Circular Attribute Grammars with Remote Attribute References and their Evaluators

Akira SASAKI and Masataka SASSA

Tokyo Institute of Technology
2-12-1 O-okayama, Meguro-ku, Tokyo 152-8552 JAPAN

sasaki@is.titech.ac.jp, sassa@is.titech.ac.jp

Received June 7, 2002

Abstract Attribute grammars (AGs) are a suitable formalism for development of language processing systems. However, for languages including unrestricted labeled jumps, like "goto" in C, optimizers in compilers are hard to write in AGs. This is due to two problems which few previous researches could deal with simultaneously, i.e., references of attribute values on distant nodes and circularity in attribute dependency. This paper proposes *circular remote attribute grammars* (CRAGs), an extension of AGs which allows (1) direct relations between two distant attribute instances through pointers referring to other nodes in the derivation tree, and (2) circular dependencies under certain conditions including those which arise from remote references. This extension gives AG programmers a natural way to describe language processors and programming environments for languages including any type of jump structures. We will also show a way to construct an efficient evaluator for CRAGs called a *mostly static evaluator*. Performance of the proposed evaluator has been measured and compared with dynamic and static evaluators.

Keywords Attribute Grammars, Circular Attribute Grammars, Attribute Evaluators, Declarative Languages, Compiler Construction.

§1 Introduction

Many applications of attribute grammars (AGs)[?] are proposed because of the advantages of its formalism in that the description of grammars is simple and evaluators can be generated mechanically from the description. However, as for pure AGs, relatively few applications are practically used. One reason for this is its strictly restricted description style that we point out in the following.

First, AGs that include circularly defined attributes were considered meaningless, or not well-defined. Applications which inherently lead to circular dependencies, for example data-flow problems which are solved well by recursive equations, were hard to write in pure AG descriptions. To overcome this problem, Babich and Jazayeri,[?, ?] Farrow[?] and Jones[?] proposed extended AGs that allow attribute instances to have circular dependencies.

Second, in traditional AGs, a semantic rule in one production must be defined only from the attributes that occur in the same production. This restriction causes difficulty in writing relations between attribute instances belonging to two distant nodes. Many researches cope with this problem by adding schemes for non-local dependencies or remote references, that is, allowing attributes to refer to values of attributes in far away nodes [?, ?, ?, ?, ?, ?]. However, there has been few published researches that could remove the above two restrictions simultaneously, i.e. circular dependencies and remote references. For example, data-flow problems on languages including unrestricted labeled jumps such as "goto"s were hard to describe because of the circular dependencies and references to distant nodes. Boyland in his PhD work implemented a dynamic evaluator that can treat this problem[?], but static evaluation strategies are not shown.

To solve the above mentioned problems, this paper proposes *circular remote attribute grammars* (CRAGs), an extension of AGs which allows (1) direct relations between two distant attribute instances through links referring to other nodes in the derivation tree, and (2) circular dependencies under certain conditions including those which arise from remote references. This extension gives AG programmers a natural way to describe language processors and programming environments for languages including any type of jump structures. For simplicity, we assume that such links between nodes are already established.

We also propose an efficient evaluator for CRAGs called a *mostly static evaluator*. The evaluator consists of a preprocessor and a main driver based on a static evaluator for absolutely non-circular AGs.[?] The preprocessor collects remote attribute information which could not be obtained at the evaluator generation time in general. By incorporation of the preprocessor, the mostly static evaluator can avoid redundant computation which could not be achieved by naive static evaluators. Performance of the proposed evaluator has been also measured and compared with dynamic and static evaluators.

The structure of this paper is as follows. In the next section we outline attribute grammars and explain some terminologies. In Section 3 we propose our extended AGs called CRAGs. Section 4 shows a way to construct the static evaluator for CRAGs. In Section 5, a revised version of this evaluator, the mostly static evaluator, is described. Section 6 evaluates performance of the mostly static evaluator by comparing with static and dynamic evaluators. Section 7 discusses related work. Section 8 gives future work and conclusion.

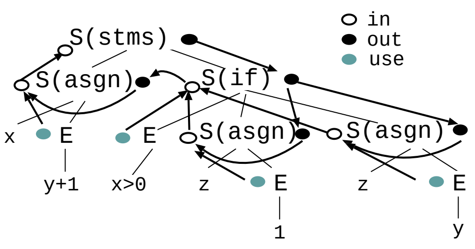
§2 Preliminaries

Attribute grammars are a formalism which can describe the syntax and semantics of languages in an integrated fashion. In this section, we present a simple example and an outline of attribute grammars and explain some terminologies used in this paper. For details, refer to the original paper by Knuth.[7)]

An attribute grammar is, in short, a context-free grammar with attribute definition rules added. Fig. **??**(a) gives an example of an attribute grammar that computes live variables for a program of a simple language.

```
S ::= S ";" S   /* (stms) */  .....(1)
{ S3.out = S.out;
  S2.out = S3.in;
  S.in = S2.in;               .....(2)
}
S ::= "if" E "then" S "else" S  /* (if) */
{ S2.out = S.out;
  S3.out = S.out;
  S.in = union(S2.in, S3,in, E.use);
}
S ::= varid ":=" E  /* (asgn) */
{ S.in = union(remove(varid, S.out), E.use);
}
S ::= "use" E  /* (use) */
{ S.in = union(S.out, E.use);
}
```

(a)



(b)

Fig. 1 Attribute grammar of live variable analysis

Lines which include "`::=`" like (1) of this figure are productions of the context-free grammar, and the parts enclosed by {} following these lines are the definitions of attributes corresponding to these productions. Each symbol X of the context free grammar may have *attributes* that represent information of the nodes with the symbol X on a syntax (or parse) tree. In this example symbol `S` (that represents a "statement") has attributes `in` and `out` which represent sets of variables that are "live" at the entry and exit of the statement, respectively. Attribute `use` of `E` ("expression") represents all the variables used in an expression. Definitions for `E` are omitted in Fig. ??(a).

The value of each attribute is defined by a *semantic rule* for a production. The definition (2) is the semantic rule to define the value of attribute `in` of the left-hand-side symbol `S` of the production. Attributes in the production are called *attribute occurrences* and are denoted as `S.in` or `S2.in`. Subscript `2` of `S2` means `S`'s second occurrence in the production.

Generally, semantic rules are represented in the following form:

$$X_0.a_0 = f(X_1.a_1, \ldots, X_k.a_k)$$

where $X_i.a_i$ for each $i \geq 0$ is an attribute occurrence and f is a function called *semantic function.* A semantic rule may cause *attribute dependency*: we say "$X_0.a_0$ depends on $X_i.a_i$ $(i \geq 1)$", since $X_0.a_0$ cannot be calculated until all of the arguments of f are determined.

Defining the value of an attribute after parsing the sentence according to the grammar is called evaluating an attribute, and the procedure to evaluate attributes is called *attribute evaluator.* When a source program "`x := y + 1; if x > 0 then z := 1 else z := y`" is analyzed according to the above grammar, we can obtain a graph shown in Fig. ??(b). Note that for simplicity of explanation, we use abstract syntax trees rather than parse trees, and omit terminal symbols. We call them simply syntax trees hereafter. Each node in this figure is called *attribute instance*, which represents an attribute associated with each grammar symbol on the syntax tree. The edges represent attribute dependency between the attribute instances. Such a graph is called *dependency graph for a syntax tree* and denoted as $DG(T)$ for syntax tree T. Evaluation of attributes should be performed in an order consistent with the dependency graph for a syntax tree.

Attributes are classified into two kinds, according to the way their values are defined. *Inherited attributes* are assigned for the nonterminal symbols on the right-hand-side of a production, whereas *synthesized attributes* are assigned for the nonterminal symbol on the left-hand-side of a production. In Fig. ??(a), `out` is inherited attribute and `in` and `use` are synthesized attributes.

§3 Circular Remote AGs (CRAGs)

3.1 Remote Attribute Grammars (RAGs)

In data-flow analysis in compiler optimizations, data-flow information is propagated between two contiguous structures of a program, for example, two statements in a program text which are executed sequentially. If the language has a labeled jump control structure (e.g. “goto”-“label”), data-flow information needs to be transferred between the points of “goto”s and those of the associated “label”s. However, in traditional AGs it is hard to write relations between two such distant points in a program text or between two nodes that do not occur in the same production. This is because in traditional AGs direct dependencies are restricted only to attributes in a parent-child relationship in the syntax tree.

We propose *remote attribute grammars* (RAGs) [*1] which allow non-local dependencies by introducing pointers (or links) that connect distant nodes of the syntax tree. The following example shows an RAG description of liveness analysis for a small language which includes the “goto”-“label” control structure. The declarations except for “goto” and “label” are the same as the previous example of Fig. **??**(a) and are omitted here.

Remote Attribute Grammar $G1$

```
/* The declarations for production stms, asgn, use and if
   are the same as Fig.(1)(a) */

      S ::= labid ":" S  %prod label  /* label */
      { S2.out = S.out;
        S.in = S2.in;
      }
      S ::= "goto" labid  %ref label  /* goto */
      { S.in = label.S.in; }
```

The declaration “`%prod label`” denotes that this production label is named “`label`”. Its accompanying declaration “`%ref label`” in the production goto allows semantic rules in goto to refer to attribute instances that are defined in production label. Here, we assume that in the syntax tree for $G1$, links are properly provided between the “goto” nodes and their associated “label” nodes, e.g. by a previous phase. The actual link is not described in the grammar. Note that although our formalism omits establishing such links, we can define it declaratively in traditional AG where attribute values may include references to nodes.[*2]

Using the above links, `label.S.in` in production goto remotely refers

[*1] Hedin[?)] has proposed *reference attributed grammars (RAGs).* Our work is independent from her work and the RAG proposed in this paper should not be confused with her definition.

[*2] However, attribute accesses via such references are not allowed here.

to the value `S.in` defined in production named "`%prod label`", i.e. production label. This remote reference is achieved by the pointer owned by the node representing "goto statement". Fig. **??** shows an example of the attribute dependency graph where the broken arrow represents dependency that arises from a remote reference.

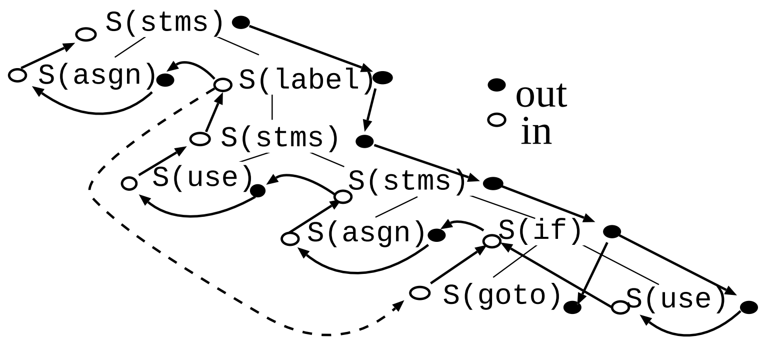


Fig. 2 Attribute dependency graph with remote references

The semantic rule in production goto describes the data-flow analysis naturally, that is,

"the set of live variables at the entry of a goto (`S.in`) is equal to the set of variables which are live at the entry of an associated label (`label.S.in`)"

Generally, the values of data-flow equations are defined recursively due to loop structures. Actually, in the example above, the attribute dependency contains a cycle. Our formulation allows such circularly defined attributes to be included if the appropriate conditions described later are satisfied. We call these RAGs *circular remote attribute grammars* (CRAGs). The grammar $G1$ is a CRAG since the appropriate conditions are satisfied.

3.2 Definition of RAGs

Let G be a context-free grammar of an attribute grammar and P be the set of productions in G. The inter-production relation PR is defined as

$$PR \subseteq \{(p_i, p_j) |\ p_i, p_j \in P \text{ and } i \neq j\}$$

Here, p_i is called the remotely referenced production (or simply remote production) and p_j is called the remotely referencing production (or simply reference production). Then, *remote attribute grammars* (RAGs) are defined by the following extension to traditional AGs.

> The arguments of semantic functions in the reference production p_j can include attribute occurrences in the remote production p_i.

In other words, this extension says that the semantic rules can be described as

$$a_{j\ell}(0) = f_{j\ell}(a_{j\ell}(1), \ldots, a_{j\ell}(n_{j\ell}))$$

where $a_{j\ell}(0)$ is attribute occurrence in p_j, $f_{j\ell}$ is its associated semantic function and each $a_{j\ell}(m)$ $(1 \leq m \leq n_{j\ell})$ is either an attribute occurrence in p_j, or one in remote production p_i. If $a_{j\ell}(k)$ is an attribute occurrence in remote production p_i for some k $(1 \leq k \leq n_{j\ell})$, we call it a *remote attribute* and $a_{j\ell}(0)$ is called a *reference attribute.* A remote attribute can be inherited (e.g. constant propagation analysis) or synthesized (e.g. live-variable analysis).

3.3 Circular RAGs and their Property

As defined previously, a semantic rule in RAGs is described in the same manner as that of traditional AGs except that we can put remote attributes on the right hand side of the semantic rule. As a natural consequence, the concept and conditions of well-defined-ness in RAGs are also analogous to those of traditional AGs.

First, RAGs which do not contain cycles in attribute dependency are well-defined similarly to traditional non-circular AGs. In this case, the values of attribute instances can be simply determined by applying semantic rules in the topological order of $RDG(t)$ – the directed graph where dependency edges made by remote and reference attributes are added to the attribute dependency graph $DG(t)$ of the syntax tree t.

Next, we show well-defined-ness of RAGs which contain circular dependencies between attribute instances. Previous researchers have suggested extensions to traditional AGs that allow such circular dependencies.[?, ?] In Farrow's scheme, under the conditions described below, values of circularly defined attribute instances are determined uniquely by a finite number of repetitive evaluation of the associated semantic rules. These circular but well-defined AGs are defined in the following way: let ATTRIBS be the set of attributes that are defined circularly and FUNCTS be the set of associated semantic functions for these attributes. An AG is a *finitely recursive attribute grammar* iff:

- The domain of all attributes in ATTRIBS constitutes a complete partial order (c.p.o.), in which it is possible to test pairs of elements for equality, and
- All functions in FUNCTS are monotonic and converge (an ascending chain condition), that is,

  $$f(s[0]) < f(s[1]) < \ldots\ldots < f(s[k]) = f(s[k+1])$$

  for

  $$s[0] < s[1] < \ldots\ldots$$

  where $f \in$ FUNCTS, $s \in$ ATTRIBS, i of $s[i]$ is the count of iteration.

The property of convergence in traditional circular AGs is trivially applicable to the one in RAGs containing circular dependencies. Thus, RAGs which satisfy the conditions above are also well-defined and we call them *circular remote attribute grammars* (CRAGs).

The decidability whether an RAG is circular or not is unknown, but at least it should take exponential time for precise circularity test. Dependency graphs for productions shown in the next section determines the circularity conservatively, that is, every circular RAG is identified as circular but some non-circular RAGs could be taken as circular. Since exactly finding circularly defined attributes is known to be a difficult problem (NP-hard even for a subclass of AGs), we can apply a conservative algorithm proposed by Rodeh and Sagiv[?].

Note that our CRAG evaluator, to be shown in later sections, also requires the monotonic condition (the second condition above) for attributes that are never included in cycles. This is due to our evaluation strategy in which successive approximation may be mixedly applied to non-circular attributes as well as to circular attributes. This requirement is not severe as it is supposed in most previous work. [*3]

§4 Static Attribute Evaluator for CRAGs

In this section we show a way to construct the static evaluator for CRAGs, which can be generated automatically from CRAG descriptions.

4.1 Static Evaluator

[*3] Dynamic evaluators can remove this requirement by analysis, e.g. Boyland's APS system[?]

Some static evaluators for AGs do not always achieve the optimal order of attribute evaluation because of recomputation of attributes. However, they usually give nearly optimal order and provide efficient run-time performance since partial orders of attribute evaluation are determined statically, i.e. at generation time. As we mentioned before, a static evaluator for finitely recursive AGs was proposed by Farrow.[?] Similarly to his work, the evaluator for CRAGs can be constructed by extending the recursive style evaluator for absolutely (strongly) non-circular AGs proposed by Katayama.[?]

The evaluator for CRAGs consists of a set of recursive functions, each of which is in the following form:

$$R\_X.s(T, i_0 \dots i_n)$$

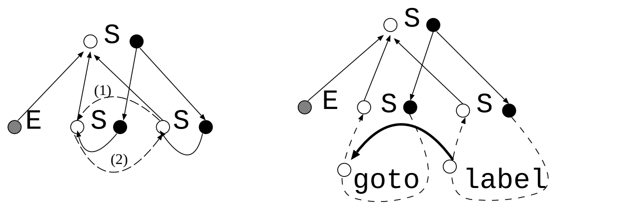
Here, each function $R\_X.s$ is associated with synthesized attribute s of nonterminal X and is called a *synth-function* because it is intended to compute the value of $X.s$ at the root of syntax (sub)tree T. Parameters $i_0 \dots i_n$ are inherited attributes of X that are needed to calculate the value of $X.s$ for all possible syntax trees generated from X.

The synth-function $R\_X.s$ takes the following form:

```
function $R\_X.s(T,\ i_0 \dots i_n)${
  case production($T$) of
    $p_1$: $H_{p_1,s}$
       ...
    $p_n$: $H_{p_n,s}$
}
```

where $p_1 \dots p_n$ are productions whose left-hand side nonterminal is X and $H_{p_i,s}$ represents the evaluation sequence to calculate $X.s$ for p_i.

Each element of sequence $H_{p,s}$ is one of the following computation: an assignment statement corresponding to the application of a semantic function of p, a call of a synth-function, or a fixed-point calculation routine to be described in Section **??**. Ordering of such computation in $H_{p,s}$ can be determined from a dependency graph defined within a production. In the original static evaluator for absolutely noncircular AGs,[?] the evaluation order within production p is determined from the augmented dependency graph $DG^*(p)$, whose nodes are attribute occurrences in p and edges represent transitive (or possibly indirect) dependency as well as direct dependency. In our evaluator for CRAGs, the ordering of computation in each $H_{p,s}$ is determined by *augmented remote dependency* *graph* $RDG^*(p)$, which extends $DG^*(p)$ by adding edges called *indirect remote dependency edges*. An indirect remote dependency edge is a relation between two attributes in p, and represents dependency by remote references which may occur considering all possible syntax trees generated from production p. For example, Fig. **??**(a) represents $RDG^*(\mathsf{if})$ for the previous example $G1$, where broken arrows (1) and (2) are indirect remote dependency edges.



(a) *RDG**(`if`) (b) a remote edge that causes edge(1) in (a)

Fig. 3 $RDG^*(\mathsf{if})$

In this example, indirect remote dependency edge (1) in (a) appears since `S2` (the subtree representing the then-clause) may contain one or more "goto"s whose destinations are in `S3` (the subtree representing the else-clause) as in (b). Edge (2) also occurs in the reverse case. Whether each of the indirect remote dependency edges actually arises or not depends on syntax trees and references existing at evaluation time. However, $RDG^*(p)$ should include all of these indirect remote dependency edges since the static evaluator for CRAGs cannot predict which remote dependencies actually arise.

Using $RDG^*(p)$, the sequence of attribute evaluation is computed. First, graph $RDG^*(p)/CDC$ is constructed in order to collect circularly defined attributes into groups. Each vertex is a CDC (Circular Dependency Class)[7] which is given by grouping together vertices of $RDG^*(p)$ included in the same cycle. In other words, each vertex of the graph represents a strongly connected component of $RDG^*(p)$. The edge of $RDG^*(p)/CDC$ represents dependency between two CDCs. Since $RDG^*(p)/CDC$ is an acyclic (cycle-free) directed graph, a total order of vertices can be calculated by the topological sorting of $RDG^*(p)/CDC$. This total order represents the sequence of attribute evaluation in the synth-function where attribute values in each circular dependency are regarded to be a single value. Attribute values inside the circular dependency will be evaluated separately as in Farrow's evaluator to be shown in Section **??**.

The values of remote attributes are "cached" into a global "dictionary" (or attribute table) in addition to storing their values into local variables of a synth-function. At reference sites, values of remote attributes can be obtained by referring to these cached values. The cache is needed because it is not generally assured that the value of a remote attribute is on the run-time stack for synth-functions.

4.2 Iterative Evaluation

Next, we discuss the part of the evaluation of circularly defined attributes. The case that a node in $RDG^*(p)/CDC$ contains two or more attribute occurrences means that they are defined circularly. To determine their values in this case, semantic rules for them are evaluated repeatedly until convergence, starting from their initial values (i.e. bottom of the c.p.o. domain).

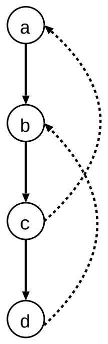
[1] Check of Convergence

A simple way to determine whether the iterated evaluation has converged is to check if every value of attribute instances in cycles is unchanged between the $(n-1)$th iteration and the n-th iteration. But provided that cycles do not occur without remote references, that is, if only remote references cause circular dependencies, the following property holds:

Property 1
All of the attribute instance values in cycle C have converged if the n-th iteration does not change any values of remote attributes in C.

This property means that only remote attributes in C have to be checked for termination of evaluation loop. The proof is omitted, though an intuitive explanation is shown in the following.

From the assumption, we can say that for C's edges except those edges which are made by remote references, a total order of attribute instances can be made. The iterative evaluation of attributes in C will be made according to this total order. Fig. **??**(a) illustrates an example of the attribute dependency graph including a cycle, where two broken arrows, `c->a` and `d->b`, represent remote references (remote edges). This means node `c` and `d` are the remote attributes, and node `a` and `b` are the reference attributes. By removing remote edges `c->a` and `d->b` from the original cycle, we can get an acyclic graph and total order ⟨`a b c d`⟩. Fig. **??**(b) is the corresponding iterative evaluation routine based on that total order.



(a) example dependency graph

```
REPEAT
  a=Fa(c);
  b=Fb(a,d);
  c=Fc(b);
  d=Fd(c);
UNTIL convergence
```

(b) iterative evaluation

Fig. 4 Example dependency graph and its corresponding iterative evaluation routine

Here, we call edges following this total order “forward edges” (`a->b`, `b->c` and `c->d` in the example), and those in the reverse direction “backward edges” (`c->a` and `d->b` in the example). A forward edge represents a relation where the attribute value at the k-th iteration can be determined from the other attribute values at the k-th iteration. In the example, forward edge `b->c` represents that the value of `c` at the k-th iteration can be determined by the value of `b` at the same iteration by “`c = Fc(b)`”. On the other hand, a backward edge represents an anti-dependence relation in the sense of dependence among attributes in a program loop, which means that the attribute value at the k-th iteration is calculated using the value determined at the $(k-1)$th iteration. In the example, backward edge `c->a` represents that at the k-th iteration, the evaluation of “`a = Fa(c)`” refers to the value of `c` at the $(k-1)$th iteration.

According to the assumption of the property and the above definitions, it can be said that all backward edges are made by remote references in cycle C. In the example, all backward edges (i.e. `c->a` and `d->b`) are remote edges. This implies that the values of reference attributes (i.e. `a` and `b`) at the k-th iteration may be determined by the values of remote attributes (i.e. `c` and `d`) at the $(k-1)$th iteration while all the other values (i.e. `c` and `d`) at the k-th iteration can be determined by the values at the same k-th iteration. From the other point of view, if both values of remote attributes c and d have converged at the n-th iteration, the values of reference attributes a and b (and of course the values of c and d) at the $(n+1)$th iteration are the same as those at the n-th iteration.

Considering that all attribute values in cycle C at the k-th iteration can be determined from values of remote attributes at the $(k-1)$th iteration and values of attributes outside C on which attributes in C depend, we can conclude that Property **??** holds in general. (end of the explanation of Property 1)

As a consequence of this property we do not need to initialize all the attribute instances in a cycle before successive approximation. Remote attributes included in the cycle are the only attribute instances that should be initialized.

In fact we can make Property 1 true even when some cycles are caused solely by direct (i.e. non-remote) edges, e.g. in the case such as "while statement" in our live-variable example, by converting at least one edge from being a direct edge to being a remote edge. That is, without loss of generality, we can assume cycles are constructed only via remote references, by performing a simple transformation as follows. When cycles arise from normal dependencies, $DG^*(p)$ for some production p should include cycles. In that case, delete some direct edges in cycles of $DG^*(p)$ so that the dependency without these edges does not contain any cycle, then replace the deleted edges with remote edges referring to attributes in the same production p[*4].

[2] Iterative Evaluation Routine

In Farrow's evaluator, iterated evaluation may recursively include another iterated evaluation, i.e. nested loops, because the evaluator is realized with a group of mutual recursive functions along the tree. The nested loops causes inefficiency since the number of iteration of the innermost loop becomes an exponential factor of the nested level of the loop in the worst case. For preventing this, the evaluator for CRAGs is made in a way that it does not execute iterations on the inner loops. This strategy makes the maximum number of evaluation in a cycle less than the depth of the graph[?)] plus two in our live-variable example. [*5]

Taking this into account, the routine for calculating attribute values in a cycle becomes as Fig. **??**.

Here, "`initialize remote attributes in this cycle;`" means the assignment of the initial value (i.e. bottom in c.p.o.) to remote attributes. As described in Property 1, the termination check of loop is made by all values of

[*4] For efficient iterative evaluation, the choice of such replaced edges should be determined by depth-first ordering [?)].

[*5] Intuitively, the depth of a graph is the largest number of backward edges on any cycle-free path of the graph. For example the graph of Figure **??**(a) is of depth 2, since a cycle-free path ⟨`d b c a`⟩ includes the two backward edges. If a cyclic graph includes only independent nested cycles, the depth of the graph is one, whatever the nested levels each cycle has.

```
if (!isInCircle) {
  isInCircle = TRUE;
  initialize remote attributes in this cycle;
  REPEAT
    a sequence of attribute evaluation in CDC;
  UNTIL convergent;
  isInCircle = FALSE;
} else {
    a sequence of attribute evaluation in CDC;
}
```

Fig. 5 iterative evaluation routine

these remote attributes being unchanged (i.e. by "`UNTIL convergent`"). Variable `isInCircle` is a global boolean variable, initialized to `FALSE`, representing whether the computation is within iterative evaluation or not. If the above routine is called during some iterative evaluation, the execution of the nested loop can be avoided since `isInCircle` must have been set to `TRUE`.

4.3 Example Evaluator

The following program fragment is the "goto" and "label" part of the synth-function for evaluating synthesized attribute `S.in` of $G1$. The "label" part contains an iterative evaluation routine, because RDG^*(label) includes an indirect remote dependency edge which causes a cycle.

```
function R_S.in(T, S.out) {
 case production(T) of
   assign:
      ...
   label:  /* label: S ::= labid S */
     S2.out = S.out;
     if (!isInCircle) {
       isInCircle = TRUE;
       initRemote(T, "label.S.in");
       REPEAT
         S2.in = R_S.in(T[2], S2.out);  /* T[2] corresponds to S2 */
         S.in = S2.in;
         storeAttrValue(nodeID(T), "S.in", S.in);
       UNTIL isConvergent(T, "label.S.in");
       isInCircle = FALSE;
     } else {
       S2.in = R_S.in(T[2], S2.out);
       S.in = S2.in;
       storeAttrValue(nodeID(T), "S.in", S.in);
     }
   goto:
     S.in = readAttr(nodeID(refNode(T,"label")), "S.in");
return S.in; }
```

Here, `initRemote` corresponds to `initialize remote attributes` in the code of iterative evaluation routine in Section **??**. In this example, all instances of the remote attribute `label.S.in` in the tree `T` are initialized to the bottom, i.e. empty set.

The function `storeAttrValue` and `readAttr` are accessors for the cache storing values of remote attributes. `storeAttrValue` is for storing the value of a remote attribute into the cache implemented in a "dictionary"(a repository in a Smalltalk-like naming). The "key" of the dictionary is a pair of the node number and the name of the attribute occurrence (i.e. remote attribute). `readAttr` is for referring these cached values. In this example, `refNode(T, "label")` in the part of "goto" gets the reference to the corresponding "label" node in `T`.

§5 Optimization of CRAG evaluators

5.1 Basic Idea

The static evaluator described above does not need any calculation of the evaluation order at evaluation time. However, unnecessarily iterated execution may arise. We will show this fact by the example of the static evaluator for $G1$.

In $RDG^*(p)$ for $G1$ many cycles arise, that is, in the case of p being "if", "stms" and "label". This means that for most syntax trees, the evaluation may encounter the part of iterative evaluation for calculating circularly defined attribute values at the synth-function of the root node. So, even if the remote dependency graph for the given syntax tree T, or $RDG(T)$ (mentioned in Section **??**), actually does not contain cycles, the evaluator will execute the same calculation twice, which causes a long evaluation sequence.

To overcome this inefficiency, we propose a refined version of the evaluator for CRAGs. The above problem stems from the over-estimation of dependency by taking the union of all possible indirect remote dependency edges when an $RDG^*(p)$ is made. So the basic idea for the refinement is to divide those unioned edges.

$H_{p_i,s}$ of the synth-function of the static evaluator associated with $X.s$ (Section **??**) now has several versions of attribute evaluation sequences in the revised synth-function. These versions correspond to the partial orders generated from $RDG^*(p)$ by classifying patterns of the actual occurrence of indirect remote dependency edges. Considering $RDG^*(\mathsf{if})$ of $G1$ as an example, there exist four patterns of partial orders, which are illustrated by the graphs in Fig. **??**. Each partial order reflects the following patterns of subtrees: (a): then-clause includes "goto"s whose destinations are in else-clause, (b): the reverse pattern of (a), (c): (a) and (b) occur simultaneously and (d): none of (a)(b)(c) occurs.

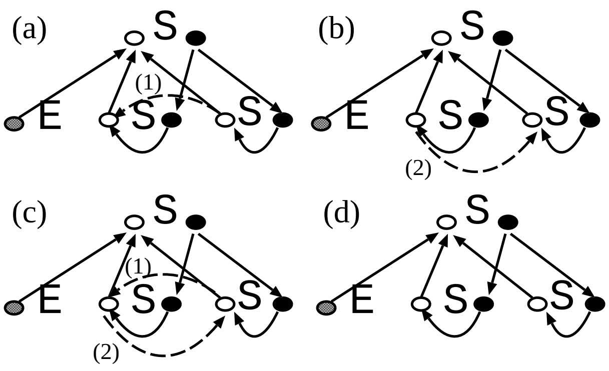


Fig. 6 Four patterns of partial orders in $RDG^*(\mathsf{if})$

Fig. **??** is the optimized evaluator for $G1$ based on the above idea. Here, we focus on the routine to evaluate "if node". Routine $H_{\mathsf{if},in}$ is divided into

```
function $R\_S.in(T, S.out)${
  case production($T$) of
  ....
   if: case $IRD[T, \mathsf{if}]$ of
       $\{(S_3.in, S_2.in)\}$: $H_{\mathsf{if},in,\{(S_3.in,S_2.in)\}}$  /* corresponds to (a) in Fig. ??*/
       $\{(S_2.in, S_3.in)\}$: $H_{\mathsf{if},in,\{(S_2.in,S_3.in)\}}$  /* (b) */
       $\{(S_3.in, S_2.in), (S_2.in, S_3.in)\}$: $H_{\mathsf{if},in,\{(S_3.in,S_2.in),(S_2.in,S_3.in)\}}$  /* (c) */
       $\{\}$: $H_{\mathsf{if},in,\{\}}$  /* (d)*/
}
```

Fig. 7 structure of the optimized evaluator

four versions which are here referred to as *versioned evaluation routines*. Here, subscript E of $H_{\mathsf{if},in,E}$ means a set of edges (called an *edge pattern*) that is added to $DG^*(\mathsf{if})$. For example, $H_{\mathsf{if},in,\{(S_3.in,S_2.in)\}}$ is based on the partial order of evaluation made by graph Fig. **??** (a), or $DG^*(\mathsf{if}) \cup \{(S_3.in, S_2.in)\}$. By making these versions, iterative evaluation can be avoided in case (a), (b) and (d).

Generally, at evaluation time, one of $H_{p,s,E}$'s will be selected at each node before evaluation. $IRD[T,p]$ represents information of indirect remote references in T and enables to select one routine that leads to most efficient evaluation for T. A preprocessor, which will be shown in Section **??**, stores $IRD[T,p]$ on each tree node. For example, at each "if node" n , the preprocessor for the evaluator above will analyse "which case of (a)(b)(c)(d) should n have?", and this information will be stored as $IRD[n, \mathsf{if}]$ into n.

We call the evaluator above *mostly static evaluator*. Here "mostly static" means although evaluation order is selected "dynamically" according to remote references in the given syntax tree, the partial order of evaluation in each production is still determined "statically" at evaluator generation time.

5.2 Generation of versioned evaluation routines

$RDG^*(p)$ can be obtained by adding all possible indirect remote edges to $DG^*(p)$. On the other hand, by adding some of indirect remote edges to $DG^*(p)$, we can obtain a specialized version of $RDG^*(p)$. Here, such a set of indirect remote edges added to $DG^*(p)$ is called *a pattern of indirect remote edges* or simply *edge pattern*.

Definition 5.1 (pattern of indirect remote edges)
Now let $IRD(p)$ be the set of all indirect remote dependency edges that belong to $RDG^*(p)$ and n_p be its number of edges, or $|IRD(p)|$. Then *a pattern of indirect remote edges for* p, (or simply a pattern of $IRD(p)$) is a set that is composed by taking k arbitrary edges ($0 \leq k \leq n_p$) from $IRD(p)$. The power set of $IRD(p)$ or $P(IRD(p))$ represents all possible patterns of $IRD(p)$.

Example 5.1 (if node)
$IRD(\mathsf{if})$ is $\{e_1, e_2\}$ where $e_1 = (S_3.in, S_2.in)$ and $e_2 = (S_2.in, S_3.in)$. Then, $P(IRD(p)) = \{\{e_1\}, \{e_2\}, \{e_1, e_2\}, \{\}\}$.

Now, one specialized version of $RDG^*(p)$ can be simply constructed by adding one of patterns of $IRD(p)$ to $DG^*(p)$.

Definition 5.2 ($RDG^*(p,E)$, $RDG^*(p)$ specialized by an edge pattern)
Let E be a pattern of $IRD(p)$, that is $E \in P(IRD(p))$. *Augmented remote dependency graph of* p *specialized by* E, $RDG^*(p,E)$, is defined by $RDG^*(p,E) = DG^*(p) \cup E$

Note that clearly, $RDG^*(p, IRD(p)) = RDG^*(p)$ and $RDG^*(p, \{\}) =$

$DG^*(p)$ hold. The number of $RDG^*(p, E)$'s derived from $RDG^*(p)$ is 2^{n_p} that is the size of $P(IRD(p))$.

Example 5.2 (if node)
From $P(IRD(\mathsf{if}))$, we can obtain four specialized versions for $RDG^*(\mathsf{if})$, that is, $RDG^*(\mathsf{if}, \{e_1\})$, $RDG^*(\mathsf{if}, \{e_2\})$,$RDG^*(\mathsf{if}, \{e_1, e_2\})$ and $RDG^*(\mathsf{if}, \{\})$ that correspond to (a)(b)(c) and (d) in Fig. **??** respectively.

Now, we can obtain a versioned evaluator as in Fig. **??** by putting together all the evaluation routines $H_{p,s,E}$'s for all indirect remote edges $E \in P(IRD(p))$. The algorithm to generate $H_{p,s,E}$ is almost the same as one that generates $H_{p,s}$ from $RDG^*(p)$ as shown in Section **??**. The only difference is that each routine is made from a specialized version of $RDG^*(p, E)$ not from $RDG^*(p)$.

Note that the number of versions or routines to be needed is always smaller than $|P(IRD(p))|$, since we can often share routines for different edge patterns. In Fig. **??**, for example, the routine drawn from the pattern of (d) is the same as one from either (a) or (b), and therefore the routine for (d) is not needed. Moreover, we can often ignore some indirect remote edges, which makes the number of routines reduced by half. For example, if a remote edge represents the same dependency as a direct (or possibly transitive) dependency, we do not require another routine for the remote edge. In CRAG descriptions of SSA transformation [?] and partial redundancy elimination[?] in our C subset compiler, the number of routines for one production is actually at most three.

But if a large number of routines are required for one production, we could fuse routines that seem rarely executed in a single routine and make the actual evaluation order in the routine dynamically scheduled.

5.3 Analysis of remote dependency in syntax tree

The mostly static evaluator performs evaluation by two steps: (1) a preprocess that analyzes remote edges and selects versioned routines, and (2) the actual evaluation by the evaluator shown above. This section shows how to construct the preprocessor.

First, we recall the relationship between the two steps and give some definitions. In the evaluator for $G1$, a total of four evaluation routines is generated from $RDG^*(\mathsf{if}, E)$'s, one for each edge pattern E of $IRD(\mathsf{if})$ (Fig. **??**). The preprocessor determines appropriate E in $P(IRD(\mathsf{if}))$ that is actually needed in each "if node" of the given syntax tree T. The actual E is represented by $\mathit{IRD}[T, \mathsf{if}]$ in Fig. **??**. For example, the pattern E for the tree in Fig. **??** (b) is $\{(S_3.\mathit{in}, S_2.\mathit{in})\}$ and routine $H_{\mathsf{if},\mathit{in},\{(S_3.\mathit{in},S_2.\mathit{in})\}}$ will be applied.

For each "if node" in a syntax tree, the candidates of indirect remote dependency edges are $(S_3.\mathit{in}, S_2.\mathit{in})$ and $(S_2.\mathit{in}, S_3.\mathit{in})$. In the case of Fig. **??** (b), $(S_3.\mathit{in}, S_2.\mathit{in})$ is actually needed but $(S_2.\mathit{in}, S_3.\mathit{in})$ is not. Here, we say $(S_3.\mathit{in}, S_2.\mathit{in})$ is an *actual indirect remote edge with respect to subtree* T. Whether an indirect remote edge is actual one or not is determined by the following definition.

Definition 5.3 (actual indirect remote edge w.r.t. syntax tree)
Let T be a syntax (sub)tree, T_i be the i-th child of T, $p : N_0 ::= N_1 \dots N_{n_p}$ be the production rule applied at the root of T, and $\mathit{REdge}(T)$ be all (direct) remote dependency edges in $\mathit{RDG}(T)$ (Section **??**).

Then edge $(N_k.s, N_{k'}.s') \in \mathit{IRD}(p)$ is an *actual indirect remote edge of* p *with respect to* T, *iff* there exists at least one remote edge $(r, \mathit{ref}_r) \in \mathit{REdge}(T)$ such that $r \in \mathit{sd}(T_k.s)$ and $\mathit{ref}_r \in \mathit{sd}(T_{k'}.s')$ where $T_i.a$ is an attribute instance a of T_i [*6], and $\mathit{sd}(T_i.a)$ is a set of attribute instances in $\mathit{DG}(T_i)$ on which $T_i.a$ directly or indirectly depends.

For example, $(S_3.\mathit{in}, S_2.\mathit{in})$ is an actual indirect remote edge with respect to the tree in Fig. **??** (b) (say T) because $\mathit{sd}(T_3.\mathit{in})$ includes remote attribute r ($S.\mathit{in}$ of the "label node") and $\mathit{sd}(T_2.\mathit{in})$ includes reference attribute ref_r($S.\mathit{in}$ of the "goto node") that refers to r. In Fig. **??**(b), $\mathit{sd}(T_2.\mathit{in})$ and $\mathit{sd}(T_3.\mathit{in})$ are the set of attributes on the broken arrows below the then-clause and the else-clause, respectively.

The actual pattern for a tree whose root is p is calculated by determining whether each edge e in $\mathit{IRD}(p)$ is an actual indirect remote edge or not.

Definition 5.4 ($\mathit{IRD}[T, p]$)
Pattern of $\mathit{IRD}(p)$ in syntax (sub)tree T, $\mathit{IRD}[T, p]$ is the unioned set of all the actual indirect remote edges of p with respect to T.

In this way, the evaluation routine that should be applied at T can be determined, that is, the routine for $\mathit{RDG}^*(p, \mathit{IRD}[T, p])$. This analysis is done by the preprocessor described in the rest of this section in detail.

[*6] Note that $T_k.s$ corresponds to $N_k.s$ and $T_{k'}.s'$ corresponds to $N_{k'}.s'$, but $T_i.a$ represents an *attribute instance*, not an *attribute occurrence* such like $N_i.a$.

[1] Generation of the preprocessor

The preprocessor computes $IRD[T,p]$ for each subtree T of the given syntax tree. $IRD[T,p]$ can be obtained by an algorithm induced from the previous definition. This algorithm does not employ $sd(T.s)$ to obtain $IRD[T,p]$, but uses $sd_{rem}[T.s]$ and $sd_{ref}[T.s]$ instead, where $sd_{rem}[T.s]$ represents the set of remote attributes in $sd(T.s)$ and $sd_{ref}[T.s]$ represents the set of remote attributes referred to by attributes in $sd(T.s)$. Fig. **??** describes the structure of the preprocessor for $G1$ and its routine that collects actual indirect remote edges with respect to each "if node" in the syntax tree.

```
function $CompIRD(T)$ {
  case production($T$) of
    assign: $CompIRD_{\mathsf{assign}}$
      ...
    if: $CompIRD_{\mathsf{if}}$
      ...
}
$CompIRD_{\mathsf{if}}$ { /* $\mathsf{if} : S_1 ::= E\ S_2\ S_3$ */
  /* compute IRDs in subtrees and collect
     remote and reference attribute information */                              /* (1)*/
    $CompIRD(T_2)$;    /* then clause */
    $CompIRD(T_3)$;    /* else clause */
  /* compute $IRD[T, \mathsf{if}]$ */
    if $(sd_{rem}[T_3.in] \cap sd_{ref}[T_2.in] \neq \phi)$ $e_1 = \{(S_3.in, S_2.in)\}$; else $e_1 = \{\}$;    /* (2a)*/
    if $(sd_{rem}[T_2.in] \cap sd_{ref}[T_3.in] \neq \phi)$ $e_2 = \{(S_2.in, S_3.in)\}$; else $e_2 = \{\}$;    /* (2b)*/
    $IRD[T, \mathsf{if}] = e_1 \cup e_2$;                                                  /* (2c)*/
  /* synthesize remote and reference attribute information */                  /* (3) */
    $sd_{ref}[T.in] = sd_{ref}[T_2.in] \cup sd_{ref}[T_3.in]$;
    $sd_{rem}[T.in] = sd_{rem}[T_2.in] \cup sd_{rem}[T_3.in]$;
}
```

Fig. 8 the preprocessor for $G1$ and the part of production "if"

The preprocessor can be implemented as a recursive procedure that takes tree T as the argument and computes $IRD[T,p]$ where p is the production applied at the root of T. The body of the procedure consists of routines for each p.

Here, the part of "if" is taken as an example. In step (1) of Fig. **??**, for each subtree (then clause, else clause), *CompIRD* is called recursively to compute the actual indirect remote edges. Steps (2a)-(2c) are the part where $IRD[T, \mathsf{if}]$ is calculated. (2a) means that $IRD[T, \mathsf{if}]$ should include edge $(S_3.in, S_2.in)$ if there exists remote attribute r such that $r \in sd(T_3.in)$ and reference attribute $ref_r \in sd(T_2.in)$ that refers to r. In a similar fashion as (2a), (2b) computes whether or not $IRD[T, \mathsf{if}]$ should include $(S_2.in, S_3.in)$. Although the calculation of $IRD[T, \mathsf{if}]$ does not straightly follow Definitions 3 and 4, the same result can be calculated. In (3), the information of actual remote dependency in subtrees is merged, that will be used in ancestors of T. In a routine for a production containing remote and/or reference attributes (in the routines for label and goto nodes for this example), information of the remote and/or reference attributes should be also merged with subtree's information at step (3).

Thus, actual indirect remote dependency edge sets are computed on the syntax tree in a one-pass bottom-up manner. For making the preprocessor faster, bit-vectors should be used for representing $sd_{ref}[T.s]$ and $sd_{rem}[T.s]$.

§6 Empirical Results and Discussion

6.1 Empirical Results

We have implemented a generator of mostly static evaluator where both the generator and the generated evaluators works on Squeak Smalltalk system. This system is an extension of our previous system [?]. In this section, we will show and compare performance of CRAG evaluators based on straightforward strategies and the mostly static one. To measure performance of these evaluators, we chose "live variable analysis" as a typical application, which is frequently used in many compiler phases. The description used here is almost the same as grammar $G1$ shown in Section **??**. For the sake of fair comparison, the measured evaluators are rewritten in Lisp and then compiled to native codes. The mostly static evaluator has been hand translated straightforwardly from the generated evaluator in Smalltalk. Details of implementation of the measured evaluators are given in the following.

- Dynamic evaluator based on Jones [?] (or simply **Dynamic** for further explanation), which consists of three phases, dependency graph construction (**Graph**), scheduling of evaluation sequence including detection of cycles (**SCC**) and attribute evaluation (**EvalD**)
- Naive static evaluator (**Static**) based on Babich,[?] which performs evaluation by iteratively applying a static evaluator [?] made from an attribute grammar ignoring remote references, where the iteration terminates when all the remote attributes converge [*7]
- Mostly static evaluator (**MStatic**) shown in the previous section, which consists of remote edge analysis (**Edge**) and attribute evaluation (**EvalM**)

Table 1 gives the elapsed time of evaluation for various programs, to-

[*7] **Static** is simpler than the evaluator shown in Section 4, and reevaluates the whole derivation tree until convergence.

Table 1 Performance of CRAG evaluators for various source programs (elapsed time in msec.)

	Graph	**SCC**	**EvalD**	**Dynamic**	**Static**	**Edge**	**EvalM**	**MStatic**
Loop10	6.5	8.5	6.0	21.0	6.0	1.0	4.0	5.0
Loop30	6.0	8.5	7.5	22.0	6.5	1.0	4.5	5.5
Loop50	6.0	9.5	8.0	23.5	6.5	1.0	5.0	6.0
Loop70	6.0	9.5	8.5	24.0	6.5	1.5	5.5	7.0
Loop90	6.0	10.0	10.0	26.0	6.5	1.5	6.0	7.5
LoopSeq	6.5	12.0	11.0	29.5	9.5	1.5	8.5	10.0
Nest2	6.0	10.0	8.5	24.5	7.0	1.0	5.5	6.5
Nest3	6.0	11.0	8.5	25.5	7.5	1.0	5.5	6.5

gether with the time of each phase. Each of the evaluation time is the mean of a total of 20 consecutive runs. All of the source programs have 1000 assignment statements and the length of each bit-vector is 300. Programs *Loopn* ($n = 10, 30, 50, 70, 90$) include a single loop with 100, 300, 500, 700, 900 assignment statements respectively inside their body. Program *LoopSeq* includes 10 independent loops with 50 statements in each. Programs *Nest2* and *Nest3* include 500 statements in their loop body like *Loop50* but their loops are nested in double and triple, respectively.

Here, the results do not include the time for parsing source programs and linking remote references between "label" and "goto" nodes. In **Static** and **EvalM** of **MStatic**, for saving information of live variables, the value of each "out" attribute is stored into the tree node which the attribute belongs to[*8].

The performance has been measured on CMU Common Lisp 18d running on a UltraSparc IIi 333MHz with memory 128MB and Cache (Instruction, Data, External) 16KBytes, 16KBytes and 2MBytes under Solaris 2.8. Each of the evaluators has been compiled by CMUCL's native compiler.

6.2 Evaluation and discussion

In this section, we will discuss advantages and disadvantages of mostly static evaluator, by comparing with the dynamic evaluator and the static evaluator for CRAGs. Empirical comparison of static and dynamic evaluators for attribute grammars are few, as far as we know. Therefore we think our results give new insight to a variety of evaluators for attribute grammars.

[*8] Otherwise, live variable information cannot be obtained since attribute values are intermediate results to compute the synthesized attribute of the root of the tree in **Static** and **EvalM**.

[1] Comparison with dynamic evaluator

The dynamic evaluator for CRAGs needs a long time for **SCC**, that detects cycles and determines the evaluation order. **SCC** takes a time proportional to the number of attribute instances. The algorithm of **Edge** in **MStatic** is also almost scalable to the number of the synthesized attributes. However, **Edge** is an order of magnitude faster than **SCC**, because **Edge** can be realized as a simple bottom-up visit over the tree and its set operations based on bit-vectors are very fast.

As for the total number of applications of semantic functions, **EvalD** is the optimal but **EvalM** is not in general. For example, the dependency graph in Fig. **??** shows that the iterative evaluation by **EvalD** will be done on the part of SCCs made by the "goto" and the "label", but in **EvalM**, which is realized by recursive functions along the tree, all the attributes below the label node will be iteratively evaluated.

However, the result shows superior performance of **EvalM** regardless of such redundant computation of attributes. There are two reasons of this. First, the cost of the redundant computation is low, since almost all the part of such redundant computation is applications of "copy rules". This pattern is made by "threading" of attributes, which generally appears not only in backward data-flow analysis like this example but also in forward data-flow analysis like constant propagation analysis. Second, **EvalM** inherently has better performance than **EvalD**. This is because **EvalM** does not need to store all of the attribute values into tree nodes in contrast to **EvalD**. It is also noteworthy that **EvalM** can easily get more performance by optimizing the evaluator itself by an existing optimizing compiler.

[2] Comparison with static evaluator

Table 1 shows that **MStatic** runs faster than **Static** for source programs *Loop10*, *Loop30* and *Loop50*. These gains of time for **MStatic** account for prevention of unnecessary iterative evaluation by the remote edge analysis (i.e. **Edge**). Such unnecessary computation cannot be avoided by **Static.** In **Static**, whenever a dependency graph includes some cycles, that is, the source code includes loops, iterative evaluation is applied to whole of the source program. This causes a large amount of redundant computation if attributes inside cycles are very few as in the case of *Loop10*. Moreover, this overhead becomes greater especially when the number of the iterative application of the evaluator rises [*9], or applications of semantic functions cost high.

But for the source programs *Loop70* and *Loop90*, **MStatic** is slower than **Static**. The reason is that the cost of **Edge** does not pay to remove **Static**'s redundant computation. These are cases that there is little difference in numbers of the attributes iteratively computed between in **EvalM** and in **Static**. Such cases occur when most of the assignment statements are in a loop. [*10]

When a given source program has nested loops (as in *Nest2* and *Nest3*), the mostly static evaluator will skip iterative evaluation within a cycle as mentioned in Section **??**. The result shows that this evaluation strategy works effectively as the level of a nested loop does not affect the performance.

As for the result of *LoopSeq* where all the three kinds of evaluators make convergency tests for ten remote attributes, all the evaluators take more time than in other examples. But this is due to naive implementation of the convergency test. By comparing the result of **Mstatic** to that of **Static**, we can say the number of loops does not affect **Mstatic**'s performance so much.

6.3 Future work with mostly static evaluator

Although the mostly static evaluator effectively works, there is one shortcoming in the current implementation of the mostly static evaluator. The shortcoming is that the evaluator cannot avoid redundant computation effectively when some form of syntax tree is given. In Fig. **??**, attributes that need to be iteratively evaluated are those in the cycle occurring between the "label" and the "goto" node. But in **MStatic**, iterative evaluation will be performed for all the attributes below the label statement. This includes a large number of attributes below the else node, which are not in the cycle. Although this problem has not been carefully investigated, caching of relevant attributes may solve the problem.

§7 Related Work

Babich and Jazayeri[?, ?] first showed a way to construct data flow analyzers of optimizers by means of attribute evaluation. Their algorithm is described by simply applying unidirectional attribute evaluation (i.e. tree traversal either

[*9] The number of iterative application depends on AG descriptions and given syntax trees.

[*10] Although such cases exist in some application, it seems that loops with large bodies are rare. For example, Knuth found in his research on various Fortran programs that 87 percent of DO loops has less than six statements in their body. [?]

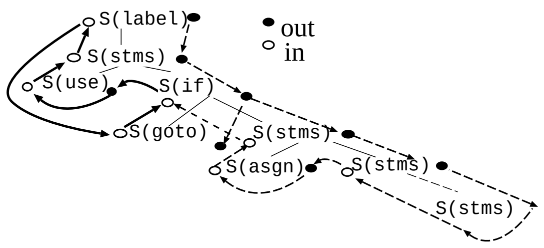


Fig. 9 Case that **MStatic** cannot effectively avoid redundant computation (represented by the broken arrows)

in left to right or right to left order) repeatedly until convergence. However, this evaluation algorithm is inefficient in the sense that the whole tree should always be evaluated although circular dependencies may arise only in a part of the tree.

In succeeding work, circular but well-defined AGs have been proposed as a formulation of AGs that include circular dependencies:

- Farrow[?] has proposed the finitely recursive AGs and showed a way to make a static evaluator for it as we have mentioned before (see Section **??** and **??**).
- Jones[?] has proposed another circular AGs, where the conditions for the termination of the fixed-point calculation are almost the same as those of Farrow. He has also shown a dynamic evaluator that can run in efficient evaluation order and that supports incremental evaluation. His group has applied the extended AGs to a VLSI design system. This dynamic evaluator, however, is not suitable for the application area where the cost of the run-time calculation of evaluation order cannot be ignored, like optimizers of compilers.

On the other hand, there are many extensions to traditional AGs that allow remote accesses to attributes.

- In *GAG*,[?] direct non-local dependencies can be described, though it is only applicable in restricted patterns that are indicated by special instructions, i.e. "including" (reference to distant attribute instances in an ancestor node) and "constituents" (aggregation of attribute values of distant nodes within the subtree).
- Johnson and Fischer proposed nonlocal attribute grammars[?], which can declaratively define both how links are made and how non-local attribute accesses are performed via these links. They also showed an incremental

evaluator for the extended grammars. For proper evaluation ordering, the evaluator uses priorities of attributes which are computed statically from the grammar, but sometimes it cannot resolve evaluation order between attributes.

- Vorthmann proposed an incremental evaluator for attribute grammars on DR (Declaration-Reference)-threaded trees, i.e. trees with what we call links, which allow non-local attribute references[?]. The proposed evaluator employs dependency graphs called ICG graph which is similar to our $RDG^*(p, E)$ for the scheduling of incremental evaluation. Unlike our evaluation strategy, the evaluation order of attributes are dynamically scheduled in this evaluator.
- The extension shown by Boyland [?] allows both remote reference and remote definition. He showed a technique that translates extended AGs to traditional AGs (actually, conditional AGs proposed by himself [?]). This is achieved by introducing the *control attribute* that is used only for scheduling, and the existing evaluator for conditional AGs can be applied to the transformed AGs.
- The *reference attributed grammars*[?] have been proposed as a natural and powerful formulation supporting remote access to attributes. The extension allows attributes to be references to nodes, or what we call links or pointers.

However, all these extensions for remote attribute accesses have not been considered for circular AGs that need fixed-point computations.

Boyland in his thesis made an attribute grammar system that allows both circularity and remote attributes[?]. Although his system has more expressiveness and supports incremental evaluation, the evaluator is demand driven and completely dynamic.

Note that those extensions by Johnson and Fischer[?], Boyland[?, ?] and Hedin[?] allow references to attributes or to nodes to be first-class attribute values and they also allow remote accesses of attributes via such references. These approaches enable one to declaratively describe incremental evaluation of trees which involve complicated manipulation for both dynamic changes of links as well as dynamic updates of the tree.

Compared to them, in our formalism, references to nodes (or links between nodes) are given externally and no attribute values include references to nodes. This allows simple grammar descriptions and enables construction of efficient static evaluators by means of relatively simple extension of existing evaluators.

Although we omit formalism of how to establish links at present, it could also be described declaratively in a traditional AG where attribute values may include references to nodes (but accesses via these references not allowed.) Our CRAG formalism with such a declarative description of links will give a completely declarative framework[*11].

§8 Conclusion

We have presented the circular remote attribute grammars (CRAGs) as an extension to traditional AGs which allows remote attribute references and circular dependencies of attributes. This greatly broadens the application area which can be described in AGs. We have also shown a way to construct an efficient evaluator for CRAGs called mostly static evaluator. Performance of this evaluator has been measured and compared with dynamic and static evaluators. The result shows that the mostly static evaluator dominates the dynamic evaluator and outscores the static evaluator when a typical source program is given. This fact shows the efficiency of mostly static evaluators for language tools, which leads to the usability of CRAGs. We made several applications using CRAGs, such as an SSA translator [?] and optimizers like one for partial redundancy elimination with lazy code motion.[?]

There is some room for further investigation. First, the mostly static evaluator should be revised so as to further reduce redundant evaluation. As mentioned in Section 6.3, for some form of input trees a few redundant evaluation of circularly defined attributes still remains which arises from the structure of our tree-based evaluator. Second, although our experience of realization of translators and optimizers have shown usability of CRAGs, we should have more applications based on CRAGs and evaluate it in terms of both conciseness of the specification and performance of the attribute evaluator.

Acknowledgements

The authors would like to thank John Tang Boyland and the anonymous referees for their comments on an earlier draft.

[*11] Boyland's circular and remote AGs have more expressive power in the sense that they allow arbitrary accesses to remote attributes via references.

References

1) Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

2) Babich, W. A. and Jazayeri, M., "The Method of Attributes for Data Flow Analysis Part I," *Acta Inf.*, *10*, *3*, pp. 245–264, 1978.

3) Babich, W. A. and Jazayeri, M., "The Method of Attributes for Data Flow Analysis Part II," *Acta Inf.*, *10*, *3*, pp. 265–272, 1978.

4) Boyland, J. T., "Conditional Attribute Grammars," *ACM Trans. Prog. Lang. Syst.*, *18*, *1*, pp. 73–108, 1996.

5) Boyland, J. T. *Descriptional Composition of Compiler Components.* PhD thesis, University of California Berkeley, 1996.

6) Boyland, J. T., "Analyzing Direct Non-local Dependencies in Attribute Grammars," in *Proceedings of International Conference on Compiler Construction (CC'98)*(Koskimies, K. ed.), volume 1383 of *Lecture Notes in Computer Science*, pp. 31–49. Springer-Verlag, 1998.

7) Farrow, R., "Automatic Generation of Fixed–Point–Finding Evaluator for Circular, but Well–Defined, Attribute Grammars," in *ACM SIGPLAN '86 Symposium on Compiler Construction*, pp. 85–98, 1986.

8) Hedin, G., "Reference Attributed Grammars," in *Proceedings of Second Workshop on Attribute Grammars and their Applications (WAGA'99)*(Parigot, D. and Mernik, M. eds.), pp. 153–172. INRIA rocquencourt, 1999.

9) Johnson, G. F. and Fischer, C. N., "A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 141–151, 1985.

10) Jones, L. G., "Efficient Evaluation of Circular Attribute Grammars," *ACM Trans. Prog. Lang. Syst.*, *12*, *3*, pp. 429–462, 1990.

11) Kastens, U., Hutt, B. and Zimmermann, E., *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

12) Katayama, T., "Translation of Attribute Grammars into Procedures," *ACM Trans. Prog. Lang. Syst.*, *6*, *3*, pp. 345–369, 1984.

13) Knuth, D. E., "Semantics of Context-Free Languages," *Math. Syst. Th.*, *2*, *2*, pp. 127–145, 1968. correction: *5*, *1*, pp. 95–96, 1971.

14) Knuth, D. E., "An Empirical Study of FORTRAN Programs," *Software–Practice and Experience*, *1*, *2*, pp. 105–133, 1971.

15) Rodeh, M. and Sagiv, M., "Finding Circular Attributes in Attribute Grammars," *Journal of the ACM*, *46*, *4*, pp. 556–575, 1999.

16) Sasaki, A. and Sassa, M., "Circular Attribute Grammars with Remote Attribute References," in *Proceedings of 3rd Workshop on Attribute Grammars and their Applications (WAGA 2000)*(Parigot, D. and Mernik, M. eds.), pp. 125–140. INRIA rocquencourt, 2000.

17) Tachikawa, S., "Implementation of Lazy Code Motion with Circular Remote Attribute Grammars," 2002, Bachelor's thesis, Dept. Information Science, Tokyo Institute of Technology (in Japanese).

18) Vorthmann, S. A., "Coordinated Incremental Attribute Evaluation on a DR-Threaded Tree," in *Attribute Grammars and their Applications. International Conference WAGA Proceedings* (Deransart, P. and Jourdan, M., eds.), volume 461 of *Lecture Notes in Computer Science*, pp. 207–221. Springer-Verlag, 1990.

Authors' profile

Akira Sasaki, Master of Science: He is a research fellow of the Advanced Clinical Research Center, the Institute of Medical Science, the University of Tokyo. He received his BSc and Master of Science from Tokyo Institute of Technology, Japan in 1994 and 1996, respectively. His research interests include: programming languages, programming language processors and programming environments, especially, compiler compilers, attribute grammars, and systematic debugging. He is a member of Japan Society for Software Science and Technology.

Masataka Sassa, Doctor of Science: He is Professor of Computer Science at Tokyo Institute of Technology. He received his BSc, MSc and Doctor of Science from the University of Tokyo, Japan in 1970, 1972 and 1978, respectively. His research interests include: programming languages, programming language processors and programming environments, currently focusing on compiler optimization, compiler infrastructure, attribute grammars, and systematic debugging. He is a member of the ACM, IEEE Computer Society, Japan Society for Software Science and Technology, and Information Processing Society of Japan.