defined by potentially expensive module invocations. We finesse this issue by looking at the cost of incrementally updating each module instance independently – the cost of evaluating a module invocation is charged to the incremental evaluator for that module instance.

Presume that an attribute tree consisting of new attribute tree nodes, a fringe of interface nodes, and a set of reused attribute subtrees has somehow been constructed. The set of attribute tree nodes visited by the incremental evaluator as it traverses the new attribute tree to visit attribute tree nodes with inconsistent attribute values is referred to as AFFCROWN. The nodes in the attribute tree that are not traversed by the incremental evaluator will be referred to as the $\overline{\text{AFFCROWN}}$ set.

Since an incremental evaluator is required to traverse all of the nodes in the set AFFCROWN, if only a constant amount of work is required at each attribute tree node in AFFCROWN, ignoring the actual cost of computing attribute values, then minimizing the cost of updating a module instance will require selecting old attribute subtrees in a manner that minimizes the size of AFFCROWN, or, equivalently, maximizing the size of $\overline{\text{AFFCROWN}}$.

Under the assumption that entries can be inserted and retrieved from the association map in constant time, the context-based evaluator can be shown to perform only a constant amount of work at each node in AFFCROWN and at the fringe of AFFCROWN. Since each of the productions in the phylum/operator grammar underlying each module specification has a fixed number of children, evaluation of a module by the context-based evaluator requires $O(|\text{AFFCROWN}|)$ time. We define AFFCROWN$'$ to be the set of attribute tree nodes in the minimum-sized AFFCROWN, and $\overline{\text{AFFCROWN}}'$ to be the set of attribute tree nodes in the maximum-sized $\overline{\text{AFFCROWN}}$. We now argue that the context-based evaluator only requires $O(|\text{AFFCROWN}'|)$ time.

The goal of the entire process of selecting appropriate attribute subtrees in the context-based evaluator is to ensure that the minimum possible number of new attribute tree nodes are created in the new attribute tree since AFFCROWN is just the set of new attribute tree nodes. The number of new attribute tree nodes is minimized by ensuring that new attribute tree nodes are inserted into the new attribute tree only when it can be guaranteed that no retained attribute subtree could be used instead. New nodes are inserted into the new tree by *BuildInitial* and by *Expand*. *BuildInitial* inserts a new attribute tree node *AttrNode* only if there does not exist a retained attribute subtree that represents the syntax subterm rooted by *AttrNode.dagnode*. *Expand* inserts a new attribute tree only after *Lookup* has failed to find an appropriate retained attribute subtree for an interface node. Failure to identify an appropriate attribute subtree means that no retained attribute subtree can be used to represent that interface node without needing to recompute some of its attributes, and therefore, the new attribute tree node represents a node that must be in AFFCROWN. Therefore, the size of AFFCROWN is minimized, and the cost of evaluation by the context-based evaluator is $O(|\text{AFFCROWN}'|)$.

Since each module instance that is invoked during evaluation of a hierarchical system will be updated optimally, and since applicative reuse of retained attribute subtrees guarantees that the evaluation of a module instance cannot interfere with the evaluation of another module instance, the complete hierarchical system will be updated optimally, and Theorem 2 holds. □

The cost of applying the CBOE algorithm to a hierarchical specification with a modified input term is the sum of the costs of evaluating each of the module instances in AFF-MOD-INSTS, where AFF-MOD-INSTS is the set of module instances that are invoked by the incremental evaluator during evaluation of the modified input term. The total cost of evaluation is, therefore, $O(\sum_{aff \in \text{AFF-MOD-INSTS}} |\text{AFFCROWN}'_{aff}|)$.

5.3 Storage Requirements of the CBOE Algorithm

The CBOE algorithm requires storage for syntax terms, the attributed tree for each module instance, attribute values, and most significantly the database of previously evaluated attribute subtrees accessible via the association map entries for each module instance. Storage for the first three of these categories is clearly required in order to perform any sort of evaluation. The fourth category, the database of previously computed values, is the hallmark of incremental algorithms in their attempt to swap space for time. The database will grow without bound as updating operations are performed if no action is taken to limit the size of the database. To combat this problem, we suggest that an implementation of the CBOE algorithm attempt to determine an appropriate working set size for the cache as updates occur. Whenever the memory requirements exceed a reasonable working set size, retained attribute subtrees can be flushed from the database and deallocated.

This may, of course, require additional computation at some later time to recreate those attribute subtrees.

5.4 The CBOE Algorithm Revisited

In the previous section we claimed that efficient incremental evaluation of modules in a Modular Attribute Grammar can be achieved by selecting old attribute subtrees in a way that minimizes the size of AFFCROWN. Mapping the CBOE process to the standard incremental evaluation process of initialization followed by change propagation would allow us to define AFFECTED in terms of the results of initialization, but, this is impossible since initialization and change propagation are intertwined by design.

Since we cannot separate initialization and change propagation in the actual CBOE algorithm, in order to define AFFECTED, we appeal to an oracle to perform initialization, and define AFFECTED in terms of this oracle-based initialization. The oracle is asked to construct a new attribute tree for a module instance consisting of a minimal-sized crown of new attribute tree nodes and a collection of maximal-sized reused attribute subtrees such that all of the attributes associated with attribute tree nodes in the reused subtrees are in RETAINED. (Recall that RETAINED is defined to be the set of attributes that do not require new values after editing.) By construction, the number of attribute tree nodes in the reused subtrees must be $|\overline{\text{AFFCROWN}'}|$, and therefore, the number of new attribute tree nodes must be $|\text{AFFCROWN}'|$.

Given the initialization by the oracle, the set AFFECTED is seen to contain all of the attributes associated with new attribute tree nodes, since each of the attributes in AFFECTED must be assigned a value during change propagation. Change propagation can easily be performed over this new attribute tree in $O(|\text{AFFECTED}|)$, which is just $O(|\text{AFFCROWN}'|)$, exactly as in the CBOE algorithm.

6 The Applicative Approaches

Two purely applicative evaluation schemes, one based on semantic function caching and the other based on visit function caching, have been investigated for incremental evaluation of hierarchical attribute grammars. Both of these schemes avoid storing computed attribute values in nodes of any sort of attribute tree.

In a language-based editing context, where all of the attributes computed during evaluation of an input term are considered to be the "result" of evaluation, we feel that it is natural to construct an explicit representation of attributed trees as in the CBOE algorithm. For applications in which only the values of a set of synthesized attributes associated with the root of the input term being evaluated are considered to be the "result" of evaluation, purely applicative approaches may be

appropriate.

We now describe both of the applicative schemes and compare them to our context-based evaluation algorithm.

6.1 Pugh's Incremental Evaluator

Attribute grammars can automatically be translated into recursive functional programs using well-known techniques [15, 16]. Here it is assumed that the attribute grammar has only a single result attribute associated with its root. Evaluation over an input tree T is performed by applying the function defining the value of the result attribute to the input tree T. The set of recursive functions acts as an "output oriented" evaluator [17], since attribute evaluations are performed on demand leading from the original request to compute the result attribute.

Translation of attribute grammars to functional programs is general; techniques have been demonstrated for translation of well-formed attribute grammars into recursive functions [18, 15]. Simplified techniques for subclasses of well-formed attribute grammars are easy to construct. The translation of ordered attribute grammars, a subclass of well-formed attribute grammars introduced by Kastens [11], is based on the notion that each nonterminal symbol in an attribute grammar represents a set of functions, one function per synthesized attribute associated with the nonterminal symbol, and each mapping a syntactic subtree and a subset of inherited attributes to a synthesized attribute value [18].

Although execution of the program constructed by translation of an attribute grammar will correctly compute the result attribute given a tree T to be attributed, execution of the functional program may take time exponential in the size of T since computed attribute values are not stored by the recursive program and attributes may be reevaluated multiple times [19]. This exponential behavior can be avoided by explicitly caching computed attribute values into the nodes of the tree T and then reusing those values anytime a request is made to recompute one of them. This caching of computed values can also be achieved by using function caching [20] to "short-circuit" attempts to invoke the same recursive procedure to recompute attribute values. Applying an efficient caching technique gives a linear time attribute grammar evaluator.

A functional program can be generated for a hierarchical specification by translating each module into a set of recursive functions named to identify the module for which that function was created, and to translate each module invocation into an invocation of the function that computes the synthesized result attribute for that module. This approach is free from complications arising from the fact that the input term to modules may be dag-structured values since it does not store any information into the nodes of the syntactic term being evaluated and since it does not use parent pointers to traverse the syntactic input term. Again, efficient batch evaluation requires that

attribute values computed during evaluation of the functional program be cached to prevent unnecessary reevaluations of those attributes. The cost of evaluating a complete hierarchical specification using the resulting evaluator is $O(\sum_{inst \in \text{MOD-INSTS}} |TREE(S_{inst})|)$, where $TREE(S_{inst})$ is a tree that represents the term S_{inst}, and MOD-INSTS is the complete set of module instances that will be invoked during the evaluation of the hierarchical specification, i.e., the cost of evaluation is linear in the size of the trees representing each input term for each module instance.

Pugh investigated applicative schemes for the incremental evaluation of functional programs using function caching [21]. Pugh demonstrated that an incremental evaluator for pure functional programs could be constructed by combining the interpreter for the functional language with efficient function caching algorithms. Each time a function is invoked with a set of arguments, the function cache is queried to determine if that function has been previously invoked with those arguments. A new function invocation with identical arguments, in a value-oriented sense, to a previous invocation's arguments can simply extract the stored result from the cache without reinvoking the function. Application of an efficient function caching interpreter provides incremental evaluation of any functional program, although the efficiency of incremental evaluation depends on the computation. If computations are suitably stable, i.e., if in response to changes to the input of a computation, many of the functions invoked during the computation are again invoked with the same arguments, then the function caching technique will prove to be quite effective for the incremental evaluation of those computations.

In order to compare incremental evaluation by function caching with incremental evaluation of attribute grammars, Pugh showed that an incremental evaluator for attribute grammars can be created by applying the function caching interpreter to the functional program constructed by translating an attribute grammar. If edits to syntax trees are performed applicatively, then incremental evaluation simply requires using a function cache to retain computed values across evaluations of the attribute grammar.

Consider an edit to an input tree T in which a subtree S of T is replaced with a new subtree S' to create the tree T'. Pugh argues that applying the function caching interpreter to perform incremental evaluation requires $O(|\text{AFFECTED}| + |\text{PATH}|)$ function applications, where PATH is the set of nodes on the path from the root of T' to the root of the subtree S', inclusive. It can easily be shown that this result can be generalized for a multiple subtree replacement model in which multiple subtrees $S_1, S_2, \ldots, S_n$ of T are simultaneously replaced with the subtrees $S'_1, S'_2, \ldots, S'_n$ to create T'. In this case, the incremental evaluator runs in $O(|\text{AFFECTED}| + |\text{PATHS}|)$ time, where PATHS includes all of the nodes in the new syntax tree T' reaching from the root of T' to the roots of each of the subtrees $S'_1, S'_2, \ldots, S'_n$.

The cost of applying Pugh's evaluator to a hierarchical specification with a modified input term is the sum of the costs of evaluating each of the module instances in AFF-MOD-INSTS, where

AFF-MOD-INSTS is the set of module instances that are invoked by the incremental evaluator during evaluation of the modified input term. Let $TREE(S_{aff})$, for $aff \in$ AFF-MOD-INSTS, be an attribute tree representing the syntax term S_{aff} that is the input to module instance aff. The cost of updating a single module instance $aff \in$ AFF-MOD-INSTS is $O(|\text{AFFECTED}_{aff}| + |\text{PATHS}_{aff}|)$, where AFFECTED_{aff} is the set of attributes associated with nodes in $TREE(S_{aff})$ that must be assigned new attribute values during evaluation of module instance *aff*, and PATHS_{aff} is a set of nodes forming a crown of $TREE(S_{aff})$. We now attempt to provide a basis for comparing this running time to the running time of the context-based evaluator.

The context-based evaluator evaluates attributes associated with nodes in a crown of the new attribute tree and skips visits to those subtrees for which old attribute subtrees can be tentatively selected based on the values of inherited attributes associated with the roots of those subtrees. As discussed above, the context-based evaluator minimizes the size of the crown of nodes traversed by the context-based evaluator during the evaluation of a module instance. Therefore, the size of the crown of nodes traversed by the context-based evaluator during the evaluation of module instance *aff* is $|\text{AFFCROWN}'_{aff}|$.

It is relatively easy to correlate the function evaluations performed by Pugh's evaluator during evaluation of a term d with the nodes of an attribute tree representing d, even though the attribute tree is not actually constructed. Pugh's evaluator can be seen as evaluating attributes associated with a set of nodes in the crown of an attribute tree, skipping visits to attribute tree nodes whenever the function cache can be used to retrieve the value of a previously computed synthesized attribute associated with those nodes. In the notation used above, the crown of the attribute tree traversed by Pugh's evaluator when applied to evaluate a module instance *aff* has size $|\text{PATHS}_{aff}|$.

```
...
N ::= M(S S)
  N.S1 = N.I1 * N.I1
  N.S2 = N.I2 * N.I2
...
```

Figure 11 A Fragment of a Simple Specification

We now present a simple example demonstrating that, in general, the set PATHS_{aff} will be smaller than the set $\text{AFFCROWN}'_{aff}$. Figure 11 presents a single production from a larger specification in which the semantic functions for an operator M define the value of a synthesized attribute *S1* to be the square of the value of inherited attribute *I1*, and the value of a synthesized attribute *S2* to be the square of the value of inherited attribute *I2*. Assume that *AvailInhAttrs(N,1)* contains *N.I1* and *N.I2*, and *AvailSynAttrs(N,1)* contains *N.S1* and *N.S2*.

Figure 12 presents a pair of retained attribute subtrees, *R1* and *R2*, whose roots are both labeled

Figure 12 A Pair of Retained Attribute Subtrees

with operator M, with *R1.dagnode* = *R2.dagnode* = *dagnode*. Now consider what happens when the context-based evaluator attempts to locate an attribute subtree that is appropriate for the first visit to an interface node f with *f.dagnode* = *R1.dagnode* = *dagnode*, and with *I1* = 3 and *I2* = 2. Neither *R1* nor *R2* is appropriate for f, and, therefore, the interface node f must be expanded to create a new attribute tree node. This, of course, is the correct action for the context-based evaluator, since the evaluator will use the retained attribute subtrees that it finds to construct the new consistent attribute tree, and since neither *R1* nor *R2* can be used in the new tree without updating some of its attributes.

The applicative evaluator, however, when applied to this same problem, initially contains a function cache containing the set of tuples:

$$\langle \text{dagnode, “N.S1”, } 1\rangle \rightarrow 1,$$
$$\langle \text{dagnode, “N.S2”, } 2\rangle \rightarrow 4,$$
$$\langle \text{dagnode, “N.S1”, } 3\rangle \rightarrow 9,$$
$$\langle \text{dagnode, “N.S2”, } 4\rangle \rightarrow 25.$$

During evaluation, the cache will be queried to find an entry for ⟨R1.dagnode, “N.S1”, 3⟩ and to then find an entry for ⟨R1.dagnode, “N.S2”, 2⟩. The function cache will return the results 9 and 4, respectively, for these two queries. The function caching algorithm, therefore, is able to skip visits to attribute subtrees that must be visited by the context-based evaluator. Therefore, the set PATHS_{aff} will be smaller than the set $\text{AFFCROWN}'_{aff}$ in this situation. As demonstrated, the function caching approach may use asymptotically less time than the CBOE algorithm to update a module instance.

6.2 Visit Function Caching

Vogt, Swierstra and Kuiper [10] describe an applicative technique for incremental evaluation of Higher Order Attribute Grammars that is quite similar to the way that the CBOE algorithm identifies and reuses previously computed attribute values. Instead of caching individual semantic function invocations as in Pugh's technique described above, entire subtree visits are cached, where visits correspond to the visit operations that occur in a visit-sequence based attribute grammar evaluator. Each visit is translated into a visit function that takes a subtree to be evaluated, a sequence of inherited attribute values needed by that visit to compute a set of synthesized attribute values, and a set of bindings. Bindings, sets of attributes and their values, are introduced as a mechanism for making attributes computed by earlier visits to a subtree available during later visits to that subtree. In the CBOE algorithm, the attributed trees themselves provide access to the values of attributes computed during previous visits to a subtree. Each visit function computes values for a set of attributes at the root of a subtree and constructs a set of bindings for use by later visits. An incremental evaluator for Higher Order Attribute Grammars results from caching visit function invocations.

Vogt, Swierstra and Kuiper state that their evaluator runs in $O(|\textsc{affected}| + |\textsc{paths}|)$ time, where PATHS is defined as above. By constructing bindings carefully, the visit function caching evaluator is able to avoid evaluating attributes that the CBOE algorithm must evaluate, in the same manner as the semantic function caching evaluator.

7 Summary and Conclusions

In this paper, we have presented a new tree-walking incremental evaluation algorithm for hierarchical attribute grammars which we have shown can evaluate entire hierarchical attribute grammar specifications optimally under the assumption that an incremental evaluator for modules of a hierarchical specification must traverse all of the nodes in the new attribute tree on paths from the root of the tree to each attribute tree node having an inconsistent attribute. The key insight is the use of context in choosing old attribute subtrees to reuse by intermingling attribute evaluation with the identification of old attribute values. Although we have observed that an applicative approach based on function caching can "skip visits" to attribute subtrees that the context-based evaluator must visit, the context-based evaluator provides a similar use of context information completely within the realm of tree-walking attribute evaluators, and in fact, is based on a simple modification to a visit-sequence interpreter for non-incremental evaluation of non-hierarchical attribute grammars.

One interesting avenue for future work is an experimental comparison of the context-based,

applicative, and matching-based incremental evaluators for hierarchical attribute grammars.

References

[1] Harald Ganzinger and R. Giegerich. Attribute Coupled Grammars. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170, Montreal, Canada, June 1984.

[2] Harald Ganzinger, Robert Giegerich, and Martin Vach. MARVIN: A tool for applicative and modular compiler specifications. Forschungsbericht Nr. 220, July 1986.

[3] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.

[4] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation*, pages 131–145, June 1989.

[5] D. Swierstra and H. Vogt. Higher order attribute grammars. *Attribute Grammars, Applications and Systems*, pages 256–296, 1991.

[6] Alan Carle. *Hierarchical Attribute Grammars: Dialects, Applications and Evaluation Algorithms.* PhD thesis, Dept. of Computer Science, Rice University, Houston, Texas, May 1992.

[7] Alan Carle and Lori L. Pollock. Matching-based incremental evaluators for hierarchical attribute grammar dialects. Technical Report 94–03, University of Deleware, August 1993.

[8] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, April 1990.

[9] Tim Teitelbaum and Richard Chapman. Higher–order attribute grammars and editing environments. In *Proceedings of the SIGPLAN '90 Symposium on Programming Language Design and Implementation*, pages 197–208, June 1990.

[10] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In J. Maluszyński and M. Wirsing, editors, *5rd International Symposium, PLILP '91*, pages 231–242. Springer-Verlag, July 1991.

[11] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[12] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *Proceedings of the SIGSOFT/SIGPLAN '84 Software Engineering Symposium on Practical Software Development Environments*, April 1984.

[13] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[14] Thomas W. Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, Massachusetts, 1984.

[15] Takuya Katayama. Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, July 1984.

[16] P. Franchi-Zannettacci. *Attributs Sémantiques et Schémas de Programmes.* PhD thesis, Université de Bordeaux I, March 1982.

[17] Mikko Saarinen. On constructing efficient evaluators for attribute grammars. In G. Ausiello and C. Böhm, editors, *5th ICALP*, pages 382–397. Springer-Verlag, July 1978.

[18] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemas I and II. *Theoretical Computer Science*, 17(2):163–191, 235–257, 1982.

[19] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[20] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[21] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the Sixteenth Annual Symposium on Principles of Programming Languages*, pages 315–328, Austin, Texas, January 1989.