Incremental Attribute Evaluation with Side-effects

Görel Hedin

Lund Institute of Technology
Dept. of Computer Science
Box 118, S-221 00 Lund, Sweden
email: gorel@dna.lth.se

1. Introduction

Attribute grammars is a declarative formalism used to express context-sensitive constraints, e.g. the static semantic constraints on a program [Knuth 68]. Incremental attribute evaluators can be derived automatically from the grammar [Yeh 83], [Reps 84]. The incremental evaluator can be interfaced to a language-based editor, in order to give the user continuous feedback on static semantic errors during editing. It would also be desirable to be able to integrate other tools in the programming environment, e.g. code-generators and cross-reference tools, and let them utilize the attribute information.

However, there are drawbacks with the standard form of attribute grammars. One problem is that the incremental evaluation technique does not scale up for large programs. This is because reevaluation of complex attributes, such as symbol tables, triggers reevaluation in more places than is intuitively necessary. E.g. adding a new global procedure triggers reevaluation throughout the program, even if there are no calls to the procedure.

Another problem with attribute grammars is that the constraints are often unnatural to express and auxiliary attributes are needed which are only used to gather and propagate information between nodes. E.g. building up a symbol table must be done by propagating the partly built symbol table around the declaration nodes, possibly in several passes, depending on how the declarations are allowed to depend on each other. It would be more natural to put constraints on the final symbol table directly in terms of each of the declarations.

Several efforts have been made recently to solve these problems. E.g. Hoover and Reps have proposed mechanisms for incremental updating of aggregate attributes, and remote access to attributes [Hoover 86], [Hoover et al. 86], [Reps et al. 86]. Johnson & Fischer have proposed allowing non-local dependencies between attributes [Johnson et al. 82], [Johnson et al. 85]. Beshers & Campbell have proposed usage of maintained attributes which can be incrementally modified by automatically reversible operations during attribute evaluation [Beshers et al. 85]. Demers, Rogers & Zadeck have proposed using a scheme based on message-passing instead of attribute evaluation [Demers et al. 85]. Horwitz & Teitelbaum have proposed a combination of relations and attributes [Horwitz et al. 86].

In this paper we describe the technique used to perform incremental semantic analysis in the programming environment Mjølner [Dahle et al. 87]. Mjølner is a grammar-driven environment, supporting programming in statically typed object-oriented languages in the Simula tradition [SIMULA-67 68], [SIMULA 87]. Typed object-oriented languages have more complex scope rules than ordinary block structured languages such as Algol and Pascal. This is because of the type hierarchy introduced through inheritance. The technique used in Mjølner is based on attribute grammars, but also allows constraints to be expressed, which have side-effects when evaluated. The side-effects are automatically reversed when needed during incremental evaluation. The technique is thus similar to the one proposed by Beshers & Campbell. Mjølner includes a package for handling complex combinations of scope rules, allowing the scope rules

of object-oriented languages to be expressed in an elegant way. The constraint facility is utilized to perform incremental updates and lookups in symbol tables, without having to copy information throughout the tree, as is normally done in attribute grammars. Since many non-local dependencies are handled directly rather than via the tree, the attribute dependencies become very simple, making it possible to use a simple and fast evaluator. Currently an ordered one-pass attribute evaluator is used in Mjølner. The evaluation technique has also been combined with demand attributes, and only very few attributes need to actually be stored in the tree. Default rules are elegantly expressed using an object-oriented grammar notation.

The rest of this paper is organized as follows: Section 2 describes the scope rules of object-oriented languages. Section 3 describes how scope-rules and symbol tables are represented in Mjølner. Section 4 describes the object-oriented grammar formalism. Section 5 describes the technique for incremental evaluation. Section 6 compares our technique to other proposed techniques. Section 7 concludes the paper.

2. Scope rules in object-oriented languages

In object-oriented languages in the Simula tradition, the normal scope rules for block structure are combined with scope rules for inheritance. In addition, via remote access ("dot-notation") and Simula's "inspect"-statement (similar to the "with"-statement of Pascal), attributes of an object can be accessed from the outside. This complexity of scope rules is not unique to Simula. Many new object-oriented languages combine block structure with inheritance, e.g. Beta [Kristensen et al. 87], Eiffel [Meyer 87], C++ [Stroustrup 86], and Trellis-Owl [Schaffert et al. 86]. There are also examples of procedural languages which use inheritance for records, e.g. PLANC [PLANC 79] and Oberon [Wirth 88]. Although inheritance is here restricted to data, such languages have scope rules similar to object-oriented languages.

2.1 Scope rules for Simula

In this section we briefly review the scope rules for Simula, and discuss the consequences of their combination.

Block structure

With *block* we mean a program construction which contains a local set of declarations. Simula has three kinds of blocks: *classes*, *procedures*, and *Algol block-statements*. In the Beta language, these constructions have been unified into one kind of block: the *pattern*. Both Simula and Beta allow full block structure, i.e. blocks may be nested to any level. Most other object-oriented languages have a simpler block structure allowing only three levels: Procedures nested inside classes, and classes nested inside a program module.

With *block-structured scope rules*, we mean the normal scope rules of Algol [Naur 63]. I.e. the scope of a declared identifier is the block it is declared in, and all inner blocks. Conversely the environment of a free identifier can be described as a search path, starting at the block in which the identifier occurs, and continuing in successively outer blocks.

Inheritance

In Simula, a class can be declared as a subclass of another class. The subclass will then inherit all the attributes of its superclass. The scope of a declaration in a class is thus the class itself, and all subclasses. In combination with block structure, we also have to add the inner blocks of the class and its subclasses. The environment of a free identifier now becomes more complex: It can be described as a search path, starting in the block, *B*, of the occurrence, continuing in the superblocks of *B*, then continuing in the enclosing block *B_f* of *B*, then continuing in the superblocks of *B_f*, and so on. E.g. in the figure below, the search path for free identifiers in class *Rectangle*'s procedure *Draw*, will consist of the following blocks: *Draw*, *Rectangle*, *Element*, *LINK*, *LINKAGE*, *GraphicPackage*, *SIMSET*, and *Main program*.

Figure 1. A block structured object-oriented program. The solid lines indicate block nesting. The dashed lines indicate subclass relationship. E.g. GraphicPackage is a subclass of SIMSET.

Remote-access

A declaration in a class may be accessed remotely, using "dot-notation". In Simula, this is used to call procedures in objects ("message-passing" in Smalltalk terminology) and to access data in objects. The environment of a remotely accessed declaration is a search path along the inheritance chain of the accessed object, starting at the static qualification of the object (the declared class of the object). E.g. in remote access to an object of class Rectangle, the search path will consist of: Rectangle, Element, LINK, and LINKAGE.

Inspect

Simula's inspect-statement "opens" an object, so the declared attributes of the object can be referenced by free identifiers. The search path for free identifiers inside an inspect statement is the inheritance chain of the inspected object, followed by the search path of the environment outside the inspect statement. E.g. if a Rectangle object is inspected in the procedure BuildWindow in figure 1, the search path will consist of the following blocks: Rectangle, Element, LINK, LINKAGE, BuildWindow, BuildHouse, GraphicPackage, SIMSET, and Main program.

It may of course be debated if the inspect statement is in agreement with object-oriented programming or not. Nevertheless, the inspect statement is widely used, and it is interesting to be able to handle it in an incremental environment.

2.2 Finding affected use sites

A problem in incremental semantic analysis is to quickly find affected identifier use sites, *IdUses*, after changes to declarations. With *affected IdUses* we mean those identifiers which refer to the added/removed/changed declaration either before or after the change. It is desirable that the affected IdUses are found in linear time, i.e. in time proportional to the number of affected IdUses. In the case of changing or removing a declaration, the problem is easily solved by maintaining cross-reference information at each of the declarations, e.g. a list of references to IdUses. However, when a new declaration is added, the problem is harder. There may exist IdUses which previously either were undeclared or referred to some other declaration, but will now refer to the new declaration.

Finding affected IdUses by checking traces

In Mjølner, a general method is used for finding affected IdUses after adding a declaration, by keeping track of *traces* when binding IdUses. This means that when we look for the declaration of an IdUse, the blocks are searched in the order prescribed by the environment of the IdUse, and in each block where the declaration is not found, a trace is left. A trace in a block thus means that there exists an IdUse which would rather refer to a declaration in that block than to whatever declaration it currently refers to. When a new declaration is added, the traces are checked to see if there are any IdUses which should be rebound, and information in the traces is used for efficient location of these IdUses.

Finding affected IdUses by searching environment

Another method for finding affected uses after adding a declaration is to search the environment of the declaration in order to find other declarations which will be hidden by the new declaration. This method is used e.g. in [Johnson et al. 82]. However, this method works only in simple cases, when the environment of an affected IdUse is a pure extension of the declaration's environment, e.g. in the case of pure block structure or pure inheritance. When block structure and inheritance is combined, or when environments are combined using the inspect-statement, this method fails.

3. Representation of symbol tables and scope rules

In a standard attribute grammar, symbol tables are built by propagating partly built symbol tables along the declarations and letting each declaration define a new symbol table value by concatenating its own information with an incoming symbol table. In the block bodies, the symbol tables are combined according to the scope rules of the language, and the appropriate combination of symbol tables is propagated to each IdUse.

We find the standard attribute-grammar approach unnatural, and it is also hard to implement efficiently. In Mjølner, each block has one symbol table attribute which is updated incrementally as the declaration information in the block is changed. Rather than propagating symbol tables to the IdUses, the appropriate symbol table *search path* is propagated to each IdUse.

The symbol table of a block contains an entry for each of the declared identifiers in the block. An entry contains information about the name, type, kind etc. of the declaration. Components of entries may contain references to other entries or to whole symbol tables. E.g. if the type of a variable is structured, the variable entry will contain a reference to the entry of the type. If a declaration is itself a block, the entry will contain a reference to the symbol table of that block. Each declaration has a binding-attribute which refers to the corresponding symbol-table entry. The declaration uses its binding-attribute to specify the component values of the entry. Each IdUse also has a binding-attribute which refers to the appropriate symbol-table entry, according to its search path. The IdUse can use the binding-attribute to look up information about its declaration, e.g. type and kind. Entries and components of entries are added, deleted, or changed as a consequence of changes to the declarations.

The scope-rules are represented by *search paths*. A search path is a list of references to symbol tables and other search paths. Each block has a number of search paths which define different kinds of accesses to the declarations of the block. E.g. local access, access from local blocks, access from subclasses, and remote access. In the syntax tree, these search paths are used to represent the environments of IdUses. Search paths of different blocks can be looked up and combined in order to form the appropriate environment for IdUses in different contexts. E.g. for Simula, the following search paths are used (see also figure 2):

The *static search path* defines the permitted access from local blocks. A Simula procedure has a static search path consisting of two elements: The first refers to the procedure's symbol table, and the second refers to the enclosing block's static search path. A Simula class has a three-

element static search path: the class' symbol table, followed by the superclass' subclass search path, followed by the enclosing block's static search path.

The *local search path* defines the environment for free IdUses in the block. In Simula (and also in most other languages), the local search path coincides with the static search path.

The *subclass search path* of a class defines the permitted access from subclasses.

The *remote search path* of a class defines permitted remote access.

The subclass search path and the remote search path have the same structure. The reason for distinguishing between them is that Simula allows declarations to be "protected" or "hidden" from subclass and remote access respectively. It is e.g. often the case that local variables in a class should be hidden from remote access, but should be accessible from the subclasses. By associating filters with the search paths, individual declarations can be filtered out from a specific search path. Figure 2 gives a simplified view however, and does not distinguish between these search paths.

Figure 2. Search paths for some of the blocks in the program shown in figure 1.

4. Grammar formalism

In a standard attribute grammar, all attributes have functional values, defined by functional constraints. An attribution is said to be *valid* when all the constraints are satisfied. The above described symbol-table structure cannot be defined by functional constraints since it contains circular references both within and between individual symbol tables. Therefore, in addition to the functional constraints, we introduce *operational* constraints. In contrast to the functional constraints, the operational constraints may have side-effects when evaluated. Nevertheless, the formalism is still declarative, and we can use the same definition of a valid attribution.

In this section we describe the grammar formalism and give examples describing some aspects of the static semantics of Simula.

4.1 An object-oriented grammar notation

The grammar notation is object-oriented in that it supports abstract production specifications and specializations of these productions. The nodes follow the production description in a similar way as objects in an object-oriented language follow their class-descriptions.

The formalism supports four different kinds of productions: constructions, lists, lexemes, and abstract productions. Each production has a name, and defines a syntax node type. We will use the terms "production" and "node type" synonymously. Productions specify the node structure, the attributes of the node, and the constraints on those attributes. Construction productions specify that their nodes will have a specific number of son-nodes with certain names and types. List productions specify that their nodes can have an arbitrary number of sons of a specific type. Lexeme productions are used to model syntactic data such as identifiers and numerical constants. Each lexeme node has an attribute "string" which contains this data. Abstract productions model "un-expanded" nodes of a general type, e.g. statements, declarations, expressions. Any production may be a specialization of an abstract production. Specialized productions augment the definition of their super production. A constraint defining a particular attribute, will override constraints in super-productions defining the same attribute. The following notation is used for the node structure:

```
<cons-prod>:<super-prod> ::= {<son1:son1Prod>&<son2:son2Prod>...}
<list-prod>:<super-prod> ::= <son:sonProd>*
<lexeme-prod>:<super-prod> ::= LEXEME
<abstract-prod>:<super-prod> ::= ABSTRACT
```

Attributes may be declared as synthesized, inherited, or local, with the usual meanings. An attribute defined by a functional constraint may also be specified "demand". This means that the attribute is not stored in the syntax tree, but instead evaluated each time it is accessed.

A kernel of pre-defined data types with associated operational constraints and functions are provided by the system. The implementation of the operational constraints is currently hand-coded, and new operational constraints cannot be defined within the grammar. Because of this, only a limited class of languages can be supported by the system in its present state.

4.2 Example: Variable declarations

The following example shows operational constraints that control a symbol-table entry for a variable declaration. Words in boldface are keywords in the grammar formalism. Words in italics are syntactic sugar for operational constraints. Two minus-signs at the end of a line indicates that the rest of the line is a comment.

```
-----
Production <Node> ::= ABSTRACT
Attributes
  Inh path : ref (SearchPath) demand
Constraints
  for all sons (x)
    Son(x) qua Node.path = path
-----
Production <Decl>:<Node> ::= ABSTRACT
Attributes
  Inh symTab : ref (SymbolTable) demand
  Loc declBinding : ref (STentry)
  [declBinding] REPRESENTS THIS DECLARATION IN [symTab] -- 1
-----
Production <VarDecl>:<Decl> ::= {<varId:IdDecl>&<varType:Type>}
Attributes
  Loc error : ErrorMarking
Constraints
  THE NAME OF [declBinding] IS [varId.string] -- 2
  THE KIND OF [declBinding] IS [VarKind] -- 3
  THE TYPE OF [declBinding] IS [varType.typeValue] -- 4
  [error] MARKS THIS NODE WITH ["Double declared"] -- 5
```

```

      WHEN [DoubleDeclared(declBinding)]

-----
Production   <IdDecl> ::= LEXEME
-----
Production   <Type>:<Node> ::= ABSTRACT
Attributes
  Syn   typeValue : TYPE demand
Constraints
  typeValue = BottomType
-----
Production   <IntegerType>:Type ::= {}
Constraints
  typeValue = IntegerType
-----
Production   <RefType>:Type ::= {<classId:IdUse>}
Constraints
  typeValue = MakeRefType(classId.UseBinding)
-----

```

The production Node is an abstract production whose only purpose is to establish a default rule for propagation of SearchPaths. I.e., the default is that a specialization of Node propagates its own search path to all of its children of production Node. See sections 4.3 and 4.4 for the usage of search paths.

Constraint 1 in production Decl expresses that the attribute declBinding in Decl refers to the declaration's entry in the symbol table symTab. VarDecl is a specialization of Decl, and constraints 2, 3, and 4 define the name, kind, and type components of the entry referred to by declBinding. The name depends on the identifier name in the IdDecl node. The kind is a constant, VarKind. The type of the variable depends on what specialization of production Type the son-node varType is. If varType is "unexpanded", i.e. of production Type, then typeValue is "BottomType", a general type compatible with everything. This default value is overridden in the specialized productions IntegerType and RefType. Constraint 5 expresses that the VarDecl should be marked on the screen, if there already exists another declaration with the same name. A message associated with the error-marking makes it possible for the user to interactively ask for an explanation of the error.

4.3 Example: Expressions and Identifier Uses

Identifier uses appear in statements, expressions, and also in some declarations (e.g. in the production RefType above). An identifier use is bound to the appropriate symbol-table entry by the operational constraint 1 in production IdUse as follows:

```

-----
Production   <Expr>:<Node> ::= ABSTRACT
Attributes
  Syn   exprType : TYPE demand
  Syn   exprQual : TYPE demand
Constraints
  exprType = BottomType
  exprQual = BottomClassType
-----
Production   <IdUse>:<Expr> ::= LEXEME
Attributes
  Syn   useBinding : ref (STentry)
  Loc   error : ref (ErrorMarking)
Constraints
  [useBinding] IS THE DECLARATION OF [string]           -- 1
    FOUND ON THE PATH [path]
  exprType = GetType(useBinding)                         -- 2
  exprQual =                                             -- 3
    if exprType.InLattice(BottomRefType, TopRefType)
    then exprType.Qualification
    else BottomClassType
  [error] MARKS THIS NODE WITH ["Not declared"]         -- 4
    WHEN [useBinding = NullEntry]

```

Constraint 1 expresses that useBinding will refer to the first symbol-table entry with the name "string" which is found when following the search path "path". If no such entry is found, the useBinding will refer to the constant "NullEntry". This is used in constraint 4 to decide if the IdUse node should be marked with an error or not. Constraints 2 and 3 define the exprType and exprQual attributes of the IdUse. These attributes are declared in IdUse's superproduction (Expr), and are used for type-checking. ExprQual is only relevant when exprType is a reference type, and will then be the appropriate class type. Types are arranged in a lattice in the following style:

Figure 3. A type lattice.

BottomType is the least specific type. E.g. undeclared identifiers will have the type BottomType. The higher up in the lattice, the more specific is the type. Lattice operations such as InLattice, Above, Below, Meet, and Join, make it easy to specify type-checking.

4.4 Example: Overriding default search paths

Remote access (dot-notation) and inspect statements are examples when the default search paths are overridden. This is specified as follows:

```
-----
Production  <Dot>:<Expr> ::= {<receiver:Expr>&<selector:IdUse>}
Attributes
  Loc  error : ref (ErrorMarking)
Constraints
  selector.path =
    if receiver.exprType.InLattice(BottomRefType, TopRefType)
    then RemoteSearchPath(receiver.exprQual)
    else NullSearchPath
  exprType = selector.exprType
  exprQual = selector.exprQual
  [error] MARKS THIS NODE WITH ["Prefix not defined"]
  WHEN [not (receiver.exprType.Below(TopRefType))]
-----

Production  <InspectStmt>:<Stmt> ::= {<obj:Expr>&<do:Stmt>}
Attributes
  Loc  error : ref (ErrorMarking)
Constraints
  do.path =
    if obj.exprType.InLattice(BottomRefType, TopRefType)
    then Concatenate(RemoteSearchPath(obj.exprQual), path)
    else path
  [error] MARKS THIS NODE WITH ["Prefix not defined"]
  WHEN [not (obj.exprType.Below(TopRefType))]
-----
```


Remote access in Simula (receiver.selector) is analogous to message-passing in Smalltalk [Goldberg et al. 83]. The "receiver" in production Dot thus represents the remotely accessed object, and "selector" represents the name of the accessed entity. The search path of the selector is the remote search path of the class of the remotely accessed object. This is expressed by constraint 1. In case the receiver is not a reference to an object, e.g. if it is an integer variable, the Dot-node will be marked on the screen, as defined in constraint 4. Constraints 2 and 3 express that the whole dot-expression has the same type as the selector.

The first constraint in production InspectStmt specifies that the search path of its interior statements (the do-part) is the concatenation of the remote search path of the inspected object and the search path of the inspect-statement itself.

4.5 Demand attributes

The high consumption of space is a well-known problem for attribute grammars, and the problem is even worse for incremental systems where the attribute values have to be saved in order to allow incremental update. Demand attributes have been used extensively in the examples above. Since demand attributes are evaluated each time they are accessed, no space needs to be reserved for them in the nodes. In combination with operational constraints, this allows the total space consumption for attributes to be kept very low. E.g. in the examples above, the only attributes which are actually stored, are one symbol table for each block, a binding for each declaration and for each IdUse, and error-marking attributes.

Using demand attributes instead of stored attributes need not slow down the evaluation. In many cases, an attribute is never accessed at all. E.g. the path attribute is propagated to all expressions. However, numeric constants (which are specializations of expressions) will not utilize the path information. If the path attribute was stored, this would take extra time both for evaluating it and for keeping it up to date.

The use of demand attributes thus makes it possible to define attributes which are used only in certain syntactic combinations, without penalizing the performance in other contexts. This is of course of most value when the attribute is used only in unusual syntactic combinations. E.g. test of assignability can be modelled as follows:

```
-----
Production   <AssignStmt>:<Stmt>::= {<to:Expr> & <from:Expr>}
Attributes
  Loc   error : ref (ErrorMarking)
Constraints
  [error] MARKS THIS NODE WITH ["Left of := cannot be assigned"]
          WHEN [not (to.assignable)]
-----
```

The left hand side of the assignment is here modelled as a general expression. However, the attribute "assignable" is only accessed in expressions on the left hand side of an assignment. Most expressions will not be in this context, and their "assignable" attribute will never be evaluated. Furthermore, most specializations of Expr are not assignable, e.g. all arithmetic, boolean, and relational expressions. By using default constraints, these productions need not include a constraint for "assignable".

Demand attributes may cause more reevaluation in an incremental system because old attribute values cannot be compared with new values in order to stop the evaluation. However, the time it takes to compare the attribute values is not necessarily less than what is won by stopping the evaluation.

Synthesized demand attributes are analogous to virtual procedures in object-oriented languages in that the "calling" node knows the general type of its son-node, but it does not know the actual type. To implement efficient access to demand attributes, the function defining the demand attribute must be located quickly. This is done by including a table of demand attribute functions in each production. In all specializations of a production, the function defining a particular

demand attribute is located at the same offset. Accesses can then be done efficiently by indexing into the table. This technique is similar to the one used to implement virtual procedures in e.g. Simula and Beta.

For inherited demand attributes, the situation is not quite as simple, since a production says nothing about the father node's type. E.g. an IdUse-node may be the son of a Dot-node, an Add-node or a RefType-node. In our system we have solved this by actually including a constraint on the father node type of each production. This does not put any practical constraints on what kinds of grammars can be defined, since the inherited demand attributes can always be declared in the most general production class, which all productions are specializations of. Because a production now knows the general type of the father node, the functions defining inherited demand attributes can be looked up using the same technique as for synthesized demand attributes.

The inherited demand attribute "path" is e.g. actually specified as follows:

```
-----
Production <SuperNode> ::= ABSTRACT
Attributes
  Exp path
Constraints
  for all sons(x)
    son(x) qua Node.path = NullSearchPath
-----
Production <Node>:<SuperNode> son of <SuperNode> ::= ABSTRACT
Attributes
  Inh path : ref(SearchPath) demand
Constraints
  for all sons(x)
    son(x) qua Node.path = path
-----
```

SuperNode defines the behavior of being able to "export" the demand inherited attribute "path" to its sons of type Node. It also defines a default value for this attribute. Typically, the root node of a syntax tree is a specialization of SuperNode.

Node is the general production for nodes which have an inherited demand attribute "path". It also specializes the behavior of SuperNode by overriding the value it propagates to its sons. Typically, all nodes in the syntax tree, except for the root node, are specializations of Node.

5. Incremental Evaluation

The incremental attribute evaluator is activated after each of the user's editing actions. The evaluator is similar to an incremental evaluator for a standard attribute grammar, following evaluation orders derived from the attribute dependencies. The main differences are that 1) the side-effects introduced when evaluating an operational constraint have to be undone properly when needed, and 2) the attribute evaluation can spread to distant locations in the syntax tree during evaluation of an operational constraint.

5.1 Undoing side-effects

Each operational constraint has two associated operations: one *evaluation* operation for establishing the constraint, and one *de-evaluation* operation for cancelling the side-effects introduced by the previous evaluation operation. E.g., when evaluating the constraint defining the useBinding attribute, trace-information will be inserted into the symbol tables as described in section 3. When the constraint is de-evaluated, that trace-information is removed. This makes it possible to *re-evaluate* an operational constraint simply by de-evaluating it and then evaluating it. For efficiency reasons one may also define a re-evaluation operation which has the same net effect as a de-evaluation followed by an evaluation. It is also necessary that the (de-)evaluation of one constraint does not interfere with the (de-)evaluation of other operational constraints. I.e. the order of (de-)evaluation must only depend on the attribute dependencies in the grammar.

These properties make it possible to let the incremental evaluator treat operational and functional constraints in the same way. The evaluator can evaluate, re-evaluate, and de-evaluate a constraint. For functional constraints, re-evaluation is the same thing as evaluation, and de-evaluation of functional constraint is an empty operation.

In order to do a proper de-evaluation of an attribute defined by an operational constraint, one may need access to the arguments used at the previous evaluation. E.g. when de-evaluating the `useBinding` attribute, the old search path is needed in order to find the appropriate symbol tables where trace-information should be removed. Thus although it is not directly visible in the grammar, the old search path is actually stored in the node together with the `useBinding` attribute.

There are four basic editing actions: *subtree replacement*, when an old subtree is removed from the syntax tree and replaced by a new subtree. *Subtree insertion*, when a new subtree is inserted as a son to a list node. *Subtree deletion*, when a son of a list node is removed. Finally, *lexeme-string replacement*, when the string attribute of a lexeme node is replaced.

After a subtree replacement, the attribute evaluator goes through three phases: In the first phase, *monolithic de-evaluation*, all attributes of the old subtree are de-evaluated to remove side-effects. New subtrees are considered unattributed, so in the second phase, *monolithic evaluation*, the inherited attributes in the root of the new subtree are given values, and all attributes in the new subtree are evaluated. The third phase, *incremental re-evaluation*, re-evaluates all attributes dependent on the synthesized attributes in the root of the new subtree.

The subtree insertion/deletion actions can be seen as special cases of subtree replacement, with one of the monolithic phases missing. The lexeme-string replacement action will trigger incremental re-evaluation of attributes dependent on the string attribute.

5.2 Propagation to distant nodes

During the (de-)evaluation of operational constraints which add, remove, or change symbol-table entries, affected `IdUses` are located as described in section 3 and incremental re-evaluation is started in these nodes. In general, such re-evaluations must be queued until ongoing evaluations are completed. This is in order to avoid conflicts by incrementally re-evaluating constraints whose arguments are not yet initialized, or are no longer relevant.

Our system supports incremental evaluation of L-attributed grammars [Lewis et al. 74], but because of the possibility to incrementally update symbol tables, and to propagate evaluation to distant nodes, a larger class of languages can be handled, than is normally the case with L-attributed grammars. E.g. in Simula, a declaration may depend on any other declaration in the same block. In a standard attribute grammar for Simula, the declarations in the block need to be traversed in three passes: one pass for gathering information about the class hierarchy, a second pass for gathering information about individual classes, and a third pass for propagating the information to the variables. In our system, only one pass is needed. However, when evaluating one declaration, all information may not yet be available in the symbol table. Through the propagation to distant nodes, the evaluation will then return to that declaration when the needed information is added to the symbol table. E.g. suppose the following two declarations are inserted at once:

```
ref (A) rA;
class A;
begin
end;
```

The declaration for the variable `rA` will be added first to the symbol table. Since `A` is not yet in the symbol table, the type component of `rA` will be set to a reference to an unknown type. When the class `A` is inserted into the symbol table, evaluation will propagate back to the `rA` variable declaration and the type of `rA` will be re-evaluated. This evaluation scheme is clearly sub-optimal, but in an incremental system where the user edits small increments at a time, adding

several declarations at once is probably very unusual. Also, the sub-optimal behavior will only occur when the declarations are in the "wrong" order, with uses before declarations.

5.3 Circular dependencies

Because of the possibility in operational constraints to propagate the evaluation to distant nodes, there are attribute dependencies which are not visible in the grammar. In fact, these dependencies may give rise to circularities. However, rather than to prohibit circular dependencies, we allow them to exist, and instead break the circular evaluation by other means. One example of circular dependencies is the following:

```
ref (B) A;
ref (A) B;
```

This example is an erroneous program fragment since A and B are variables, and the qualification of a reference variable must be a class. If no special precaution was taken in this case, the evaluation would continue for ever as follows:

```
1. add A-entry
2. set type of A-entry to ref(BottomType) since B is not in symbol table
3. add B-entry
4. set type of B-entry to ref(A)
5. since the type of B is changed, reevaluate IdUses referring to B
6. reevaluate the type of A (set it to ref (B))
7. since the type of A is changed, reevaluate IdUses referring to A
8. reevaluate the type of B (set it to ref (A))
9. same as 5
10.same as 6
...
```

This circularity was broken by changing the evaluation operation for setting the type of a symbol-table entry. A check was added to see if the new type was the same as the previous one. In that case, the re-evaluation of distant nodes was stopped.

Another example of a circularity is when the inheritance chain is circular (this is also an illegal program fragment), e.g. as follows:

```
A class B;
begin
end;

B class A;
begin
end;
```

If the symbol table would reflect this circular structure, this would lead to loops in the search paths, and endless searching when binding IdUses. To avoid this, a check was inserted into the constraint defining the superclass of a class declaration as follows:

```
-----
Production <ClassDecl>:<Decl> ::=
    {<superId:IdUse> & <classId:IdDecl> &...}
Attributes
Constraints
    THE SUPERCLASS OF [declBinding] IS
    [if superId.exprType.Above(GetType(declBinding))
     then BottomClassType
     else superId.exprType]
-----
```

If the intended super-class is already a subclass to the ClassDecl (subclasses are above their superclasses in the type lattice), the superclass of ClassDecl's entry will instead be set to BottomClassType, the most general class type.

6. Related Work

In related work on making incremental evaluators more useful in practice, all examples are given for very simple block-structured languages. It is unclear if the proposed techniques are sufficiently efficient for object-oriented languages.

Hoover [Hoover et al. 86] and Reps [Reps et al. 86] introduce aggregate valued attributes which can be incrementally updated and are thus suitable to represent symbol tables. New symbol tables can be constructed from other symbol tables in order to represent concatenated scopes in a space efficient way, and usage information is maintained in each symbol table about attributes depending on it as a whole, or on specific elements in it. This makes it possible to do selective propagation to use sites, so only those use sites which are actually affected by the change will be visited. However, when changing a symbol table, all dependent symbol tables have to be updated. This may prove to be very expensive for object-oriented languages. E.g. if the environment of each selector in each remote access is represented by an individual symbol table, adding a new procedure in a class would cause updating of the symbol tables of all remote accesses to that class. Another problem is that the proposed aggregate values may not be part of other attributes [Hoover 87]. This will probably make it hard to combine symbol tables to represent e.g. inheritance.

Demers et al. propose a declarative message-passing formalism as an alternative to attribute grammars [Demers et al. 85]. It allows a natural modelling of information similar to ours, in that symbol table information is maintained locally in the block and not propagated to all use sites. Identifier use sites send messages to the symbol tables to find out typing information. However, messages can only be propagated according to the syntax tree structure. This would probably make it hard to propagate messages along an inheritance search path as would be needed for an object-oriented language.

The technique we use of augmenting attribute grammars with constraints which have side-effects when evaluated, is very similar to the work of Beshers and Campbell [Beshers et al. 85]. In their terminology, our symbol tables would be classified as "maintained attributes", and our operational constraints as "constructor attributes".

Our use of demand attributes solve similar problems as do the copy rules of Hoover [Hoover 86] and the remote attributes of Reps [Reps et al. 86]. However, due to our object-oriented notation, identity constraints need not be repeated in all intervening nodes as in the copy-rules of Hoover, and son-nodes do not need to know the exact type of their remote father node as in Reps. Also, the demand attributes are more general than copy rules and remotely accessed attributes, since they may refer to information which is not explicitly stored in other attributes. The Cornell Synthesizer Generator also includes a mechanism for demand evaluation of attributes, but it differs from our demand attributes in that the attributes are stored when they have been evaluated the first time [Reps et al. 85].

Our use of default constraints for general productions also solve similar problems as the default message propagation orders of Demers et al. [Demers et al. 85], in that information can be propagated via nodes without having to add propagation rules in each individual node type. However, their approach is different in that they associate the propagation order with the messages (the attributes) rather than with the node types.

7. Conclusion

We have demonstrated how an attribute grammar, augmented with operational constraints, can be used to specify the static semantics of an object-oriented language. The operational constraints have side-effects which make it possible to incrementally update attributes. This makes it possible to represent symbol tables in a natural and efficient way. A tracing technique is used to efficiently locate all affected identifier use sites after changes to symbol tables. The technique is general enough to handle the various combinations of scope rules which appear in object-oriented languages.

A mechanism for demand attributes is used, which together with the operational constraints allows the total space consumption for attributes to be kept very low. This is extremely important in order to be able to represent large programs. An object-oriented notation has been introduced which allows default definition of constraints. This gives a high degree of flexibility in the grammar specification, and is particularly useful in combination with demand attributes.

Acknowledgements

This work has been carried out within the Mjølner project. I would like to thank the other participants in this project, especially Lars-Ove Dahlin who has designed and implemented the search path mechanism, and with whom I have had many constructive discussions.

The Mjølner project is partially financed by the Nordic Fund for Technology and Industrial Development (Nordisk Industrifond) and by the Swedish national board for Technical Development (STU).

References

- [Beshers et al. 85] Beshers G., Campbell R.: *Maintained and Constructor Attributes*. In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments. Seattle, June 1985.
- [Dahle et al. 87] Dahle, H.P., Löfgren, M., Madsen, O.L., Magnusson, B., *The MJØLNER Project*, Proceeding of the Conference held at Software Tools, Online Publications, London, 1987.
- [Demers et al. 85] Demers A., Rogers A., Zadeck F.K.: *Attribute Propagation by Message Passing*. In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments. Seattle, June 1985.
- [Goldberg et al. 83] Goldberg A., Robson D.: *Smalltalk 80. The language and its implementation*. Addison-Wesley 1983.
- [Hoover 86] Hoover R.: *Dynamically bypassing copy rule chains in attribute grammars*. In Conference Record of the 13th Symposium on Principles of Programming Languages, St. Petersburg, Fla. Januray 1986.
- [Hoover et al. 86] Hoover R., Teitelbaum T.: *Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars*. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, June 1986.
- [Hoover 87] Hoover R.: *Incremental Graph Evaluation*. PhD Thesis. Cornell University, Dept. of Computer Science, May 1987.
- [Horwitz et al. 86] Horwitz S., Teitelbaum T.: *Generating Editing Environments Based on Relations and Attributes*. ACM TOPLAS, Vol 8, No 4, October 1986. Pages 577-608.
- [Johnson et al. 82] Johnson G. F., Fischer C. N.: *Non-syntactic attribute flow in language based editors*. In Conference Record of the 9th ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., January 1982. 185-195.
- [Johnson et al. 85] Johnson G. F., Fischer C. N.: *A meta-language and system for nonlocal incremental attribute evaluation in language-based editors*. In Conference Record of the 12th ACM Symposium on Principles of Programming Languages, New Orleans, La. January 1985. 141-151.
- [Knuth 68] Knuth D. E.: *Semantics of context-free languages*. Math. Syst. Theory 2:2, June 1968, 127-145.
- [Kristensen et al. 87] Kristensen B. B., Madsen O.L., Møller-Pedersen B., Nygaard K.: *The BETA Programming Language*. In Shriver B., Wegner P. (eds.) *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.
- [Lewis et al. 74] Lewis P.M., Rosenkrantz D.J., Stearns R.E.: *Attributed translations*. J. Computer and Systems Sciences 9:3, 279-307.
- [Meyer 87] Meyer B.: *Eiffel: Programming for Reusability and Extendibility*. SIGPLAN Notices, Vol. 22, No. 2, Feb 1987.

- [Naur 63] Naur P.: *Revised report on the algorithmic language ALGOL 60*. Comm. ACM 6:1, 1-17.
- [PLANC 79] PLANC Reference Manual. Norsk Data A.S. Oslo, Norway, 1979.
- [Reps 84] Reps T.: *Generating Language-Based Environments*. The MIT Press, 1984.
- [Reps et al. 85] Reps T., Teitelbaum T.: *The Synthesizer Generator. Reference Manual*. Cornell University, Dept. of Computer Science. 1985 (revised 1987).
- [Reps et al. 86] Reps T., Marceau C., Teitelbaum T.: *Remote attribute updating for language-based editors*. In Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1986. pp 1-13.
- [Schaffert et al. 86] Schaffert C., Cooper T., Bullis B., Kilian M., Wilpolt C.: *An Introduction to Trellis/Owl*. In *Proceedings of OOPSLA '86*. The 1986 Conference on *Object-Oriented Programming Systems, Languages and Applications*. SIGPLAN Notices. Vol 21 No 11, Nov 1986.
- [SIMULA 87] Data Processing - Programming Languages - *SIMULA*. Swedish Standard SS 63 61 14. SIS. Stockholm, Sweden, June 1987.
- [SIMULA-67 68] Dahl, O.-J., Nygaard, K.: *SIMULA 67 Common Base Language*. Norwegian Computing Center, Oslo, 1968. Revised 1970, 1972, and 1984.
- [Stroustrup 86] Stroustrup B.: *The C++ Programming Language*. Addison-Wesley, 1986.
- [Wirth 88] Wirth N.: From Modula to Oberon. Tech. Rep. ETH, Zurich, 1988.
- [Yeh 83] Yeh D.: *On Incremental Evaluation of Ordered Attribute Grammars*. BIT 23:3, 1983.