Single-pass Static Semantic Check for Efficient Translation in YAPL

Zafiris Karaiskos, Panajotis Katsaros and Constantine Lazos

Department of Informatics, Aristotle University
Thessaloniki, 54124, Greece
email: {karaisko, katsaros, clazos}@csd.auth.gr

Abstract. Static semantic check remains an active research topic in the construction of compiler front-ends. The main reasons lie into the ever-increasing set of semantic properties that have to be checked in modern languages and the diverse requirements in the timing of checks during the compilation process. Challenging single-pass compilers, impose the development of appropriate parse-driven attribute evaluation mechanisms. LR parsing is amenable to L-attributed definitions, where, in every syntax tree, the attributes may be evaluated in only one left to right depth-first-order traversal. In such definitions, for every production of the grammar, the inherited attributes on the right side depend only on attributes to the left of themselves. When the LR-parser begins to analyze a word for a non-terminal, the values of its inherited attributes must be made available. This outstanding difficulty constitutes the major concern of the existing attribute evaluation mechanisms, found in the literature. We review them and then we introduce the evaluation mechanism, which we successfully utilized in YAPL, a complete programming language build exclusively for educational purposes. Our approach takes advantage of an effective attribute-collection management scheme, between a parse-driven symbol node stack and the symbol table, for delivering synthesized and inherited attribute values. Finally, we demonstrate the applicability of the suggested approach to representative problems of enforcing language rules for declaration and scope related properties and in type checking of assignments.

Index terms – Compilers, Programming Languages, Static Semantic Check

1 Introduction

The main problem in parser controlled attribute evaluation is the design of an efficient mechanism for delivering the inherited attributes involved in semantic check computations. In LR parsing, these values may be only determined, when the grammar rule to be used becomes known. Thus, all attribute evaluation actions, should be associated, with reductions of the LR-parser.

Let us consider an LR-state item

$$[X_0 \rightarrow X_1 X_2 ... \cdot X_i ... X_n]$$

with a non-terminal X_i. Then, one possible interpretation for this state is that a word for X_i is next to be analysed. To evaluate the attributes in the sub-tree of X_i, which must then be

constructed, the values of the inherited attributes of X_i must be available. Since, by assumption, the attribute grammar is L-attributed ([5], [6]), we may suppose that all arguments of the inherited attributes of X_i are somehow available in the parsing stack.

Yacc-like parser generators, support the use of the `%union` declaration for defining possible data types of the symbols pushed into and popped out of the parsing stack. In addition, access to non-positive positioned symbols (`$0`, `$-1` etc.) is allowed, for obtaining the arguments of the inherited attribute computations. However, an important limitation of the Yacc maintained parsing stack, is that due to its fixed access scheme, there is no way to alter symbol values and push them back into the stack.

Alternatively, in [7] the authors suggest the use of a stack made by lists of attributes and maintained, in parallel to the parsing stack. However, this complex structure is still non applicable to grammars, where the positions of the arguments to be used in inherited attribute computations, may not be predicted (see [1] for such an example grammar). Thus, a new non-terminal, say *N*, is suggested to be placed before X_i and a new production rule $N \rightarrow \varepsilon$ to be added to the grammar. If reduction to this non-terminal *N* occurs, the attribute evaluator obtains the arguments for the inherited attribute computations at stack positions, addressed relative to the top of the stack, evaluates the attributes and leaves the values behind, at the top of the stack. In any case, this transformation yields attribute values, placed in known stack positions.

It is important to note that a similar behavior in respect to the parsing stack is also supported, by the Yacc generated parsers, when using actions embedded into the grammar rule productions. However, this approach could even lead to the transformed grammar to exhibit artificially introduced parsing conflicts, except of the more restricted class of the LL(1) based L-attributed definitions. In response to this limitation, Wilhelm & Maurer suggest, in [7], a complicated LR-property preserving grammar transformation, based on the previously described attribute lists stack.

In this work, we introduce the static semantic check mechanism that we have successfully implemented in YAPL (Yet Another Programming Language). YAPL is a programming language build exclusively for educational purposes:

- it supports a rich set of C-like language constructs (almost all the widely used),
- it uses four instances of a carefully designed symbol table structure, for the efficient delivery of inherited attribute values (one for variables and functions, one for structures and unions, one for enumerations and the last one for control flow labels),
- its semantic check mechanism is based on an effective composite data structure, which provides a means for by-passing the need for embedded actions and their aforementioned side-effects.

Next section describes the basic symbol table and the overall attribute-collections management structure. In section 3, the introduced semantic check mechanism is applied to problems of enforcing language rules, for declaration and scope related properties. Section 4 demonstrates the applicability of the suggested approach, in a representative type checking problem and the paper concludes with a summary of its main advantages and a short note on its future research prospects.

2 Static Semantic Check in YAPL

The basic symbol table structure (Figure 1), is based on the use of a separate chaining Hash table, in conjunction with a composite symbol node cross-link structure that connects all symbols recognized in the same nested scope level.

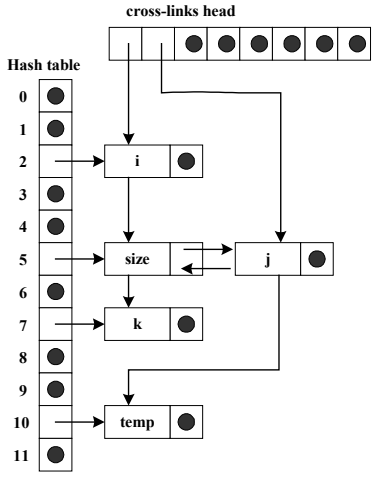

Fig. 1. The basic symbol table structure in YAPL.

Each symbol node is expressed as an attribute-collection, double linked to a specific Hash table chain. At the same time, it is also linked to another chain, associated with the cross-link head of the nested scope level, where the symbol was recognized. The hash function used in our compiler is the one suggested in [1].

YAPL' s extensible symbol table structure supports the use of highly complex C-like declarations ([4], [3]). Focusing in variable declarations, each one of them is composed of two parts: the specifier, which is basically a list of keywords (int, long, extern, struct etc.) and the declarator that is made up of the variable's name and an arbitrary number of stars, array-size specifiers and/or parentheses (used both for grouping and as a reference to a function). There is only a limited number of legal keyword combinations for the specifier and they are expressed based on a fixed number of entry fields in the attribute-collection, referred to the corresponding symbol (sclass, type, user_type, short, sign). In Figure 2, we present the attribute-collection corresponding to the array of pointers declaration:

```
static unsigned short int *ap[4];
```

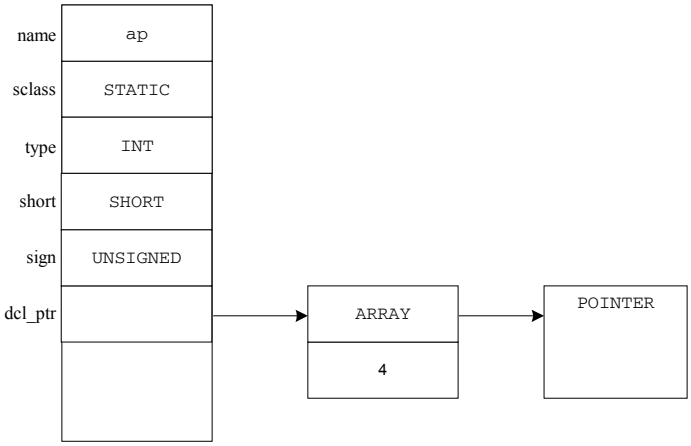

Fig. 2. Symbol node attribute-collection structure.

The array bracket, indirection and structure member selection operators generate temporary variables, whose types may be derived from the original type representation, possibly, by removing the first element of the declaration list. As an example consider the expression `ap[2]` that generates a temporary variable represented as in Figure 3.

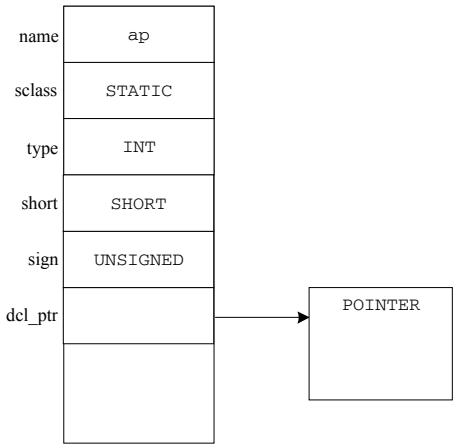

Fig. 3. Temporary attribute-collection corresponding to a pointer array element.

The overall mechanism is completed by a parse-driven symbol node stack, which is maintained in parallel to the Yacc generated parsing stack, for delivering synthesized attribute-collections.

3 Declarations

There may be more than one independent declarations of the same name, in different parts of the source program. The portion of the program, where a declaration applies, is the scope of that declaration. Language scope rules determine which declaration of a given name is applied, when the name appears in the program text.

In YAPL, we have implemented four types of scope (as in C):

- Function scope, which applies only for label names, followed by ":" and a statement. The label name must be unique within a function and may be referenced anywhere within that function.
- File scope, for declarations placed outside of any block that are not part of a parameter declaration. File scope extends to the end of file.
- Block scope, for declarations placed within a block, extended to the portion of the program between "{" and the right brace "}" that terminates the block.
- Function prototype scope, for names contained within the list of parameter declarations in a function prototype that is not part of a function definition. Function prototype scope extends to the end of the function declarator.

The linkage property forces a name that is declared in more than one scope or more than once in the same scope, to refer to the same object or function. We discriminate three types of linkage: external, internal and no linkage. The type of linkage for a particular name depends on whether it is a function or an object and on its storage class. Table I summarizes the scope linkage properties that have been implemented in YAPL.

STORAGE CLASS	FILE SCOPE	BLOCK SCOPE
None	If the declared name is a function, then it has the same linkage as any visible file scope declaration of that name or, if none exists, it has external linkage. Names representing any other type of object (variable, struct etc.) have external linkage.	No linkage.
Extern	The name being declared has the same linkage as any visible file scope declaration of that name or, if none exists, it has external linkage.	As in file scope.
Static	Internal linkage.	No linkage.

Table 1. Scope Linkage.

The suggested semantic check mechanism delivers the described declaration related properties, as following:

- The parser actions create symbol attribute-collections, which are appropriately updated and pushed into the parse-driven symbol node stack (not the Yacc generated one).
- For a list of declarators, a separate attribute-collection is created for each of them and they are all connected by their cross-link dedicated attribute. Only the last one remains in the stack.
- When parsing proceeds with a declaration statement reduction, then two attribute-collections are popped out of the stack: the one that corresponds to the specifier part and

the one that corresponds to the declarator that was last encountered. The non-null valued attributes of the specifier part are copied to the corresponding attributes of the delivered declarator and propagated by its cross-link, to the previously encountered declarators, contained in the same statement. Thus, by delaying all the attribute-collections updates up to the declaration statement reduction, we avoid non-positive positioned stack accesses or use of embedded actions, with possibly undesirable side effects.

- The following static semantics is then checked:
  i. only one storage class may be specified,
  ii. only one type may be specified,
  iii. if the type is STRUCT, UNION or ENUM, the corresponding attribute values can not be short, long, signed or unsigned,
  iv. if there is a user-defined type, the respective attribute value can not be short or long,
  v. if there is a character type, the respective attribute value can not be short or long,
  vi. if there is a float or a double type, the corresponding attribute values can not be short, long, signed or unsigned and
  vii. if there is a void type, the corresponding attribute values can not be short, long, signed or unsigned.
- The properties of Table I, are then enforced, together with other declaration specific semantic properties. These include top level or local variable declaration semantics, structure and union declaration semantics, enumeration declaration semantics, function definition and declaration semantics accompanied by appropriate compatibility checks and label declaration semantics. In this paper, we do not proceed to the detailed description of them, due to space limit reasons.
- Finally, the declared names' attribute-collections are inserted into the appropriate symbol table instance, for later use.

4 Type Checking

The suggested attribute-collections representation made also feasible the incorporation of type compatibility checks, in an elegant way. In YAPL, the notion of type compatibility is summarized in the following language rules:

- Two arithmetic types are compatible only if they are the same type. If a type can be written using different combinations of type specifiers, all the alternate forms are considered to refer to the same type. Thus, the types `short` and `short int` are the same, but the types `unsigned`, `int` and `short` are all different.
- Each enumerated type definition yields a new integral type. There is no case, for two different enumerated types that are defined in the same source file, to be compatible.
- Two array types are compatible, if their element types are compatible. If both types specify sizes, the sizes must be the same. However, if only one of them specifies a size - or if neither do - the two types are still compatible.
- Two function types are compatible, if they define compatible return types.
- Each occurrence of a type specifier for a structure or a union type definition introduces a new structure or union type that is neither the same nor compatible to any other such type, in the same source file. Any type specifier that is a structure, union, or enumerated type

reference is the same type as the one introduced in the corresponding definition. The type tag is used to associate the reference to the definition and in this sense it may be considered as the name of the type. As an example we have the following:

```
struct S {char c; int i;} u;
struct S v;
```

- Two pointer types are compatible if they point to compatible types.

We proceed to the description of the type checks that are to be applied to the complicated assignment expression of the following code fragment:

```
typedef struct s2 {
    struct { int i, j; } arr[123];
    int b;
} strct_type;
strct_type met[30], drt;

void main(void)
{
    . . .
    met[14].arr[28].j=drt.b;
    . . .
}
```

In YAPL, an expression may either specify the computation of a value or designate an object or a function or generate side effects or combine any of them. It is expressed as a sequence of operators and operands that is possibly containing one or more balanced pairs of parentheses. Expressions are build-up from primary expressions, which may take the form "(" expression ")" or may be simply represented by an identifier or a literal. The evaluation order is affected by the operators' precedence and associativity properties, as well as the parentheses and it is defined exclusively in the language syntax.

Thus, type checking makes use of the operands' attribute-collections and proceeds in a parse-driven incremental fashion: starting with identifier and literal attribute-collections, each operator occurrence causes the generation of a new intermediate attribute-collection, carrying the results of the type compatibility checks applied to the operands. The whole type checking process is completed, when the parser encounters a statement's occurrence.

As regarding the expression contained in the previously stated code fragment, Figure 4 shows the attribute-collection corresponding to the identifier "`met`".

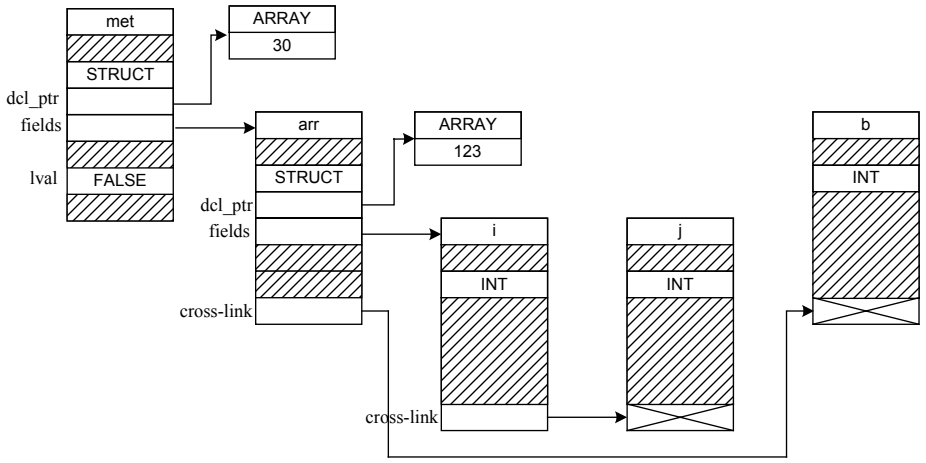

Fig. 4. The attribute-collection of the identifier "`met`".

The square-brackets operator is then encountered and applied to the identifier "`met`". However, as in C, the expression `met[14]` is equivalent to `*(met+14)` and results in enforcing first, the language rules for '+' and then, the rules associated to the indirection operator. This yields the lvalue attribute-collection shown in Figure 5.

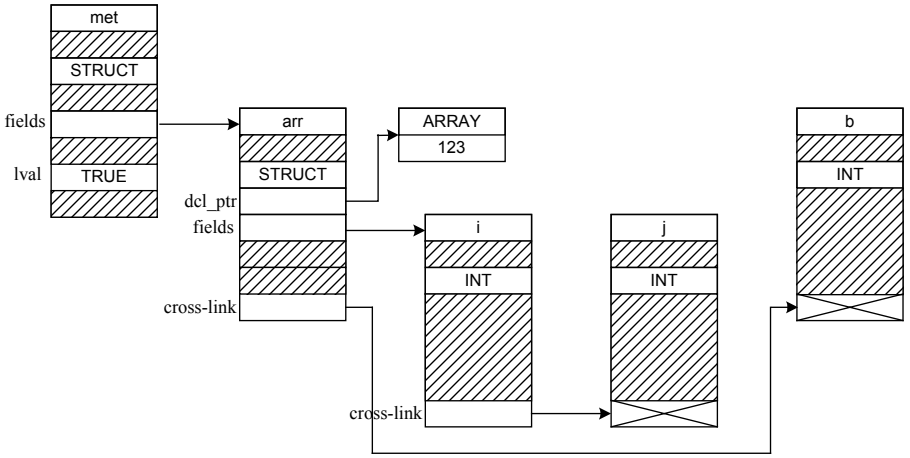

Fig. 5. The lvalue attribute-collection for the expression `met[14]`.

Next, the parser invokes the semantic actions associated to the dot operator, for checking the expression `met[14].arr`: the attribute-collection has to refer to a structure or a union type and the name next to the dot to coincide to one of its components (connected to each other by the cross-link attribute). The result is the named component shown in Figure 6.

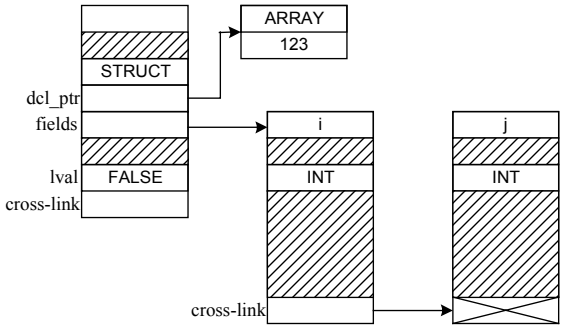

Fig. 6. The named component `met[14].arr`.

The type checking proceeds in the same way and concludes to the attribute-collections (Figure 7) for the operands `met[14].arr[28].j` and `drt.b` of the initial expression. The left-hand side attribute-collection corresponds to an lvalue and both of them refer to compatible types that happen to be permissible in an assignment operation. The expression's type check terminates successfully.

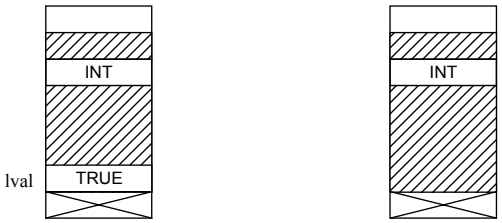

Fig. 7. Attribute-collections for the `met[14].arr[28].j` and `drt.b` operands.

5 Conclusion

This paper introduces a novel composite data structure that provides a means for bypassing the use of embedded actions that may cause undesirable side effects in the parsing procedure. The resulted semantic check mechanism is based on an effective attribute-collection management scheme, between a parse-driven symbol node stack and the symbol table, for delivering synthesized and inherited attribute values. The overall approach has been successfully applied in a C-like educational language compiler, named YAPL. For the whole language implementation,

- we did not make use of any non-positive positioned stack accesses and
- we used just a single embedded action, for changing the nested scope level, in compound statement blocks.

Finally, we demonstrate the use of the suggested mechanism for semantic checking of declaration related properties and in a representative types checking problem.

The only alternative found to be related, to the problems attacked, is the data structure, suggested in [7], together with a complicated LR-property preserving grammar transformation, to avoid the introduction of parsing conflicts.

Future research work will focus on two distinct directions, namely,

- the formal description of the introduced semantic check framework and
- the appropriate customization, for use in static semantic checks of object-oriented and concurrent programming language constructs ([2]).

References

1. Aho, A. V., Sethi, R. and Ullman, J. D. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
2. Baiardi, F., Ricci, L. and Vanneschi, M. Static type checking of interprocess communication in ECSP. In: *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices (ACM)*, 19 (6): 290-299, 1984.
3. Harbison, S. P. and Steele, G. L. Jr: C. A Reference Manual, 4th edition, Prentice-Hall, 1995.
4. Kernighan, B. W. and Ritchie, D. M. The C programming Language, 2nd edition, Prentice-Hall, 1988.
5. Knuth, D. E. Semantics of context-free languages, *Mathematical Systems Theory*, 2 (2): 127-145, 1968.
6. Knuth, D. E. Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory*, 5 (1): 95, 1971.
7. Wilhelm, R. and Maurer, D. Compiler Design, Addison-Wesley, 1995