A Door Attribute Grammar for an Object-Oriented Language

Görel Hedin

LU-CS-TR:94-134

June 1994

Department of Computer Science
Lund University

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

A Door Attribute Grammar for an Object-Oriented Language

Görel Hedin

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

June 1994

Abstract. This paper gives a specification of the static semantics of an example object-oriented language. The specification is done using the Door AG formalism which is an extension to standard AGs. The language contains typical fundamental object-oriented constructs found in languages like Simula, BETA, Eiffel, and C++.

1 Introduction

Door Attribute Grammars [Hed92, Hed94] are an extension to standard attribute grammars [Knu68]. A standard AG can be considered as a declarative description of syntax trees, where a syntax tree consists of attributed tree node objects. Taking this perspective, the additional facilities available in Door AGs are the following:

- The possibility to extend the syntax tree nodes with additional objects which can be used to describe static-semantic information such as symbol tables, types, etc.
- The possibility to have reference attributes, i.e. attributes whose value is an object identity.
- The possibility to distribute the definition of a collection-valued attribute.

The motivation for introducing these extensions were both to allow a more natural description of the static semantics of programming languages and to allow the description of attributions which are well suited for incremental updates in the context of language-based editors. In particular, the goal was to support description of object-oriented languages.

The objects which extend the syntax tree are of two kinds: *semantic objects* and *door objects*, where door objects can be directly attached to syntax nodes whereas the semantic objects must be attached via a door object. The door objects thus serve as interfaces between the syntax nodes and the semantic objects. The distinction between doors and semantic objects is important in order to develop effective incremental attribute evaluators.

The purpose of this paper is to give a concrete example of how Door AGs can be used to specify the static semantics of an object-oriented language. The example language, Tooy, is based on the running example used in [Hed92]. For an introduction and overview of Door AGs, we refer to [Hed94].

In analogy with standard AGs, a Door AG defines an equation system over syntax trees where the variables are the attributes occurring in the equations of the Door AG. A solution for a given syntax tree is an assignment of attributes such that all the equations of the Door AG are fulfilled. To evaluate the attributes, i.e. to find a solution, is more difficult than for a standard AG due to non-local dependencies introduced by the reference attributes. To describe evaluation techniques is outside the scope of this paper. However, one efficient incremental evaluation technique is developed in [Hed92] where it is also shown how it can be applied to a language in the style of Tooy.

2 The Tooy language

Our example language, Tooy, is a toy language with object-oriented features. It contains classes, subclasses, methods, basic types such as integers and booleans, reference types, qualified access, assignments, while-statements, and conditional statements. The Tooy language is primarily inspired by Simula, but the constructs include the main OO constructs which typically appear also in other statically typed object-oriented programming languages such as BETA, Eiffel, and C++. Most of the constructs (except for those related to type checking) are also similar to those in Smalltalk.

Below is an example Tooy program.

```
Program
  A: class          -- A is a class
    x: integer;       -- x is an instance variable in A
  endclass

  B: class A        -- B is a subclass of A
    inc: method       -- inc is a method in B
      x := x+1;       -- which increments the instance variable x
    endmethod;
  endclass;

  rB: ref B;        -- rB is a reference variable denoting an object of class B

  rB :- new B;      -- A new B object is created and a reference to it
                    -- is assigned to rB
  rB.inc;           -- The method inc of rB is called.
endprogram;
```

3 Informal static-semantics

3.1 Scope rules

We will use the term *block* to mean a language construct which introduces a set of declarations. In Tooy, the different kinds of blocks are: program, class, and method.

Homogeneous scopes
As in Algol, the order of declaration within a Tooy block is without importance. Thus, it is for example possible for a method to call another method declared earlier in the program. We call this property *homogeneous scopes*.

Block structure
Also as in Algol, names declared in a block are visible in interior blocks, and a declaration of a given name in an interior block will hide other declarations of the same name in enclosing blocks. For example, an instance variable x declared in a class is visible inside the methods of the class, except for in methods with a local declaration of x.

Most object-oriented programs contain a number of classes which in turn contain methods. However, similar to BETA and Simula, Tooy allows general nesting of classes and methods, allowing for example a class to be nested within another class. See, e.g., [MMN93] for a discussion of the dynamic semantics of nested classes.

Subclassing
Names declared in a class are visible also in subclasses. Similar to the block structure case, a declaration of a given name in a subclass will hide other declarations of the same name in superclasses. Subclassing has precedence over block structure. For example, consider the following Tooy program:

```
Program
   x: integer;          (1)

   A: class
      x: integer;       (2)
   endclass

   B: class A
      inc: method
         x := x+1;      (3)
      endmethod;
   endclass;
endprogram;
```

For use sites inside class B, declarations in the superclass A will hide declarations of the same name in B's enclosing block (the program). Thus, at the point (3), the declaration (1) is hidden by declaration (2). The uses of x at (3) thus refer to the instance variable declared at (2).

Qualified access
The expression *x.y* is an example of a *qualified access*. The *x*-component is a reference expression qualified by some class *C*, and the *y*-component is a name declared in *C* or in one of its superclasses. Both instance variables and methods can be accessed in this way. For example:

```
rB.inc        (* The method "inc" of the object denoted by "rB" is called *)
```

new B.x *(* A new object of class B is created and its instance variable x is accessed *)*

rA.rB.rC.m *(* The method "m" of the object denoted by "rA.rB.rC" is called, where rB is an instance variable of rA and rC is an instance variable of rB. *)*

3.2 Type checking

We differ between basic types (integer and boolean) and reference types. The value assignment $x := y$ is used for assigning variables of basic types. In this case, the types of x and y must be equal. The reference assignment $z :- v$ is used for assigning to variables of reference types. In this case, z and v must be qualified by classes C_z and C_v. The assignment is considered correct if C_z and C_v are equal or if one is a subclass of the other. However, if C_z is a subclass of C_v, a runtime qualification check will be needed. Some OO languages, e.g. Eiffel, have special syntax for the latter case of assignment.

3.3 Error detection

As in most other programming languages, there are the following basic kinds of static-semantic errors:

- Multiple declarations of the same name in the same block
- Missing declaration of a name
- Violation of the type checking rules

In addition, there is the following kind of error related to subclassing:

- A class may not, directly or indirectly, have itself as a superclass.

4 Context-free syntax

The context-free syntax is expressed in terms of *node classes*. General language constructs are represented by general node classes (superclasses) whereas specific language constructs are represented by specialized node classes (subclasses). Compared to an ordinary CFG, the superclasses correspond roughly to nonterminals and the subclasses to productions. However, some node classes correspond to both a nonterminal and a production. There are four different kinds of node classes: alternation, construction, list, and option:

- *Alternations* represent abstract language constructs.
- *Constructions* represent constructs with a given number of components of given classes.
- *Lists* represent constructs with 0 or more components of a given class.

- *Option* s represent unexpanded optional constructs. Each option node class has exactly one direct subclass, representing the expanded optional construct.

Furthermore, there are two predefined node classes ID and INT which represent identifiers and integer constants respectively .

Below is the context free syntax of T ooy in terms of node classes. For comparison, a corresponding ordinary CFG for T ooy is given in Appendix A.

Decl: **alt**;
Stmt: **alt**;
Exp: **alt**;
Program: **cons** (‘program’ Block ‘endprogram’);
Block: **cons** (Decls Stmts);
Decls: **list** (Decl ‘;’);
Stmts: **list** (Stmt ‘;’);
ClassDecl: **cons** Decl (ID ‘:’ ‘class’ SuperOpt Block ‘endclass’);
SuperOpt: **opt**;
Super: **cons** SuperOpt (SimpleUse);
MethodDecl: **cons** Decl (ID ‘:’ ‘method’ Block ‘endmethod’);
VarDecl: **cons** Decl (ID ‘:’ DeclType);
DeclType: **alt**;
IntDeclType: **cons** DeclType (‘integer’);
BoolDeclType: **cons** DeclType (‘boolean’);
RefDeclType: **cons** DeclType (‘ref’ SimpleUse);
RefAssignStmt: **cons** Stmt (Use ‘:-’ Exp);
ValAssignStmt: **cons** Stmt (Use ‘:=’ Exp);
WhileStmt: **cons** Stmt (‘while’ Exp ‘do’ Stmts ‘endwhile’);
IfStmt: **cons** Stmt (‘if’ Exp ‘then’ Stmts ‘else’ Stmts ‘endif’);
CallStmt: **cons** Stmt (Use);
Use: **alt** Exp;
SimpleUse: **cons** Use (ID);
QualUse: **cons** Use (Use ‘.’ SimpleUse);
NewExp: **cons** Exp (‘new’ SimpleUse);
NoneExp: **cons** Exp(‘none’);
Add: **cons** Exp (Exp ‘+’ Exp);
Sub: **cons** Exp (Exp ‘-’ Exp);
GTexp: **cons** Exp (Exp ‘>’ Exp);
LTexp: **cons** Exp (Exp ‘<‘ Exp);
EQexp: **cons** Exp (Exp ‘=’ Exp);
IntConst: **cons** Exp (INT);
TrueConst: **cons** Exp(‘true’);
FalseConst: **cons** Exp(‘false’);

5 Design of the context-sensitive part

To describe the context-sensitive properties of T ooy we will extend the CFG of section 4 to a Door AG by adding attributes, equations, semantic objects, and door objects. The most important kinds of semantic objects used are the following:

- *types* (basic types, class types, and reference types)
- *entries* (containing the name and type of a declaration)
- *symbol tables* (which collect entries for the local declarations in a block)
- *paths* (which combine symbol tables to describe the visibility rules applicable at a specific point in the program)

5.1 Scope rules

Each block in the program is extended with a symbol table object which collects entry objects, one for each declaration in the block. The symbol tables can be combined to form declarative environments, to be used for doing lookup of names. The declarative environments are modelled by so called *path objects*. A path object has reference attributes which connect to symbol table objects and/or to other path objects. This way, a directed acyclic graph is formed, where the symbol tables and path objects constitute the vertices, and the reference attributes of the path objects constitute the edges. By doing an ordered depth-first traversal of this graph, starting at a path object, one will find the symbol tables containing visible declarations, in an order corresponding to the declaration hiding rules.

To define ordinary block structure, each block is extended by a so called *static* path object which represents the declarative environment for free identifiers in the block. Each class is, in addition, extended by a so called *prefix* path object which represents the declarative environment for accesses qualified by that class. By connecting these path objects in suitable ways, the appropriate definition of scope rules are obtained. See [Hed93] for a more detailed description of how this technique is used to define visibility rules for different language constructs. The figure below shows the visibility graph of symbol tables (t), static paths (s) and prefix paths (p) for the program in section 2.

5.2 Type checking

Different Tooy types are represented by different objects with the common superclass Type. A constant object unknownType is used for cases when the type can not be decided, e.g. in the case of a missing declaration.

Basic types
Two constant objects intType and boolType represent the integer and boolean types.

Methods
A constant object methodType represents the type of method identifiers. [1]

1. The methods in Tooy are very simple since they have neither return type nor parameters. In a slightly more advanced language, the method type could instead be represented by a unique object per method, containing information about return type and parameters.

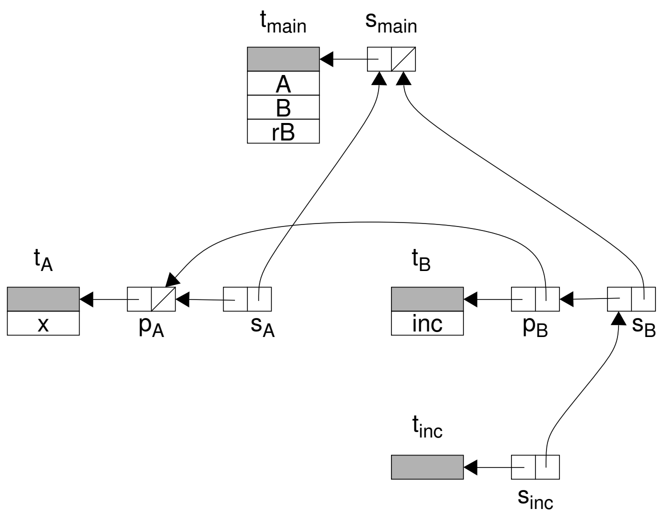

Figure 1 *Visibility graph for the program in section 2*

Classes

Each class *X* in a T ooy program gives rise to two type objects: a RefType object, for references qualified by *X*, and a ClassType object, for the qualification *X* itself. T ooy also has two built-in classes: ObjectClass and NoClass. ObjectClass is a most general class of which all user -defined classes are specializations. NoClass is the qualification of the built-in reference NONE, and has no subclass or superclass relation to any other class. Constant objects objectClassType, objectRefType, noClassType, and noRefType are used to represent the corresponding class types and reference types.

The ClassType and RefType objects for a given user -defined class are connected to each other by reference attributes. In addition, each ClassType object has a reference attribute denoting the ClassType object of its superclass (or objectClassType in case of a top-level class). Each ClassType object also has a reference attribute denoting the prefix path object for the class.

The figure below shows the graph of class type and ref type objects for the program in section 2.

5.3 Error detection

To specify error conditions we will use local boolean attributes with the suffix Error. A Tooy program for which all such attributes are false has no compile-time errors.

To detect multiple declarations of the same name within a block, and circularly defined class hierarchies, *underdetermined* grammars will be used. An underdetermined grammar is a grammar which has more than one attribute solution for some syntax trees. In the case of multiple declarations, one of the declarations is regarded as visible, and the others as non-visible, but exactly which of the declarations to be chosen as the visible one is not specified by the grammar .

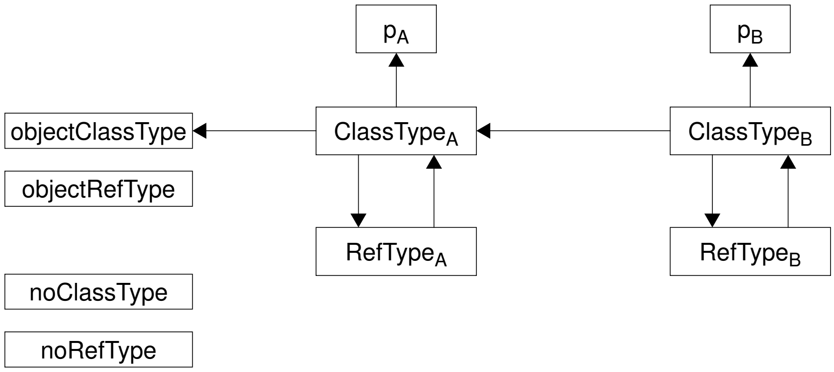

Figure 2 *Type graph for the program in section 2*

For circularly defined class hierarchies, one of the classes on the cycle is regarded as causing the cycle, but which one is chosen is not specified in the grammar. The visibility graph introduced in section 5.1 must be a DAG, and may thus not contain any cycles. To accomplish this, the class chosen as causing the cycle is treated as if it had ObjectClass as its superclass. This way, the cyclic class hierarchy will not cause cycles in the visibility graph.

The use of underdetermined grammars leads to simpler specifications and is also beneficial in incrementally updating systems because it leads to fewer attribute re-evaluations. See [Hed92] for a further discussion of these issues.

6 The Door AG specification

The Door AG is partitioned into a *main grammar*, containing the syntax node description, and a *door package*, containing the description of semantic objects and doors. The door package is self-contained, and can be reused by several different main grammars. Furthermore, the door package is split into an *interface*, containing only the parts that are directly used by a main grammar, and a *body*, which contains the detailed specification of the door package. It is necessary to read both the interface and the body of the door package to understand the specification in detail. But hopefully, it is sufficient to read the interface to get an overall understanding of how to use the door package. The door package interface contains the following parts:

- door class declarations, including the inherited and synthesized attributes of these classes
- declarations of the semantic classes used as types of the door class attributes
- declarations of constant semantic objects which the main grammar may use

We use the term *inherited* in the sense of attribute grammars. If we need to use this term in the sense of object-orientation, we will say *oo-inherited*. For example, all attributes (both inherited and synthesized) and all equations declared in a class are oo-inherited by the subclasses.

6.1 The door classes

The door package contains the following door classes: BlockDoor, ClassDoor, DeclDoor, UseDoor, and CompareClassDoor.

BlockDoor
BlockDoor is intended to extend the syntax tree at each Algol-style block, i.e. at each main program and each method declaration. The BlockDoor inherits one attribute: the static path for the enclosing block, and synthesizes two attributes: the static path for the block and the symbol table for the block. The static path should be transferred to all UseDoor objects at free local use sites within block. The symbol table should be transferred to all DeclDoor objects at local declarations within the block.

ClassDoor
ClassDoor is intended to extend the syntax tree at each class declaration. In addition, the syntax tree should be extended with a DeclDoor at this point, to add the class to the symbol table of its enclosing block.

The ClassDoor handles the visibility rules for classes (in a way analogous to BlockDoor), introduces a unique ClassType object and a unique RefType object for the class, and checks for circular class definitions. It inherits the type for the superclass, and will use it to define the visibility rules. However, in case the superclass type would introduce a cyclic structure among the ClassType objects, this will be prevented by instead treating the class as if it were a subclass to ObjectClass. In this case, a synthesized attribute causesCycle will be True.

DeclDoor
DeclDoor is intended to extend the syntax tree at each declaration. It inherits the symbol table to which to add the declaration, the name of the declaration, and the type of the declaration. It synthesizes a boolean attribute visible. In case there is more than one declaration of the same name within a symbol table, the attribute visible will be true for one of these declarations and false for the others.

UseDoor
UseDoor is intended to extend the syntax tree at each identifier use site. It inherits the path to use for lookup, the name of the identifier, and synthesizes the type of the corresponding declaration. It also synthesizes a boolean attribute to indicate if a declaration was found or not.

CompareClassDoor
CompareClassDoor is intended to extend the syntax tree at each place a comparison of class types is needed. Typically, this is at assignment statements and at actual parameters in method calls. Class types are considered to be arranged in a lattice where ObjectClass is at the top and NoClass at the bottom, and a more

general class (a superclass) is placed above a more specific class (a subclass). The door inherits the two class types, guessing that the one is more general than the other, and synthesizes two boolean attributes, stating if the guessed order was correct, and if the classes were at all comparable in the lattice.

6.2 About the grammar notation, OOSL

The grammar is written in an object-oriented specification language, OOSL, whose constructs are inspired by those in Simula and BETA. OOSL contains both declarative and imperative constructs. However, for the grammar description, only the declarative constructs are used. (The imperative constructs are used for specifying the attribute evaluator algorithms.) For a more detailed description of OOSL, see [Hed92].

Notational conventions
By convention, all class names start with an upper-case letter, whereas attributes, objects, and functions start with a lower-case letter. However, door objects will often be named by upper-case letters, as follows:

- BD - for a BlockDoor object
- CD - for a ClassDoor object
- DD - for a DeclDoor object
- UD - for a UseDoor object
- CCD - for a CompareClassDoor object

The following names are often used for attributes and functions of common types:

- tp - for a Type reference
- tbl - for a SymbolTable reference
- pth - for a Path reference

6.3 The door package interface

```
(* --- Semantic classes and objects --- *)

Path: class;
emptyPath: object Path;
   (* A path through which no declarations are found *)

SymbolTable: class;

Type: class;
unknownType: object Type; (* The type of undeclared identifiers, etc. *)

AbstractClassType: class Type
   (* Abstract definition of a class type, either built-in or user-defined *)
{  getRefTp: func ref AbstractRefType;
```

```
        (* Returns the corresponding ref type object *)
    getPrefixPth: func ref Path;
        (* Returns the prefix path of the class *)
};

ClassType class AbstractClassType;
    (* The class type of a user-defined class *)
objectClassType: object AbstractClassType;
    (* The class type of the built-in class ObjectClass *)
noClassType: object AbstractClassType;
    (* The class type of the built-in class NoClass *)

AbstractRefType: class Type
    (* Abstract definition of a reference type, either built-in or user-defined *)
{   getClassTp: func ref AbstractClassType;
        (* Returns the corresponding class type object *)
};
RefType: class AbstractRefType;
    (* The reference type of a user-defined class *)
objectRefType: object AbstractRefType;
    (* The reference type of the built-in class ObjectClass *)
noRefType: object AbstractRefType;
    (* The reference type of the built-in class NoClass.
        I.e., the type of the NONE reference *)


(* --- Door classes --- *)

BlockDoor: door
    (* Intended to extend each main program, block statement, or method *)
{   inh encPth: ref Path; (* Static path for the enclosing block *)
    syn locPth: ref Path; (* Static path for this block *)
    syn tbl: ref SymbolTable; (* Symbol table for this block *)
};

ClassDoor: door (* Intended to extend each class declaration *)
{   inh encPth: ref Path; (* Static path for the enclosing block *)
    inh superClassTp: ref AbstractClassType;
        (* The type of the superclass, according to the syntax *)
    syn locPth: ref Path; (* Static path for this class *)
    syn tbl: ref SymbolTable; (* Symbol table for this class *)
    syn classTp: ref ClassType; (* The ClassType for this class *)
    syn causesCycle: boolean;
        (* True if this class is on a cycle and if it is this class which is
            considered to cause the cycle *)
};

DeclDoor: door (* Intended to extend each declaration *)
{   inh tbl: ref SymbolTable;
```

```
    inh tp: ref Type;
    inh ident: string;
    syn visible: boolean;
};

UseDoor: door (* Intended to extend each identifier use site *)
{   inh pth: ref Path; (* The path to be used for lookup *)
    inh ident: string; (* The used name *)
    syn tp: ref Type; (* The type of the matching declaration *)
    syn declared: boolean;
        (* True if there is a declaration for this use. *)
};

CompareClassDoor: door (* Intended to use for comparing two class types *)
{   inh generalClassTp: ref AbstractClassType;
        (* One of the class types *)
    inh specialClassTp: ref AbstractClassType;
        (* The other of the class types *)
    syn reverseOrder: boolean;
        (* True if generalClass is instead more special than specialClass *)
    syn incomparable: boolean;
        (* True if neither of generalClass and specialClass is more general
           than the other one *)
};
```

6.4 The main grammar

The main grammar for T ooy uses the door package interface as shown below . The main grammar contains the node classes of section 4, with attributes, equations, and door objects added. Boolean attributes with the suffix Error indicate different kinds of context-sensitive errors.

The main grammar also contains some additional constant semantic objects, particular to this grammar: intType, boolType, and methodType.

For clarity , we have not utilized the possibility to introduce additional abstract superclasses to the context-free grammar , which would otherwise have allowed us to oo-inherit commonly occurring copy equations, thus saving us from repeating these equations in the different node classes.

```
(* --- Semantic classes and objects --- *)

ValueType: class Type; (* Types for which value assignment is meaningful *)

intType: object ValueType;
boolType: object ValueType;

methodType: object Type;
```

```
(* --- Node classes --- *)

Decl: alt
{   inh tbl: ref SymbolTable;
    inh pth: ref Path;
};
Stmt: alt
{   inh pth: ref Path;
};
Exp: alt
{   inh pth: ref Path;
    syn tp: ref Type;
};
Program: cons ('program' Block 'endprogram')
{   BD: doorobject BlockDoor;
    eq BD.encPth :- emptyPath;
    eq Block.pth :- BD.locPth;
    eq Block.tbl :- BD.tbl;
};
Block: cons (Decls Stmts)
{   inh pth: ref Path;
    inh tbl: ref SymbolTable;
    eq Decls.tbl :- tbl;
    eq Decls.pth :- pth;
    eq Stmts.pth :- pth;
};
Decls: list (Decl ';' )
{   inh pth: ref Path;
    inh tbl: ref SymbolTable;
    eq Decl.pth :- pth;
    eq Decl.tbl :- tbl;
};
Stmts: list (Stmt ';')
{   inh pth: ref Path;
    eq Stmt.pth :- pth;
};
ClassDecl: cons Decl (ID ':' 'class' SuperOpt Block 'endclass')
{   CD: doorobject ClassDoor;
    DD: doorobject DeclDoor;
    loc cyclicClassError: boolean; (* True if this class causes a cycle *)
    loc multipleDeclaredError: boolean;
        (* True if this declaration has a namesake in the current block,
           but is not regarded as the visible one. *)
    eq SuperOpt.pth :- pth;
    eq CD.encPth :- pth;
    eq CD.superClassTp :- SuperOpt.tp;
    eq Block.pth :- CD.locPth;
    eq Block.tbl :- CD.tbl;
    eq DD.tbl :- tbl;
```

```
   eq DD.ident :- ID.ident;
   eq DD.tp :- CD.classTp;
   eq cyclicClassError := CD.causesCycle;
   eq multipleDeclaredError := not DD.visible;
};
SuperOpt: opt
{  inh pth: ref Path;
   syn tp: ref AbstractClassType;
   eq tp :- objectClassType;
      (* default equation for tp, overridden in the subclass*)
};
Super: cons SuperOpt (SimpleUse)
{  loc superclassTpError: boolean;
      (* True if the SimpleUse is something else than a class*)
   eq SimpleUse.pth :- pth;
   eq tp :-
         inspect x :- SimpleUse.tp
         when AbstractClassType do x
         otherwise objectClassType;
   eq superclassTpError :=
         not (SimpleUse.tp in AbstractClassType or
              SimpleUse.tp == unknownType);
};
MethodDecl: cons Decl (ID ':' 'method' Block 'endmethod')
{  BD: doorobject BlockDoor;
   DD: doorobject DeclDoor;
   loc multipleDeclaredError: boolean;
      (* True if this declaration has a namesake in the current block,
         but is not regarded as the visible one. *)
   eq BD.encPth :- pth;
   eq Block.pth :- BD.locPth;
   eq Block.tbl :- BD.tbl;
   eq DD.tbl :- tbl;
   eq DD.ident :- ID.ident;
   eq DD.tp :- methodType;
   eq multipleDeclaredError := not DD.visible;
};
VarDecl: cons Decl (ID ':' DeclType)
{  DD: doorobject DeclDoor;
   loc multipleDeclaredError: boolean;
      (* True if this declaration has a namesake in the current block,
         but is not regarded as the visible one. *)
   eq DeclType.pth :- pth;
   eq DD.tbl :- tbl;
   eq DD.ident :- ID.ident;
   eq DD.tp :- DeclType.tp;
   eq multipleDeclaredError := not DD.visible;
};
```

```
DeclType: alt
{   syn tp: ref Type;
    inh pth: ref Path;
};
IntDeclType: cons DeclType ('integer' )
{   eq tp :- intType;
};
BoolDeclType: cons DeclType ('boolean' )
{   eq tp :- boolType;
};
RefDeclType: cons DeclType ('ref' SimpleUse)
{   loc refDeclError: boolean;
        (* True if the SimpleUse is something else than a class *)
    eq SimpleUse.pth :- pth;
    eq tp :-
        inspect x :- SimpleUse.tp
        when AbstractClassType do x
        otherwise objectClassType;
    eq refDeclError :=
        not SimpleUse.tp in AbstractClassType and
        not SimpleUse.tp == unknownType;
};
RefAssignStmt: cons Stmt (Use ':-' Exp)
{   loc lhsError: boolean; (* True if left-hand-side is not a ref type *)
    loc rhsError: boolean; (* True if right-hand-side is not a ref type *)
    loc incomparableError: boolean;
        (* True if both sides are ref types, but incomparable *)
    loc runtimeCheckNeeded: boolean;
        (* True if runtime check is needed, i.e. if the left-hand-side type is a
           specialization of the right-hand-side type. *)
    CCD: doorobject CompareClassDoor;
    eq Use.pth :- pth;
    eq Exp.pth :- pth;
    eq CCD.generalClassTp :-
            inspect x :- Use.tp
            when AbstractRefType do x.getClassTp
            otherwise objectClassType;
    eq CCD.specialClassTp :-
            inspect x :- Exp.tp
            when AbstractRefType do x.getClassTp
            otherwise noClassType;
    eq lhsError :=
            not (Use.tp in AbstractRefType or Use.tp == unknownType);
    eq rhsError :=
            not (Exp.tp in AbstractRefType or Exp.tp == unknownType);
    eq incomparableError := CCD.incomparable;
    eq runtimeCheckNeeded := CCD.reverseOrder;
};
```

```
ValAssignStmt: cons Stmt (Use ':=' Exp)
{  loc incompatibleTypesError: boolean;
      (* True if both sides are value types, but incompatible *)
   loc nonValueTypesError: boolean;
      (* True if one or both sides are non-value types *)
   loc BothValueTypes: boolean; (* True if both sides are value types *)
   eq Use.pth :- pth;
   eq Exp.pth :- pth;
   eq bothValueTypes :=
         (Use.tp in ValueType or Use.tp == unknownType) and
         (Exp.tp in ValueType or Exp.tp == unknownType);
   eq incompatibleTypesError :=
         bothValueTypes and
         not ( Use.tp == Exp.tp or
               Use.tp == unknownType or
               Exp.tp == unknownType);
   eq nonValueTypesError := not BothValueTypes;
};
WhileStmt: cons Stmt ('while' Exp 'do' Stmts 'endwhile')
{  loc condTypeError: boolean; (* True if the condition Exp is non-boolean *)
   eq Exp.pth :- pth;
   eq Stmts.pth :- pth;
   eq condTypeError :=
         not(Exp.tp == boolType or Exp.tp == unknownType);
};
IfStmt: cons Stmt ('if' Exp 'then' Stmts 'else' Stmts 'endif')
{  loc condTypeError: boolean; (* True if the condition Exp is non-boolean *)
   eq Exp.pth :- pth;
   eq Stmts1.pth :- pth;
   eq Stmts2.pth :- pth;
   eq condTypeError :=
         not(Exp.tp == boolType or Exp.tp == unknownType);
};
CallStmt: cons Stmt ( Use )
{  loc typeError: boolean; (* True if Use is not a method *)
   eq Use.pth :- pth;
   eq typeError := not (Use.tp == methodType or Use.tp == unknownType);
};
Use: alt Exp;
SimpleUse: cons Use (ID)
{  UD: doorobject UseDoor;
   loc undeclaredError: boolean; (* True if the declaration is missing *)
   eq UD.pth :- pth;
   eq UD.ident := ID.ident;
   eq tp :- UD.tp;
   eq undeclaredError := ID.ident <> "" and not UD.declared;
      (* Don't report error if the ID is unexpanded (empty string) *)
};
```

```
QualUse: cons Use (Use '.' SimpleUse)
{   loc lhsNotRefError: boolean;
        (* True if the left-hand-side is not a reference *)
    eq lhsNotRefError :=
            not (Use.tp in AbstractRefType or Use.tp == unknownType);
    eq Use.pth :- pth;
    eq SimpleUse.pth :-
            inspect x :- Use.tp
            when AbstractRefType do x.getClassTp.getPrefixPth
            otherwise emptyPath;
    eq tp :- SimpleUse.tp;
};
NewExp: cons Exp ('new' SimpleUse)
{   loc useNotClassError: boolean;
        (* True if the SimpleUse is not a class *)
    eq SimpleUse.pth :- pth;
    eq useNotClassError :=
        not (SimpleUse.tp in AbstractClassType or
                SimpleUse.tp == unknownType);
    eq tp :-
            inspect x :- SimpleUse.tp
            when AbstractClassType do x.getRefTp
            otherwise objectRefType;
};
NoneExp: cons Exp('none')
{   eq tp :- noRefType;
};
Add: cons Exp (Exp '+' Exp)
{   loc typeError: boolean; (* True if any of the operands is a non-integer *)
    eq Exp1.pth :- pth;
    eq Exp2.pth :- pth;
    eq typeError :=
            not ((Exp1.tp == intType or Exp1.tp == unknownType) and
                    (Exp2.tp == intType or Exp2.tp == unknownType));
    eq tp :- intType;
};
Sub: cons Exp (Exp '-' Exp)
{   loc typeError: boolean; (* True if any of the operands is a non-integer *)
    eq Exp1.pth :- pth;
    eq Exp2.pth :- pth;
    eq typeError :=
            not ((Exp1.tp == intType or Exp1.tp == unknownType) and
                    (Exp2.tp == intType or Exp2.tp == unknownType));
    eq tp :- intType;
};
GTexp: cons Exp (Exp '>' Exp)
{   loc typeError: boolean; (* True if any of the operands is a non-integer *)
    eq Exp1.pth :- pth;
    eq Exp2.pth :- pth;
```

```
   eq typeError :=
         not ((Exp1.tp == intType or Exp1.tp == unknownType) and
               (Exp2.tp == intType or Exp2.tp == unknownType));
   eq tp :- boolType;
};
LTexp: cons Exp (Exp '<' Exp)
{  loc typeError: boolean; (* True if any of the operands is a non-integer *)
   eq Exp1.pth :- pth;
   eq Exp2.pth :- pth;
   eq typeError :=
         not ((Exp1.tp == intType or Exp1.tp == unknownType) and
               (Exp2.tp == intType or Exp2.tp == unknownType));
   eq tp :- boolType;
};
EQexp: cons Exp (Exp '=' Exp)
{  loc typeError: boolean; (* True if any of the operands is a non-integer *)
   eq Exp1.pth :- pth;
   eq Exp2.pth :- pth;
   eq typeError :=
         not ((Exp1.tp == intType or Exp1.tp == unknownType) and
               (Exp2.tp == intType or Exp2.tp == unknownType));
   eq tp :- boolType;
};
IntConst: cons Exp (INT)
{  eq tp :- intType;
};
TrueConst: cons Exp('true')
{  eq tp :- boolType;
};
FalseConst: cons Exp('false')
{  eq tp :- boolType;
};
```

6.5 Collection data types

Door AGs contain a facility for distributing the definition of a collection-valued attribute. I.e., instead of defining the collection-valued attribute by a single equation, each element in the collection is defined individually , at arbitrary locations in the extended syntax tree.

This facility is realized by so called *collection objects* , holding the collection-valued attribute, and so called *conditions* , defining an object to be part of zero or more collections.

The door package for T ooy makes use of a list collection for collecting declarations in symbol tables. The collection class used for this has the following interface:

```
List: class [T: class Element]
{  contains: func boolean (e: ref T);
```

```
      (* Returns True if e is an element in the list *)
    first: func ref T;
      (* Returns the first element in the list.
         Returns NONE if the list is empty *)
    sucOf: func ref T (e: ref T);
      (* Returns the successor of e in the list.
         Returns NONE if e is the last element *)
    predOf: func ref T (e: ref T);
      (* Returns the predecessor of e in the list.
         Returns NONE if e is the first element *)
};

Element: class;
```

The list class has a parameter T (which must be a subclass of class Element) which is the type of the elements in the list. The state of a list object (the collection-valued attribute) is hidden within the class, and can only be accessed via the functions in the class.

To define an object as a member of the list, one uses a condition, e.g. as follows:

```
D: door
{   inh aList: ref List;
    reg: cond aList.contains(this D);
};
```

This condition (named "reg") states that the D object is a member of the list aList.

Note that the conditions only state element memberships, not in which order the elements should appear in the list. This is thus a source of underdeterminedness in the grammar — any permutation of the members satisfies the conditions in the grammar. In the Tooy door package, the list is used in a symbol table to collect the declarations of a block. The order in the list is relevant only in the case when there are several declarations with the same name. The declaration of a particular name occurring first in the list is defined to be the visible one, and successive declarations of the same name are defined as invisible.

6.6 The door package body

The door package body adds equations, attributes, and semantic objects to the door classes declared in the door package interface. Additional semantic classes are also introduced. For readability, we repeat the door package interface in the description here.

```
(* --- SEMANTIC CLASSES AND OBJECTS --- *)

(* --- Symbol tables and entries --- *)
```

```
AbstractEntry: class Element
   (* Abstract definition of a symbol table element *)
{  getTp: func ref Type;
};

Entry: class AbstractEntry
   (* A symbol table element corresponding to a declaration in the program *)
{  loc ident: string;
   loc tp: ref Type;
   impl getTp :- tp;
};

nullEntry: object AbstractEntry
   (* The symbol table element for a missing declaration *)
{  impl getTp :- unknownType;
};

SymbolTable: class
{  collection entries: object List[Entry];
   lookup: func ref Entry(ident: string)
      (* Returns the first Entry in entries with identifier ident.
         Returns NONE if no such Entry is found *)
      :- loop x :- entries.first do
            inspect y :- x
            when Entry do
               if y.ident = ident
               then y
               else next x :- entries.sucOf(x)
            otherwise NONE;
};

(* --- Paths --- *)

Path: class
{  lookup: func ref Entry(ident: string);
      (* Returns the Entry with identifier ident which is visible according
         to this path. Returns NONE if no such Entry is found *)
};

TwoPath: class Path
   (* A path with two outgoing edges, where first has priority over second *)
{  loc first: ref Path;
   loc second: ref Path;
   impl lookup :-
      inspect x :- first.lookup(ident)
      when Entry do x
      otherwise second.lookup(ident);
};
```

```
SymbolTablePath: class Path
   (* A path with one outgoing edge, connected to a symbol table *)
{  loc tbl: ref SymbolTable;
   impl lookup :- tbl.lookup(ident);
};

emptyPath: object Path
   (* A path with no outgoing edges *)
{  impl lookup :- NONE;
};

(* --- Types --- *)

Type: class;
unknownType: object Type;

AbstractClassType: class Type
{  getRefTp: func ref AbstractRefType;
   getPrefixPth: func ref Path;
   subclassOrEqual: func boolean(classTp: ref AbstractClassType);
      (* Returns TRUE if this class type represents a subclass of the class
         type classTp, or if this class type is the same as classTp. *)
};

ClassType class AbstractClassType
{  theRefType: object RefType;
   loc owner: ref ClassDoor;
   loc superclassTp: ref AbstractClassType;
   impl getRefTp :- theRefType;
   impl getPrefixPth :- owner.prefixPath;
   impl subclassOrEqual :=
      if classTp == this ClassType
      then true
      else
         if classTp == noClassType
         then false
         else superClassTp.subclassOrEqual(classTp);
};
objectClassType: object AbstractClassType
{  impl getRefTp :- objectRefType;
   impl getPrefixPth :- emptyPath;
   impl subclassOrEqual := classTp == objectClassType;
};
noClassType: object AbstractClassType
{  impl getRefTp :- noRefType;
   impl getPrefixPth :- emptyPath;
   impl subclassOrEqual := true;
};
```

```
AbstractRefType: class Type
{   getClassTp: func ref AbstractClassType;
};
RefType: class AbstractRefType
{   loc classTp: ref ClassType;
    impl getClassTp :- classTp;
};
objectRefType: object AbstractRefType
{   impl getClassTp :- objectClassType;
};
noRefType: object AbstractRefType
{   impl getClassTp :- noClassType;
};


(* --- DOOR CLASSES --- *)

BlockDoor: door
{   inh encPth: ref Path;
    syn locPth: ref Path;
    syn tbl: ref SymbolTable;
    theTable: object SymbolTable;
    staticPath: object TwoPath;
    theTablePath: object SymbolTablePath;
    eq staticPath.first :- theTablePath;
    eq staticPath.second :- encPth;
    eq theTablePath.tbl :- theTable;
    eq tbl :- theTable;
    eq locPth :- staticPath;
};

ClassDoor: door
{   inh encPth: ref Path;
    inh superClassTp: ref AbstractClassType;
    syn locPth: ref Path;
    syn tbl: ref SymbolTable;
    syn classTp: ref ClassType;
    syn causesCycle: boolean;

    theClassType: object ClassType;
    eq classTp :- theClassType;
    eq theClassType.owner :- this ClassDoor;
    eq theClassType.superClassTp :-
        if superClassTp.subclassOrEqual(theClassType)
        then objectClassType
        else superClassTp;
      (* Make sure a cyclic structure is not created *)
```

```
    theTable: object SymbolTable;
    theTablePath: object SymbolTablePath;
    eq tbl :- theTable;
    eq theTablePath.tbl :- theTable;

    prefixPath: object TwoPath;
    eq prefixPath.first :- theTablePath;
    eq prefixPath.second :- theClassType.superclassTp.getPrefixPth;

    staticPath: object TwoPath;
    eq staticPath.first :- prefixPath;
    eq staticPath.second :- encPth;
    eq locPth :- staticPath;

    eq causesCycle := superclassTp =/= theClassType.superclassTp;
};

DeclDoor: door
{   inh tbl: ref SymbolTable;
    inh tp: ref Type;
    inh ident: string;
    syn visible: boolean;
    theEntry: object Entry;
    registered: cond tbl.entries.contains(theEntry);
    eq theEntry.ident := ident;
    eq theEntry.tp :- tp;
};

UseDoor: door
{   inh pth: ref Path;
    inh ident: string;
    syn tp: ref Type;
    syn declared: boolean;
    loc binding: ref AbstractEntry;
       (* Denotes the Entry of the matching declaration, or nullEntry,
          in case there is no matching declaration. *)
    eq binding :-
          if ident = “”
          then nullEntry
          else
             inspect x :- pth.lookup(ident)
             when Entry do x
             otherwise nullEntry;
    eq tp :- binding.getTp;
    eq declared := binding =/= nullEntry;
};
```

```
CompareClassDoor: door
{   inh generalClassTp: ref AbstractClassType;
    inh specialClassTp: ref AbstractClassType;
    syn reverseOrder: boolean;
    syn incomparable: boolean;
    orderOK: func boolean :=
       specialClassTp.subclassOrEqual(generalClassTp);
    eq reverseOrder :=
          if orderOK
          then false
          else generalClassTp.subclassOrEqual(specialClassTp);
    eq incomparable :=
          if orderOK
          then false
          else not generalClassTp.subclassOrEqual(specialClassTp);
};
```

7 Concluding remarks

The main grammar of section 6.4 and the door package of section 6.6 together constitute the Door AG for T ooy. The Door AG is similar in nature to a standard AG in that the attribute values are defined declaratively , by means of equations. However , thanks to the extensions available in Door AGs, there are several things which we are able to express in Door AGs, that cannot be expressed in standard AGs, leading to a more natural specification and to an attribution more suited to incremental updates and more useful in interactive systems:

Reference attributes allow us to directly connect objects in the extended syntax tree. For example, a use site is connected to its declaration site by the binding attribute in UseDoor, and a class to its superclass by the superclassTp attribute in ClassType. Such direct connections are not possible to define in standard AGs. In an interactive system, such connections can be used, e.g., to support browsing, allowing the user to navigate between connected sites in a program in a hypertext-like manner .

Semantic objects allow us to achieve a fine-grained definition of static-semantic structures. For example, the visibility graph discussed in section 5.1 is defined by several small semantic objects connected by reference attributes. Each individual part of the graph, for example a single edge, is defined by an equation independent of other equations defining other parts of the graph. A declarative environment is represented by a reference to a vertex in the graph.

This organization is quite different from the one in standard AGs where a graph would need to be embedded in a single attribute, defined by a single equation, and each declarative environment would be represented by a large attribute containing all the information needed for name analysis. This has a great impact on incrementality: Consider a change to an edge in the visibility graph. In the Door AG, this affects the edge attribute itself, but the declarative environment attributes are unaffected since they still denote the same vertex objects. In contrast, in the standard AG, all declarative environment attributes containing the edge will be affected. Since there are usually as many environment attributes as there are use sites in the program, there are much fewer affected attributes in the Door AG case, and the work needed to restore attribute consistency is correspondingly smaller. Notably, this difference makes the incremental Door AG technique possible to scale up for large programs because the work needed for incremental update is independent on program size.

Collection objects allow us to distribute the definition of lists, sets, and other collection-valued attributes over an arbitrary set of syntax tree sites. Each site in the syntax tree that contributes to the collection-valued attribute can define its contribution independently of other sites. For example, each declaration defines its membership in the local symbol table independently of all other declarations by the condition registered in DeclDoor. In contrast, to construct a collection-valued attribute in a standard AG one needs to introduce auxiliary attributes containing partially built collection values and thread these attributes around in the tree. One effect of this is that the standard AG will contain more complex local dependencies, due to the threading of the auxiliary attributes.

A second effect is incrementality: whereas individual members of collections can be updated individually in a Door AG, the updating in a standard AG will require all partial collections containing the updated member to be re-evaluated.

A third effect is that of underdeterminedness: the use of underdetermined collections introduces the possibility to define grammars with more than one solution to a given syntax tree. We use this in our Door AG to associate static-semantic errors with different locations in different solutions. This is the case for the attributes causesCycle in ClassDoor and visible in DeclDoor. Such underdeterminedness is beneficial in an interactive system, both because it leads to less work during incremental reevaluation, and because it gives the opportunity to let editing order have effect on the error reporting; if changes lead to errors, the errors can be associated with the newly changed parts of the tree rather than with previously existing parts of the tree.

References

[Hed92] G. Hedin. *Incremental semantic analysis*. PhD thesis, Lund University, Lund, Sweden, 1992. Tech. Rep. LUTEDX/(TECS-1003)/1-276/(1992).

[Hed93] G. Hedin. Incremental name analysis for object-oriented langauges. In J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object oriented environments: the Mjølner approach*. Prentice Hall, 1993.

[Hed94] G. Hedin. An overview of door attribute grammars. In *Proceedings of the 5th international conference on Compiler Construction (CC'94)*, LNCS 786, pp 31–51, Edinburgh, April 1994. Springer-Verlag.

[Knuth68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[MMN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

Appendix A CFG for Tooy

For comparison to the OO CFG given in section 4, we here give an ordinary CFG for Tooy. For node classes in the OO CFG which correspond to both a nonterminal and a production, we have introduced additional production names ending in “Prod”

Nonterminals:

Decl, Stmt, Exp, Program, Block, Decls, Stmts, SuperOpt, DeclType, Use, SimpleUse

Productions

ProgramProd:	Program ::= ‘program’ Block ‘endprogram’
BlockProd:	Block ::= Decls Stmts
DeclsProd:	Decls ::= (Decl ‘;’)*
StmtsProd:	Stmts ::= (Stmt ‘;’)*
ClassDecl:	Decl ::= ID ‘:’ ‘class’ SuperOpt Block ‘endclass’
Super:	SuperOpt ::= SimpleUse
NoSuperProd:	SuperOpt ::= ε
MethodDecl:	Decl ::= ID ‘:’ ‘method’ Block ‘endmethod’
VarDecl:	Decl ::= ID ‘:’ DeclType
IntDeclType:	DeclType ::= ‘integer’
BoolDeclType:	DeclType ::= ‘boolean’
RefDeclType:	DeclType ::= ‘ref’ SimpleUse
RefAssignStmt:	Stmt ::= Use ‘:-’ Exp
ValAssignStmt:	Stmt ::= Use ‘:=’ Exp
WhileStmt:	Stmt ::= ‘while’ Exp ‘do’ Stmts ‘endwhile’
IfStmt:	Stmt ::= ‘if’ Exp ‘then’ Stmts ‘else’ Stmts ‘endif’
CallStmt:	Stmt ::= Use
ExpUseProd:	Exp ::= Use
UseSimpleUseProd:	Use ::= SimpleUse
SimpleUseProd:	SimpleUse ::= ID
QualUse:	Use ::= Use ‘.’ SimpleUse
NewExp:	Exp ::= ‘new’ SimpleUse
NoneExp:	Exp ::= ‘none’
Add:	Exp ::= Exp ‘+’ Exp
Sub:	Exp ::= Exp ‘-’ Exp
GTexp:	Exp ::= Exp ‘>’ Exp
LTexp:	Exp ::= Exp ‘<‘ Exp
EQexp:	Exp ::= Exp ‘=’ Exp
IntConst:	Exp ::= INT
TrueConst:	Exp ::= ‘true’
FalseConst:	Exp ::= ‘false’