# Circular Attribute Grammars with Remote Attribute References

Akira Sasaki & Masataka Sassa

*Department of Mathematical and Computing Sciences*
*Graduate School of Information Science and Engineering,*
*Tokyo Institute of Technology*
{sasaki, sassa}@is.titech.ac.jp

**Abstract**

Attribute grammar (AG) is 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 AG. This is due to two problems which former researches could not deal with simultaneously, i.e., references of attribute values on distant nodes and circularity in attribute dependency. This paper proposes *circular remote attribute grammar* (CRAG), an extended AG which allows (1) direct description of relations between two distant attribute instances through pointers referring to other nodes in the derivation tree, and (2) circular dependency 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 labeled jumps. We will also show a way to construct a static evaluator for CRAGs that has efficient run-time performance. The usability of CRAG is also presented by an example application, the SSA translator, which is a part of our compiler for a subset of C language.

## 1. Introduction

Many applications of attribute grammars (AGs) [11] 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, relatively few AG 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 thought to be meaningless, or not to be well-defined. Applications which inherently lead to circular dependency, for example data-flow problems which are solved well by recursive equations, were hard to write in AG descriptions. To overcome this problem, Babich and Jazayeri [1] [2], Farrow [6] and Jones [8] proposed extended AGs that allow attribute instances to have circular dependency.

Second, in traditional AGs, a semantic rule in one production must be defined only from the attribute occurrences in the same production. This restriction causes difficulty in writing relations between attribute instances belonging to two distant nodes, since extra attributes for passing values to remote nodes are needed in order to conform to the restriction of traditional description style of AGs. This gives rise to not only complex AG descriptions which are hard to read but also inefficient attribute evaluation. Many researches (e.g. [9] [7] [4]) cope with this problem by adding schemes for non-local dependency or remote references, that is, allowing attributes to refer to values of attributes in far away nodes.

However, there has been no research that could remove the above two restrictions simultaneously, i.e. circular dependency and remote references. For example, data-flow problems on languages including unrestricted labeled jumps such as "goto"s were hard to be described because of the circular dependency and references to distant nodes.

To solve the above problem, this paper proposes *circular remote attribute grammars* (CRAGs) – circular attribute grammars that allow remote attribute references. CRAGs are an extension to traditional AGs which

1. allow direct description of relations between two distant attribute instances through pointers referring to other nodes in the derivation tree, and
2. allow circular dependency 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 labeled jumps.

In addition, we will show a way of constructing a static evaluator for CRAGs which is an extension of the recursive style evaluator [10] for absolutely non-circular AGs. The evaluator for CRAGs can calculate attribute values efficiently since it considers attribute dependency which may be caused by remote references at generation time.

The structure of this paper is as follows. In Section 2 we explain our extended AG called CRAG. Section 3 shows a way to construct the static evaluator for CRAG. Section 4 illustrates an application written in CRAG which is a part of our compiler. Section 5 discusses related work. Section 6 gives future work and our conclusion.

# 2. Remote Attribute Grammar (RAG)

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 the labeled jump control structure (i.e. "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. This is because in traditional AGs direct dependency is restricted only to attributes in a parent-child relationship in the derivation tree.

We propose *remote attribute grammars* (RAGs) [1] which allow non-local dependency by introducing pointers on nodes of the derivation tree. The example shows a RAG description for liveness analysis in a small language which includes the "goto"-"label" control structure.

Remote Attribute Grammar G1

```
S ::= S ";" S   /* (stms) */
{ S3.out = S.out;
  S2.out = S3.in;
  S.in = S2.in;
}
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);
}
```

[1] Hedin [7] has proposed *reference attributed grammars (RAGs)*. However, for the explanation of our circular remote attribute grammars we also use this abbreviation *RAG* in Section 2.

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

In this example, "`%prod label`" declares that this production (label) has the name "`label`". The associated 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 derivation tree for G1, associated links are properly provided between the "goto" nodes and the "label" nodes, e.g. by a previous phase. The actual link is not described in the grammar.

Using the above links, `label.S.in` in production (goto) remotely refers 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". This dependency is described by the broken arrow in Figure 1.

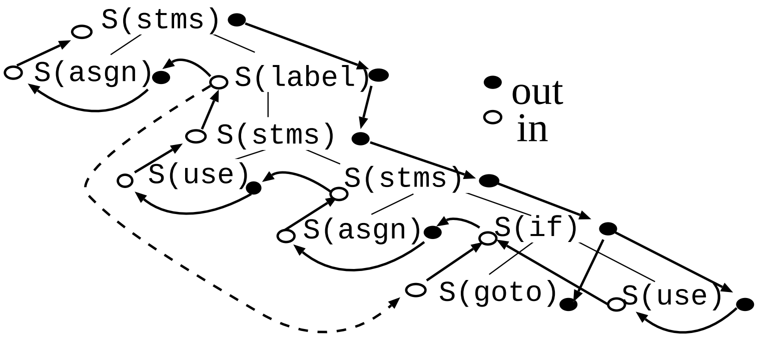



Figure 1 Attribute dependency graph with remote references

The semantic rule in production (goto) describes a part of 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 structure. 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.

## 2.1. Definition of RAGs

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

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

Here, $p_i$ is called the remote referenced production (or simply remote production) and $p_j$ is called the remote 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

$$a_{jl} = f_{jl}(\ldots a_{ik} \ldots)$$

where $a_{jl}$ is an attribute occurrence in $p_j$ , $f_{jl}$ is its associated semantic function and $a_{ik}$ is an attribute occurrence in remote production $p_i$ . Here, we call $a_{jl}$ a reference attribute and $a_{ik}$ a remote attribute respectively.

### 2.2. Well-Defined RAGs and Circular RAGs

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 derivation tree $t$.

Next, we show well-defined-ness of RAGs which contain circular dependency between attribute instances. Former researchers suggested extensions to traditional AGs that allow such circular dependency[6][8]. In Farrow's scheme, under the conditions described below, values of circularly defined attribute instances are determined uniquely by means of a finite iteration of the evaluation of 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.

Here, $f(s[k]) = f(s[k+1])$ is called the least fixed-point.

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

## 3. 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.

### 3.1. Static Evaluator

Static evaluators for AGs do not always achieve the optimal order of attribute evaluation. 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 [6]. Similarly to his work, the evaluator for CRAGs can be constructed by extending the recursive style evaluator for absolutely (strongly) non-circular AG [10].

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

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

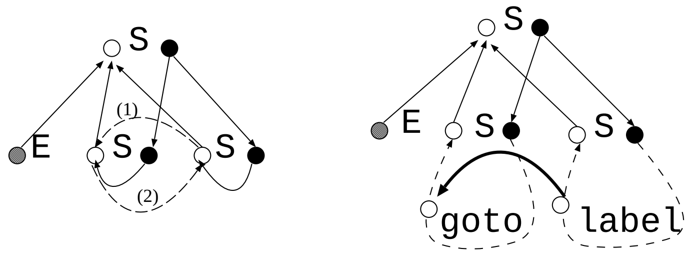
Here, each function $R\_X.s$ is associated with synthesized attribute $s$ of nonterminal $X$ and is called a *synth-function* because it is intended to compute the value of $X.s$ at the root of derivation tree $T$. Parameters $i_0 \dots i_n$ are inherited attributes of $X$ that are needed to calculate the value of $X.s$ for all possible derivation 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
    $p_1$: $H_{p_1}$
      $\dots$
    $p_n$: $H_{p_n}$
$\}$

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

Each element of $H_p$ may be 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 3.2. The ordering of elements in each $H_p$ is determined by the *augmented remote dependency graph* $RDG^*(p)$, which is obtained from $DG^*(p)$ for production $p$ [10] with additional edges, *indirect remote dependency edges.* Here, 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 derivation trees generated from production $p$. For example, Figure 2(a) represents $RDG^*(\texttt{if})$ 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)

Figure 2 $RDG^*(\texttt{if})$

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 derivation 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 dependency actually arises.

Using $RDG^*(p)$, the sequence of attribute evaluation is computed. First, $RDG^*(p)/CDC$ is constructed in order to collect circularly defined attributes into groups. Its vertex is a CDC (Circular Dependency Class) which is given by grouping together vertices of $RDG^*(p)$ included in the same cycle. Its edge 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 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 3.2.

When computing the values of remote attributes, i.e. remotely referenced attributes, they are "cached" into a global "dictionary" (or symbol table) in addition to storing their values into local variables of a synth-function. At reference sites, on the other hand, 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.

## 3.2. 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).

### 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 dependency, 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 $C$* [2].

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

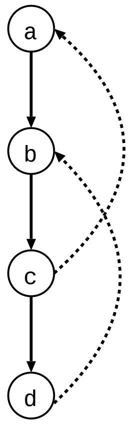
From the assumption, we can say that for $C$'s edges except those which are made by edges representing remote references, a total order of attribute instances can be made. The iterative evaluation of attributes in $C$ will be made according to this total order. Figure 3(a) illustrates an example of the 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. Figure 3(b) is the corresponding iterative evaluation routine that can be made according to the total order ⟨`a b c d`⟩.

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.

Now, we can conclude the following: 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

[2] In the case that the cycle arises from only normal (i.e. not remote) dependency, this property also holds by transforming such dependency as follows: delete some direct edges in cycle $C$ of production $p$ so that the dependency without these edges does not contain any cycle, then regard such deleted edges as remote edges referring to attributes in the same production $p$.




(a) example dependency graph

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

(b) iterative evaluation

Figure 3 Example dependency graph and its corresponding iterative evaluation routine

outside $C$ on which attributes in $C$ depend. This means that Property 1 holds. In the example, if both values of remote attribute `c` and `d` have converged at the $n$-th iteration, all attribute values have converged. (end of Property 1)

### Iterative Evaluation Routine

In Farrow's evaluator, if there is another cycle during the iterated evaluation of circularly defined attribute values (i.e. nested cycles), nested loops are constructed. This 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 CRAG is made in a way that it does not execute iterations on the inner loops. Taking this into account, the routine for calculating attribute values in a cycle becomes as follows.

```
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
}
```

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 these remote attributes being unchanged (i.e. by "`UNTIL convergent`"). Variable `isInCircle` is a global boolean variable representing whether the computation is within iterative evaluation or not, for preventing the problem of nested loops described before.

## 3.3. Example Evaluator

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

```
function R_S.in(T, S.out){
 case production(T) of
   assign:
      ...
   label:
     S2.out = S.out;
     if(!isInCircle){
       isInCircle = TRUE;
       initRemote(T, "label.S.in");
       REPEAT
         S2.in = R_S.in(T, S2.out);
         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, 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 3.2. 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 "label" in `T`, i.e. the node representing "goto".

## 3.4. Selecting Appropriate Evaluation Order at Evaluation Time

The static evaluator described above does not need any calculation of the evaluation order at evaluation time, while 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 are contained, that is, in the case of $p$ being "if", "stms" and "label". This means that for most derivation trees, the execution may encounter the part of iterative evaluation for calculating circularly defined attribute values at the synth-function of the root node. So, even if $RDG^*(t)$ 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 $RDG^*(p)$s are made. So the basic idea for the refinement is to divide those unioned edges.

$H_p$ of the revised synth-function associated with $X.s$ (Section 3.1) now has several versions of attribute evaluation sequences. These versions correspond to the partial orders generated from $RDG^*(p)$ by classifying patterns of the actual occurrence of indirect remote dependency edges. Considering $RDG^*(\texttt{if})$ of G1 as an example, there exist four patterns of partial orders, that is: (a) each subtree does not contain "goto"s, (b) then-clause includes "goto"s whose destinations are in else-clause, (c) the reverse pattern of (b), and (d) (b)and(c) occur simultaneously. The dependency graphs for these patterns are illustrated in Figure 4.

To select an appropriate version at evaluation time, information of the remote edges on nodes

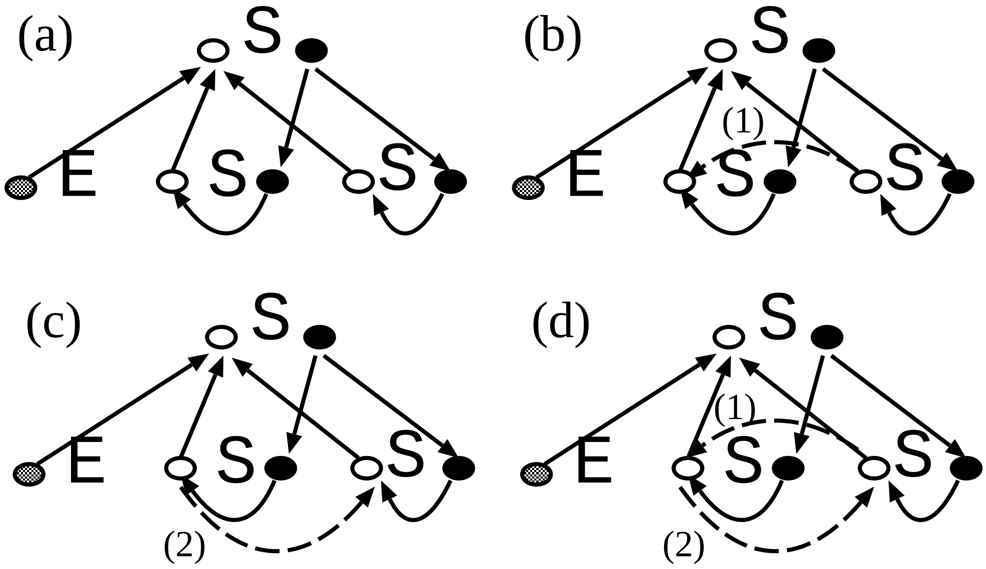



Figure 4 Four patterns of order in $RDG^*(if)$

of the given derivation tree is needed in advance. In previous example, each "if" node needs to be classified by the above patterns (a)-(d) before evaluation, which means each "if" node should have information about whether indirect remote dependency edges (1) and/or (2) of Figure 4 exist or not. This information can be stored by a pre-processor that collects all "goto" and "label" nodes under the then-clause and the else-clause of each "if" node, and then finds associated "goto"-"label" nodes, i.e., indirect remote dependency edges. Such pre-process can be made by a 1-pass traversal of the tree. This achieves the efficient evaluation based on the precise partial order instead of that by the rough estimation, preventing unnecessary iterations which cannot be detected at generation time.

## 4. Compiler Construction using CRAGs

In this section we will show how CRAGs are applied to compiler construction, by presenting an example description in CRAG which is the specification of a part of a compiler we have been developing.

Our research group has been investigating the method to obtain optimizing compilers from AG specifications throughout all compiler phases and has successfully developed an experimental compiler. The compiler is for a subset of C programming language where one of its restricted features compared to C is the lack of labeled jump control structure. One major obstacle for generation of compilers from AG specifications was circular attribute dependency in optimizers. But this has been overcome by Jun [12], an attribute evaluator generator that accepts finitely recursive AGs as input.

Now we are developing a compiler for the full set of C. This can be achieved by CRAGs defined previously. We have implemented a prototype system that generates an attribute evaluator from CRAG description in the way as proposed in Section 3. By this system, several phases of the compiler for a language which is almost the full set of C have been made.

## SSA transformation

An example of CRAG is illustrated in the following. This is a specification of a phase called "SSA transformation", which transforms a source program into an intermediate representation. The intermediate representation is based on the static single assignment form (SSA form) [5] that is proposed as a suitable intermediate representation for compiler optimizers. The program in SSA form has the property that there is only one assignment to each variable. The way to obtain SSA form is the following:

1. Each variable `V` in the program is renamed to $\mathtt{V_i}$, each of which has only one assignment and the uses of `V` are replaced by the appropriate variable $\mathtt{V_i}$ (`i` is called index).
2. At each join node, the *pseudo-assignment*, which is of the form $\mathtt{V_i} \leftarrow \phi\ (\mathtt{V_j},\ \mathtt{V_k})$, is added. This means that if control reaches this node along the first input edge, $\mathtt{V_i}$ is assigned the value of $\mathtt{V_j}$, and if it is along the other edge, $\mathtt{V_i}$ is assigned the value of $\mathtt{V_k}$. Here the right hand side of the pseudo-assignment is called $\phi$-function.

| (a) original program | (b) corresponding SSA form |
|---|---|
| $a = 1;$<br>L1:<br>if $(a < 3)$ {<br>  $a = a + 1;$<br>  goto L1;<br>  $a = 2;$<br>} else {<br>  $a = a + 3;$<br>}<br>$a = a + 2$ | $a_1 = 1;$<br>L1:<br>$a_2 = \phi(a_1, a_4);$<br>if $(a_2 < 3)$ {<br>  $a_4 = a_2 + 1;$<br>  goto L1;<br>} else {<br>  $a_6 = a_2 + 3;$<br>}<br>$a_7 = a_6 + 2;$ |

Figure 5 SSA transformation

Figure 5 is an example of a source program and its corresponding program in SSA form [3]. The property of SSA form naturally expresses use-def chains of a program. Optimizers that process the SSA form can be made in a simple way and can give faster algorithms for code transformations.

Our SSA based intermediate representation is in the form of an abstract syntax tree, and each node of the tree representing a "goto statement" is connected with a node of the associated "label statement" by a pointer. We call this intermediate representation "the intermediate tree", which is processed by the attribute evaluators generated from the specification of optimizers written in CRAG.

The SSA transformation consists of two phases, the analysis phase and the code transformation phase (simply, the analyzer and the code transformer). The analyzer collects information for the transformation into nodes of the intermediate tree. This information is used by the code transformer that performs the actual transformation of the intermediate tree. The actual passing of the information is in such a way that the analyzer (i.e. the attribute evaluator) stores attribute values into corresponding nodes of the intermediate tree, and the code transformer refers to these stored values.

We now explain the specification for the analyzer written in CRAG. The analysis phase is divided into two subtasks.

1. giving a unique number, i.e. index, to each definition of a variable.
2. solving which definition each variable at use-site should refer to (i.e., reaching definition)

In the specification below, we will highlight only the subtask 2 due to space limitation. Throughout the description, a synthesized and an inherited attribute, "`var_in`" and "`var_out`", play a main

[3] The control doesn't reach "a=2" of Figure 5(a) that has been removed in the corresponding SSA form (b). This can be done by our SSA transformation phase described later.

role on the process. Both values indicate a mapping from names of the original variables to indices representing their reaching definitions at every point of the program. For nonterminal `A`, the values of "`A.var_in`" and "`A.var_out`" are such mappings at the entry of and at the exit of the point that `A` represents respectively. For example, if "`var_in`=((v 3)(u 5))", it means definition 3 of variable "v" (i.e. $v_3$) and definition 5 of variable "u" (i.e. $u_5$) reach the entry of `A`. In the case that no control reaches `A`, the value of these attributes is "NULL".

```
assign  => CVAR, EXP;
{
   assign.var_out =
          replace-element(EXP.var_out, CVAR.name, assign.def_id );
   EXP.var_in = assign.var_in;
}
```

In this part, the attribution at a node representing "assignment statement" is specified. Since our evaluator runs on the abstract syntax tree, the part of the production of AG is described in the form of the tree. Namely, the part "`assign => CVAR, EXP;`" means that each node for "assignment statement" is labeled as "`assign`" and has two subtrees, `CVAR` and `EXP` that represent a variable and an expression respectively.

Here, "`CVAR.name`" indicates the name of the assigned variable and "`assign.def_id`" is the unique number given to the "assignment statement". Each value of these two attributes is given previously by the subtask 1 of the analyzer described before and can be treated as a constant. The attribution rule here means that after "assignment statement" the reaching definition for "`CVAR.name`" is updated to this assignment itself (i.e. "`assign.def_id`"). The function "`replace-element`" describes this update of the mapping. For example, if "`CVAR.name`=v", "`assign.def_id`=4", and "`EXP.var_out`=((v 3)(u 5))", then "`assign.var_out`" is updated to ((v 4)(u 5)).

```
stms => CSTM, CSTM;
{
   CSTM[1].var_in = stms.var_in;
   CSTM[2].var_in = CSTM[1].var_out;
   stms.var_out = CSTM[2].var_out;
}
```

This is the part of nodes representing a sequence of statements. The "threading" technique is used for propagating a value from one statement to the next statement.

```
goto => LEXICAL %prod goto;
{
   goto.var_out = NULL;
}
```

This is the part of the node of "goto statement". Here, "`LEXICAL`" represents the destination label for a "goto statement". As described before, the production is named "`goto`" by declaration "`%prod goto`". The semantic rule here means that no control reaches the statement after this "goto statement". The passing of the attribute value to the associated "label statement" is not described here but in the "label statement" part coming next.

```
label => LEXICAL, CSTM %ref goto;
{
   CSTM.var_in =
      merge (label.var_in, goto.goto.var_in label.def_ids);
   label.var_out = CSTM.var_out;
}
```

Here "`LEXICAL`" represents the label name of a "label statement". Two kinds of control can reach a "label statement"; the control from the statement before that "label statement" (`label.var_in`) and incoming control from "goto"s jumping to this label (`goto.goto.var_in`). As described before

"`%ref goto`" means that production "`goto`" is remotely referenced, and "`goto.goto.var_in`" means "`goto.var_in`" of that remotely referenced production "`goto`".

Here, the function "`merge`" computes a new mapping by comparing the reaching definitions for each (original) variable: for the variable whose reaching definition is not unique, an assignment using the $\phi$-function should be inserted and a new reaching definition is set. When the "label statement" has only one possible incoming control, the result of "`merge`" is the mapping coming along with that control. "`label.def_ids`" is similar to "`assign.def_id`" given before but contains new definition points *for every variable*, e.g. ((v 7)(u 8)).

For example, if "`label.var_in`=((v 4)(u 6))", "`goto.goto.var_in`=((v 5)(u 6))", and "`label.def_ids`=((v 7) (u 8))", then "`CSTM.var_in`" becomes ((v 7) (u 6)). This means that the insertion of $v_7 = \phi(v_4, v_5)$ is needed but no insertion of the $\phi$-function for $u_8$ is needed since the reaching definitions from both control edges are $u_6$.

The remote reference to "goto" made by "label" may cause circular dependency. But the domains of possible circularly defined attributes, i.e., `var_in` and `var_out`, make a c.p.o. and both of the functions "`merge`" and "`replace-element`" ensure the monotonicity and the ascending chain condition. Therefore, the values of these circularly defined attributes have a fixed-point.

Figure 6 and 7 illustrate the evaluation process by the evaluator generated from the above description. The intermediate tree corresponds to the example program of Figure 5(a). The number at the lower left of the "assign" nodes is the value of "`def_id`" and the value below the "label" node represents "`def_ids`". These attribute values are given by the previous phase. The values of "`var_in`" and "`var_out`" are placed on the left and right of the nodes respectively.

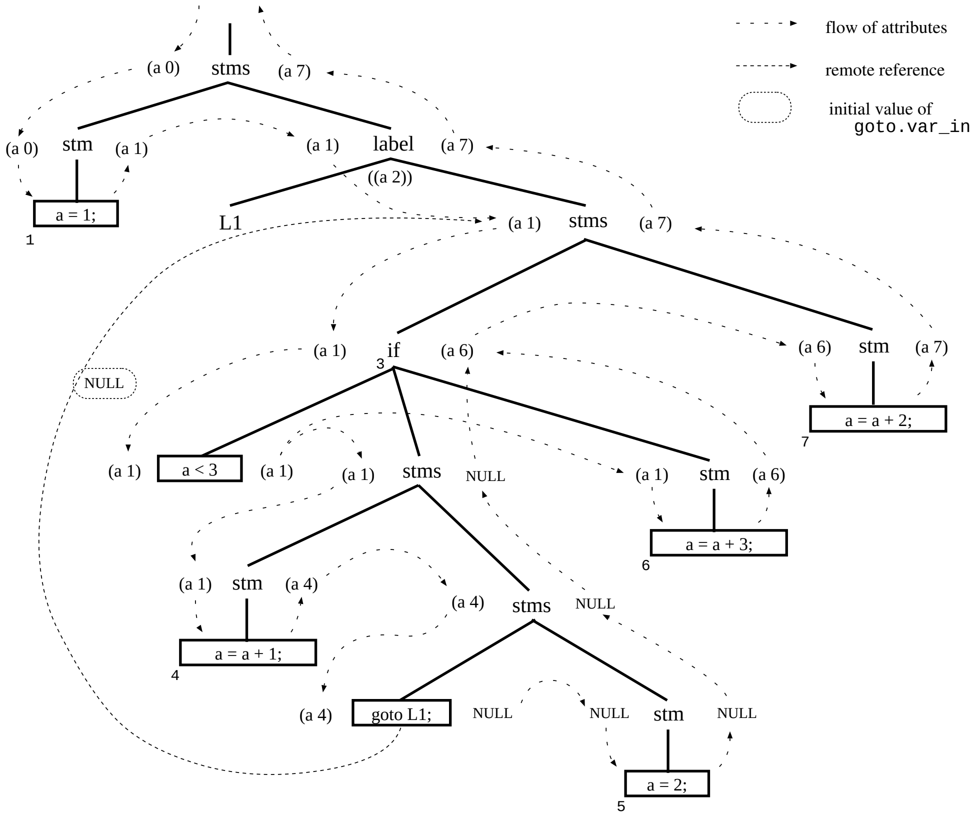



Figure 6 Flow of attributes(1): the first iteration for computing the fixed-point

In the case of this intermediate tree as input, the evaluator needs fixed-point computation, since there is a circular dependency among attributes. The iterative evaluation is terminated when the value of remote attribute "`var_in`" of the "`goto`" node has converged. The least fixed-point

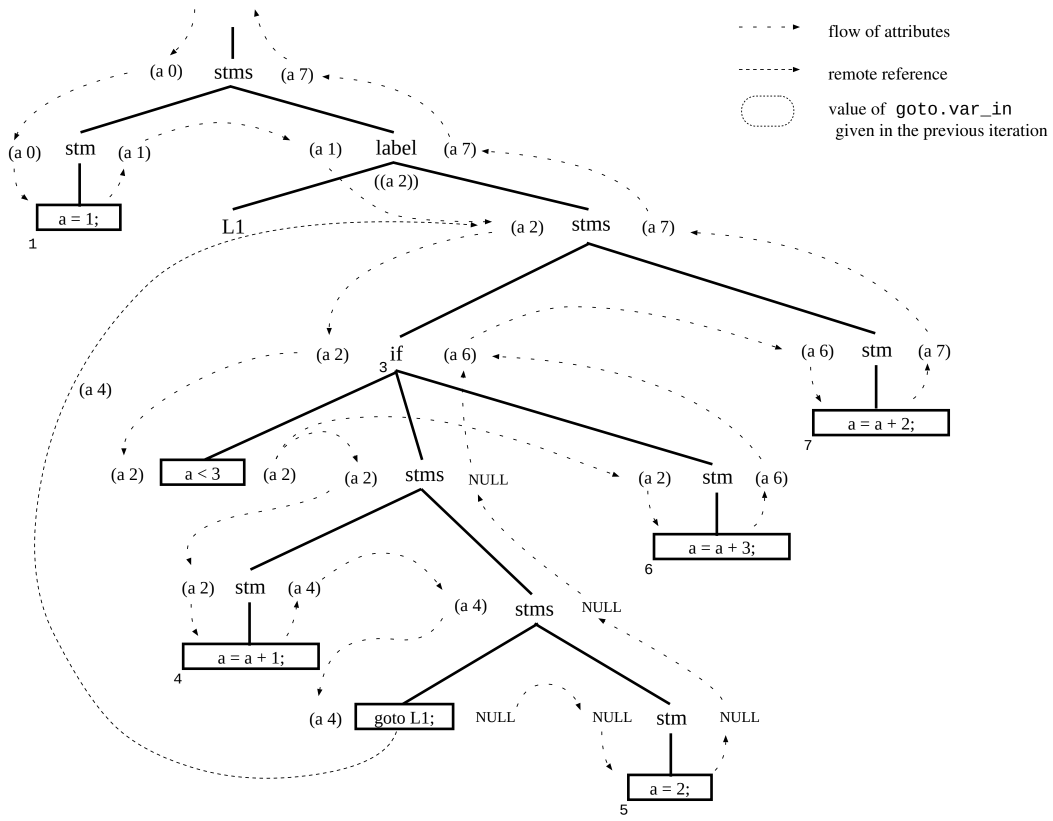



Figure 7 Flow of attributes(2): the second and third iteration of Figure 6

is calculated after the second iteration. However, to check the convergence, one more iteration is needed.

Figure 8 shows how the transformer rewrites the intermediate tree using the information passed from the analyzer. The new variables with unique indices are appropriately given to each assigned variable and the uses of each variable. At the "`label`" node, the assignment using the $\phi$-function, $a_2 = \phi(a_1,\ a_4)$, is inserted, since the indices of variable "a" along the two control edges are 1 and 4 (i.e. the values of "`label.var_in`" and "`goto.goto.var_in`") respectively, and $a$'s part of "`def_ids`" of the node is 2.

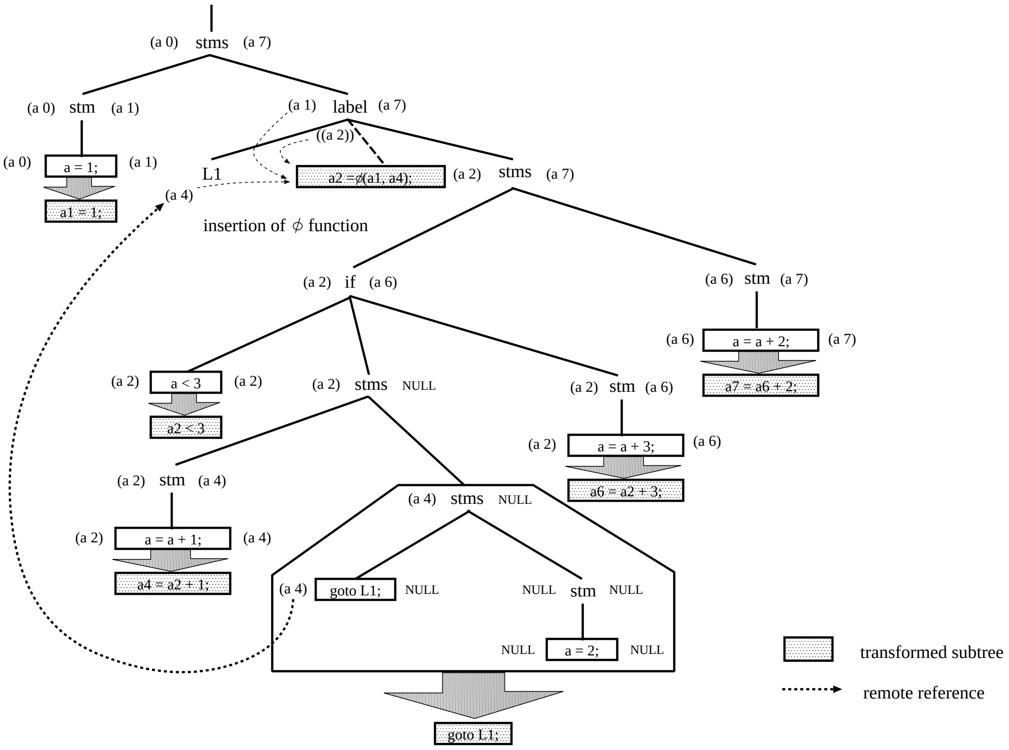



Figure 8 Transformation phase

## 5. Related Work

Babich and Jazayeri [1] [2] 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 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 dependency may arise only in a part of the tree.

In the succeeding work, circular but well-defined AGs have been proposed as a formulation of AG that includes circular dependency:

- Farrow [6] has proposed the finitely recursive AG and showed a way to make a static evaluator for it as we have mentioned before (see Section 2 and 3).
- Jones [8] has proposed another circular AG, 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* [9], direct non-local dependency 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).
- The extension shown by Boyland [4] allows both remote reference and remote definition. He showed a technique that translates extended AGs to traditional AGs (actually, conditional AGs proposed by himself [3]). 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.
- In the recent work by Hedin [7], 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.

## 6. Concluding Remarks

We have presented the circular remote attribute grammars (CRAGs) as an extension to traditional AGs which allows remote attribute references and circular dependency of attributes. We also showed a way to construct a static evaluator for CRAGs and its prototype implementation. Using this prototype generator, the SSA translator has been implemented as a part of our compiler from the specification written in CRAG. CRAG and its evaluator overcome the problem of circular dependency and remote attribute reference simultaneously. This will enable AG programmers to describe compilers for languages including any type of control structure in a more natural way.

There is some room for further investigation. First, CRAG has achieved the references to long-distant attributes through links owned by nodes of the tree. However, by now, this formulation does not determine precisely how those links are connected between nodes. One way to formalize this will be to use reference attributes as in [7]. By a precise description of links, the evaluation method based on the run-time (evaluation time) selection of attribution order (Section 3.4) can be made clearer. Second, although our experience of the realization of the SSA translator given in Section 4 shows usability of CRAG, we should have more applications based on CRAG and evaluate it in terms of both conciseness of the specification and performance of the attribute evaluator.

## Bibliography


[1] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis Part I. *Acta Inf.*, 10:245–264, 1978.

[2] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis Part II. *Acta Inf.*, 10:265–272, 1978.

[3] J. T. Boyland. Conditional attribute grammars. *ACM Trans. Prog. Lang. Syst.*, 18(1):73–108, 1996.

[4] J. T. Boyland. Analyzing direct non-local dependencies in attribute grammars. In *International Conference on Compiler Construction (CC'98)*, volume 141 of *Lecture Notes in Computer Science*, pages 31–49. Springer-Verlag, 1998.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, 1991.

[6] R. Farrow. Automatic generation of fixed–point–finding evaluator for circular, but well–defined, attribute grammars. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, 1986.

[7] G. Hedin. Reference attributed grammars. In *Second Workshop on Attribute Grammars and their Applications (WAGA'99)*, pages 153–172, 1998.

[8] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Trans. Prog. Lang. Syst.*, 12(3):429–462, 1990.

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

[10] T. Katayama. Translation of attribute grammars into procedures. *ACM Trans. Prog. Lang. Syst.*, 6(3):345–369, 1984.

[11] D. E. Knuth. Semantics of context-free languages. *Math. Syst. Th.*, 2(2):127–145, 1968. correction: 5(1):95–96, 1971.

[12] M. Sassa. Rie and Jun: Towards the generation of all compiler phases. In *3rd Int. Workshop on Compiler Compilers (CC'90)*, volume 477 of *Lecture Notes in Computer Science*, pages 56–70. Springer-Verlag, 1991.