Incremental Static Semantic Analysis for Object-Oriented Languages Using Door Attribute Grammars

Görel Hedin

LU-CS-TR:91-79

Also published in: Proceedings of the International Summer School on Attribute Grammars, Applications, and Systems, Prague, vol. 545 of Lecture Notes in Computer Science, 1991

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Incremental Static Semantic Analysis for Object-Oriented Languages using Door Attribute Grammars

Görel Hedin

Lund University, Department of Computer Science
P.O. Box 118, S-221 00 Lund, Sweden.
e-mail: Gorel.Hedin@dna.lth.se

EXTENDED ABSTRACT

1. Introduction

Interactive programming environments in the tradition of the Cornell Program Synthesizer [TeR81] represent static semantic information as an attribution of the program syntax tree. The attribution is derived from the syntax tree, and is incrementally maintained as the user edits the program. The static semantic information may serve many purposes in the environment; as a base for incremental code generation, to give the user feedback on compile-time errors, and to provide advanced interactive support, e.g. context-sensitive editing and browsing. The design of the attribution is extremely important in order to obtain a system which works well in practice. It should be structured in a simple and natural way, not consume excessive storage, and it must allow efficient incremental updates. To obtain these goals we suggest the use of a *semantic object structure* as part of the attribution. This means attaching semantic objects to syntax nodes and allowing these objects to be linked to objects attached at other places in the syntax tree.

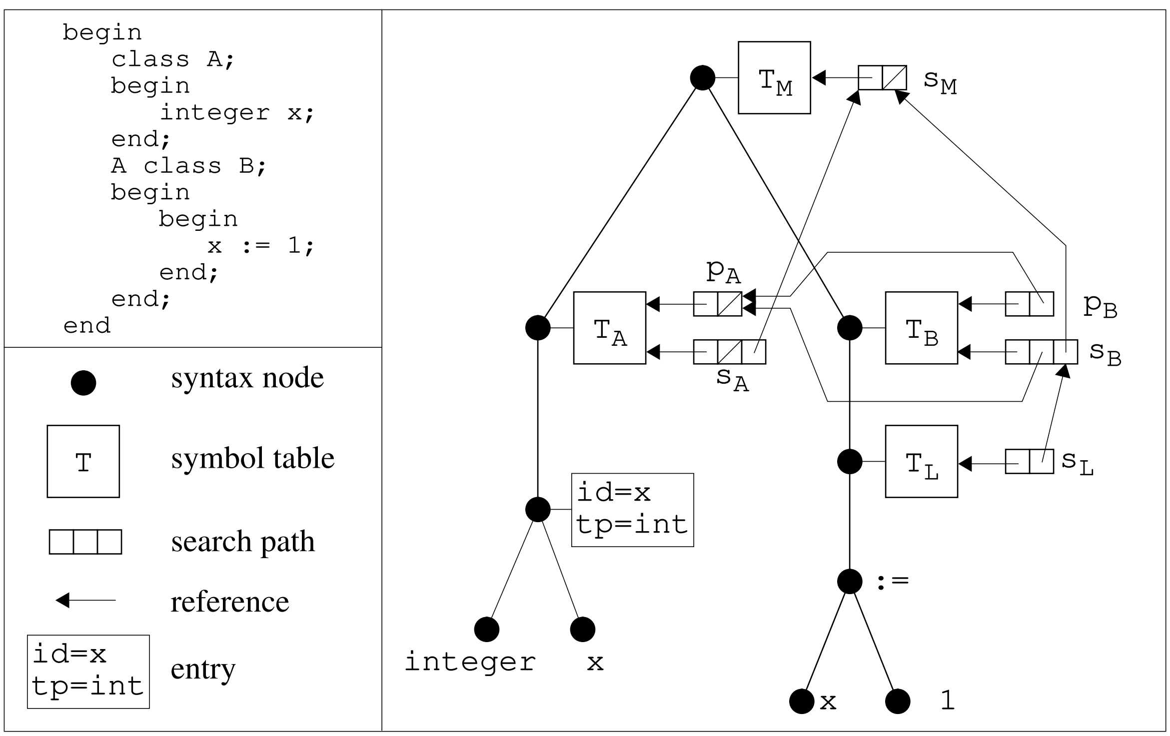
Attribute grammars, as defined by Knuth [Knu68], support the idea of an attribution derived from the syntax tree, but only in a limited way. In particular,

- Attributes may not have reference values.
- There are no direct non-local attribute dependencies.
- The attribute dependencies must not be circular.

We introduce an extension to attribute grammars, called *Door Attribute Grammars*, which removes the above limitations while retaining the declarative nature of attribute grammars. Door-AGs support the notion of a semantic object structure. Attributes of the nodes in the syntax tree may have reference values, referring to objects in the semantic object structure. The use of references leads to information sharing and direct non-local dependencies. In the framework we are suggesting, circular dependencies also fall in naturally . The grammars are named after the special *door* objects which serve as connections between the syntax tree and the semantic object structure. Complex scope rules, such as those appearing in object-oriented languages, are easily described by semantic object structures. We give an example of this and we also show how use/declaration bindings can be specified in a Door -AG. A more detailed description of Door -AGs appears in [Hed91]. There are many other approaches to extending attribute grammars to improve their incremental behavior , see e.g. the references in [Alb91]. However , these approaches do not support specification of semantic object structures.

2. A Semantic Object Structure

The Simula program below exemplifies the scope rules for block structure in combination with subclassing. Let M denote the main program, let A and B denote the classes A and B respectively , and let L denote the local block inside class B. The declaration of x is visible in A, in B (since B is a subclass of A), and in L (since L is nested inside B). The figure shows part of the syntax tree for the program and an associated semantic object structure. Each block is associated with a *symbol-table*



object and each declaration is associated with an *entry* object. A symbol table keeps track of all the entries in its block and provides a lookup function for looking up the entry of a particular identifier . Scope rules are modelled by *search-path* objects which define in which order to look in the symbol tables when binding identifiers. A search path consists of a sequence of slots which refer to symbol tables or other search paths. Each block b has a *static* search path s_b which specifies the lookup order for free identifiers in b. A class has in addition a *prefix* search path p_b which is used for binding qualified accesses and for defining the search paths of its subclasses.

For example, the use site x occurs free in L and is bound according to s_L, the static search path of L. s_L specifies that one should first look in the symbol table of L and then according to s_B (the static search path of the block enclosing L). s_B specifies to first look in the symbol table of B, then according to p_A (the prefix search path of the superclass of B), and finally according to s_M (the static search path of the block enclosing B). The search-path objects also have lookup functions which make use of the lookup functions in connected search paths and symbol tables. The declaration of x can therefore be found simply by calling the lookup function of s_L.

3. Door Attribute Grammars

A Door-AG specification consists of classes which specify the objects appearing in an attributed syntax tree. The *node classes* model the syntax nodes and specify the context-free syntax, the attributes (inherited, synthesized, and local), and the equations defining the attributes. Roughly, if X is a nonterminal and p is a production with left-hand side X, then p will be a subclass of X in our formulation. We use the same principles for casting the attribute grammar into an object-oriented form as reported in [Hed89] and which is also reviewed in this volume [Kos91]. The *semantic classes* model the objects in the semantic object structure. These objects may have local attributes, functions, and equations defining attributes. The syntax nodes interact with semantic objects via special objects called *doors*, modelled by *door classes*. Doors can be seen as half syntax nodes and half semantic objects. They have synthesized and inherited attributes to mimic a syntax node but are otherwise similar to semantic objects.

Each door and semantic object has a specific *owner* object. Let X and Y be two objects such that X owns Y. This means that Y is created at the same time as X and will have the same life-time as X. Doors are owned by syntax nodes whereas semantic objects can be owned by either another semantic object or by a door. Each object in the semantic object structure is therefore transitively owned by a particular syntax node and will be part of the attributed syntax tree as long as that syntax node is part of the tree. Thus, the whole semantic object structure can be seen as an extension of the syntax tree. Ownership is specified by declaring static reference attributes. A static reference is constant and will always refer to the owned object. In contrast, dynamic reference attributes are defined by equations and their values may change at attribute re-evaluation.

As an example, consider the Door-AG fragment below. `Decl` is a node class which models a declaration. It has two inherited attributes, `table` and `path`. `VarDecl` models a variable declaration as a subclass (specialization) of `Decl`. “<u>`alt`</u>” and “<u>`cons`</u>” are different types of node classes, roughly corresponding to nonterminal and production respectively. `VarDecl` has two son nodes, `DeclType` and `LexID`, for the type and the name of the variable. B, T, and N are static references to door objects. The role of B (a DeclBindingDoor object) is to own an Entry object and connect it to the symbol table so it can be reached at lookup. The role of T and N (the DeclareTypeDoor and the DeclareNameDoor) is to define the type and the name attributes of the Entry object. The reason why this functionality is split into three different doors, rather than putting it all together in one door , is to achieve efficient re-evaluation. Each door is re-evaluated as a unit, so this design enables the name and the type to be changed independently .

```
Decl: alt
{  inh table: ref SymbolTable;
   inh path: ref SearchPath;
};

VarDecl: cons Decl (DeclType, LexID)
{  B: doorobject DeclBindingDoor;
   T: doorobject DeclareTypeDoor;
   N: doorobject DeclareNameDoor;
   DeclType.path := path;
   B.table := table;
   T.declBinding := B.declBinding;
   T.tp := DeclType.tp;
   N.declBinding := B.declBinding;
   N.id := LexID.string;
};

Entry: class
{  table: ref SymbolTable;
   id: text;
   tp: ref Type;
};

DeclBindingDoor: door
{  inh table: ref SymbolTable;
   syn declBinding: ref Entry;

   binding: object Entry;
   binding.table := table;
   binding.id := "";
   binding.tp := bottomType;
   table.contains(binding);
   declBinding := binding;
};
```

```
DeclareTypeDoor: door
{  inh declBinding: ref Entry;
   inh tp: ref SimpleType;
   declBinding.tp := tp;
};

DeclareNameDoor: door
{  inh declBinding: ref Entry;
   inh id: text;
   declBinding.id := id;
};

Exp: alt
{  inh path: ref SearchPath;
   syn tp: ref Type;
};

IdUse: cons Exp (LexID)
{  U: doorobject UseBindingDoor;
   U.path := path;
   U.id := LexID.string;
   tp := U.tp;
};

UseBindingDoor: door
{  inh path: ref SearchPath;
   inh id: text;
   syn tp: ref Type;

   binding: ref Entry;
   binding := path.lookup(id);
   tp := binding.tp;
};
```

The following things can be noted in this example. First, a door object is treated in exactly the same way as a son node: its inherited attributes are defined by equations in the owning syntax node and its synthesized attributes are used in the syntax node to define other attributes. Second, the attributes may have reference values. E.g., the table and path attributes are references to symbol table and search path objects respectively . For reference values, the equations denote reference equality . For instance, the equation

```
DeclType.path := path;
```

means that both DeclType.path and path refer to the same object.

Third, the DeclBindingDoor contains the clause

```
table.contains(binding);
```

This is a *condition* rather than an equation. Conditions are constructive. This means that each condition has evaluation code associated with it. Executing the evaluation code makes the condition hold. In this case, the evaluation code would add `binding` to a list in `table`.

A door must contain equations for defining all of its attributes and all of the attributes of its owned objects. In addition, it may contain equations which define attributes in other semantic objects. This means that the original equation in the door owning that object is overridden. `DeclareNameDoor` and `DeclareTypeDoor` exemplify this. They define the `id` and `tp` attributes of the entry object and thereby override the corresponding equations in `DeclBindingDoor`.

The attributes and functions of semantic objects may not be accessed in the syntax nodes. Instead, these accesses must be placed in door objects. The reason for this is to simplify the analysis of non-local dependencies. The restriction in effect maps non-local dependencies between syntax nodes to non-local dependencies between door objects. The `IdUse` node class, modelling identifier use sites, exemplifies this. The `IdUse` owns a `UseBindingDoor` which has a reference "`binding`" to the appropriate entry object. The `binding` must be stored in the door, rather than directly in the `IdUse`, since the equation defining `binding` accesses the lookup function of the search path. Similarly, the access to the `tp` attribute of `binding` must be done in the door. Here we see examples of direct non-local dependencies. The value of `binding` depends not only on the values of `path` and `id`. It depends also on what symbol tables are found along the path, on what entries are found in each of those symbol tables, and on the `id` attribute of those entries. Thus, changing the `id` of an entry object may invalidate the `binding` attribute of some `UseBindingDoor` objects. We say there is a *direct non-local dependency* from the `DeclareNameDoor` to each of those `UseBindingDoors`.

4. Incremental Attribute Evaluation

Incremental evaluation of Door-AGs can be based on standard incremental visit-oriented algorithms, e.g. the one for ordered attribute grammars [Kas80] proposed by Reps and Teitelbaum in [ReT89]. Actually, it will often suffice to base the evaluation on a simpler 1-visit algorithm, because the local dependencies are usually very simple in a Door-AG. The algorithm involves attaching special visit procedures to the doors which handle evaluation of attributes in the semantic object structure. These visit procedures also schedule visits to non-local dependent doors. To find these dependents in an efficient way, additional dependency structures are usually needed. These structures are defined as additional local attributes of the doors and the semantic objects. Circular dependencies are handled simply by inserting a convergence test in the visit procedure. The details of the evaluation algorithm and of the visit procedure construction are given in [Hed91].

5. Conclusion

The Door-AGs introduced in this article are relevant for constructing practical incremental attribute evaluators. The most important difference between Door-AGs and standard AGs is that Door-AGs allow objects and reference-valued attributes to be part of the attribution. Object structures are natural for modelling many static semantic properties, in particular the complex scope rules of object-oriented languages. The techniques are, however, relevant also for simpler procedural languages.

To demonstrate the use of Door-AGs, a prototype editor with incremental static semantic analysis has been implemented for a subset of Simula. The subset includes classes, subclassing, block nesting, arbitrary declaration/use order, reference variables, qualified accesses and reference assignments. An earlier, less formalized variant of Door-AGs, has been used in implementing Mjølner/Orm, a complete incrementally compiling environment for Simula [Hed88], [Mag90].

Acknowledgment

This work was supported by Nordisk Industrifond and STU within the Mjølner II project.

References

[Alb91] Alblas H. Incremental Attribute Evaluation. In this volume.

[DMN68] Dahl O.-J., Myhrhaug B., Nygaard K. SIMULA 67 Common Base Language. Norwegian Computing Centre, Oslo, 1968. Revised 1970, 1972, and 1984. Swedish standard SS 63 61 14, 1987.

[Hed88] Hedin G. Incremental Attribute Evaluation with Side-effects. Proceedings of the Workshop on Compiler Compilers and High Speed Compilation. Berlin, October 1988. 175-189.

[Hed89] Hedin G. An Object-Oriented Notation for Attribute Grammars. Proc. of the European Conference on Object-Oriented Programming, Nottingham. Cambridge University Press, 1989, 329-345.

[Hed91] Hedin G. Door Attribute Grammars (preliminary title). Ph.D. thesis, to appear.

[Kas80] Kastens U. Ordered Attribute Grammars. Acta Informatica 13 (1980), 229-256.

[Knu68] Knuth D. E. Semantics of Context-Free Languages. Math. Syst. Theory 2:2, June 1968, 127-145.

[Kos91] Koskimies K. Object-Orientation in Attribute Grammars. In this volume.

[Mag90] Magnusson B., et al. An Overview of the Mjølner/Orm Environment: Incremental Language and Software Development. Proc. of the 2nd Int. Conf. on Technology of Object-Oriented Languages and Systems. Paris 1990, 635-646.

[TeR81] Teitelbaum T., Reps T. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. CACM, Sept. 1981, 563-573.

[ReT89] Reps T., Teitelbaum T. The Synthesizer Generator. Springer-Verlag, 1989.