Resolving Circularity in Attribute Grammars with Applications to Data Flow Analysis (*Preliminary-Version*)

S. Sagiv[†] O. Edelstein[*] N. Francez[†] M. Rodeh[*‡]

Abstract

Circular attribute grammars appear in many data flow analysis problems. As one way of making the notion useful, an automatic translation of circular attribute grammars to equivalent non-circular attribute grammars is presented. It is shown that for circular attribute grammars that arise in many data flow analysis problems, the translation does not increase the asymptotic complexity of the semantic equations. Therefore, the translation may be used in conjunction with any evaluator generator to automate the development of efficient data flow analysis algorithms. As a result, the integration of such algorithms with other parts of a compiler becomes easier.

1 Introduction

Attribute grammars (AGs) were introduced by Knuth [Knu68] as a means for defining the semantics of context-free languages. Non-circular AGs (NCAGs) were shown to be useful for specifying many compilation tasks, and are employed to automate compiler development [ASU85]. Until recently, circular AGs (CAGs) were considered ill formed and meaningless. However, if the semantic equations employ monotonic operators (over some partial order), they define a unique greatest (least) fixed point which may be interpreted as the meaning of the equations [JS86,Far86]. This kind of circularity arises naturally in many data flow analysis (DFA) problems [BJ78,ES86].

[*]IBM Israel Scientific Center, The Technion City, Haifa, 32000 Israel

[†]Department of Computer Science, The Technion, Haifa, 32000 Israel

[‡]Current address: IBM T.J. Watson Research Center, Yorktown Heights, P.O. Box 704 N.Y. 100522

For structured languages, various DFA problems, including all the classical ones (see [ASU85]), have been specified using CAGs [ES86]. CAGs may also handle, at a moderate cost, certain less structured constructs (e.g. the BREAK and CONTINUE statements), and constructs with side effects. In addition, we can automatically transform other uniform formalisms for specifying DFA problems (e.g. [Kil73]) into CAGs.

Evaluator generators [DJL86b] accept an AG as input and generate an ***evaluator***, which accepts a syntax tree and decorates it with correct attribute values. Current evaluator generators are restricted to NCAGs. Non-circularity guarantees that the number of iterations of the generated evaluator is linear in the size of the input tree. Furthermore, these tools generate optimized evaluators based on non-circularity. Therefore, translating a CAG into an equivalent NCAG has both practical applications and theoretical value. As a by-product of the translation process, the uniqueness of the decoration which will be obtained at evaluation time can be checked. Thereby, assuring that for every input tree the result is well defined.

1.1 Results and Related Works

In this paper an algorithm is presented which resolves the circularity in each production individually. As Knuth has noted, AGs can simulate Turing machines by encoding the syntax tree itself as an attribute. Hence every CAG may be trivially converted into an equivalent NCAG. The resulting NCAG first records all the equations as an attribute of the start non-terminal. Then, it computes the greatest (simultaneous) fixed point. However, this AG is not a 'true' AG, since instead of using the power of the AG paradigm it uses the power of the language in which the equations are expressed (the *specification language*). In contrast to the trivial translation, our algorithm generates 'true' AG in the sense that all the computation are local. Furthermore, the local transformation guarantees that for CAGs arising in DFA problems, the complexity of solving the equations in the generated NCAG is the same as that of solving the original equations.

In contrast to iterative evaluation[JS86,Far86], our method may be used with an ordinary AG generator to generate a linear evaluator.

For several DFA problems, our algorithm transforms the natural CAGs describing these problems into NCAGs which are similar to those composed manually [BJ78,Ken81,Far86,ES86]. Therefore, our algorithm also unifies the treatment of DFA problems using AGs, and supports natural specification of DFA problems.

1.1.1 Production Based Translation

The equations associated with a production p may refer to *free* attributes, i.e., attributes with no defining equations in p. Furthermore, the missing defining equations are not determined until the input tree is given. To overcome this problem, we extend the conventional fixed point theory [Tar55] by considering *free fixed points*, i.e., fixed points of systems of equations with free variables. The greatest free fixed point is used to compute the greatest simultaneous fixed point using a divide-and-conquer (DAC) principle.

Every CAG may be converted into an equivalent NCAG by using the free fixed point operator. The generated NCAG defines a DAC computation of the greatest fixed point of the original CAG. The generated NCAG includes new functional attributes which summarize the input-output relation between circular inherited and synthesized attributes. The circularity is avoided by using the functional attributes and free fixed points.

Input-output relations have been introduced by Knuth [Knu68,Knu71] for circularity testing and were also used for static evaluation [KW78,CF82,Far86] and other analysis problems in NCAGs (see [DJL86a] for more references). Mayoh used input-output relations to transform an NCAG into an NCAG without inherited attributes but with functional attributes which summarize the input-output relations [May81].

In contrast to Mayoh, we do not eliminate inherited attributes but only eliminate references to circular synthesized attributes. Therefore, our construction minimizes the number of added attributes, and yields an NCAG which maintains the same attributes as the original CAG and not only the attributes of the start non-terminal.Furthermore, for CAGs the same tree may have more than one fixed point and therefore many values may be assigned to the functional attributes. An immediate consequence of our DAC principle is that the choice of the greatest free fixed points as *the values* of the corresponding functional attributes is the only choice which preserves the greatest fixed point.

1.1.2 Computability of Free Fixed Points

Our construction yields computable equations whenever direct evaluation may be used. The complexity of solving the generated equations may be the same as that of a direct evaluator due to the complexity of fixed point computation.

The closure assumptions of [GW76,Ros80,Tar81b] are used to ensure that for DFA problems, the free fixed point may be expressed in the specification language of the original CAG. Usually, the DFA problems are also *bounded* ([Ros80]), i.e., the number of iterations needed to compute the fixed point of a single function need not depend on the input. In these cases, the complexity of solving the equations in the generated NCAG is the same as that of solving the original equations. This follows from the fact that the number of equations associated with a production does not depend on the tree size. Therefore, in these cases, the generated evaluator is more efficient than the iterative evaluation by an order of magnitude.

Many of the DFA problems may be expressed in terms of simple set expressions. In these cases, the generated NCAG does not include functional attributes but only set attributes.

1.2 Outline of the Paper

In Section 2, we review the fixed point and AG notations. Then, we present our novel schematic translation of CAGs into NCAGs (Section 3). In Section 4, we address the algorithmic aspects of the translation process as they arise in DFA problems. We conclude by describing the applications of our translation.

2 Preliminaries

2.1 Fixed Point Theory

In this Section, we review the needed fixed point notations and results. In particular, Tarski's[Tar55] conditions are somewhat relaxed, and required to hold only for circular arguments.

Let $X = \{x_1, x_2, \ldots, x_n\}$ where with each $x \in X$ is associated a poset D_x. Let

$$D = D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$$

and let $f: D \rightarrow D$. A fixed point of f is sometimes called a *simultaneous* fixed point since the equation $X = f(X)$ can be viewed as a simultaneous system of equations

$$S :: \{x = f_x(X) : x \in X\} \tag{1}$$

To use positional notations, X is (arbitrary) ordered and each of the f_x has the form $f_x(x_1, x_2, \ldots, x_n)$. In the sequel, we use systems and functions interchangeably. In particular, we sometimes use νS instead of νf to denote the greatest fixed point of f.

We are interested in systems S in which f_x need not depend on all its arguments. The *characteristic graph* $G_S = (X, E)$ is a directed graph in which $(x_i, x_j) \in E$ if x_i (semantically) depends on x_j, i.e., $f_{x_i}(d_1, \ldots, d_j, \ldots, d_n) \neq f_{x_i}(d_1, \ldots, d'_j, \ldots, d_n)$ for some $d_1, \ldots, d_j, d'_j, \ldots, d_n$. An argument $x \in X$ *transitively depends* on $y \in X$ in S if there exists a directed path from y to x in G_S. In the sequel, we do not distinguish between transitive and direct dependencies. An argument $x \in X$ is *cyclic* if it depends on itself. It is *transitively cyclic* if it is either cyclic or it depends on a cyclic argument and otherwise it is *acyclic*. The system S is *cyclic* if it contains cyclic arguments, and otherwise it is *acyclic*.

Tarski's theorem guarantees that for every system S in which *all* the functions f_x are monotonic in *all* their arguments, and *all* the domains are complete partial orders(CPOs) νS exists. The theorem also holds when S is only *well defined*, i.e., for every transitively cyclic argument $x \in X$, f_x is monotonic in all its transitively cyclic arguments, and the domains of all the cyclic arguments are CPOs. Moreover, we prove a simple representation theorem which allows to transfer to well defined systems, the results about systems in which all the equations are monotonic and all the domains are CPOs.

To that end, the set X of arguments of a system S is partitioned into the cyclic arguments (denoted by C), the acyclic arguments (denoted by A) and transitively cyclic arguments which are not cyclic (denoted by T). Every well defined system S has an equivalent representation (one preserving all fixed points):

$$S'' :: \left\{ \begin{array}{lcll} A & = & h_A(C) & \equiv \; d(S, A) \\ C & = & h_C(C) & \\ T & = & h_T(C) & \end{array} \right\} \tag{2}$$

This representation can be derived from S by the following phases:

1. Find the unique simultaneous fixed point of the acyclic part of S, $d(S, A)$.

2. Replace the references to the acyclic arguments by their values to get a new system S'.

3. Restructure S' to have the form (2), by eliminating the references to transitively cyclic arguments in S'. Each such reference is replaced by its defining equation.

An element $d \in D$ is a *pre-fixed point* of S, if $d \leq f(d)$ and $d_A = f_A(d)$. Notice that this definition is non-standard by requiring equality for acyclic arguments since their fixed point component is unique. A way to circumvent this non-standard definition is by using equality as the order imposed on the acyclic arguments. In the following theorem, $FP(f)$ and $PRE(f)$ are the set of fixed points and pre-fixed points of f, respectively.

Theorem 2.1 (Representation of well defined systems)
For every well defined system S of the form (1), *there exists a unique $d(S, A) \in D_A$ and a monotonic function $h: D \rightarrow D$ which depends only on its cyclic arguments such that*

$$\forall d \in D: h_A(d) = d(S, A) \tag{3}$$
$$FP(S) \equiv FP(h) \tag{4}$$
$$PRE(S) \subseteq PRE(h) \tag{5}$$

The representation theorem implies that for every well defined system S, νS exists. The existence of νS can be also proved by a direct inductive proof by partitioning G_S into its strongly connected components, and constructing νS using a topological order on the strongly connected components. The direct proof has the advantage that it suggests an algorithm to compute νS when the domains of the cyclic arguments are well founded sets (WFS). However, the representation theorem may be also used to extend Park's Theorem[1] and to prove the monotonicity of the ν operator.

Systems of equations may contain free variables. Let $Y = \{y_1, y_2, \ldots, y_m\}$. The *free system* $S(Y)$ has the form:

$$S(Y) :: \{x = f_x(Y, X) : x \in X\} \tag{6}$$

The functional representation of $S(Y)$ has the form $f: D_Y \times D_X \rightarrow D_X$, or in its Curried form $\hat{f}: D_Y \rightarrow (D_X \rightarrow D_X)$, where $\hat{f}(d_Y)(d_X) = f(d_Y, d_X)$. The notions of characteristic graph, cyclic and transitively cyclic arguments and well defined systems of

[1] One of the variants of Park's theorem is that $d \leq \nu f$ for every pre-fixed point $d \in D$ of a monotonic function $f: D \rightarrow D$ where D is a CPO.

equations are extended to free systems in a natural way. Furthermore, for well defined free systems S, $\nu(S(d_Y))$ (the greatest fixed point of the system of the form (1) obtained by substituting d_Y for Y) exists for every $d_Y \in D_Y$.

To allow transformations on free systems of equations, we define the notion of weak equivalence of two free systems S^1 and S^2 of the form (6). S^1 and S^2 are *weakly equivalent* ($S^1 \cong S^2$) if for every $d_Y \in D_Y$, $\nu S^1(d_Y) = \nu S^2(d_Y)$. For example, the equations $x = x \cap y$ and $x = y$ are weakly equivalent on powerset domains ordered by set inclusion.

2.2 Attribute Grammars

For a review of the standard imperative semantics of AGs see [DJL86a]. Here we exemplify the notations via the AG AEP in Table 1, which describes the formal available expressions problem[ASU85, pp. 627-631], [BJ78]. The intended value of the inherited attribute *St.before* is the set of available expressions before the execution of the statements derived from *St*. Similarly, the value of the synthesized attributes *St.after* and *Prog.after* are the set of available expressions after the execution of *St* and *Prog*, respectively. The synthesized attributes *Exp.s* and *Cond.s* hold the set of expressions which appear in the expression and the condition, respectively. The attribute **id**.*name* contains the name of the variable. The function $NotArg(n, E)$ used in 5.1 returns the expressions in E in which n does not appear as argument. Thus, equation 5.1 first adds the new expression, and then deletes the expressions which become unavailable by the assignment. The other equations are straightforward. This AG is used as a running example in this paper.

Apart from the standard imperative semantic definition of AGs, an AG defines a class of systems of equations ([CM79]). With each production p in the grammar is associated a system $S[p]$ over the attributes $Attr[p]$ of the grammar symbols in p. The set $Attr[p]$ is partitioned into the *defined* attributes $Output[p]$ and the *free* attributes $Input[p]$. Let T be a tree over a given AG and let v be a vertex of T in which a production p has been applied. The system $S(v)$ is an instantiation of $S[p]$ obtained by replacing the attributes of $S[p]$ by the corresponding *attribute occurrences*. $S(T)$ is obtained by pasting together the systems $S(v)$ for all v in T. The set $Attr(T)$, the attribute occurrences of T, is partitioned into $Input(T)$, which are the free attribute occurrences and are initialized during the tree construction, and the defined attribute occurrences $Output(T)$. Figure 1 contains an input program for AEP and a possible tree. Table 2 contains the system of equations of this tree as well as the greatest simultaneous fixed point. Here b, a and n are shorthand for the attribute names *before, after* and *name*, respectively.

An AG is *circular* (CAG) if there exists a tree T for which $S(T)$ is cyclic; otherwise it is *non-circular* (NCAG). Table 2 shows that AEP is circular since $v_{12}.a$ depends on $v_{12}.b$ in equation 13, and $v_{12}.b$ depends on $v_{12}.a$ in equation 11.

A CAG is *well defined* if for every T, the system $S(T)$ is well defined. The example AG AEP is well defined since the finite powerset AE of available expressions ordered by set inclusion is a CPO and the functions used in AEP employ set operations which are monotonic in the (transitively) cyclic attributes. Some care has to be taken in this example in order to cope with the absence of a maximum element.

Since the equations of any AG are syntactically sparse, the characteristic graph $G(T) \overset{\text{def}}{=} G_{S(T)}$ of $S(T)$ may be approximated by considering syntactic rather than semantic dependencies. As a result, $G(T)$ may have more edges than the real characteristic graph of $S(T)$.

AGs may be analyzed in order to find out which of the semantic functions need to be monotonic, and which of their domains need to be a CPO. Unfortunately, the circularity testing problem is exponential ([JOR75]). Still, for all practical purposes, Knuth's algorithm [Knu68] can be used as an approximation to circularity testing. The idea is to precompute the IO relation $IO[Z] \subseteq Inh[Z] \times Syn[Z]$ which is a superset of the set of all the possible dependencies of the synthesized attributes of Z on the inherited attributes of Z. Thus, for every tree T with a root of type Z (where Z is not necessarily the start symbol of the grammar), $IO(T) \subseteq IO[Z]$, where $IO(T)$ denotes the relation obtained by restricting the transitive closure of $G(T)$ to the attributes of Z. An AG is *uniform* if for every non-terminal Z, there exists a tree T with a root of type Z such that $IO[Z] \equiv IO(T)$. Non-uniform AGs are considered ill formed and tend not to appear in practice. In this paper the IO relations are useful for two purposes:

1. To derive an algorithm for finding the cyclic and transitively cyclic attributes of every production p, (denoted by $CAttr[p]$ and $TAttr[p]$), and the circular attributes of a non-terminal Z (denoted by $CAttr[Z]$ and $TAttr[Z]$). The details of the algorithm are not give here.

2. In the next section, we shall use the IO relations to define the translation of a CAG into an equivalent NCAG.

In the rest of this paper we treat only uniform CAGs although this restriction can be relaxed

3 Schematic Resolution

The *free greatest fixed point* νS of a free system $S(Y)$ of the form (6) is a function of the free arguments which yields the greatest fixed point for every input value (thus $(\nu S)(d) \equiv \nu S(d)$). For example, consider a powerset domain ordered by set inclusion. Then, for $S(y_1, y_2) :: x = (x - y_1) \cap y_2$, $\nu S = f(y_1, y_2)$ where $f(y_1, y_2) \overset{\text{def}}{=} y_2 - y_1$.

If the domains of the cyclic attribute occurrences of a tree T over a CAG are WFS then a straightforward iterative procedure may be used to compute the greatest fixed point of $S_{AG}(T)$. Alternatively, the free fixed point may be used to compute $\nu S_{AG}(T)(Input(T))$ in the following two phase procedure:

Bottom-Up For each vertex v find the free fixed point $\nu S(v)$ by scanning T bottom-up.

Top-Down Use the values of the inherited attributes top-down to substitute for the free arguments of each free fixed point to obtain the values of all the attributes.

This procedure has the desired property of being local. However, it is carried out at *evaluation* time when T is given. Instead, we would like to concentrate on *generation* time and to construct an NCAG, $\widehat{AG}$, for a given CAG AG, which induces the above DAC process for computing the greatest fixed point of $S_{AG}(T)$ for every T. The AG $\widehat{AG}$ would include the original set of attribute occurrences and new synthesized functional attribute occurrences to represent $\nu S(v)$. Since we deal with generation time, $S[p]$ will be considered.

By solving the systems $S[p]$ once and for all, the complexity of the generated evaluator is reduced. However, the cost of maintaining the functional attributes may be considerable. Therefore, we only add enough functional attributes to break all the cycles. As an additional benefit, this allows easy integration with non-circular components of a given CAG.

3.1 Production Based Translation

Consider a production $p: Z \rightarrow Z_1 Z_2 \cdots Z_n$. To get rid of cycles, it is sufficient to eliminate the following two sources of circularity in p:

1. The system $S[p]$ is cyclic by itself (**direct circularity**)
2. The system $S[p]$ contains a reference to a cyclic synthesized attribute $Z_j.cs$ (**transitive circularity**).

In AEP, there is no direct circularity since for all p, $S[p]$ is acyclic. However, all the productions other than p_5 are transitively circular. For example, p_4 is transitively circular since the attribute $St_1.after$ is used in $S[p_4]$ for defining $St_1.before$.

Our approach to first eliminate the transitive circularity in p, thereby introducing direct circularity and then to eliminate direct circularity altogether.

3.1.1 Elimination of Transitive Circularity

Transitive circularity is a chicken and egg problem where cyclic synthesized attributes are used in $S[p]$ for defining cyclic inherited attributes, and cyclic inherited attributes are used in productions applied below p for defining cyclic synthesized attributes.

The transitive circularity may be eliminated by adding a functional synthesized attribute $Z_j.fcs$ which holds the input-output relation from inherited attributes to a circular synthesized attribute $Z_j.cs$ and using the expression $Z_j.fcs(Inh[Z_j])$ instead of $Z_j.cs$.

Table 3 contains the system obtained from $S[p_4]$ where the cyclic attribute $St_1.before$ was replaced by $St_1.fafter(St_1.before)$. Thus, $St.fafter$ is a new functional synthesized attribute of type $AE \rightarrow AE$ (remember that AE is the powerset of available expressions). Of course, this replacement is valid only if all the semantic equations associated with productions in which Z_j is derived maintain the invariant $Z_j.cs = Z_j.fcs(Inh[Z_j])$. In AEP, each of the productions p_2, p_3, p_4 and p_5 need to be associated with an equation defining $St.fafter$ such that $St.after = St.fafter(St.before)$. We defer the definition of the semantic functions to maintain these invariants to Section 3.1.3.

In general, the new functional attribute $Z_j.fcs$ is a function from $D_{Inh[Z_j]}$ to $D_{Z_j.cs}$. However, the number of inherited attributes may be large and therefore $Z_j.fcs$ may have many arguments. Thus, we take only those arguments which are cyclic in p and used in some of the trees below p to define $Z_j.cs$.

Let $CSyn[Z]$ ($CInh[Z]$) be the circular synthesized (inherited) attributes of a non-terminal Z. Formally, for every non-terminal Y in AG, and an attribute $cs \in CSyn[Y]$, $\widehat{AG}$ includes a synthesized functional attribute fcs associated with Y. For a circular synthesized attribute $cs \in CSyn[Y]$, let $Inp[Y, cs] \overset{\text{def}}{=} \{Y.ci : (ci, cs) \in IO[Y], ci \in CInh[Y]\}$. Each cyclic synthesized attribute $Z_j.cs \in CAttr[p]$ is replaced by

$Z_j.fcs(Inp[Z_j, cs])$. Let $MS[p]$ be the modified system and let $f[p]$ be the corresponding function. Let $Input^1[p]$ be the free attributes in $MS[p]$. The Curried representation of $MS[p]$ is

$$MS[p] :: \begin{array}{r} \{x = f[p]_x(Input^1[p])(Output[p]) : \\ x \in Output[p]\} \end{array} \quad (7)$$

In AEP, $Input^1[p_4] = \{Cond.s, St_1.fafter, St.before\}$. Table 4 contains the system $MS[p_4]$ where the semantic functions are explicitly written using Curried notations.

3.1.2 Elimination of Direct Circularity

Direct circularity may be avoided by finding an acyclic system $AS[p]$ such that $AS[p] \cong MS[p]$. This can be done by 'symbolically' solving all the cycles in $MS[p]$, and finding an expression over the free attributes of $MS[p]$ which computes the free greatest fixed point of $MS[p]$.

Applying the free fixed point operator to $f[p]$ of (7) we get:

$$AS[p] :: \begin{array}{r} \{x = (\nu f[p])_x(Input^1[p]) : \\ x \in Output[p]\} \end{array} \quad (8)$$

In Table 5, we applied this transformation to $MS[p_4]$ to obtain $AS[p_4]$. By definition $AS[p]$ is acyclic and has the same greatest fixed point as $MS[p](d)$ for every input $d \in D_{Input^1[p]}$.

If the original AG is well defined, then all the equations in $S[p]$ are monotonic in $Z_i.cs$. Therefore, if the semantic equations set $Z_i.fcs$ to a monotonic function then $\nu f[p]$ exists. For example, equation (4.1) in the original AG AEP is monotonic in $St_1.after$. Therefore, if the value of the functional attribute $St.fafter$ is always a monotonic function, then we can guarantee at construction time that $\nu f[p_4]$ exists.

3.1.3 Maintaining the Invariants

To complete the construction described above, we have to introduce equations which establish the invariants $Z.fcs(Inp[Z, cs]) = Z.cs$ for every circular synthesized attribute cs of the left hand side non-terminal Z.

Let $cs \in CSyn[Z]$. Let $Other[p, cs] \overset{\text{def}}{=} Input^1[p] - Inp[Z, cs]$. The function $(\nu f[p])_{Z.cs}$ may be presented in a Curried notation as:

$$(\nu f[p])_{Z.cs}: D_{Other[p,cs]} \rightarrow (D_{Inp[Z,cs]} \rightarrow D_{Z.cs})$$

Now define the system $FS[p]$ as follows:

$$FS[p] :: \begin{array}{r} \{Z.fcs = (\nu f[p])_{Z.cs}(Other[p, cs]) : \\ cs \in CSyn[Z]\} \end{array} \quad (9)$$

The system $\hat{S}[p]$ associated with p in $\widehat{AG}$ is $AS[p] \cup FS[p]$. Table 6 contains $\hat{S}[p_4]$.

3.2 Correctness of the Translation

The non-circularity of $\widehat{AG}$ stems from the following:

1. An AG which does not contain transitive and direct circularity is non-circular.

2. For every production $p: Z_0 \rightarrow Z_1 Z_2 \cdots Z_n$, each of the functional attributes of Z_0 may depend only on functional attributes of Z_i for $1 \leq i \leq n$ and on non-cyclic attributes of Z_i. This follows from the elimination of references to cyclic synthesized attributes as well as the elimination of cyclic inherited attributes, by using them as arguments of the functional attributes.

To guarantee that the new functional attributes are non-circular, we eliminate *all* the references to cyclic synthesized attributes even though not all the eliminations expose direct circularity. In fact, new direct circularity is created in $MS[p]$ only when a cyclic inherited attribute depends on a cyclic synthesized attribute in $S[p]$.

In AEP, the only new direct circularity is in $MS[p_4]$ since $St_1.before$ depends on $St_1.after$, in $S[p_4]$. Still, to guarantee that $St.fafter$ is non-circular we need to eliminate $St.after$ from $S[p_2]$ and $S[p_3]$.

A tree T is *complete* if its leaves contain only terminals.

Theorem 3.1 (Equivalence Theorem) *Let AG be a well defined CAG and $\widehat{AG}$ be the resulting NCAG. Then, for every complete tree T, $S_{AG}(T) \cong S_{\widehat{AG}}(T)$.*

The system $S_{\widehat{AG}}(T)$ may be obtained from the system $S_{AG}(T)$ by a sequence of transformations each of which preserves the greatest fixed point and the well definedness of the system. Therefore, the unique fixed point of (the acyclic) $S_{\widehat{AG}}(T)$ is the greatest fixed point of $S_{AG}(T)$. Due to space limitations we cannot rigorously prove it here. The proof is based on the following mathematical properties:

1. The weak equivalence relation is closed under systems union, i.e., if $S^1{}_{X_1} \cong S^2{}_{X_1}$ and $S^1{}_{X_2} \cong S^2{}_{X_2}$ for an arbitrary partition of the defined arguments X of S^1 and S^2 into X_1 and X_2, then $S^1 \cong S^2$. This allows us to show that $S_{AG}(T) \cong S_{\widehat{AG}}(T)$ by structural induction on T.

2. Let $S(Y)$ be a well defined system of the form (6). Then νS can be found by first finding νS_1 for an arbitrary sub-system S_1 of S and then finding $\nu S'$ where S' is the system obtained from S by replacing the arguments of S_1 by νS_1. The

above may be proven using Park's Theorem. In particular, for a complete tree T rooted by r, $\nu S_{AG}(T)$ may be found by first finding the free fixed point of cyclic arguments of the children of r, and then finding the solution of the resulting system as done in $S_{\widehat{AG}}(T)$.

4 Application to Data Flow

We now apply the translation scheme of Section 3 to CAGs arising in DFA. In these problems each of the semantic functions defining circular attributes is an expression in some specification language. If in addition we make the standard DFA assumptions regarding the expressibility of the specification language, then each of the $AS[p]$ systems can be also written in the same specification language.

4.1 A Data Flow Framework

Let the domain of all the transitively circular attributes D be a complete semi-lattice, $\wedge$ the meet operator over D and let F be set of functions on D. Then, CAG is a ***circular data flow analysis AG*** with respect to D and F (CDFAG) if each of the transitively circular attributes a is defined by an equation of the form:

$$a = \wedge_{a' \in I_a} R_{a'}(a') \tag{10}$$

where each $R_{a'}$ is an expression over the acyclic attributes which yields a function in F, and I_a is the set of transitively cyclic arguments on which a depends. The set F is the 'valid' functions and (10) defines the specification language.

A special case of the above parametric specification language is when $D = 2^P$ for some finite set P and $\leq$ is set inclusion. In this case, $\wedge$ is $\cap$ and $\top = P$. A function $f: D \rightarrow D$ is a ***uniform modification function*** (UMF) with respect to P if there exists $K, G \subseteq P$ such that for every $d \subseteq P$, $f(d) = (d - K) \cup G$. The set $F(P)$ is the set of UMFs with respect to P. UMFs are used to describe most of the classical DFA problems[GW76,RP86]. We use K and G as a generalized shorthand for *kill* and *gen* which are frequently used in the literature to describe classical problems such as available expressions, live variables and reaching definitions[2][ASU85]. For UMFs, (10) has the form:

$$a = \cap_{a' \in I_a}((a' - K_{a'}) \cup G_{a'}) \tag{11}$$

where $K_{a'}$ and $G_{a'}$ are functions of acyclic attributes which yields subsets of P. By convention, CDFAG which uses UMFs is a ***uniform modification circular attribute grammar***(UMCAG).

[2]The last two problems need least and not greatest fixed point.

For example, Table 7 presents a variant of $S[p_4]$ in which both equations have the form (11). All the Equations of AEP but (5.1) can be easily put in the form (11). To use UMFs in (5.1), we modify it into:

$$(St.b - Arg(\text{id}.n, \top)) \cup NotArg(\text{id}.n, Exp.s)$$

where $\top$ denotes the universal set (all the expressions in program) and $Arg(n, \top)$ is the set of expressions in the program in which n is an argument. Notice that $\top$ can be replaced by an inherited attribute $St.universe$, and by writing new semantic equations to define this attribute in a left-to-right manner. In general, the requirement of having P finite can be relaxed by requiring that only the G sets are finite. Thus, for an input tree T, we can set $P(T) \overset{\text{def}}{=} \cup_{i=1}^{n} G_i$ where G_i are the G sets which appear in the equations. Notice that the sets G are known after computing non-circular attributes.

To guarantee that $\widehat{AG}$ uses the same specification language, we require that the new equations also have this form over the same F. To this end, we say that an AG is ***closed*** with respect to F if for every production $p: Z_0 \rightarrow Z_1 Z_2 \cdots Z_n$ and a transitively circular attribute $Z_i.a \in Output[p]$, the defining equation of $Z_i.a$ in $AS[p]$ may be written as an expression of the form (10) over F where $a' \in CInh[Z_0]$. As we shall see, every UMCAG is closed with respect to UMFs. Therefore, the schematic translation of Section 3 will produce an NCAG where the modified equations use only sets expressions.

4.2 Distributive Frameworks

We now impose more restrictions on F to guarantee that the CDFAG is closed. Furthermore, the restrictions allow us to construct functional expressions for computing the greatest free fixed point, by an efficient algorithm. In particular, it will enable the automatic conversion of UMCAG into an equivalent NCAG since it will satisfy the restrictions.

A function $f: D \rightarrow D$ is ***distributive*** (and therefore monotonic) if for every $d_1, d_2, \ldots \in D$, $f(\wedge d_i) = \wedge f(d_i)$. For example, UMFs are distributive since for every $d_1, d_2, \ldots$, $((\cap_i d_i) - K) \cup G \equiv \cap_i((d_i - K) \cup G)$.

A set of distributive functions F on D is a ***distributive data flow analysis framework*** (DFAF) if:

Identity There exists an identity function $\iota \in F$ such that for every $d \in D$: $\iota(d) = d$.

Top There exists a function $\top \in F$ such that for every $d \in D$: $\top(d) = \top$.

M-Closure F is closed under meet operation, i.e.,

$$\forall f_1, f_2 \in F, \exists (f_1 \bigwedge f_2) \in F, \forall d \in D: \\ (f_1 \bigwedge f_2)(d) = f_1(d) \wedge f_2(d)$$

C-Closure F is closed under composition, i.e.,

$$\forall f_1, f_2 \in F, \exists (f_1 \, o \, f_2) \in F, \forall d \in D: \\ (f_1 \, o \, f_2)(d) = f_2(f_1(d))$$

S-Closure F is closed under the star operation, i.e.,

$$\forall f \in F, \exists f^* \in F, \forall d \in D: \\ f^*(d) = d \wedge f(d) \wedge f^2(d) \wedge \cdots$$

The requirement of having a top function is an extra requirement with respect to [Ros80,Tar81b], and may be replaced by some connectivity requirements on the characteristic graph of each of the systems.

The set $F(P)$ of UMFs is closed where $\iota(d) \stackrel{\text{def}}{=} (d - \phi) \cup \phi$ and $\top(d) \stackrel{\text{def}}{=} (d - \phi) \cup P$. For the closure rules let $f_i(d) \stackrel{\text{def}}{=} (d - K_i) \cup G_i$ for $i = 1, 2$. Then:

M-Closure $(f_1 \bigwedge f_2)(d) \equiv (d - (K_1 \cup K_2)) \cup (G_2 \cap G_2)$.

C-Closure $(f_1 \, o \, f_2)(d) \equiv (d - (K_2 \cup (K_1 - G_2))) \cup (G_2 \cup (G_1 - K_2))$.

S-Closure Every UMF f is idempotent, i.e., $f^k \equiv f$, for $k \geq 1$. in particular, $f_1{}^*(d) \equiv d \cap (d - K_1) \cup G_1 \equiv (d - (K_1 - G_1)) \cup \phi$.

Theorem 4.1 *Let AG be a CDFAG with respect to D and a DFAF F. Then, AG is closed with respect to F.*

The proof of Theorem 4.1 (not given here) is constructive. The idea is to maintain stronger invariants by guaranteeing that the functional attributes also have the form (10). For example, in AEP we maintain the invariant that

$$St.fa(St.b) = (St.b - St.k) \cup St.g \qquad (12)$$

where k and g are new synthesized attributes. Therefore, in this case, these set attributes may be used instead of the functional attribute $St.fa$.

Using the stronger invariants, the construction is similar to the construction of the functional expressions which compute the *meet over all paths*[Tar81b][3]. For example, consider the system of Table 7. After eliminating the cyclic reference to $St_1.a$ in (4.1), we get:

$$St_1.b = ((St_1.fa(St_1.b) - \phi) \cup Cond.s) \cap ((St.b - \phi) \cup \phi)$$

[3]This is not exactly true since we allow many sources. Thus, we need to use the $\top$ function to reduce our system into Tarjan's.

By using the invariant (12), we get

$$St_1.b = ((((St_1.b - St_1.k) \cup St_1.g) - \phi) \\ \cup Cond.s) \cap ((St.b - \phi) \cup \phi)$$

Now using the closure of UMFs under composition $MS[p_4]$ yields:

$$MS[p_4] :: \left\{ \begin{array}{ll} St_1.b = f_1(St_1.b) \wedge f_2(St.b) & (4.1) \\ St.a = f_3(St_1.b) & (4.2) \end{array} \right\}$$

where:

$$f_1(d) \stackrel{\text{def}}{=} (d - \overbrace{St_1.k}^{K_1}) \cup \overbrace{St_1.g \cup Cond.s}^{G_1} \qquad (13)$$

$$f_2(d) \stackrel{\text{def}}{=} (d - \overbrace{\phi}^{K_2}) \cup \overbrace{Cond.s}^{G_2} \qquad (14)$$

$$f_3(d) \stackrel{\text{def}}{=} (d - \overbrace{\phi}^{K_3}) \cup \overbrace{\phi}^{G_3} \qquad (15)$$

The characteristic graph of this system is shown in Figure 2 where the edges are marked with the function names.

Tarjan's method may be used to construct $\nu MS[p_4]$ in two conceptual phases:

1. Find regular path expressions $\alpha(St.b, St.a)$ and $\alpha(St.b, St_1.b)$ over the edges which summarize the set of paths from $St.b$ to $St.a$ and $St_1.b$ respectively. In the graph of Figure 2 we will get:

$$\begin{array}{rcl} \alpha(St.b, St.a) & = & (St.b, St_1.b).(St_1.b, St_1.b)^*. \\ & & (St_1.b, St.a) \\ \alpha(St.b, St_1.b) & = & (St.b, St_1.b).(St_1.b, St_1.b)^* \end{array}$$

2. Convert $\alpha(St.b, St.a)$ and $\alpha(St.b, St_1.b)$ into functions in F which compute $(\nu MS[p_4])_{St.a}$ and $(\nu MS[p_4])_{St_1.b}$, respectively. This is done by induction of the structure of the regular path expression and using the closure operations on F. The expressions in the system $MS[p_4]$ are:

$$\begin{array}{rcl} St_1.b & = & f_2 \, o \, f_1{}^*(St.b) \\ & \stackrel{\text{def}}{=} & (St.b - (K_1 - G_1) \cup (K_2 - \phi)) \cup \\ & & \phi \cup (G_2 - (K_1 - G_1)) \\ & \equiv & (St.b - (K_1 - G_1) \cup K_2) \cup \\ & & (G_2 - (K_1 - G_1)) \\ & \stackrel{\text{def}}{=} & (St.b - (St_1.k - (St_1.g \cup Cond.s)) \cup \phi) \cup \\ & & (Cond.s - (St_1.k - (St_1.g \cup Cond.s))) \\ & \equiv & (St.b - St_1.k - (St_1.g \cup Cond.s)) \cup \\ & & Cond.s \\ & \equiv & (St.b - (St_1.k - St_1.g)) \cup Cond.s \\ St.a & = & f_2 \, o \, f_1{}^* \, o \, f_3(St.b) \\ & \equiv & (St.b - (St_1.k - St_1.g)) \cup Cond.s \end{array}$$

Table 8 contains the constructed NCAG obtained by finding $\nu MS[p]$ for all the productions p in the modified AEP, using the above two phases algorithm. To make this algorithm efficient, the regular path expressions (and the functional expressions) need to be maintained in a single labeled direct acyclic graph.

For general graphs, the complexity of computing the path expressions is $O(n^3)$ where n is the number of vertices[4]. Therefore, the complexity of constructing the expressions for CDFAG, is $O(pl^3)$ where p is the number of productions and l is the maximal number of circular attribute appearances in a production.

Some of the DFA problems are not distributive. Furthermore, the star operation may be too expensive to compute. In these cases, a functional expression can still be computed [GW76,Ros80,Tar81b] but the expression may yield a value which is between the meet over all paths and νS. Thus, the resulting NCAG is not weakly equivalent but yields safe results which may be greater than νS.

Since we resolve circularity at generation time, the major concern is the complexity of the generated semantic functions. To estimate the complexity of the star operation (which is the main problem), the notion of k-boundedness is defined: F is ***k*-bounded** if for every $f \in F$, $f^{k+1} \geq \iota \bigwedge f \bigwedge \cdots \bigwedge f^k$. For example, UMFs are 1-bounded. For a k-bounded function f, $f^* \equiv (\iota \bigwedge f)^k$ may be computed by $O(\log k)$ composition operations. As a result, the generated semantic equations have the same asymptotic complexity as the original ones.

5 Conclusions

The resolution algorithm presented here may be used to convert a CAG into a linear evaluator in the following steps:

1. The IO relations and the set of (transitively) cyclic attributes are found. This information is used to guarantee that the CAG is a data flow analysis CAG and therefore well defined.

2. The algorithm of Section 3 is applied to yield an NCAG which includes the free fixed point operator.

3. The free fixed point operator is replaced by an expression over the functions used in the original expressions and the closure operations. If the functions are bounded then the resulting expressions have the same asymptotic complexity as the original ones.

4. An ordinary evaluator generator translates the NCAG into a program.

High-level data flow analysis algorithms[Ros77] are also linear. The fact that we use AG formalism enables us to solve data flow analysis problems without interpreting the meaning of the control structures or the non circular components. As a result, our method provides a tool for developing data flow analysis algorithms.

Historically, this work began by developing an optimizer for a Pascal-like language using GAG[KHZ82]. For each data flow analysis problem we wrote[ES86] a CAG and manually translated it into an equivalent NCAG so that GAG could convert the NCAG into a Pascal program. The resolution algorithm proposed here automates the translation for most of the data flow analysis problems considered since most of them could be easily specified using uniform modification CAGs. One exception is the constant propagation problem in which symbol tables are needed. In this case, we can still resolve the circularity but the resulting NCAG yields less informative results. For example, the NCAG in [Wil81] is a natural approximation for a CAG to the constant propagation problem.

This work also emphasizes the the trade-off between the class of AGs and the power of semantic functions. For uniform modification CAGs, the expressive power of NCAGs and CAGs are essentially the same. This is also true for monotonic computable functions on well founded sets. However, in problems such as the constant propagation or when using GOTOs, symbol-tables need to be used. Therefore, the resulting NCAG includes an explicit free fixed point operator.

A direct efficient algorithm may be used instead of translating CAGs into NCAGs. In particular, methods of finite differencing[CP87] may be used to avoid the cumulative cost of repeatedly computing the functions. Our initial study shows that this technique provides an alternative linear computation for a subclass of CAGs including UMCAGs.

Acknowledgments

We would like to thank Peter Lipps, Robert Paige, Ken Perry, Ron Pinter, Reinhard Wilhelm and Shaula Yemini for their help during our research and writing up this paper. The work of Shmuel Sagiv was Partially supported by SFB 124-VLSI design and parallelism of the Deutsche Forschungsgemeinschaft. The work of Nissim Francez was partially supported by the foundation for research in Electronics, Computers and Communication (administered by Israeli academy of sciences and humanities), and by the fund for the promotion of research in the Technion.

[4]If the graph is reducible then a more efficient algorithm due to Tarjan [Tar81a] may be used.

References

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[BJ78] W.A. Babich and M. Jazayeri. The method of attributes for data flow analysis, part i: exhaustive analysis, part ii: demand analysis. *Acta Informatica*, 10:245–272, 1978.

[CF82] B. Couracelle and P. Franchi-Zanettachi. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

[CM79] L. Chirica and D. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.

[CP87] J. Cai and R. Paige. Binding performance at language design time. In *ACM Symposium on Principles of Programming Languages*, pages 85–97, 1987.

[DJL86a] P. Deransart, M. Jourdan, and B. Lhorho. *A Survey on Attribute Grammars Part I: Main Results on Attribute Grammars.* Research Report 485, I.N.R.I.A, 1986.

[DJL86b] P. Deransart, M. Jourdan, and B. Lhorho. *A Survey on Attribute Grammars Part II: Review on Existing Systems.* Research Report 501, I.N.R.I.A, 1986.

[ES86] O. Edelstein and S. Sagiv. *Machine Independent Optimizations via Attribute Grammars.* Technical Report TR88.187, IBM Israel Scientific Center, 1986.

[Far86] R.W. Farrow. Automatic generation of fixed-point-finding evaluators for circular but well-defined attribute grammars. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, 1986.

[GW76] S.L. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of ACM*, 23(1):172–202, 1976.

[JOR75] M. Jazayeri, W.F. Ogden, and W.C. Rounds. The intrinsic exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18(12):696–706, 1975.

[JS86] L.G. Jones and J. Simon. Hierarchical VLSI design systems based on attribute grammars. In *ACM Symposium on Principles of Programming Languages*, pages 58–71, 1986.

[Ken81] K. Kennedy. A survey of data flow analysis techniques. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54, Prentice-Hall, 1981.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmerman. *GAG: a Practical Compiler Generator.* Volume 141 of *Lecture Notes in Computer Science*, Springer Verlag, 1982.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[Knu68] D. E. Knuth. Semantics of context free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] D. E. Knuth. Semantics of context free languages. *Mathematical Systems Theory*, 5(1):95–96, 1971. Errata.

[KW78] K. Kennedy and S.K. Warren. Automatic generation off efficient evaluators for attribute grammars. In *ACM Symposium on Principles of Programming Languages*, pages 32–49, 1978.

[May81] B. H. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3):503–518, 1981.

[Ros77] B. K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20:712–724, 1977.

[Ros80] B. K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal on Computing*, 9(1):159–196, 1980.

[RP86] B.G Ryder and M.C. Paul. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, 1986.

[Tar55] A. Tarski. A lattice-theoretical fixedpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tar81a] R.E. Tarjan. Fast algorithms for solving path problems. *Journal of ACM*, 28(3):594–614, 1981.

[Tar81b] R.E. Tarjan. A unified approach to path problems. *Journal of ACM*, 28(3):577–593, 1981.

[Wil81] R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3, pages 141–159, Prentice-Hall, 1981.

p_1:	$Prog \to St$	
	$St.before = \phi$	(1.1)
	$Prog.after = St.after$	(1.2)
p_2:	$St \to St_1 \,;\, St_2$	
	$St_1.before = St.before$	(2.1)
	$St_2.before = St_1.after$	(2.2)
	$St.after = St_2.after$	(2.3)
p_3:	$St \to \textbf{if}\ Cond\ \textbf{then}\ St_1\ \textbf{else}\ St_2$	
	$St_1.before = Cond.s \cup St.before$	(3.1)
	$St_2.before = Cond.s \cup St.before$	(3.2)
	$St.after = St_1.after \cap St_2.after$	(3.3)
p_4:	$St \to \textbf{while}\ Cond\ \textbf{do}\ St_1$	
	$St_1.before = Cond.s \cup (St_1.after \cap St.before)$	(4.1)
	$St.after = St_1.before$	(4.2)
p_5:	$St \to \textbf{id} := Exp$	
	$St.after = NotArg(\textbf{id}.name, St.before \cup Exp.s)$	(5.1)

Table 1: An AG (AEP) for the available expressions problem.

```
x := y + z ;
q := x + t ;
while  x < 8  do
   x := x + 1;
```

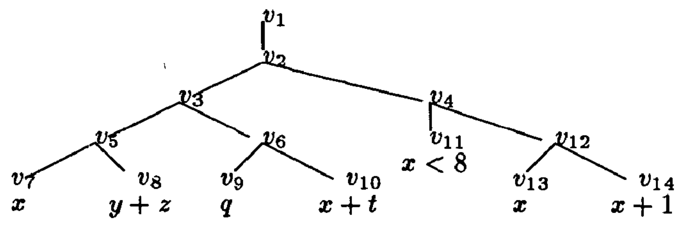

Figure 1: An input program to the available expressions problem and its tree.

	vertex	equation	actual – equation	fixed – point
1	v_1	1.1	$v_2.b = \phi$	ϕ
2		1.2	$v_1.a = v_2.a$	$\{(y+z),(x<8)\}$
3	v_2	2.1	$v_3.b = v_2.b$	ϕ
4		2.2	$v_4.b = v_3.a$	$\{(y+z),(x+t)\}$
5		2.3	$v_2.a = v_4.a$	$\{(y+z),(x<8)\}$
6	v_3	2.1	$v_5.b = v_3.b$	ϕ
7		2.2	$v_6.b = v_5.a$	$\{(y+z)\}$
8		2.3	$v_3.a = v_6.a$	$\{(y+z),(x+t)\}$
9	v_5	5.1	$v_5.a = NotArg(v_7.n, v_5.b \cup v_8.s)$	
			$= NotArg(x, v_5.b \cup \{(y+z)\})$	$\{(y+z)\}$
10	v_6	5.1	$v_6.a = NotArg(v_9.n, v_6.b \cup v_{10}.s)$	
			$= NotArg(q, v_5.b \cup \{(x+t)\})$	$\{(y+z),(x+t)\}$
11	v_4	4.1	$v_{12}.b = v_{11}.s \cup (v_{12}.a \cap v_4.b)$	
			$= \{(x<8)\} \cup (v_{12}.a \cap v_4.b)$	$\{(y+z),(x<8)\}$
12		4.2	$v_4.a = v_{12}.b$	$\{(y+z),(x<8)\}$
13	v_{12}	5.1	$v_{12}.a = NotArg(v_13).n, v_{12}.b \cup v_{14}.s)$	
			$= NotArg(x, v_5.b \cup \{(x+1)\})$	$\{(y+z)\}$

Table 2: The system $S(T)$ for the tree and its greatest fixed point.

$$p_4{:}\quad St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$$
$$\left\{\begin{array}{ll} St_1.before = Cond.s \cup (St_1.fafter(St_1.before) \cap St.before) & (4.1) \\ St.after = St_1.before & (4.2) \end{array}\right\}$$

Table 3: The system of equations obtained by eliminating the references to cyclic synthesized attributes in $S[p_4]$.

$$p_4{:}\quad St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$$
$$\left\{\begin{array}{l} St_1.before = f[p_4]_{St_1.before}(Cond.s, St_1.fafter, St.before)(St.after, St_1.before) \\ St.after = f[p_4]_{St.after}(Cond.s, St_1.fafter, St.before)(St.after, St_1.before) \end{array}\right\}$$

Table 4: The Curried representation of system of equations $MS[p_4]$ obtained by eliminating the references to cyclic synthesized attributes.

$$p_4{:}\quad St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$$
$$\left\{\begin{array}{l} St_1.before = (\nu f[p_4])_{St_1.before}(Cond.s, St_1.fafter, St.before) \\ St.after = (\nu f[p_4])_{St.after}(Cond.s, St_1.fafter, St.before) \end{array}\right\}$$

Table 5: The canonical setting to $AS[p_4]$.

$$p_4{:}\quad St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$$
$$AS[p_4] :: \left\{\begin{array}{l} St_1.before = (\nu f[p_4])_{St_1.before}(Cond.s, St_1.fafter, St.before) \\ St.after = (\nu f[p_4])_{St.after}(Cond.s, St_1.fafter, St.before) \end{array}\right\}$$
$$FS[p_4] :: \{\ St_1.fafter = (\nu f[p_4])_{St_1.after}(Cond.s, St_1.fafter)\ \}$$

Table 6: The system $\hat{S}[p_4]$.

p_4:	$St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$ $\left\{\begin{array}{ll} St_1.before = ((St_1.a - \phi) \cup Cond.s) \cap ((St.b - \phi) \cup Cond.s) & (4.1) \\ St.after = (St_1.b - \phi) \cup \phi & (4.2) \end{array}\right\}$

Table 7: A rewriting of $S[p_4]$ using UMFs.

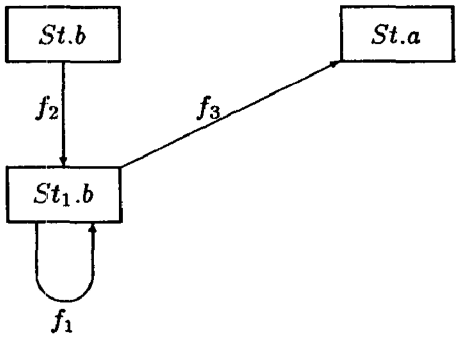

Figure 2: The characteristic graph of the transitively cyclic arguments of $MS[p_4]$.

p_1:	$Prog \rightarrow St$ $AS[p_1] :: \left\{\begin{array}{l} St.b = \phi \\ Prog.a = St.g \end{array}\right\}$
p_2:	$St \rightarrow St_1\ ;\ St_2$ $AS[p_2] :: \left\{\begin{array}{l} St_1.b = St.b \\ St_2.b = (St.b - St_1.k) \cup St_1.g \\ St.a = (St.b - (St_2.k \cup (St_1.k - St_2.g))) \cup (St_2.g \cup (St_1.g - St_2.k)) \end{array}\right\}$ $FS[p_2] :: \left\{\begin{array}{l} St.g = St_2.g \cup (St_1.g - St_2.k) \\ St.k = St_2.k \cup (St_1.k - St_2.g) \end{array}\right\}$
p_3:	$St \rightarrow \textbf{if}\ Cond\ \textbf{then}\ St_1\ \textbf{else}\ St_2$ $AS[p_3] :: \left\{\begin{array}{l} St_1.b = Cond.s \cup St.b \\ St_2.b = Cond.s \cup St.b \\ St.a = (St.b - (St_1.k \cup St_2.k)) \cup (St_1.g \cap St_2.g) \end{array}\right\}$ $FS[p_3] :: \left\{\begin{array}{l} St.g = St_1.g \cap St_2.g \\ St.k = St_1.k \cup St_2.k \end{array}\right\}$
p_4:	$St \rightarrow \textbf{while}\ Cond\ \textbf{do}\ St_1$ $AS[p_4] :: \left\{\begin{array}{l} St_1.b = (St.b - (St_1.k - St.g)) \cup Cond.s \\ St.a = (St.b - (St_1.k - St.g)) \cup Cond.s \end{array}\right\}$ $FS[p_4] :: \left\{\begin{array}{l} St.g = Cond.s \\ St.k = St_1.k - St_1.g \end{array}\right\}$
p_5:	$St \rightarrow \textbf{id} := Exp$ $AS[p_5] :: \{\ St.a = (St.b - Arg(\textbf{id}.name, \top)) \cup NotArg(\textbf{id}.name, Exp.s)\ \}$ $FS[p_5] :: \left\{\begin{array}{l} St.g = NotArg(\textbf{id}.name, Exp.s) \\ St.k = Arg(\textbf{id}.name, \top) \end{array}\right\}$

Table 8: The constructed NCAG for the AG of the available expressions problem.