Implementing High-Level Identification Specifications

Arnd Poetzsch–Heffter
Institut für Informatik der TU München
Arcisstrasse 21, D–8000 München 2
poetzsch@informatik.tu-muenchen.de
fax: 49 89 2105 8180

Abstract

Identification is the task of finding the relation between used identifier occurrences and the declared entities of a program. The paper presents the techniques needed for the implementation of high–level identification specifications based on visibility rules. The underlying specification method is related to descriptions in language reports defining the identification in terms of validity and hiding of bindings in program ranges. This leads to shorter and better to read specifications. Sematics and specification properties of the method are already considered in [PH91b], [PH92]. Here, we present the new implementation techniques developed for the generation of compiler front–ends from high–level identification specifications. The implementation combines instantiation and partial evaluation of expressions with a generated global table mechanism.

1 Introduction

Identification is the task of finding the relation between used identifier occurrences and the declared entities of a program. Identification is usually described by a symbol table (or environment) mechanism that computes for each relevant point of a program the set of visible bindings between identifiers and program entities. This mechanism can easily be used in attribute grammar systems and is often supported by parameterized symbol table modules ([Rei83], [KW91]). Unfortenately, specifications based on the symbol table method tend to be hard to read, because symbol tables are often very complex data structures, needing a couple of enter– and lookup–procedures to handle the different kinds of program entities. Another draw back of the classical symbol table method is that it bounds together different, conceptually independent tasks of language processing (e.g. identification, typing, and storage allocation), as the symbol table is the only way to propagate information from declarations to applications. This prevents more modular specifications, and makes it more difficult to prove language properties.

To meet these draw backs, we developed a specification method for identification that is related to descriptions in language reports [PH91a]. The core of such an identification specification is a set of *visibilty clauses*. A visibilty clause specifies how a declaration influences the visibilty: Usually, a declaration makes valid the binding between the corresponding identifier and the declared entity in a certain range of a program and hides other bindings in a certain range. As we will see, this can directly be formulated with visibility clauses. This method leads to more flexible, shorter and better to read specifications (cf. section 2), has a simple logic–based semantics providing a good basis to prove language properties, and solves the modularity drawback sketched above (cf. [PH93]).

This paper presents a generative implementation technique for such high–level identification specifications. Thereby, such specifications can be directly used in the analysis phase of a compiler and serve as starting point of a tool–based, stepwise refinement of compilers for realistic programming langugaes. The main difference to related approaches (see below) is that our *specification technique* is not based on a table mechanism. The data structures and algorithms for identification are automatically generated from such specifications.

Related Work The work is related to many well–known works in the field of static semantics specification and implementation. It can be understood, as closing the gap between purely specification oriented works (like that of Garrison [Gar87], Uhl [Uhl86], Odersky [Ode89]) and those works looking for general symbol table models that provide standard techniques being more abstract than hand coded symbol tables (like that of Reiss [Rei83] or Kastens and Waite [KW91]). It is as well interesting as basis for executable specifications of programming language semantics (cf. e.g. [Gan85], [Pal92]).

Paper Overview The paper is organized as follows: In section 2, we explain how a identification specification according to our method looks like, and sketch its semantics. Section 3 presents the developed generative implementation techniques for such specifications. Section 4 gives the conclusions.

2 Visibility and Identification

This section gives a short introduction in the developed method for identification specification (for more details see [PH91b]). The illustrating specification will be used as well to explain the implementation techniques in the following sections. As example, we consider a simple **b**lock–structured **i**mperative **l**anguage, called BIL, allowing to express declaration and use of variables. The abstract syntax of BIL is given by the following productions:

```
Block              (  DeclList  StatList  )
DeclList           *  VarDecl
VarDecl            (  Ident  TypeSpec  )
TypeSpec           =  TypeBool  |  TypeInt
TypeBool           ()
TypeInt            ()
StatList           *  Stat
Stat               =  Assign  |  Block  |  ...
Assign             (  Ident  Exp  )
Exp                =  ArithmExp  |  Int  |  Bool  |  Ident
ArithmExp          (  Exp  Exp  )
```

We use three kinds of productions: *tupel, list*, and *class productions.* Each production defines one *sort*, denoted by the name on the left–hand side of the production. Tupel productions are denoted by enclosing the right–hand side in parentheses. List productions are denoted by an `*`–symbol. In contrast to purely order–sorted term algebras, we use "flat" lists, i.e. a list term may have an arbitrary number of subterms; this has shown to be much more convenient for our framework. The class productions can be understood as defining a sort that is the "union" of the *subsorts* on the right–hand side. The sorts `Int`, `Bool` and `Ident` are predefined.

2.1 Program Representation

The abstract syntax constitutes the interface to the parser; i.e., for each program being correct w.r.t. the concrete syntax, the parser provides a term according to the abstract syntax. If we only have order–sorted term representations of programs, we cannot express global relations between distant parts of a program, as e.g. the function that yields for each identifier occurrence the corresponding declaration. To overcome this problem, we enrich each program term by the set of its occurrences or, as we call it, its *nodes.* Furthermore, we introduce for each node two additional elements to represent the *program points* before and after the node. Program points are mainly used to model program *ranges*, that are parts of a program relevant for visiblity. Altogether, we get the following syntax tree representation:

- the set of syntax tree nodes and the set of its program points;
- functions yielding the father of a node (`fath`), the first son (`fstson`), the left and right brother (`lbroth,rbroth`), and a constant yielding the root; in cases where these functions are not defined they yield the extra element `nil`;
- a function `term` yielding for each node the corresponding order–sorted term; in particular, `term` applied to a leaf node returns the terminal value of the leaf (identifier, integer, ..).
- functions yielding the point before a node (`before`) and after a node (`after`).

The sorting on nodes can be imported from the corresponding terms: For example, let n be a node such that the corresponding term is of sort `Block`; then we say n is of *node sort* `Block@`.

2.2 Specification of Identification

Identification is its program pointsspecified in two steps. The first step defines the so–called visibility predicate, the second step the identification function.

Visibility The visibility (predicate) declaration consists of a head line defining the parameter sorts, and a list of so–called *visibility clauses*[1]. Here is the visibility declaration for our example language BIL:

[1]For BIL, we only need one visibility clause, as it only contains variable declarations; in realistic languages we would have several clauses.

```
VIS  is_visible [ Ident, VarDecl@ , Point ]:
   VarDecl@ V :
      VALID   id(V), V
      FROM    after( V )  TO  after( encl_block(V) )
      HIDDEN  id(V), OUTERV:  desc[ encl_block(V), encl_block(OUTERV) ]
      FROM    before( encl_block(V) ) TO  after( encl_block(V) )
```

The visibility predicate for BIL, named `is_visible`, takes three parameters: an idententifier, a program point, and a declaration `VarDecl@` to represent the program entities that can be visible at program points. The visibility clause consists of a *quantification part*, a *V–clause*, and an *H–clause*. The quantification parts tell which language constructs influence the visibility, the V– or H–clauses tell which bindings are valid/hidden in which program range. Thus, the above visibility clause for BIL can be read as follows: A variable declaration makes valid the binding between its identifier and itself in the range from the point after the declaration up to the point after the directly enclosing block; a declaration d hides all bindings between d's identifier and those declarations that are declared in an outer block; the hiding ranges from the *beginning* of the directly enclosing block to its end. That is, according to scope boundaries, BIL adopts PASCAL's visibility rules. For C or Ada like hiding, the hiding range starts at the point before the declaration.

Visibility is then defined as usual: A declaration d is said to be *visible* at a program point p under identifier id, iff the binding (id, d) is valid at p and not hidden[2]. The auxiliary functions `id(X)` and `encl_block(X)` are just abbreviations for `term(fstson(X))` and `fath(fath(X))` respectively. The predicate `desc`endant is predeclared.

Identification The result of the identification will be a function from identifier occurrences to declarations. Thus, in BIL we get a function from `Ident`–nodes to variable declaration nodes. This function is specified using the determination operator `THAT`. The determination operator yields the element that satisfies the corresponding condition, if that element is uniquely determined, and otherwise the extra element `nil`. For BIL, we have:

```
IDENTIFICATION  declof( Ident@ ID ) THAT VarDecl@ V :
       is_visible[ term(ID), V, before(ID) ]
```

As in the example, the body of an identification declaration may contain an application of the visibility predicate such that neither the first nor the third actual parameter may depend on the identifier bound by the determination operator; this is, of course, always fulfilled in practice. That there has to be a declaration for each applied identifier, must be formulated by a context condition. How this is done in a declarative way, is out of the scope of this paper (see [PH93]).

[2]This is precisely the semantics formally described in [PH92]. The semantics given in [PH91b] is a bit more comfortable, as it has a built–in block structure mechanism; but it is less general and harder to implement.

3 How to Generate Identification Implementations

This section presents the generative implementation of identification specifications. The main idea is as follows: During semantic analysis, the generated analyser first computes a global table data structure representing the visibility aspects of the input program. Then, on the basis of this table, an array representation of the identification function is computed. This identification function makes available the result of the identification for succeeding compiler phases. (Thus, the table data structure can be freed after identification.)

3.1 Evaluating the Visibility Clauses

In order to compute the identification function, we have to know all declarations that are visible under a given identifier at a given program point. This information is provided by a table data structure that is constructed during evaluation of the visibility clauses. How these *visibility tables* look like, is shown by figure 1: For each identifier, we have a list of *disjoint* program ranges represented by their starting and end point. Attached to a program range are those declarations that are visible at all points contained in the range. Using this data structure, we can enumerate all declarations visible under identifier *id* at point *p* by looking whether *p* is contained in a range of the range list belonging to *id*. If this is the case, enumerate the attached declarations, otherwise there are none. (Of course, in practice nobody would use a range list, but an appropriate tree structure to represent the ranges. We choose lists here and in the following only to concentrate on the main aspects).

The visibility table is constructed in two steps. In the first step, we construct an *auxiliary data structure* that allows to find out where an identifier–declaration–binding is hidden. The second step computes the visibility table.

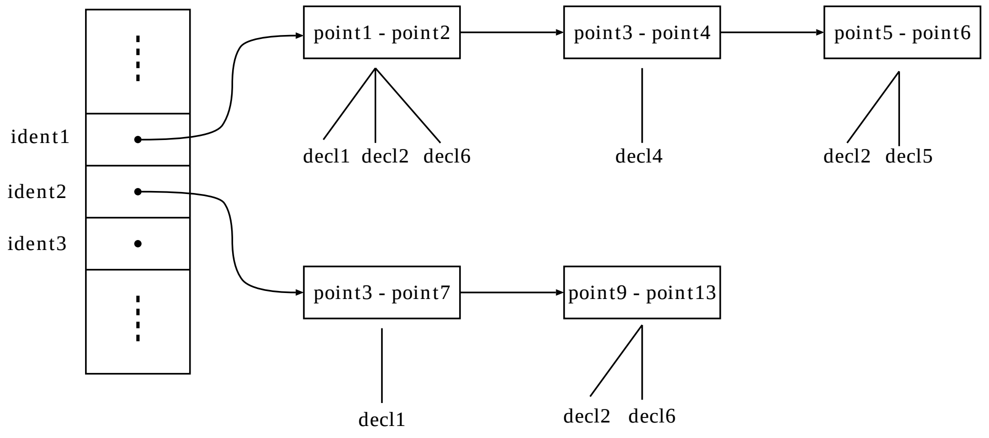

figure 1

First Step: The auxiliary data structure representing the hiding information consists of a list of disjoint program ranges. Each of these program ranges has attached a list of *hiding pairs* expressing the sets of bindings that are hidden in the range. A hiding pair *HP* consists of a declaration `D` and a check procedure allowing to check whether a given declaration is hidden by `D` in the range *HP* is attached to. For brevity, we have to ommit details here (cf. [PH91a]).

Second Step: In the second step, the visibilty table is constructed (cf. figure 1). The main structure of the algorithm is as follows:

```
allocate an array [1,$maxident$] and initialize the entries to empty_list ;
for each visibility clause $VC$ having a V–clause :
    for each element $E$ of the node sort in the quantification part of $VC$ :
        compute the starting and end point $(SP,EP)$ corresponding to $E$ ;
        compute the identifier–declaration–binding $(ID,D)$ corresponding to $E$ ;
        let $vis\_ranges$ = $eliminate\_hidden\_parts(ID, D, (SP, EP))$ ;
        for each range $R$ in $vis\_ranges$ :
            let $ranges$ be the entry for $ID$ in the visibility table ;
            insert $(D, R)$ into $ranges$ ;
```

Thereby, visibility ranges of an identifier–declaration–binding *(ID,D)* (cf. the sixth line of the algorithm) are computed by eliminating all parts from the validity range *(SP,EP)* in which *(ID,D)* is hidden. These parts are figured out using the auxiliary data structure for the hiding information sketched above.

To get a fast element enumeration in the for–loop for the quantification part, we code the tree nodes in such a way that all nodes of a node sort are continuously coded (for details see [PH93]).

3.2 Computing the Identification Function

In order to have an efficient implementation of the identification function for succeeding phases in the specified language processor, we compute an array having nodes of identifier occurrences as indices and the corresponding declarations as entries. To do so, we code the tree nodes in such a way that all identifier nodes are contained in a compact integer intervall (for details see [PH93]). Thus in BIL, we get an array $[min_{\texttt{Ident@}} .. max_{\texttt{Ident@}}]$ of sort `VarDecl@`. As the body of the identification declaration has to contain an application of the visibility predicate where neither the first nor the third actual parameter may depend on the searched declaration, we can evaluate these parameters given an identifier occurrence *occ*. Let us call the results *id* and *point*. By a lookup with *id* in the visibility table, we get a range list. As these ranges are disjoint, there is at most one range r containing *point*. If there is none, set the value for *occ* to `nil`. Otherwise check for all declarations attached to r whether they satisfy the rest of the identification declaration body. If there is exactly one such d, the value of *occ* is set to d; otherwise to `nil`. Obviously, the method is sufficiently powerful to handle overloading by selecting the declaration in question using additional information; this is explained in the long version of this paper (contact the author).

4 Conclusions and Future Research

The paper presented the fundamental implementation concepts for identification specifications based on visibility rules. Combined with an attribute based compiler specification system (cf. [PH93]), the presented method provides a very flexible high–level specification tool allowing

- formal specifications following the paradigm of informal language reports, so that the reader can easily switch between the informal and formal description;
- execution and prototyping of realistic full size languages; thereby, providing a starting point of a stepwise refining development of compiler front–ends.

Another strength of the presented method is that it enables high–level representation of identified programs by syntax graphs interpreting the identification function as additional edges in the syntax tree. So, table data structures at the interface between front– and back–ends can be avoided; in this respect the approach follows the lines of the Ada intermediate language DIANA [GWEB87].

References

[Gan85] H. Ganzinger. Modular first-order specifications of operational semantics. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, LNCS 217, 1985.

[Gar87] P. E. Garrison. *Modeling and Implementation of Visibility in Programming Languages.* PhD thesis, University of California, Berkley, 1987.

[GWEB87] G. Goos, W. A. Wulf, A. Evans, Jr., and K. J. Butler, editors. *DIANA An Intermediate Language for Ada.* LNCS 161. Springer-Verlag, 1987.

[KW91] U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica 28*, 1991. Kopiensammlung.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1989. Diss. ETH No. 8938.

[Pal92] J. Palsberg. A provably correct compiler generator. In *ESOP'92 Symposium on Compiler Construction*, 1992.

[PH91a] A. Poetzsch-Heffter. *Formale Spezifikation kontextabhängiger Syntax von Programmiersprachen.* PhD thesis, Technische Universität München, July 1991.

[PH91b] A. Poetzsch-Heffter. Logic-based specification of visibility rules. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, LNCS 528, pages 63–74. Springer-Verlag, 1991. Kopiensammlung.

[PH92] A. Poetzsch-Heffter. Identification as programming language principle. Technischer Bericht TUM-I9223, Technische Universität München, July 1992. (ausführliche Version zu Poetzsch-Heffter, A.: 1992).

[PH93] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX System. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, LNCS 714, pages 137–150. Springer-Verlag, 1993.

[Rei83] S. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):127–163, April 1983. Kopiensammlung.

[Uhl86] J. Uhl. *Spezifikation von Programmiersprachen und Übersetzern.* GMD-Bericht 161. R. Oldenbourg Verlag, 1986.