Another Solution of Scoping Problems in Symbol Tables

Werner Aßmann
German National Research Center for Computer Science
Research Center for Innovative Computer Systems and Technology
Rudower Chaussee 5, D - 1199 Berlin
assmann@first.gmd.de

Abstract

The central task of symbol table administration is identification of all identifiers used in the source program under consideration of visibility rules varying between the different programming languages. Usually this problem is handled by a hash search of the identifier and an additional decision about the visibility of this identifier in the current program region ("scoping problem"). The most used solution for the scoping problem is scope-dependent switching of hash links in such a way that only such identifiers indeed visible in this region are reachable by the hashing process.

The new approach handles the scoping problem by visibility sets defined for every program region, and additionally by overload resolution tables for multiple used identifier names. The result is constant decision time for the scoping problem (at latest) after the first access to an identifier in a program region. The algorithm is very simple so that the access time is not only (nearly) constant but also very small. The approach permits an efficient solution of all the scoping problems of contemporary programming languages such as MODULA-2, C, or Fortran90.

1. Introduction

The symbol table is one of the most important tools in compiler construction. All identification problems of the identifiers used in a program must be solved by the symbol table. Additionally, the symbol table usually administrates all the information about the program objects such as type, allocation, and other properties (this aspect will not be considered in the following).

In classical programming languages such as ALGOL-60 only simple problems arise in symbol tables: after searching (with success) in the list of identifiers only a decision is necessary, if the object is truly visible in the current block because of the scoping rule that all objects of outer blocks are readable in the inner blocks but not vice versa, and the declaration rule that objects can be overdeclared in inner blocks. In other programming languages the rules sometimes are much more simpler.

The classical solution to symbol table administration in this class of programming languages is a hash table with separate overflow table (all identifiers with the same hash code are linked together) and some switching strategy for these link chains to make visible exactly the right objects in dependence of the block structure, usually by a stack.

In modern programming languages much more complicated scoping and visibility rules are defined. In MODULA-2 (will be used in the following for the demonstration of the different features and the algorithms) firstly a few data structures are defined with special scoping rules: enumerations and records. All elements of an enumeration (the enumeration literals) are visible in the same scopes as the enumeration type itself. Because of the export/import concepts it is necessary to gather all enumeration literals of one enumeration type in an extra scope. As a result of this method we need a possibility to make visible all elements of these scopes in another scope (extension of a scope). The components of a record declaration are visible either by qualification or inside of a WITH-statement. An extra scope for the record components is necessary, too.

Another very important extension is the module concept with the export/import features. Module objects can be exported in the environment (by registering into the export list), and are directly visible there (unqualified export) or by qualification with the module name. On the other hand modules can import objects from their environment - also with direct visibility or by qualification.

The scoping respective visibility rules therefore are much more difficult as in the past. A lot of scopes with different properties must be administrated, and the visibility of scopes must be extended to (or restricted from) other scopes in a very free way.

Some solutions of this problem have already been given. Typically the hash chains are updated but there exists a solution founded on relational data bases, too. The approach described in this paper starts on another consideration. Provided that we have a matrix with one row for each identifier and one column for each scope, and the matrix elements denotes the visibility (may be: not visible, visible for read operations, visibility for write operations, or visibility for both), then the decision about the visibility of an identifier in a given scope is very easy to decide by inspection of the related matrix element (despite of overloading). This is a situation known from parser tables: the original form of these tables is very big and it is necessary to use table compaction methods. It seems to be no reason not to try such a technique for symbol table administration, especially for the solution of the scoping problem.

The result of this approach is a matrix similar to the just described one but with rows for scopes and not for every identifier (representing classes of identifiers), and an additional table with one entry for every overloaded identifier to solve the overloading problem.

2. A Simple Example

The program in Figure 1 demonstrates the scope handling for procedures, enumerations, and records (written in MODULA-2). In this little program various scopes appear: the environment of the module (scope 1), the scopes for the program objects of the module m and the procedure p (scopes 2 and 5), the scopes of the enumeration literals and the record fields (scope 3 and 4), and last but not least the scope inside the with clause (scope 6). All scopes are simply numbered starting by 1.

```
MODULE m;                                   (* scope 1: module environment     *)
  VAR  a: CARDINAL;                         (* scope 2: objects of module m    *)
       e: (e1,e2);                          (* scope 3: enumeration literals   *)
       r: RECORD f1, f2: CARDINAL END;(* scope 4: record components            *)
  PROCEDURE p;
    VAR  b: CARDINAL;                       (* scope 5: objects of procedure p *)
    BEGIN (* body p *)
      b := 1;  e := e1; r.f1 := 1;
      WITH r DO f2 := 2 END;                (* scope 6: with statement         *)
    END p;
  BEGIN (* body m *)
    a := 0:
END m.
```

Fig. 1: Sample program with enumerations, records, and procedures

For each entry of the symbol table the following information is necessary: the definition scope (the number of the scope in which the identifier has been declared), an adjacent scope (a scope associated with this identifier such as the scope 4 of the record components with the identifier r), and a list of scopes in which this identifier is visible (normally differentiated in read and write visibility). The identifier table for this example is given in Figure 2.

In this example three types of scopes are represented: the scope 2 is impermeable to objects in both directions (no export, no import), scope 3 is permeable in both directions, and scope 5 is permeable for imports only (import means in this context read visibility, export write visibility).

Entry	Def. Scope	Adj. Scope	Read Visibility	Write Visibility
m	1		{1}	{1}
a	2		{2, 3, 5, 6}	{2, 3}
e	2	3	{2, 3, 5, 6}	{2, 3}
el	3		{2, 3, 5, 6}	{2, 3}
e2	3		{2, 3, 5, 6}	{2, 3}
r	2	4	{2, 3, 5, 6}	{2}
f1	4		{4, 6}	{4}
f2	4		{4, 6}	{4}
p	2		{2, 3, 5, 6}	{2, 3}
b	5		{5, 6}	{5, 6}

Fig. 2: Extended symbol table to the program of Fig. 1

Immediately it is evident that we can reduce the symbol table to the columns entry, definition scope, and adjacent scope, and to store the visibility rules in an extra table (s. Figure 3) with one entry for every scope. If the entry is found in the symbol table, the definition scope will be read, and by looking in the extra table with this entry the decision about read/write visibility can be made: the identifier is visible, if the current scope number is a member of the set of read/write visible scopes. This test is very effective, if the visibility sets are represented in form of bitsets.

Objects of Scope	are read visible in scopes	are write visible in scopes
1	{1}	{1}
2	{2, 3, 5, 6}	{2, 3}
3	{2, 3, 5, 6}	{2, 3}
4	{4, 6}	{4}
5	{5, 6}	{5, 6}

Fig. 3: Visibility table of the program in Fig. 1

Some more simplifications are possible but should be part of real implementations.

3. Overloading Resolution

The most programming languages allow overloading of identifiers: an object with the same name has different meanings in different scopes (the other case of 'true' overloading will not be handled here!). Figure 4 shows a sample program with such situation.

Another interesting situation is included in this little program - the import of single objects through module bounds. There is a simple solution for this import problem: declaration of a second instance of the imported object as a module object and to use the overloading resolution schema already available. This technique simplifies the algorithm substantially.

Overload resolution is done by the following method:

- All symbol table entries with the same name are linked together.
- The first entry of this chain is associated with a list of pairs (scope, symbol table entry), usable for the decision which symbol table entry is valid in the current scope. The updating of this list can be delayed until an access to an entry not used in an inner scope until

now takes place indeed.

```
MODULE m;                                (* scope 1: module environment     *)
    VAR    a, b: CARDINAL;               (* scope 2: objects of module m    *)
    PROCEDURE p;
        VAR    a: CARDINAL;              (* scope 3: objects of procedure p *)
        MODULE n;
            IMPORT a;                    (* scope 4: objects of module n    *)
            BEGIN (* body n *)
                a := 0;                  (* a: imported from p              *)
        END n;
        BEGIN (* body p *)
            a := 1;                      (* a: declared in p                *)
            b := 1;                      (* b: declared in m                *)
    END p;
    BEGIN (* body m *)
        a := 2;                          (* a: declared in m                *)
END m.
```

Fig. 4: Sample program with overloaded objects

The resulting data structures are shown in the Figures 5 and 6.

Entry Number	Entry Name	Definition Scope	Adjacent Scope	Next Overloaded	Overload Table Entry
1	m	1			
2	a	2		5	1
3	b	2			
4	p	2			
5	[a]	3		7	
6	n	3			
7	<a>	4			

Fig. 5: Symbol table to the example of Fig. 4
[a] means: doubly declared entry, <a>: alias declaration

Overload Table Entry	Read visible symbol table entries	Write visible symbol table entries
1 (from a)	{1: not; 2: 2; 3: 5; 4: 7}	{1: not; 2: 2; 3: 5; 4: 7}

Fig. 6: Overload resolution table to the program of Fig. 4

4. Description of the Abstract Data Type

The algorithm based on the ideas described above will be formulated in the following using a MODULA-2 - like style. First some data structures must be defined (Fig. 7). In a real implementation dynamic instead of static data structures can be used without essential changes. An appropriate initialization of all variables is presupposed.

The following variables describe the actual analysis state. The lexical analysis writes the name of the current identifier in the variable *CurrentIdentifier*, *CurrentScope* is the number of the scope currently processed, and *CurrentEntry* is the entry in the identifier table (or 0 if not found).

In Fig. 8 the algorithms for scope handling are given. The procedure *OpenScope* prepares a new scope and handles the permeability to the surrounding scope. The other two procedures switche to another scope (typically to a 'adjacent' scope) resp. extend the visibility of a scope to another (in addition to the visibility rule defined in *OpenScope*.

```
VAR     HashTable:          ARRAY [0..MaxHash-1] OF CARDINAL;
        IdentifierTable:    ARRAY [1..MaxIdentEntry] OF RECORD
                                NextHash:          CARDINAL;
                                NextOverloaded:    CARDINAL;
                                OverloadEntry:     CARDINAL;
                                DefinitionScope:   CARDINAL;
                                AdjacentScope:     CARDINAL;
                                NameOfEntry:       STRING;
                                AttributList:      ARRAY [ ... ] OF ... ;
                            END;
        ScopeTable:         ARRAY [1..MaxScopeNumber] OF RECORD
                                FatherScope:   CARDINAL;
                                VisibleScopes: ARRAY [readvisible..writevisible] OF
                                                   SET OF [1..MaxScopeNumber];
                            END;
        OverloadTable:      ARRAY [1..MaxOverloadEntry] OF
                                ARRAY [readvisible..writevisible] OF
                                    ARRAY [1..MaxScopeNumber] OF CARDINAL;
        OverloadedScopes:   ARRAY [1..MaxOverloadEntry] OF
                                SET OF [1..MaxScopeNumber];
        CurrentIdentifier:              STRING;
        CurrentEntry, FirstEntry:       CARDINAL;
        CurrentScope, LastUsedScope:    CARDINAL;
```

Fig. 7: Global vaiables and data structures

```
PROCEDURE OpenScope(EnvReadVisible,EnvWriteVisible:BOOLEAN);
    BEGIN
        INC (LastUsedScope);
        WITH ScopeTable[LastUsedScope] DO
            FatherScope := CurrentScope;
            IF EnvReadVisible THEN VisibleScopes[readvisible] :=
                    ScopeTable[CurrentScope].VisibleScopes[readvisible] END;
            INCL (VisibleScopes[readvisible], LastUsedScope);
            ... (* the same for EnvWriteVisible *)
        END;
        CurrentScope := LastUsedScope;
END OpenScope;
PROCEDURE SwitchScope (NewScopeNumber: CARDINAL);
    BEGIN CurrentScope := NewScopeNumber; END SwitchScope;
PROCEDURE ExtendCurrentScope (ToScopeNumber: CARDINAL;
        (EnReadVisible, EnvWriteVisible: BOOLEAN);
    BEGIN
        IF EnvReadVisible THEN
            INCL (ScopeTable[ToScopeNumber].VisibleScopes[readvisible], CurrentScope);
            INCLSET (ScopeTable[CurrentScope].VisibleScopes[readvisible],
                    ScopeTable[ToScopeNumber].VisibleScopes[readvisible]);
        END;
        ... (* the same for EnvWriteVisible *)
END ExtendCurrentScope;
```

Fig. 8: Scope handling procedures

It can immediately be seen that all scope operations are very trivial. Especially the very often used switch operation (access to record components!) is reduced to a simple number assignment.

The following search operation seems to be a little more expensive. But disregarding the hash search cycle the most other program parts are straight-forward with the only exception mentioned above (overload resolution).

```
PROCEDURE SearchIdentifier (Visibility: CARDINAL);   (* readvisible | writevisible  *)
    VAR   HashCode:            CARDINAL;
          PotentialEntry:      CARDINAL;
          CurrentOverloadEntry: CARDINAL;
    BEGIN                                              (* examination of hash chain*)
        HashCode := ComputeHashCode (CurrentIdentifier);
        CurrentEntry := HashTable [HashCode];
        LOOP
            IF CurrentEntry = 0 THEN RETURN END;(* nothing found with this name *)
            IF CurrentIdentifier = IdentifierTable [CurrentEntry].NameOfEntry THEN
                EXIT;                          (* an entry with this name was found*)
            ELSE
                CurrentEntry := IdentifierTable [CurrentEntry].NextHash;
            END; (* if *)
        END; (* loop *)
        FirstEntry := CurrentEntry;                (* store for later use in a global var.  *)
        WITH IdentifierTable [CurrentEntry] DO     (* decide visibility of this entry       *)
            IF OverloadEntry = 0 THEN
                IF NOT
                 (DefinitionScope IN ScopeTable [CurrentScope].VisibleScopes [Visibility])
                    THEN CurrentEntry := 0 END; (* found entry not visible: no success*)
                RETURN;                            (* found entry visible: success!        *)
            ELSE                                   (* decision by overload resolution     *)
                PotentialEntry := OverloadTable [OverloadEntry,Visibility, CurrentScope];
                IF PotentialEntry = MAXCARD THEN
                    CurrentEntry := 0; RETURN;     (* sorry: unvisibility is proved        *)
                ELSIF PotentialEntry > 0 THEN
                    CurrentEntry := PotentialEntry; RETURN;(* visibility was proved      *)
                END;
            END (* if *)
            CurrentOverloadEntry := OverloadEntry;
        END; (* with *)
        LOOP                                          (* visibility was not decided yet *)
            IF  OverloadedScopes[OverloadEntry] AND
                ScopeTable[CurrentScope].VisibleScopes[Visibility]
                =   OverloadedScopes[OverloadEntry] AND
                    ScopeTable[IdentifierTable[CurrentEntry].DefinitionScope].
                        VisibleScopes[Visibility] THEN
             OverloadTable[CurrentOverloadEntry,Visibility,CurrentScope]:=CurrentEntry;
                RETURN;                          (* visible, overload table is updated  *)
            END;
            CurrentEntry := IdentifierTable [CurrentEntry]. NextOverloaded;
            IF CurrentEntry = 0 THEN RETURN END;(* sorry: truly nothing found!    *)
        END; (* loop *)
END SearchIdentifier;
```

Fig. 9: Search algorithm

The procedure *EnterIdentifier* has to update the hash chain, and - if used - the overload chain too. The global variable *FirstEntry* initialized in the search procedure with the first entry of the overload chain can be used for this task. Additionally the overload table (if there is an overload entry - possibly newly created) must be updated by the following statements:

```
    OverloadTable [IdentifierTable [FirstEntry].OverloadEntry, ivisible, CurrentScope]
         := CurrentEntry;     (* ivisible = readvisible, writevisible *)
    INCL(OverloadedScopes[IdentifierTable[FirstEntry].OverloadEntry],CurrentScope);
```

The situation can slightly vary, if an alias identifier should be entered. Further variations of the algorithm can be useful in an actual implementation to support other tasks of the compiler. More optimizations can be applied to the data structures to reduce the memory demands. For instance, inside a WITH-statement there is no need for a list of write visibilities because nobody will ask for it.

5. Application to MODULA-2

The most constructs of MODULA-2 can be handled very easily. The visibility problem of procedures is solved by entering the procedure scope in the read visibility list of the environment. Enumeration lists and record component lists have own scopes extended to the environment complete (for enumerations) resp. only in WITH-statements or temporary in qualified identifiers. Dependent from the application an aliasing of record fields in the WITH-scopes can be necessary. Export lists also have their own scope visible in the environment dependen on qualified or unqualified export, but in both cases they are additionally visible in the module scope. Imports (in local modules) are handled by entering an alias identifier.

A little more attention must be given to the import from separate compiled modules. All imported objects are collected in an extra scope, and this scope is completely visible in the scope of the importing module. In a sense this also is a type of aliasing but between different compilation units.

6. Conclusion

The use of visibility sets for every program region together with the concept of overloading resolution by direct access reduces the time complexity to a constant (and very small) value with only one exception: the time needed for updating the resolution tables. This situation is very seldom and then the needed time is linear to the number of identifiers with the same name (also usually a very small value) and naturally restricted by the number of scopes. The time complexity of the additionally necessary hash search has not yet been changed by the algorithm.

References

1. P. Fritzson: Incremental Symbol Processing.
   Research Report LiTH-IDA-R-88-09, Linköping University, Sweden, 1988
2. L.B. Geissmann: Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith. ETH Zürich, Diss. ETH No. 7286, 1983
3. S.L. Graham, W.N. Joy, O. Roubine: Hashed Symbol Tables for Languages with Explicit Scope Control. SIGPLAN Notices 14 (Aug. 1979), 50-57
4. R.T. House: Alternative Scope Rules for Block-Structured Languages.
   The Computer Journal 29 (1986) 3, 253-260
5. U. Kastens, W.M. Waite: An Abstract Data Type for Name Analysis.
   Research Report CU-CS-460-90, University of Colorado at Boulder, March, 1990
6. Masato Takeichi: Name Identification for Languages with Explicit Scope Control.
   Journal of Information Processing 5 (1982) 1, 45-49
7. W.M. Waite, G. Goos: Compiler Construction. Springer Verlag, New York, 1984