GENERAL FLOW PROBLEMS AND GRAPH GRAMMARS

Lishing Liu

TR 77-308

Computer Science Department
Cornell University
Ithaca, N.Y. 14853

1. Introduction

A standard approach to compile-time program optimization is to analyze the controle flow graph of a program. In the early 70's the interval approach [A, C, AU1] was introduced to solve flow problems. It has been recognized that "reducibility" is a desired property on flow graphs when flow problems are solved. Hecht and Ullman [HU1] give two simple graph grammar rules T_1 and T_2 for reducible graphs. Ullman [HU, U] shows that the common subexpression elimination problem on reducible graphs can be solved in O(eloge) bit-vector steps, where e is the number of edges in flow graphs.

Another approach is to propagate information through flow graphs until the propagating process converges. General formulation of data flow problems using lattice theoretic framework is also introduced. Representatives of such work can be found in [AU2, FKZ, HU2, KU1, Ke, Ki, U].

Recently Graham and Wegman [GW] present a new algorithm for global flow analysis which combines a modification of interval approach with a modification of the transformations T_1 and T_2 introduced by Hecht and Ullman. The idea is to decompose the graph reduction rules into simpler forms so that a flow analysis problem can easily be reduced into a smaller-sized problem when the flow graph is reduced. It has been shown that a forward information propagation problem on a reducible graph can be solved in O(eloge) functional steps.

Farrow, Kennedy and Zucconi [FKZ] define a class $\mathcal{F}_{SSFG}$

of semi-structured flow graphs. The graph grammar for $\mathscr{F}_{SSFG}$ is an extension of the B-J grammar (named for Böhm and Jacopini). The graph grammar rules are rather simple so that many flow problems on $\mathscr{F}_{SSFG}$ can be solved in O(e) functional steps.

In this paper we extend the idea of "acceptable assignment" in [GW] and formulate general data flow analysis problems in terms of lattice theoretic framework. In general data flow analysis problems can be classified into two major classes, the forward information propagation problems and the reverse information propagation problems. Modifying slightly the three transformations T_1', T_2' and T_3' introduced in [GW] we show that these data flow problems on reducible flow graphs with "fast" function spaces can be solved in O(eloge) functional steps. The generalization of the method in [GW] to general forward information propagation problems is straightforward. For the backward information propagation problems, we don't treat them as forward problems on the reversed graphs and no reverse graph reduction rules are needed. We also define new graph grammars SG(k). The class $\mathscr{F}_{SG(k)}$ of flow graphs generated by SG(k) is wider than $\mathscr{F}_{SSFG}$. Gneral data flow problems with "fast" function spaces can be solved using O(e) functional steps on a flow graph in $\mathscr{F}_{SG(k)}$.

The paper is organized as follows. In section 2 some basic definitions and results on flow analysis are presented. In section 3 we formulate general flow analysis problems and show how they are solved on reducible flow graphs. In section 4 new graph grammars SG(k) are defined and examined. Section 5 is the summary.

2. Preliminaries

A (directed) graph is a pair (N,E) where N is a finite set of nodes and $E \subseteq N \times N$ is a set of edges. Given a graph G = (N,E) we say that a non-empty sequence of nodes $p_0, p_1, \cdots$ $\cdots, p_k$ is a path from p_0 to p_k of length k provided that for all i between 0 and k-1, (p_i, p_{i+1}) is an edge. Let e=(x,y) be an edge, we say that e enters y and leaves x, x is a predecessor of y and y is a successor of x. Pred(x) will denote the set of predecessors of x, and Succ(x) will denote the set of successors of x. An edge (x,x) is called a self-loop. A path from x to x is called a cycle. A graph with a single node and no edges is called a trivial graph.

2.1 Reducible Graphs

A flow graph $G = (N,E,n_0)$ is a graph (N,E) together with a distinguished entry node n_0 such that for all x in N there is a path from n_0 to x in G.

Reducible graphs were originally defined by Allen and Cocke in terms of "intervals". We will use an equivalent definition using the transformations T_1 and T_2 introduced by Hecht and Ullman [HU1].

Let $G = (N,E,n_0)$ be a given flow graph. Transformations T_1 and T_2 are defined as follows:

T_1 —— If e is a self-loop in G then

$T_1(G,e) = (N, E-\{e\}, n_0)$.

T_2 —— If $e = (u,v) \in E$, $v \neq n_0$ and $Pred(v) = \{u\}$

then $T_2(G,e) = (N', E', n_0)$, where $N' = N - \{v\}$

and $E' = (E \cap (N' \times N')) \cup \{(u,x) \mid (v,x) \in E\}$.

T_1 and T_2 are depicted in Figure 1.

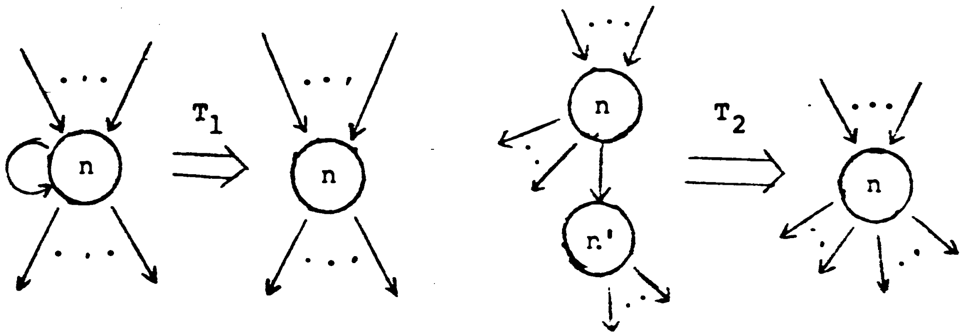


Figure 1

Definition: A flow graph G is reducible if it can be reduced to the trivial graph by successive applications of transformations T_1 and T_2.

2.1 Some Results on Reducible Graph Parsing

Hopcroft and Ullman [HU] have shown that a reducible graph with e edges can be parses using T_1 and T_2 in O(eloge) elementary steps.

Graham and Wegman [GW] decompose T_1 and T_2 into three transformations T_1', T_2' and T_3'. They are defined below and are depicted in Figure 2.

Again let $G = (N,E,n_0)$ be a flow graph.

T_1' —— If $e=(v,v) \varepsilon E$ and there is a unique edge entering v other than e then

$T_1'(G,e) = (N,E-\{e\},n_0)$.

T_2' —— If for some v in $N-\{n_0\}$, $Pred(v) = \{u\}$, $u \neq v$, and if $(v,w) \varepsilon E$, then

$T_2'(G,e) = (N',E',n_0)$, where

(1) if $Succ(v) = \{w\}$ then $N' = N-\{v\}$

and $E' = (E \cup \{(u,w)\}) - \{(u,v),(v,w)\}$,

(2) otherwise $N' = N$ and

$E' = (E \cup \{(u,w)\}) - \{(v,w)\}$.

T_3' —— If all non-looping edges leave n_0, i.e. G is a fan graph, and for some $v \varepsilon N$. $v \neq n_0$, $Pred(v) = n_0$, then

$T_3'(G,e) = (N-\{v\},E-\{e\},n_0)$.

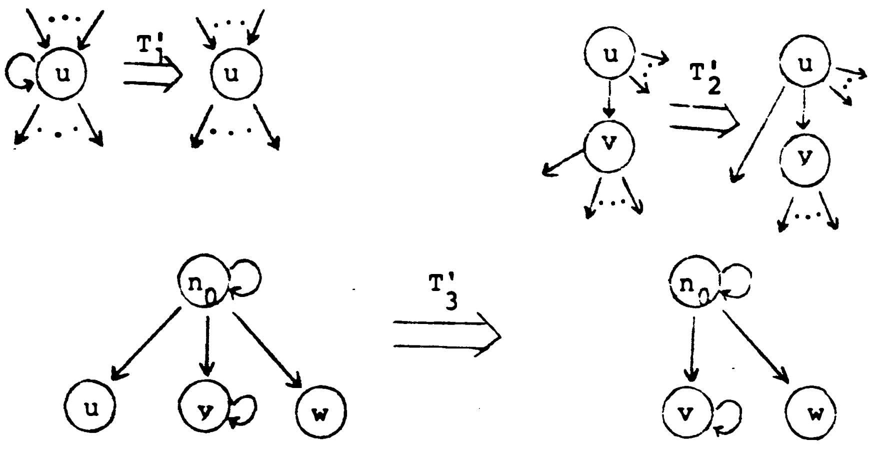


Figure 2

Theorem 2.1 (Graham and Wegman):

Let $G = (N,E,n_0)$ be a reducible flow graph. Then G can be reduced to a graph with single node using no more than $|N|$ T_1' transformations, $O(|E|\log|E|)$ T_2' transformation and $|N|$ T_3' transformations.

2.3 Finite Church-Rosser (FCR) Property

In general graph reductions are highly ambiguous. That is, there may be many different sequences of transformations by which a given graph can be reduced. It is desirable that any sequence of transformation on a given flow graph will always terminate and that the limit graph is unique independent of the order of reduction sequences.

Definition: Two flow graphs $G = (N,E,n_0)$ and $G' = (N',E',n_0')$ are isomorphic if there exists a function $f: N \to N'$ such that

(1) f is one-to-one and onto.

(2) $f(n_0) = n_0'$.

(3) $(x,y) \in E$ if and only if $(f(x),f(y)) \in E'$.

In graph parsing we treat two flow graphs as equal if they are isomorphic. That is, the names of node and edges are not important.

Definition: Let S be a set and $\Rightarrow$ be a relation on S, i.e. $\Rightarrow \subseteq (S\times S)$. The pair $(S,\Rightarrow)$ is finite if for each p in S there is a constant k_p such that if $p \overset{i}{\Rightarrow} q$ then

$i \leq k_p$, where $\overset{i}{\Rightarrow}$ is the i-fold composition of $\Rightarrow$. We say that $(S,\Rightarrow)$ is <u>finite Church-Rosser</u> (<u>FCR</u>) if the following conditions are satisfied:

(1) $(S,\Rightarrow)$ is finite, and

(2) if $p \overset{*}{\Rightarrow} q$ and $p \overset{*}{\Rightarrow} r$ and there is no t with $q \Rightarrow t$ or $r \Rightarrow t$ then $q = r$, where $\overset{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$.

The following theorem gives a test for FCR property.

<u>Theorem</u> [Se]:

Let $\Rightarrow$ be a relation on S. Then $(S,\Rightarrow)$ is FCR if and only if

(1) $\Rightarrow$ is finite, and

(2) if $p \Rightarrow q$ and $p \Rightarrow r$ then there is a t in S such that $q \Rightarrow t$ and $r \Rightarrow t$.

<u>Example</u>:

Let S be the set of flow graphs. Define the relation $\Rightarrow$ on S as $G_1 \Rightarrow G_2$ if and only if

$$G_2 = T_i(G_1), \quad i = 1,2.$$

It is shown in [HU1] that $(S,\Rightarrow)$ is FCR.

3. Global Data Flow Problems

In this section we are going to generalize the results of [GW]. In 3.1 some lattice algebra is presented. In 3.2 - 3.3 straightforward generalization of data flow problem definitions will be given. In 3.4 we first state some lemmas and the key constructions in the proofs which are direct abstractions of the results in [GW], then we show how the reverse problems can be handled in a similar fashion without introducing reverse graph transformations.

3.1 Some Lattice Algebra

As flow analysis is a general term, instead of looking at few typical examples we will formulate flow analysis problems using lattice theoretic constructs [Ki,KU2].

Definition: A semi-lattice is a set L together with a binary meet operation $\wedge$ such that for all x, y and z in L

$$x \wedge x = x \qquad \text{(idempotent)}$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z \qquad \text{(associativity)}$$

$$x \wedge y = y \wedge x \qquad \text{(commutativity)}.$$

Given a semi-lattice L, for x, y in L we say that $x \leq y$ if $x \wedge y = x$. L is said to have a zero element 0 if $0 \leq x$ for all $x \in L$. L is said to have an one element 1 if $x \leq 1$ for all $x \in L$. Let $f\colon L \to L$ be a function from L to L. We say that f is monotonic if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in L$. f is distributive if $f(x \wedge y) = f(x) \wedge f(y)$ for all $x, y \in L$. We also extend the "fastness" defined in [GW]. f is fast if for all x in L, $x \wedge f(x) \leq (f \cdot f)(x)$. A set F of functions from L to L is said to be fast (resp. distributive) if all functions in F are fast (resp. distributive). If g and h are functions from a set S to a semi-lattice L, then

$(g \wedge h)$ denotes the function from S to L such that $(g \wedge h)(s) = g(s) \wedge h(s)$ for all $s \in S$. We say that $g \leq h$ if $g \wedge h = g$. id_L will denote the identity function on L.

Now we are ready to define data flow problems formally by extending the information propagation problem definition in [GW].

3.2 Information Propagation Problems

Definition: A monotone information propagation framework is a tuple $D = (L,\wedge,F)$, where L is a semi-lattice with meet $\wedge$ and F is a set of functions from L to L such that F is closed under function composition and function meet. D is fast (resp. distributive) if F is fast (resp. distributive).

The notion of monotone framework was first introduced in [KU]. Through out this report we assume that all problems are defined under a monotone framework. In [KU] it is shown that the constant propagation problem is usually formulated under a monotone but non-distributive framswork, and it is observed that it is not quite satisfactory to solve such problems by finding fixed points of equations.

Definition: An information propagation problem is a tuple $IP = (G,D,M,x_0)$ where $G = (N,E,n_0)$ is a flow graph, $D = (L,\wedge,F)$ is a monotone information propagation framework, $M: E \rightarrow F$ is a function from E to F, and x_0 L is the boundary information associated with the entry node n_0.

For notational convenience, if $M: E \to F$ is as in the above definition, then f_e denote $M(e)$ for e in E. If $p = p_0,\cdots,p_k$ is a path in G then f_p will denote the composition of functions $f_{(p_{k-1},p_k)}\circ\cdots\circ f_{(p_0,p_1)}$. If p is the null path then $f_p = id_L$.

Notice that id_L might not be in the function space F. Actually, if F is fast then adjoining id_L to F might not preserve the "fastness" property.

Definition: Let $IP = (G,D,M,x_0)$ be an information propagation problem where $G = (N,E,n_0)$ and $D = (L,\wedge,F)$. A function $FP: N \to L$ is a fixed point for IP if $FP(n_0) \le x_0$ and $FP(n) \le f_e(FP(m))$ for all $m \in Pred(n)$. A function $S: N \to L$ is a safe assignment to IP if $S(n) \le f_p(x_0)$ for all path p from n_0 to any node n. A function $AA: N \to L$ is an acceptable assignment to IP if AA is safe and if $FP \le AA$ for all fixed point FP for IP.

The above definition is a direct abstraction of the information propagation problem definition in [GW]. To solve such a problem is to find an acceptable assignment. An acceptable assignment is an assignment of information to nodes in such a way that the information at each node never exceeds any information propagated from n_0 to that node, while the assignment is at least as good as any fixed point.

A typical example of the information propagation problem is the common subexpression elimination problem. Let $G = (N,E,n_0)$ be any flow graph for a certain program. For n N let b(n) be the computation block represented by n. An expression, say A+B, is said to be available at a node n if no path from the previous evaluation of A+B to n can modify either A or B. To formulate the problem of finding available expressions at all nodes we first introduce the well-known functions KILL and GEN used for this purpose. KILL and GEN are functions from E to the set $\mathscr{E}$ of expressions computerd in this program:

For e = (m,n) in E.

(1) GEN(e), denoted GEN_e, is the set of all expressions α in $\mathscr{E}$ such that α is computed somewhere in b(m) and that none of the variables involved in α is modified within b(m) after that computation of $\mathscr{E}$.

(2) KILL(e), denoted $KILL_e$, is the set of expressions α in $\mathscr{E}$ such that some variables involved in α are modified without recomputing α within b(m) afterwards.

Then the common subexpression elimination problem can be formulated as the information propagation problem IP = (G,D,M,x_0), where

$D = (L, \wedge, F)$ with $L = 2^{\mathscr{E}}$, $\wedge$ = set intersection

F = set of all functions from L to L,

M is define as $M(e) = GK(GEN_e, KILL_e)$ for $e \in E$, where $GK: L \times L \to F$ is the function such that $GK(x,y)(z) = (z-y) \cup z$ for all x, y and z in L,

x_0 = the empty set.

In implementation, elements of L can be represented as bit-vectors. And it can be shown [GW] that

$GK(x,y) \circ GK(x',y') = GK((x'-y) \cap x, (y'-x) \cap y)$ and

$GK(x,y) \wedge GK(x',y') = GK(x \cap x', y \cap y')$.

Hence the functional operations can be carried out efficiently using logical bit-vector operations like AND or OR.

3.3 Reverse Information Propagation Problems

Another class of flow problems requires propagation of information from the end of a program. There are some difficulties in defining this kind of problems on general flow graphs. According to the definition of a flow graph there could be several nodes without edges leaving them. A conditional program return in a basic block may not be located if we simply look at a flow graph without any return message. But fortunately the program return infromation can be collected when the flow graph is constructed from a program. Therefore, in dealing with reverse propagation problems, we may assume that the flow graph for a program is constructed in the following way:

(1) First construct an extra return node r.

(2) Construct flow graph as usaual. But when a node n is formed and the block $b(n)$ has

a return instruction at an exit, we add an edge (n,r).

(3) $b(r)$ contains no computation at all.

It is easily seen that if the original flow graph for a program is reducible then the new graph constructed as above is still reducible. The return node r constructed has the property that it is the only node without leaving edges. Hence the above steps give a construction of a "hammock graph" [GW].

The extension of the information propagation problem definition to the reverse information propagation problems is quite obvious now. The reverse problem is just like a mirror image of the forward problem, information is propagated backwards from the return node instead of being propagated from the entry node n_0. The reason we need a separate definition for the reverse problems is that the reverse of a reducible graph might not be reducible, and hence we do not want to treat the reverse problems as forward problems on the reversed graphs.

<u>Definition</u>: A <u>reverse information propagation problem</u> is a tuple $RIP = (G_r, D, M, x_0)$, where $G_r = (N, E, n_0)$ is a flow graph with a return node r such that r is the only node in N without leaving edges, $D = (L, \wedge, F)$ is a monotone information propagation framework, $M: E \to F$ is a function from E to F, and $x_0 \in L$ is the boundary information associated with r.

Again we denote $M(e)$ by f_e for $e \epsilon E$. If $p = p_0, p_1, \cdots, p_k$ is a path in G then $\tilde{f}_p$ denotes the function composition $f_{(p_0,p_1)} \circ f_{(p_1,p_2)} \circ \cdots \circ f_{(p_{k-1},p_k)}$. If p is the null path then let $\tilde{f}_p$ be id_L. Notice that in $\tilde{f}_p$ functions are composed in the reverse order. That is, when $\tilde{f}_p$ is used, $f_{(p_{k-1},p_k)}$ will be applied first.

Definition: Let $RIP = (G_r, D, M, x_0)$ be a reverse information propagation problem. A function $FP: N \to L$ is a fixed point for RIP if $FP(r) \le x_0$ and for all $e = (m,n)$ in E, $FP(m) \le f_e(FP(n))$. A function $S: N \to L$ is a safe assignment to RIP if for all path p from a node n to r, $S(n) \le \tilde{f}_p(x_0)$. A function $AA: N \to L$ is an acceptable assignment to RIP if AA is safe and $FP \le AA$ for all fixed point FP for RIP.

To solve a reverse information problem is to find an acceptable assignment. That is, to find an assignment of information to nodes such that the assignment is no more than any information which can be propagated from r and that it is at least as better as any fixed point solution.

An example of a reverse information problem is the well-known live variable analysis problem. We will follow the notation in [FKZ]. The problem is to determine for each basic block b in a program the set LIVE(b) of all variables X for which there is a path from the entry point of b to a reference of X such that this path is X-clear (i.e. contains no re-definition of X).

Let INSIDE(b) be the set of all variables X which are live on entry to block b because there is a use of X within b which is not preceded by any re-definition of X in b. Let THRU(b) be the set of all variables X for which there exists an X-clear path through b. Both INSIDE and THRU can be computed by local examination of basic blocks and can be implemented as bit-vectors.

Then the live variable analysis problem can be formulated as a reverse information propagation problem RIP = (G_r, D, M, x_0), where G_r is a flow graph with return node r for a certain program, $D = (L, \wedge, F)$ where $L = 2^{\mathcal{V}}$ is the power set of the set $\mathcal{V}$ of all variables used in a program, $\wedge$ is the set union operation, and F is the set of all functions $T_{x,y}$ of the form $T_{x,y}(z) = x \cup (y \cap z)$ for $x, y, z \in L$. It can easily be verified that such D is distributive and fast. As no variables can be live at the end of a program, $x_0 = \phi$. Notice that ϕ is the 1 element of L (and hence x_0 is again a trivial information). Initially if u is a node in G_r representing a basic block b and if (u,v) is an edge in G_r then the propagation function M((u,v)) is defined as $T_{\mathrm{INSIDE(b)},\mathrm{THRU(b)}}$. The equation among the live infromation is

$$\mathrm{LIVE(b)} = \mathrm{INSIDE(b)} \cup \bigcup_{c \in S(b)} (\mathrm{THRU(b)} \cap \mathrm{LIVE(c)}),$$

where S(b) denotes the set of all basic block represented by nodes which are entered by an edge leaving the node representing block b. It can be shown that

$$T_{x,y}\, T_{x',y'} = T_{x\cup(y\cap x'),(y\cap y')} \qquad \text{and}$$

$$T_{x,y}\, \bar{T}_{x',y'} = T_{(x\cup x'),(y\cup y')}.$$

Therefore functional operations can also be implemented efficiently using logical bit-vector operations like AND and OR.

3.4 Solving Flow Analysis Problems on Reducible Graphs

In [GW] Graham and Wegman show that an information propagation problem with boundary information $x_0 = \phi$ and a fast function space can be solved in $O(e\log e)$ functional steps. Their method also has the property that for a nicely structured program without too many exits from loops only $O(e)$ functional steps are required. In this section we extend their results to problems defined in the previous two sections. The generalization to forward problems is essentially a direct translation of the results in [GW]. The generalization to reverse problems is also straight-forward, while we avoid using reverse transformations RT_1', RT_2' and RT_3' so that we can use the same rules T_1', T_2' and T_3' to solve forward and backward problems simultaneously during a parse. Lemmas will be stated and the key constructions in proofs are also given for later use in section 4.

We generalize their results to forward problems first. In the following let $IP = (G,D,M,x_0)$ be an information propagation problem, where $G = (N,E,n_0)$ and $D = (L,\wedge,F)$.

<u>Proof</u>: Define M' such that for all u in U

$$M'((u,w)) = M((u,w)) \wedge (M((v,w)) \circ M((u,v))) \quad \text{if } (u,w) \in E,$$
$$= M((v,w)) \circ M((u,v)) \quad \text{otherwise,}$$

and for all e in $E - \{(x,w) \mid x \in U \cup \{v\}\}$, $M'(e) = M(e)$. The rest of the proof is analogous to the proof of Lemma 4.2 in [GW].

□

<u>Corollary 3.3</u>:

Let $e = (v,w)$ and $e' = (u,v)$ be in E, where $u \neq v$, $v \neq n_0$, $v \neq w$ and (u,v) is the only edge entering v. Let $G' = T_2'(G,e,)$. An information propagation problem $IP' = (G',D,M',x_0)$ can be found using one function meet and one function composition. IP' has the property that if $N' = N$ then any acceptable assignment to IP' is also acceptable to IP, and if $N \neq N'$ then an acceptable assignment to IP can be obtained from an acceptable assignment to IP' by one function application.

<u>Proof</u>: If $N \neq N'$ and AA' is accptable to IP' then then let $AA(v) = f_{(u,v)}(AA'(u))$ and $AA(n) = AA'(n)$ for all n in N'. The rest of the proof is straightforward.

□

<u>Lemma 3.4</u>:

Let $G' = T_3'(G,e)$ be defined for some $e = (n_0,v)$. An information propagation problem $IP' = (G',D,M',x_0)$ can be found using no functional operations. By one functional

To handle the arbitrary boundary information x_0 properly we will restrict the T_1' transformation so that T_1' is not allowed to apply to self-loops (n_0,n_0) at the entry node n_0. Obviously this restriction will not affect the parsing of reducible graphs.

<u>Lemma 3.1</u>:

Let $G' = T_1'(G,e)$ be defined for some e in E such that $M(e)$ is fast. An information propagation problem $IP' = (G',D,M',x_0)$ can be found using one function meet and one function composition such that any acceptable assin-ment to IP' is also acceptable to IP.

<u>Proof</u>: Let $e = (v,v)$ and let $e' = (u,v)$ be the only other edge entering v. Define M' such that

$$M'(e') = M(e') \wedge M(e) \circ M(e') \quad \text{and}$$

$$M'(a) = M(a) \quad \text{for all } a \text{ in } E - \{e,e'\}.$$

The rest of the proof is analogous to the proof of Lemma 4.1 in [GW] by noticing that $v \neq n_0$.

□

<u>Lemma 3.2</u>:

Let v be a node in $N - \{n_0\}$ such that (v,v) is not in E and let $U = Pred(v)$. Let $G' = (N,E',n_0)$, where for some $w \in Succ(v)$, $E' = (E \cup \{(u,w) | u \in U\}) - \{(v,w)\}$. Then an information propagation problem $IP' = (G',D,M',x_0)$ can be found using at most $|U|$ function meets and $|U|$ fuction compositions such that any acceptable assignment to IP' is also acceptable to IP.

application we can find an acceptable assignment to IP from an acceptable assignment to IP'.

Proof: For an acceptable assignment AA' to IP' let $AA(v) = f_{(n_0,v)}(AA'(n_0))$. The rest of the proof is easy.

□

When a reducible graph G is finally reduced to a flow graph with single node n_0, an acceptable assignment AA can be defined as

$$
\begin{aligned}
AA(n_0) &= x_0 && \text{if } (n_0,n_0) \in E, \\
&= x_0 \wedge f_{(n_0,n_0)}(x_0) && \text{otherwise.}
\end{aligned}
$$

The following theorem is a straightforward generalization of Theorem 5.3 in [GW].

Theorem 3.1:

Let IP = (G,D,M,x_0) be an information propagation prblem, where $G = (N,E,n_0)$ is reducible and D is fast. Then we can find an acceptable assignment to IP in $O(|E|\log|E|)$ functional operations.

□

In the remainder of section 3 we consider solutions to reverse information propagation problems on reducible graphs. In the following we assume that RIP is a reverse information propagation problem (G_r,D,M,x_0), where $G_r = (N,E,n_0)$ is a flow graph with return node r, and $D = (L,\wedge,F)$.

It is easily seen that, since the return node r does not have leaving edges, r will remain as a return node during reductions and no self loops can be created at r. Hence T_1' can never apply to r, and we can always use r as the return node for graphs resulted from T_1' or T_2'.

Lemma 3.5:

Let $G_r' = T_1'(G_r, e)$ be defined for some $e = (v,v)$ in E such that $M(e)$ is fast. Let $W = Succ(v) - \{v\}$. Then a reverse information propagation problem $RIP' = (G_r', D, M', x_0)$ can be found using $|W|$ function meets such that any acceptable assignment to RIP' is also acceptable to RIP.

Proof: Define M' such that for all w in W, $M'((v,w)) = M((v,w)) \wedge (M(e) \circ M((v,w)))$, and for all (x,y) in E, $x \neq v$, $M'((x,y)) = M((x,y))$. The rest of the proof is similar to the proof of Lemma 4.1 in [GW]. Notice that $v \neq r$.

□

The $2|W|$ functional operations used in the above lemma seem to be quite costly. But later we will see that the total number of such functional steps is still $O(|E|\log|E|)$.

Lemma 3.6:

Let v be a node in $N - \{n_0\}$ such that $(v,v) \notin E$ and let $U = Pred(v)$. Let $G_r' = (N, E', n_0)$, where for some $w \in Succ(v)$ such that (v,w) is not the only edge leaving v,

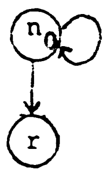
<u>Corollary 3.7</u>:

Let $G_r' = T_2'(G_r, e)$ be defined for some $e = (v,w)$ in E and $e' = (u,v)$ be the only edge entering v. A reverse information propagation problem $RIP' = (G_r', D, M', x_0)$ can be found using one function composition and, if (u,w) is in E, one function meet. RIP' has the property that an acceptable assignment AA to RIP can be obtained from an acceptable assignment AA' to RIP' using one function application if v is not in G_r', and one extra function meet if v is in G_r'.

<u>Proof</u>: If $v \in G_r'$ then the lemma follows from Lemma 3.6. If $v \notin G_r'$ then define AA such that $AA(v) = f_{(v,w)}(AA'(w))$, and for all $u \neq w$, $AA(n) = AA'(n)$. The rest of the proof is easy.

□

Due to the properties of the return node, the fan graph produced by applying T_1' and T_2' to G_r is of the form



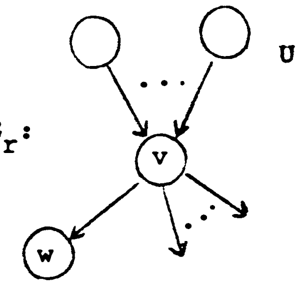
<u>Lemma 3.8</u>:

If G_r is a fan graph and, if $(n_0,n_0) \in E$, $M((n_0,n_0))$ is fast then an acceptable assignment to RIP can be found using one function application and, if $(n_0,n_0) \in E$, another function application.

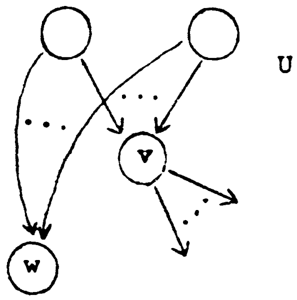
$E' = (E \cup \{(u,w) \mid u \in U\}) - \{(v,w)\}$. Then a reverse information propagation problem $RIP' = (G_r', D, M', x_0)$ can be found using at most $|U|$ function meets and $|U|$ function compositions such that an acceptable assignment AA to RIP can be obtained from an acceptable assignment AA' to RIP' using one function meet and one function application.

<u>Proof</u>: The situation of v and w in G_r and G_r' are depicted in the picture below

In G_r:



In G_r'



Define M' such that for all u in U

$$M'((u,w)) = M((u,w)) \wedge (M((u,v)) \circ M((v,w))) \quad \text{if } (u,w) \text{ is in } E,$$

$$= M((u,v)) \circ M((v,w)) \quad \text{otherwise,}$$

and for all e in $E - \{(x,w) \mid x \in U \cup \{v\}\}$, $M'(e) = M(e)$.

If AA' is an acceptable assignment to RIP' than AA is constructed such that

$$AA(v) = AA'(v) \wedge f_{(v,w)}(AA'(w)), \text{ and}$$

$$AA(n) = AA'(n) \quad \text{if } n \neq v.$$

The rest of the proof is simple.

□

Proof: If $(n_0, n_0) \notin E$ then define $AA(r) = x_0$ and $AA(n_0) = f_{(n_0,r)}(x_0)$. If $(n_0,n_0) \in E$ then we define $AA(r) = x_0$ and $AA(n_0) = f_{(n_0,r)}(x_0) \wedge f_{(n_0,n_0)}(f_{(n_0,r)}(x_0))$. The rest of the proof is easy.

$\square$

Now we are ready to look at the running-time of reverse information propagation problem solution on reducible graphs.

Theorem 3.2:

Let $RIP = (G_r, D, M, x_0)$ be a reverse information propagation problem, where $G_r = (N, E, n_0)$ is reducible and D is fast. Then an acceptable assignment to RIP can be found using $O(|E|\log|E|)$ functional operations.

Proof: Combining Theorem 2.1, Lemma 3.5, Lemma 3.6, Corollary 3.7 and Lemma 3.8, the theorem will follow if we can show that the total number of functional operations used on T_1' reductions in Lemma 3.5 is $O(|E|\log|E|)$.

Following the notation in Lemma 3.5, let us consider the case when W is non-empty. Since $v \neq n_0$, v will eventually be eliminated from N. Before v is eliminated there must be $|W|$ T_2' reductions applied to the unique edge entering v. Since T_1' can apply to each node at most once, the total number of functional operations spent on T_1' reductions is bounded by twice the number of T_2' applications. Then the theorem follows by Theorem 2.1.

$\square$

4. Data Flow Analysis on Other Classes of Flow Graphs

In this section we are interested in subclasses of reducible graphs on which flow analysis problems can be solved efficiently. In 1966 Böhm and Jacopini [BJ] showed that three control structures — sequencial statements, conditional statements and while loops — are sufficient to program any computation. Together with the "repeat-until" loop structure suggested by Wirth[Wi] a simple graph grammar, the B-J grammar, can generate the flow graph for usual "structured" programs. Kennedy [Ke] has shown that certain flow analysis problems can be solved in linear steps using node listing techniques.

Recently Farrow, Kennedy and Zucconi [FKZ] proposed a new graph grammar G_{SSFG} which extends the B-J grammar to cover bigger class of flow graphs. They also show that certain flow analysis problems can be solved in linear bit-vector steps on this class of flow graphs.

The reduction rules in B-J grammar and G_{SSFG} are rather simple, allowing at most two edges leaving each node, so that flow analysis problems can be reduced very easily. In the remaining of this section we propose new graph grammars SG(k) which can cover bigger classes of flow graphs and still allow flow problems be solved efficiently.

4.1 Graph Grammars SG(k)

For $k \geq 1$ we define SG(k) using five graph

R_3: If e and e' are duplicates and $e \neq e'$ then $R_3(G,e,e') = (N,E-\{e\},n_0)$. Therefore R_3 eliminates duplicated edges.

R_4: Let $e = (u,v)$ be the only edge entering v, $u \neq v$, $v \neq n_0$, and neither of u or v has duplicated leaving edges. Let
$W_e = \{w \mid (v,w) \in E \text{ and } u \neq w\}$ and
$S_e = \{w \mid (v,w) \in E \text{ and } (u,w) \in E,\ u \neq w\}$.
If there are no more than k non-looping edges leaving u and $|W_e - S_e| \leq 1$ then $R_4(G,e) = (N',E',n_0)$, where $N' = N - \{v\}$ and $E' = (E \cap (N' \times N')) \cup \{(u,w) \mid w \in (W_e - S_e)\}$. Hence R_4 is a special case of T_2 such that no more than k edges can leave u and that the number of leaving edges from u will not increase after collapsing u and v.

R_5: Let $e = (u,v) \in E$, $u \neq v$, $v \neq n_0$. If e is the only edge entering v and v has at most one leaving edge then $R(G,e) = (N - \{v\},E',n_0)$, where E' is $E - \{e\}$ if no edges leave v, and if $(v,w) \in E$ then E' is obtained by first deleting e and then creating a new edge from u to w.

The five transformations are depicted in Figure 3.

transformation rules. In defining the rules we allow duplicated (or parallel) edges in graphs, i.e. two edges with different names might enter the same node and leave the same node. The existence of duplicated edges is not necessary in defining SG(k). But we need to keep duplicates of edges in actual graph parsing if we want to implement it efficiently. Duplicated edges are eliminated only when it is necessary or when it will not cost too much (e.g. when no searching is required). In the five reduction rules, whenever a new edge is created a new name will be assigned. A newly created edge might be a duplicate of some existing edges, but the new name will make it a new edge and no elimination will be done unless it is specifically stated.

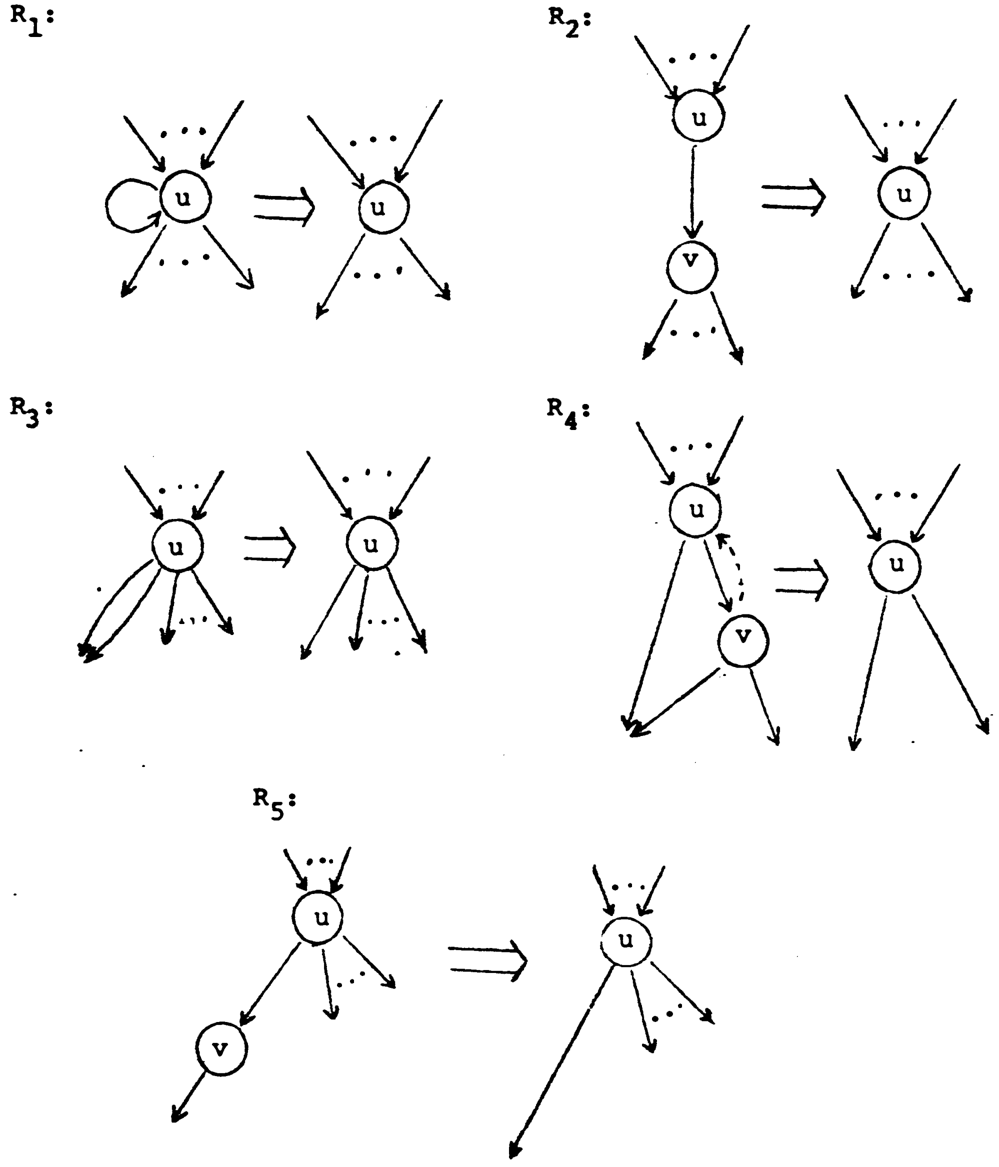
The five graph reduction rules of SG(k) are:

R_1: If $e = (v,v) \in E$ then

$$R_1(G,e) = (N, E-\{e\}, n_0).$$

Therefore R_1 is almost the same as T_1 except that the eliminated self-loop might still have other duplicates.

R_2: If $e = (u,v)$ is the only edge leaving u, $v \neq n_0$, and e is the only edge entering v. Then $R_2(G,e) = (N-\{v\}, E', n_0)$ where E' is formed by first deleting e and then replacing each edge (which could have duplicates) leaving v by a new edge which leaves u instead.

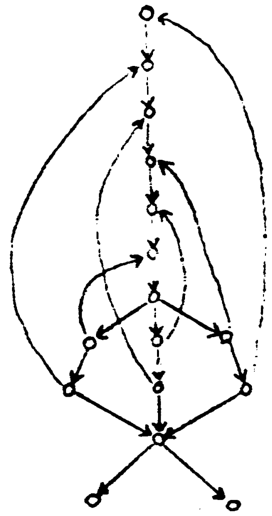


<u>Figure 3</u> SG(2) Rules

One thing worth mentioning is that among the five reduction rules in SG(k), R_5 is the only one which can create new duplicated edges. R_2 might preserve duplicated edges, but it will never increase the number of duplicated edges. The reason we do not allow duplicated edges in R_4 is that we can easily check if there are more than k non-duplicated edges leaving a certain node.

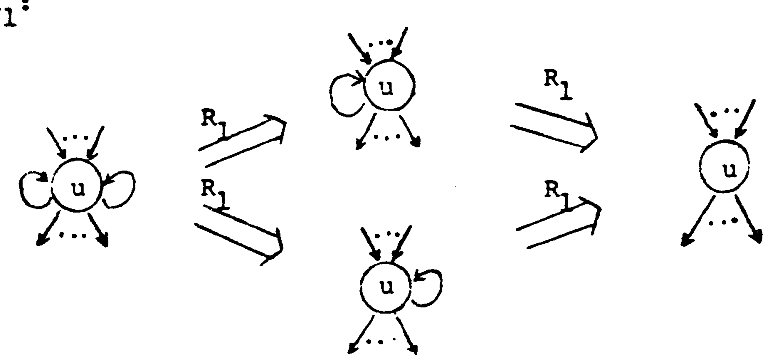
<u>Definition:</u> $\mathscr{F}_{SG(k)}$ is the family of all flow graphs which can be reduced to the trivial graph by repeated application of the five transformations in SG(k).

It is easily seen that $\mathscr{F}_{SG(k)}$ get bigger as k increases and $\mathscr{F}_{SSFG} \subseteq \mathscr{F}_{SG(1)}$. Furthermore, since G_{SSFG} allows at most two edges to leave each node, the structure of the flow graphs in $\mathscr{F}_{SG(3)}$ is richer. An example is the graph given in Figure 4.



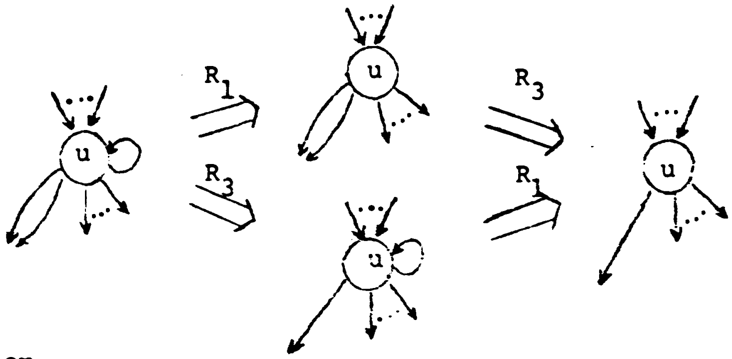
<u>Figure 4</u> A Graph in $\mathscr{F}_{SG(3)}$

R_1 - R_1:

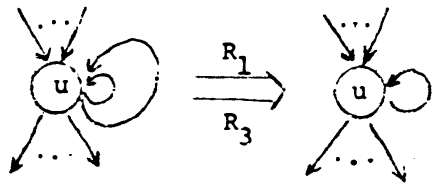


R_1 - R_2: These two operations are obviously disjoint.

R_1 - R_3:



or



4.2 Finite Church-Rosser Property for SG(k)

As explained in section 2.2 the FCR property is important for graph grammar design. For simplicity we will only prove the FCR property for SG(2). The proof can easily be generalized to show the FCR property for all SG(k).

Theorem 4.1:

Let $\mathcal{G}$ be the family of all flow graphs and let $\Rightarrow$ be the relation on $\mathcal{G}$ defined by reduction rules of SG(k), then $(\mathcal{G}, \Rightarrow)$ is FCR.

Proof: By Theorem 2.2, we need only to show that

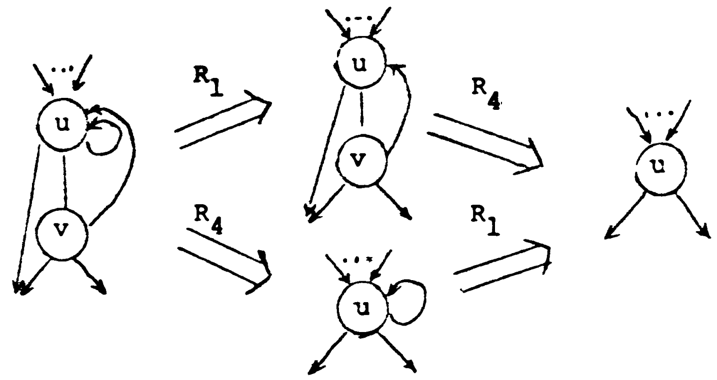
(a) $(\mathcal{G}, \Rightarrow)$ is finite, and

(b) If $G\ \mathcal{G}$ and $G \Rightarrow G_1$, $G \Rightarrow G_2$, then there is a flow graph G_3 such that $G_1 \overset{*}{\Rightarrow} G_3$ and $G_2 \overset{*}{\Rightarrow} G_3$.

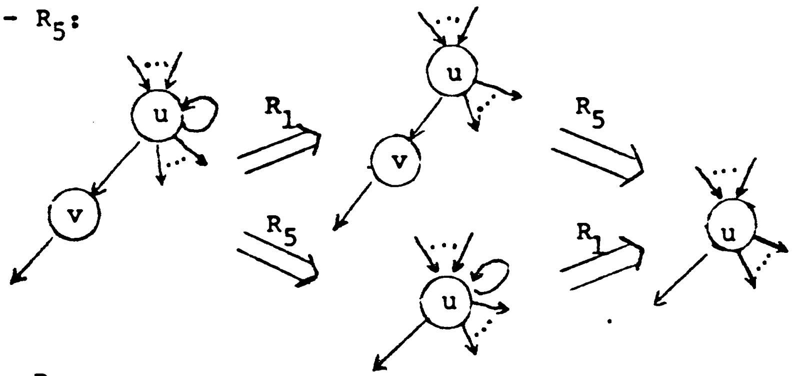
Property (a) is true since each flow graph has only finite number of edges and each of the five reduction rules decreases the number of edges by at least one.

To prove property (b) we need to consider the cases where the reductions from G to G_1 and G_2 interact with each other, for otherwise those two reductions can be performed independently and the order is not important. Therefore we assume that the reductions of G to G_1 and G_2 apply to nodes and edges which overlap. We will exhaust all possibilities of such interaction by considering the diagrams of possible interacting reductions.

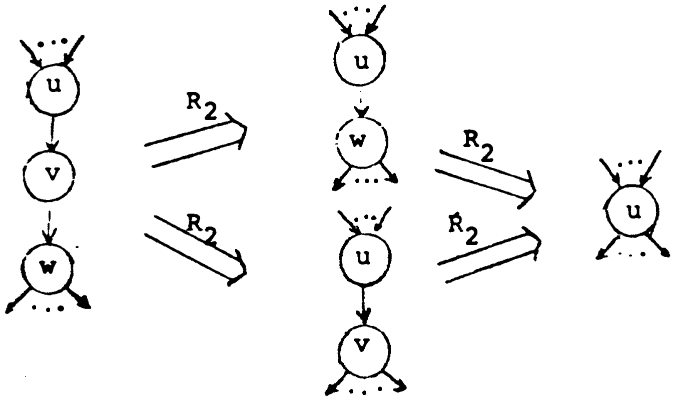
$R_1 - R_4$:



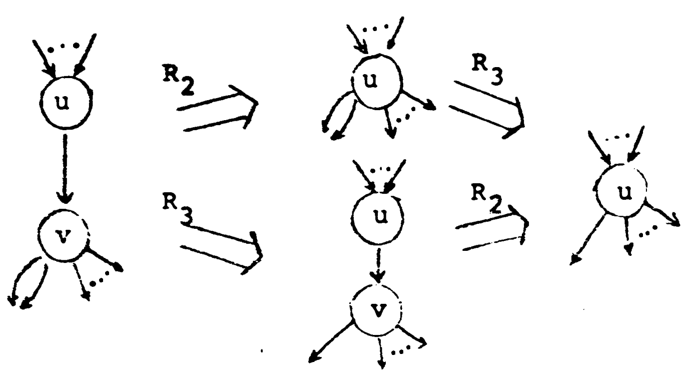
$R_1 - R_5$:



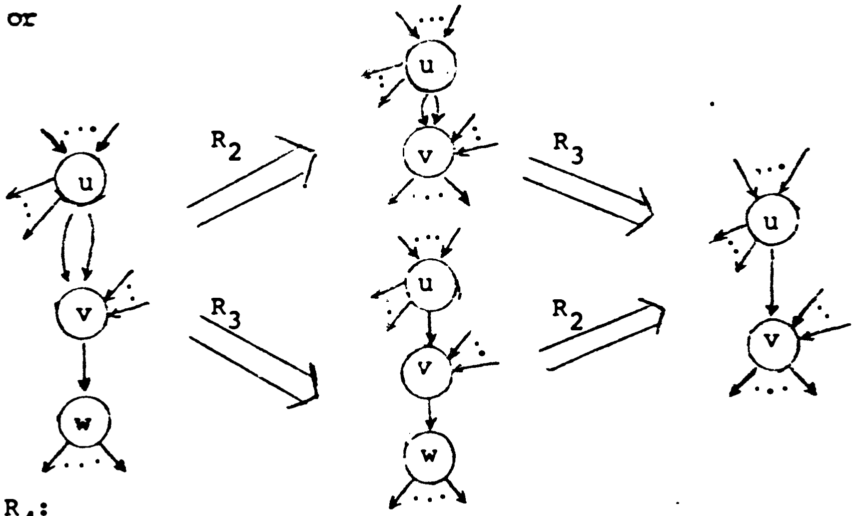
$R_2 - R_2$:



$R_2 - R_3$:

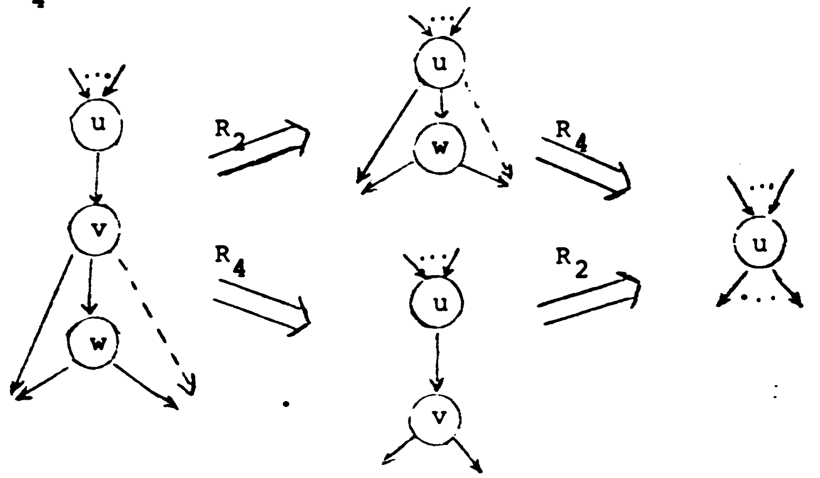


or



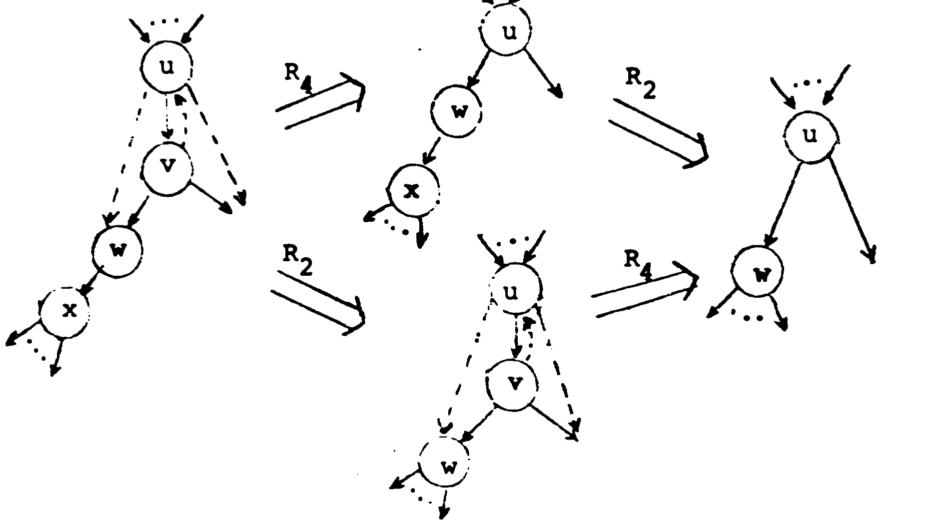
u
v
w
R_2
R_3
u
v
u
v
w
R_3
R_2
u
v

$R_2 - R_4$:



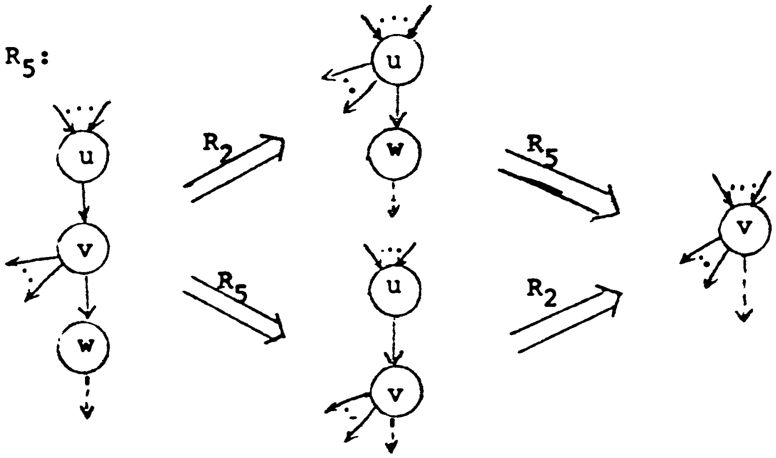
u
v
w
R_2
R_4
u
w
u
v
R_4
R_2
u

or



u
v
w
x
R_4
R_2
u
w
x
u
v
w
R_2
R_4
u
w

R_2 - R_5:

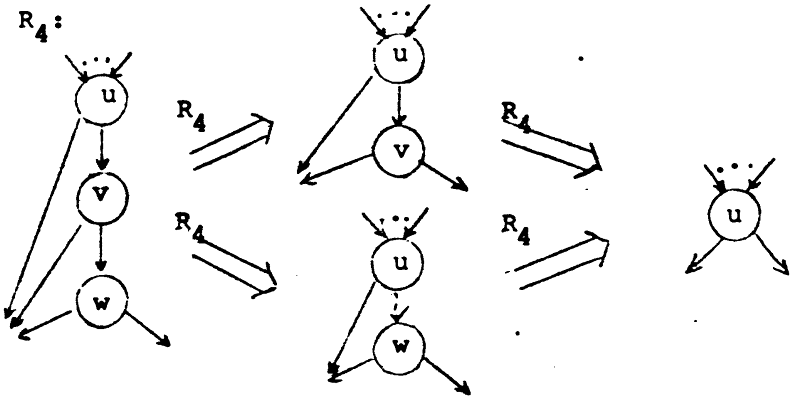


R_3 - R_3: This case is obvious.

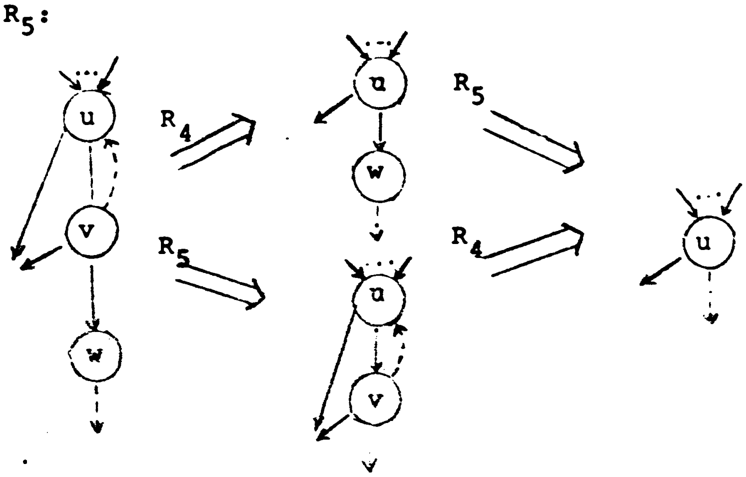
R_3 - R_4: This case is also obvious because of the non-duplicate restriction on R_4.

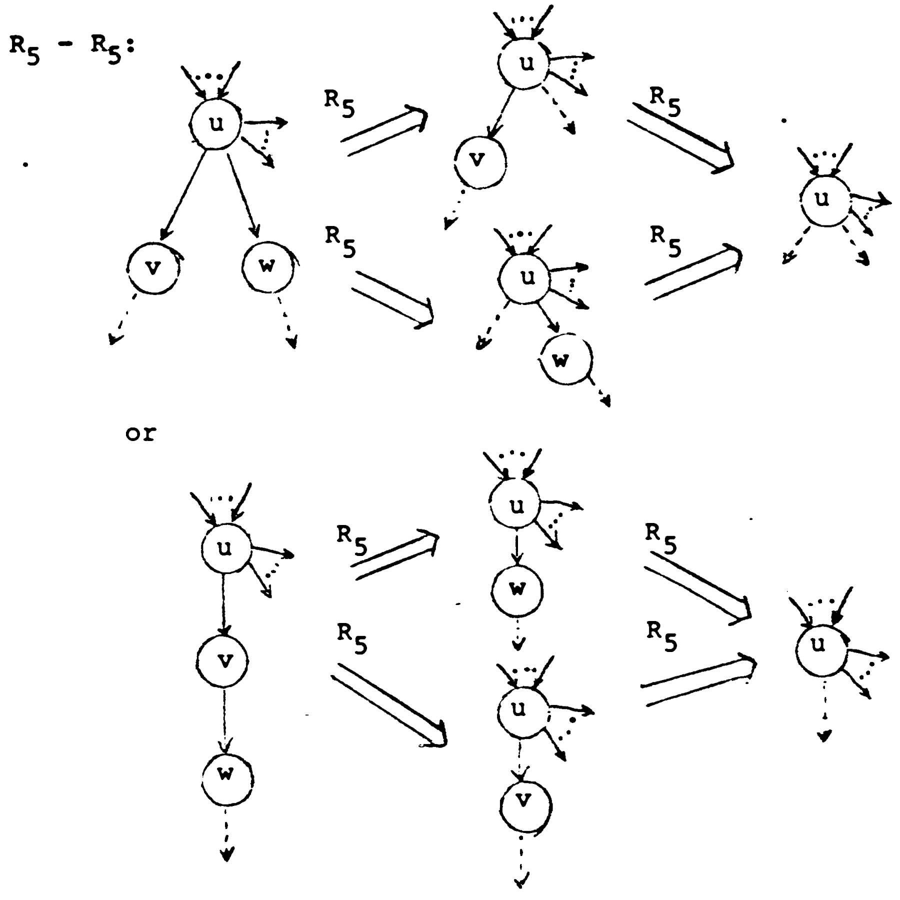
R_3 - R_5: This case is clear.

R_4 - R_4:



R_4 - R_5:





□

4.3 Parsing for $\mathscr{F}_{SG(k)}$

Our goal is to find a sequence of SG(k) reduction transformations which can apply to a given flow graph, and we want to do this in $O(e)$ steps for a flow graph with e edges.

First we introduce the data structures we use in performing SG(k) parsing. Let $G = (N,E,n_0)$ be a flow graph. We first do a depth-first search [T] starting at n_0. Then we number the nodes with integers $1,2,\cdots,|N|$ in such a way as to form an acyclic order, i.e. if (u,v) is not a backedge then the number at u is less than the number at v. This step can easily be done in $O(|E|)$ steps. Each node n in E has two pointers PIN(n) and POUT(n) pointing to two linear lists IN(n) and OUT(n) respectively. IN(n) is a list of all edges entering n, and OUT(n) is a list of all edges leaving n. Notice that both IN and OUT lists might contain duplicates of edges. And for each edge $e = (m,n)$ in E, the elements representing e in IN(n) and OUT(m) are double-linked so that insertion and deletion of edges can be handled efficiently.

When R_1 or R_3 applies to a certain edge, we only need to perform an edge deletion. When R_2 applies to $e = (m,n)$ we simply point POUT(m) to OUT(n). When R_4 applies to $e = (m,n)$ the lists can be updated in a bounded amount of time because none of m and n has more than $k+1$ leaving edges. When R_5 applies, at most one insertion and two deletions will update the structures.

To implement R_1 and R_3 efficiently in parsing, we do them only when we need to verify the applicability of other reduction rules. Also we must ensure that execution of R_1 and R_3 do not "waste" steps. Algorithm 1 which follows is designed to satisfy these requirements. When a node n is passed to Algorithm 1 the first $k+1$ edges in OUT(n) are examined. If the first $k+1$ edges in OUT(n) are all "distinct" non-looping edges then neither R_2 nor R_4 can apply to edges in OUT(n), and hence

Algorithm 1:

Input: (1) A flow graph $G = (N,E,n_0)$,

(2) a node n in N,

(3) a partial parse P which is a list of all reductions made so far.

Output: (1) A flow graph $G' = (N,E',n_0)$ resulted from some R_1 and R_3 reductions on G. G' has the following properties:

(a) There are no duplicates or self-loops among the first k+1 edges in OUT(n).

(b) Analogous property for IN(n) as in (a).

(2) P updated by all successive reductions.

Method:

```
begin
   while  there are duplicates or self-loops
          among the first k+1 edges in OUT(n)
          (or all edges in OUT(n) if there are
          less than k+1 left)    do
          begin
             eliminate all those duplicates and
                 self-loops by R1 or R3;
             update P by successive reductions
                 applied;
             end
  do same steps as above to IN(n);
  end
```

Figure 5

Algorithm 1 simply quits. On the other hand, if there are self-loops or duplicates among the first k+1 edges in OUT(n) then the self-loops and duplicates are eliminated by R_1 and R_3 and the process is repeated to the new OUT(n) list. The elimination of edges assures the efficiency of Algorithm 1.

<u>Lemma 4.2</u>:

Algorithm 1 works correctly and terminates in O(s) steps, where s is the number of edges eliminated.

Proof: Easy induction on $|E|$.

□

Now we are ready to look at SG(k) parsing. Due to the FCR property the order of reduction rule applications is not important. R_1 and R_3 are performed only in the execution of Algorithm 1 calls. Applicability of R_2 and R_4 on an edge e = (m,n) can easily be verified at either m or n. Applicability of R_5 can always be checked efficiently at node n.

The SG(k) parsing algorithm is described in Figure 5. Algorithm 2 contains calls to two subroutines UP and DOWN which are described in Figure 6 and Figure 7. DOWN(n) will collapse edges leaving n as far as possible. UP(n) is used basically for detecting candidates of reductions on edges in IN(n). At the beginning all nodes of G are on a list L in an acyclic order. Nodes are chosen from L successively. For each node chosen, if some reductions can apply to a region involving n, then Algorithm 2 tries to perform a chain of interacting reductions.

Algorithm 2: SG(k) parsing.

Input: A flow graph $G = (N,E,n_0)$ with nodes numbered in acyclic order $1,2,\cdots,|N|$.

Output: A parse P for G if G is in $\mathscr{F}_{SG(k)}$, otherwise a failure report.

Method:

```
begin
    P := ϕ;
    for s = 1 to |N| do
        begin
            n := s;
            if n has not been deleted from G yet then
                begin
                    iterate := true;
                    while iterate do
                        begin
                            call Algorithm 1 on node n;
                            call DOWN(n);
                            call UP(n);
                        end
                end
        end
    if G is now the trivial graph then P is a parse;
    else report failure;
end
```

Figure 5

```
procedure  DOWN(n)

    begin
        while  |OUT(n)| ≤ k  do
           begin
              call  Algorithm 1 on each node entered by edges
                    in  OUT(n);
              if  any of R2 and R4 can apply to an edge
                  in  OUT(n)  then
                  begin
                     choose an edge  e  from OUT(n) and a
                         reduction R from R2 and R4 such that
                         R can apply to  e;
                     apply  R  to  e;
                     update  P  by  R;
                     call Algorithm 1 on  n;
                     end

              else  go to end_of_down;
              end
        end_of_down:
        end
```

Figure 6

Notice that DOWN(n) does not test for R_5 application since R_5 is a special case of R_4 if $|OUT(n)| \leq k$.

```
procedure  UP(n)
    begin
1   .  if |IN(n)| > 1  or  n = n0  then  return;
2      let  e = (m,n) be the unique edge in IN(n);
3      if  R2 or R4 are applicable to e  then
           begin
4             apply R2 or R4 to e and update P;
5             n := m;
6             return;
              end
7      if  R5 is applicable to (m,n);
           begin
8      A:     apply  R5  to (m,n) and update P;
9             if  before applyign the R5 there is a unique
                  edge (n,n') in OUT(n), n ≠ n0  then
                  begin
10                   call Algorithm 1 on  n';
11                   if  R5 is now applicable to (m,n') then
                         begin
12                          n := n';
13                          go to  A;
                            end
14            n := m;
              end
15     else  iterate := false;
       end  of UP
```

Figure 7

When UP(n) is called in Algorithm 2, R_2 and R_4 are not applicable to any edge in OUT(n), and therefore it tries to apply R_2, R_4 or R_5 to collapse a possible edge in IN(n). If none of R_2, R_2 and R_5 is applicable to edges in IN(n) then iterate is set to <u>false</u> so that Algorithm 2 will process the next node n at statement 2. In UP we need a chain of R_5 reductions since an R_5 reduction might create another instance of R_5 application which might not be easily detected at node m. The situation is illustrated in Figure 8 below.

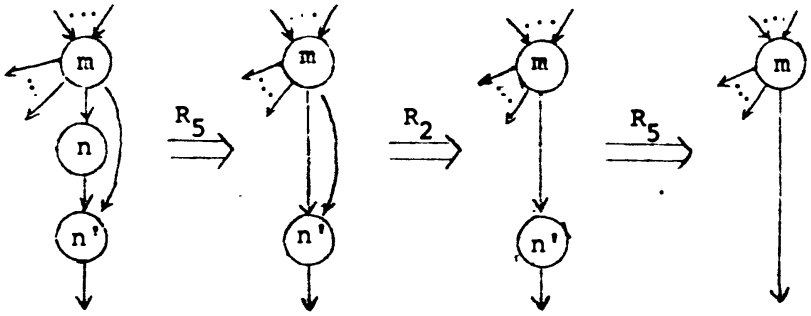


Figure 8

In Figure 8, after an R_5 applies to (m,n), it might not be easy to verify the applicability of R_5 on (m,n') as $|OUT(m)|$ could be very large.

Before we prove the correctness of Algorithm 2 we need some observations.

Observation 1: If there are no R_2, R_4 or R_5 reductions on edges in OUT(n) between two calls of Algorithm 1 on n, then the second call of Algorithm 1 will not change OUT(n).

Observation 2: During the execution of a call of UP(n), if none of R_2, R_4 and R_5 is performed then iterate will be set to false, otherwise n will be set to m upon exit of UP, where m is the unique predecessor of n upon entry to UP(n).

Observation 3: In a call of DOWN(n), n is never changed. Just before the exit from a call of DOWN(n), a call of Algorithm 1 on n will be executed.

Lemma 4.3:

Right before n is set to s, $1 \leq s \leq |N|$, at statement 2 of Alogrithm 2, for $m = 1,2,\cdots,s-1$ either m has been eliminated (by R_2, R_4, or R_5) or the following will be true:

(a) OUT(m) cannot be changed by calling Algorithm 1 on m.

(b) IN(m) cannot be changed by calling Algorithm 1 on m.

(c) Neither R_2 nor R_4 is applicable to edges in OUT(m).

(d) None of R_2, R_4 and R_5 is applicable to edges in IN(m).

Proof: The proof is by induction on s.

When s = 1 the lemma is clearly true.

Assume that the lemma is true for s < K, where $1 \leq K \leq |N|$. We need to show that the lemma is also true when s = K.

Let ϕ denote the moment when n is set to K at state-statement 2 of Algorithm 2. So there is a moment ϕ when

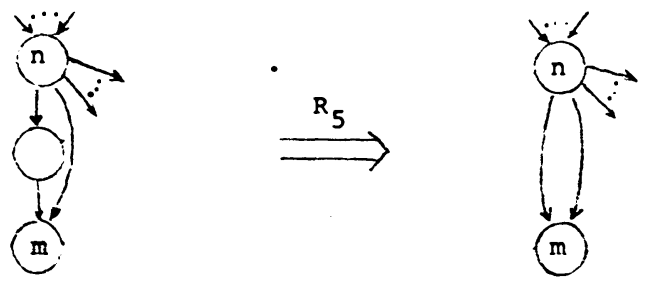
n is about to be set to $K-1$ at statement 2 of Algorithm 2. Assume that the lemma is violated by some $m < K$ at ϕ. Of course m cannot have been eliminated from G before ϕ.

First consider (a). Assume that (a) is false at ϕ. Let η be the last time before ϕ when (a) is changed from true to false for m. By Observation 1, the change of η must be causes by R_2, R_4 or R_5 on some edge $e = (m,n)$ in OUT(m). But then by Observation 2 and Observation 3, a call of Algorithm 1 on m will be executed at statement 7 of Algorithm 2. Then, by Observation 1, we get a contradiction to the choice of η. Therefore (a) must be true at ϕ.

Assume that (b) is false at ϕ. Again consider the last moment η before ϕ when (b) is changed from true to false for m. By reasoning similar to the above argument the change at η cannot be caused by an Algorithm 1 call or by R_2, R_4 or R_5 on an edge in OUT(m). The change cannot be caused by an R_2 reduction as the only possible choice of R_2 at η is an R_2 on an edge in OUT(m). The change cannot be caused by R_4 as in that case the test at statement 1 of DOWN would be true and hence the execution of DOWN would force (b) true at m again. Similarly the change cannot be caused by R_5. Therefore the change at η can never occur and so (b) must be true at ϕ.

Assume that (c) is false at ϕ. Again consider the last time η before ϕ when (c) is changed from true to false for m. The change cannot be caused by R_2 or R_4 as the construction of DOWN and UP would force the reduction to be carried out before n is set to K at statement 2 of Algorithm 2. By Observation 2 the change cannot be caused by R_5 reduction on an edge in OUT(m). Therefore the only possible change at η is an R_5 on an edge in OUT(n) such that n is successor of m. But then by construction of UP (c) has to be true again before ψ, which would violate the choice of η. Hence (c) must be true at ψ.

Assume that (d) is false at ψ. If R_2 or R_4 are applicable to an edge (n,m) in IN(m) then $n < m$ by the choice of acyclic order $1,2,\ldots,|N|$. Hence, by (a), (b) and (c), the violation must be that an R_5 can apply to an edge (n,m). Consider the last time η before ψ when (d) is changed from true to false for m. By Observation 2 the change cannot be caused by R_5 reduction on an edge in OUT(m). Then the only possible change at η is depicted in the picture below.



That is, the R_5 at η must create a duplicate of an edge from n to m such that R_5 is applicable to (n,m) after η. But after the R_5 reduction at η statements 10 - 13 and statement 8 of UP would eliminate node m by an R_5 reduction which would contradict the assumption that m has not been eliminated at ψ. Hence (d) must also be true at ψ. □

Lemma 4.4:

Algorithm 2 always terminates.

Proof: We will prove that each iteration of the for loop in Algorithm 2 will eventually terminate. Then, since s never gets modified except at statement 1 in Algorithm 2, and hence at most $|N|$ iterations can be executed, Algorithm 2 will finally terminate.

We prove by induction on s that the s-th iteration of the for loop will be executed and will terminate, for $0 \leq s \leq |N|$.

When $s = 0$ the assertion is obviously true.

Assume that the assertion is true for all s, $s < K$, where $K \leq |N|$. Then after the termination of the first $K-1$ for loop iterations n will be set to have value K at statement 2 of Algorithm 2. From this point on iterate will eventually be set to false in UP since the number of possible reductions on G is no more than $|E|$ and hence statement 8 of DOWN and then statement 15 of UP will eventually be executed. And then the K-th iteration of the for loop will also terminate.

Therefore the lemma is proved by induction principle.

$\square$

<u>Theorem 4.5</u>:

Algorithm 2 terminates with correct output.

<u>Proof</u>: During the execution of Algorithm 2 a reduction can be performed only after being verified applicable, and the parse list P is always updated after each reduction. Therefore, if G is reduced to the trivial graph at the end of Algorithm 2 then P must be a correct parse.

On the other hand, suppose G is not the trivial graph at the end of Algorithm 2. By Lemma 4.3 no further reductions is applicable. Thus by FCR property $G \notin \mathcal{F}_{SG(k)}$. Hence the failure report is correct.

□

Now let us examine the running time of Algorithm 2.

<u>Theorem 4.6</u>:

Algorithm 2 terminates in $O(|E|)$ steps on a flow graph $G = (N,E,n_0)$.

<u>Proof</u>: From Lemma 4.4 Algorithm 2 will eventually terminate on G. Let A be the number of iterations executed on the <u>for</u> loop and let b be the number of iterations executed on the inner loop of Algorithm 1. Let C be the total amount of work spent on the execution of statements 7 - 9 of Algorithm 2. The total running time of Algorithm 2 on G is $O(A+B+C)$.

Clearly $A = |N|$. B is $O(|E|)$ since in UP(n) iterate is set to <u>false</u> unless an R_5 reduction is performed and the total number of reductions is no more than $|E|$. By Lemma 4.2

the total number of steps spent on statement 7 of Algorithm 2 is $O(|E| + B)$ and hence is $O(|E|)$. The total number of steps spent on statement 8 of Algorithm 2 is clearly bounded by the amount of work spent on R_2 and R_4, and hence is $O(|E| + B)$, or $O(|E|)$. Similarly statement 9 takes $O(|E|)$ steps in total. Therefore C is $O(|E|)$. And hence Algorithm 2 terminates in $O(|E|)$ steps.

□

Corollary 4.7:

SG(k) parsing on a flow graph $G = (N,E,n_0)$ can be done in $O(|E|)$ elementary steps.

Proof: The proof follows from Theorem 4.5 and Theorem 4.6.

□

4.4 Information Propagation Problems on $\mathcal{F}_{SG(k)}$

We have shown that an SG(k) parser works in $O(|E|)$ time. In this section we present a solution method for information propagation problems on $\mathcal{F}_{SG(k)}$.

We cannot apply the problem reduction technique of [GW] directly by decomposing each SG(k) reduction rule into a series of T_1', T_2' and T_3' transformations, since R_1 and R_2 could be very costly. Furthermore, the existence of duplicated edges precludes eliminating edges in a straightforward way. To get around these difficulties we will use a weighting technique to remember information. We perform the problem reduction technique described in section 3 by manipulating the weighted information properly.

Let IP = (G,D,M,x_0) be any information propagation problem, where $G = (N,E,n_0)$ and $D = (L,\wedge,F)$. Two weighting functions W_1, W_2: $N \rightarrow F \cup \{id_L\}$ will be used. Initially $W_1(n) = W_2(n) = id_L$ for each node n. The weighting functions are used to mimic another reduction of the information propagation problem. Let $\overline{IP} = (\overline{G},D,\overline{M},x_0)$ denote the mimicked problem. When an R_1 reduction eliminates a self-loop at node n the mimicked problem does not really eliminate the self-loop and the information associated with the self-loop is accumualated in $W_1(n)$. When a reduction R_2, R_4 or R_5 applies to an edge (m,n) in G, the mimicked problem first eliminates the mimicked self-loop at n by applying the reduction technique described in the proof of Lemma 3.1 using the accumualated information $W_1(n)$. W_2 is used to store flow information for edges eliminated by R_2 reductions. When an R_2 reduction applies to an edge e = (m,n) in G we do not update $f_{e'}$ as $f_{e'} \circ f_e$ for all edges e' in OUT(n), as that could be very slow. Instead we store f_e in $W_2(m)$. The invariant relation we maintain between IP and $\overline{IP}$ is that $\overline{M}(e) = M(e) \cdot W_2(m)$ for all edges e = (m,n) in IP. Notice that $\overline{G}$ is the same as G except for several mimicked self-loops which are already eliminated from G by R_1.

The actions on each reduction rule of SG(k) are described below.

R_1: Let $G' = R_1(G,e)$ be defined for a self loop e at node v.

<u>Action</u>: $W_1(v) := W_1(v) \wedge \overline{M}(e)$;

R_2: Let $G' = R_2(G,e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.

<u>Action</u>: $M(e) := M(e) \wedge (W_1(v) \circ M(e))$; (1)

$W_2(u) := W_2(v) \circ \overline{M}(e)$;

Notice that the assignment in (1) has the side effect of modifying the value of $\overline{M}(e)$.

Any acceptable assignment AA to the reduced problem can be extended to an acceptable assignment to the original problem by defining

$AA(v) := \overline{M}(e)(AA(u))$; where

$\overline{M}(e)$ is evaluated using the $M(e)$ obtained in (1).

R_3: Let $G' = R_3(G,e,e')$ be defined for e and e'.

<u>Action</u>: $M(e') := M(e') \wedge M(e)$;

R_4: Let $G' = R_4(G,e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.

<u>Action</u>: $M(e) := M(e) \wedge (W_1(v) \circ M(e))$; (2)

<u>if</u> $(v,u) \in E$ <u>then</u>

$W_1(u) := W_1(u) \wedge (\overline{M}((v,u)) \circ \overline{M}(e))$;

<u>for</u> each $(v,w) \in E$, $v \neq w$ <u>do</u>

<u>if</u> $(u,w) \notin E$ <u>then</u>

$M((u,w)) := \overline{M}((v,w)) \circ M(e)$;

<u>else</u> $M((u,w)) :=$

$M((u,w)) \wedge (\overline{M}((v,w)) \circ M(e))$;

Any acceptable assignment AA to the reduced problem can be extended to an acceptable assignment to the original problem by defining

$AA(v) := \overline{M}(e)(AA(u))$; where $\overline{M}(e)$ is evaluated using the $M(e)$ obtained in (2)

R_5: Let $G' = R_5(G,e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.

<u>Action</u>: $M(e) := M(e) \wedge (W_1(v) \circ M(e))$; (3)

<u>if</u> $(v,w) \in E$ <u>then</u>

$M(e') := \overline{M}((v,w)) \circ M(e)$;

where e' is the new edge created from u to w.

Any acceptable assignment AA to the reduced problem can be extended to an acceptable assignment to the original problem by defining

$AA(v) := \overline{M}(e)(AA(u))$;

where $\overline{M}(e)$ is evaluated using the $M(e)$ obtained in (3).

When a flow graph is reduced to the trivial graph we simply define

$$AA(n_0) := W_1(x_0);$$

and this acceptable assignment to the trivial graph can be extended backwards to give an acceptable assignment to the original information propagation problem.

Combining the above with Corollary 4.7 we get

Theorem 4.8:

Let $G = (N,E,n_0)$ be in $\mathcal{F}_{SG(k)}$ and let $IP = (G,D,M,x_0)$ be an information propagation problem on G. If D is fast then an acceptable assignment to IP can be found using $O(|E|)$ functional operations.

□

4.5 Reverse Information Propagation Problems on $\mathcal{F}_{SG(k)}$

To solve reverse information propagation problems on $\mathcal{F}_{SG(k)}$ we also use weighting technique to solve mimicked problems.

To handle the problem reduction construction in Lemma 3.5 efficiently we eliminate a self-loop at node v in the mimicked problem only when there are no more than $k+1$ edges leaving v. But then we need to handle R_2 reduction carefully. Another problem is that if AA is an acceptable assignment to a reduced problem on $R_2(G_r,e)$ than we want to extend AA to an acceptable assignment to the original problem efficiently. According to the construction of Lemma 3.6 and Corollary 3.7, if $e = (u,v)$, in order to extend AA and to define $AA(v)$ we need to take the meet of all $M((v,w))(AA(w))$ over w's such that $w \in Succ(v)$ before the R_2 reduction. But this is quite expensive if $|Succ(v)|$ is very large before the R_2 reduction. To overcome the above two difficulties, when R_2 applies to an edge (u,v), we collapse u into v instead of collapsing v into u.

Lemma 4.10:

Let RIP = (G_r, D, M, x_0) be a reverse information propagation problem, where $G_r = (N, E, n_0)$ is a flow graph with return node r. Let $e = (u,v)$, $v \neq n_0$, be the unique non-looping edge entering v and the only edge leaving u. Then a reverse information propagation problem RIP' = (G_r', D, M', x_0), $G_r' = (N-\{u\}, E', n)$, can be found such that an acceptable assignment to RIP can be obtained from an acceptable assignment to RIP' using one function application.

Proof: Let $W = \mathrm{Pred}(u)$. Define G_r' such that $E' = (E-\{(w,u) \mid w \in W\}) \cup \{(w,v) \mid w \in W\}$, $n = n_0$ if $u \neq n_0$, $n = v$ if $u = n_0$. M' is defined as follows. For all $w \in W-\{v\}$, $M'((w,v)) = M((w,u)) \circ M(e)$. Suppose $(v,u) \in E$ then $M'((v,v)) = M((v,u)) \circ M(e)$ if $(v,v) \notin E$, and $M'((v,v)) = M((v,v))$ $(M((v,u)) \circ M(e))$ if $(v,v) \in E$. For all $(x,y) \in E$, $y \neq u$, define $M'((x,y)) = M((x,y))$. Notice that all function compositions are "backwards" as we are dealing with reverse problems.

Let AA' be any acceptable assignment to RIP'. Define AA such that $AA(u) = f_e(AA(v))$, and $AA(z) = AA'(z)$ if $z \neq u$. We need to show that AA is acceptable to RIP.

First we show that any fixed point for RIP, if restricted to $N-\{u\}$, is also a fixed point for RIP'. So let FP be a fixed point for RIP. In G_r', if $(x,y) \in E'$, $x \notin W$, then $FP(x) \leq f_{(x,y)}(FP(y)) = f'_{(x,y)}(FP(y))$. If $w \in W-\{v\}$, or if

$w = v$ and $(v,v) \notin E$, then $FP(w) \le f_{(w,u)}(FP(u)) = f_{(w,u)}(f_{(u,v)}(FP(v)))$, hence $FP(w) \le f'_{(w,v)}(FP(v))$. If $v \in W$ and $(v,v) \in E$ then $FP(v) \le f_{(v,v)}(FP(v))$ and $FP(v) \le f_{(v,u)}(FP(u)) \le f_{(v,u)}(f_e(FP(v)))$, therefore $FP(v) \le f'_{(v,v)}(FP(v))$ is also true. Hence FP, when restricted to $N-\{u\}$, is a fixed point for RIP'.

It remains to show that AA is safe for RIP. Let $p = p_0,p_1,\cdots,p_j$ be a path from p_0 to the return node r in G_r. Let p' be the path obtained from p by replacing all consecutive occurences of w,u,v with w,u, where w is any node in W. Then p' is a path in G'_r if $p_0 \ne u$. Since $\tilde{f}'_{w,v} \le \tilde{f}'_{w,u,v}$ for all w in W, $AA(p_0) = AA'(p_0) \le \tilde{f}_{p'}(x_0) \le \tilde{f}_p(x_0)$ if $p_0 \ne u$. If $p_0 = u$ then $p_1 = v$, hence $AA(u) = f_e(AA(v)) \le f_e(\tilde{f}_{p_1,p_2,\cdots,p_j}(x_0)) = \tilde{f}_p(x_0)$. Therefore AA is also safe to RIP and is then acceptable to RIP.

□

The possible change of the entry node n_0 in the above lemma could be inconvenient if we solve both forward and reverse problems at the same time. But proper renaming of nodes can overcome this difficulty easily without maintaining two different flow graphs.

To implement the construction in Lemma 4.10 efficiently we use a weighting function W_2 to store information on the edges eliminated by R_2.

Define two weighting functions $W_1, W_2: N \to F \cup \{id_L\}$. Initially $W_1(n) = W_2(n) = id_L$ for all nodes n in N. Let

$\overline{RIP} = (\overline{G}_r, D, \overline{M}, x_0)$ be the mimicked problem. Again $\overline{G}_r$ is the same as G_r except for some self-loops which are already eliminated from G_r, and the information of these self-loops is stored in W_1. W_2 is used to store information on those edges eliminated by R_2. The invariant relation between RIP and $\overline{RIP}$ is that $\overline{M}((u,v)) = M(u,v)) \circ W_2(v)$ for all edges (u,v) in G_r. Notice that the weight $W_2(v)$ above is the weight associated with node v instead of the weight associated with node u in the forward problem case.

It is worth mentioning that, by construction of Algorithm 1, if there are no more than k+1 edges leaving a node n then all self-loops at n will be eliminated if Algorithm 1 is called at n. Therefore before we do reduction R_4 or R_5 on an edge (m,n), all information on self-loops at n has been stored in $W_1(n)$.

The actions taken by each rule of SG(k) are described below.

R_1: Let $G_r' = R_1(G_r, e)$ be defined for a self-loop at v.
<u>Action</u>: $W_1(v) := W_1(v) \cdot \overline{M}(e)$;

R_2: Let $G_r' = R_2(G_r, e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.
<u>Action</u>: <u>if</u> v = r <u>then</u> ; /* see the remark after R_5 actions */
<u>else</u> /* node u will be eliminated, see Lemma 4.10 */

```
begin
   M(e) := W_1(u)∘M(e);                          (4)
   W_2(v) := M̄(e)∘W_2(v);
   end
```

Again notice that the assignment in (4) has the side effect of modifying $\overline{M}(e)$.

If AA is an acceptable assignment to the reduced problem then we can extend AA to an acceptable assignment to the original problem by defining $AA(u) := \overline{M}(e)(AA(v))$; where $\overline{M}(e)$ is evaluated using the $M(e)$ obtained in (4).

R_3: Let $G_r' = R_3(G_r, e, e')$ be defined for edges e and e' from u to v.

Action: $M(e') := M(e') \wedge M(e)$;

R_4: Let $G_r' = R_4(G_r, e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.

```
Action: if v = r then ;  /* see the remark
                               after R_5 actions */
        else
          begin
            for each (v,w) in E do
               M((v,w)) := W_1(v)∘M((v,w));    (5)
            if (v,u) ∈ E then
               W_1(u) := W_1(u) ∧ (M̄(e)∘M̄((v,w)));
```

```
for each (v,w) ∈ E, w ≠ u do
   if (u,w) ∉ E then
      M((u,w)) := M̄(e)∘M((v,w));
   else
      M((u,w)) := M((u,w)) ∧
                        (M̄(e)∘M((v,w)));
end
```

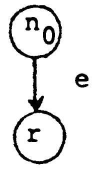
If AA is an acceptable assignment to the reduced problem then we can extend to an acceptable assignment to the original problem by defining $AA(v) := \bigwedge_{(v,w)\in E} (\bar{M}((v,w))(AA(w)))$; where each $\bar{M}((v,w))$ is evaluated using the $M((v,w))$ obtained in (5).

R_5: Let $G_r' = R_5(G_r,e)$ be defined for $e = (u,v)$, $u \neq v$, $v \neq n_0$.

Action:
```
if v = r then ; /* see remark below */
else /* let (v,w) be the unique edge
        leaving v */
  begin
     M((v,w)) := W_1(v)∘M((v,w));        (6)
     let e' be the new edge created from
     u to w ;
     M(e') := M̄(e)∘M((v,w));
  end
```

If AA is an acceptable assignment to the reduced problem then we can extend AA to an acceptable assignment to the original problem by defining $AA(v) := \bar{M}((v,w))(AA(w))$; where $\bar{M}((v,w))$ is evaluated using the $M((v,w))$ obtained in (6).

<u>Remark</u>: If $G_r \in \mathcal{F}_{SG(k)}$ and the return node r has a unique entering edge e then the graph obtained from G_r with r and e eliminated would still be in $\mathcal{F}_{SG(k)}$ and hence can still be reduced to the trivial graph. Therefore we may assume that the parser never tries to collapse the return node r. If $G_r \in \mathcal{F}_{SG(k)}$, then eventually we will get the graph



and then we simply do the following:

$M(e) := W_1(n_0) \circ M(e)$;

$AA(r) := x_0$;

$AA(n_0) := \bar{M}(e)(x_0)$;

Notice that, in the above actions, $M((x,y))$ is updated by $W_1(x)$ only when node x is about to be eliminated by the parser. Therefore the information stored in $W_1(x)$ will be handled properly. Lemma 4.10 enables us to handle W_1 on R_2 reductions efficiently.

Theorem 4.11:

Let RIP = (G_r, D, M, x_0) be a reverse information propagation problem. If $G_r = (N, E, n_0) \in \mathscr{F}_{SG(k)}$ and D is fast then an acceptable assignment to RIP can be found using $O(|E|)$ fuctional operations.

Proof: The action on each reduction described above uses only a bounded number of functional operations. Then the theorem follows from Theorem 4.6.

$\square$

5 Summary

Extending the results of Graham and Wegman we show how reverse information propagation problems on reducible graphs can be solved without introducing reverse transformations.

The arbitrary boundary formulation of flow problems is quite natural and could be useful in doing flow analysis on subroutines or in assertion search in program proving tools.

The weighting technique used in flow analysis on $\mathscr{F}_{SG(k)}$ enables us to reduce problems in a non-trivial fashion. With such weighting technique it is possible to design efficient flow analysis methods on different classes of flow graphs if reasonably good graph parsers are available.

ACKNOWLEDGEMENTS

I like to acknowledge Alan Demers for his critical comments on a draft of this paper and his suggestions on the presentation of algorithms.

Bibliography

[A] Allen, F.E., "Control Flow Analysis", ACM SIGPLAN Notices (newsletter) 5:7, July 1970, pp. 1-19.

[AM] Ashcroft, E. and Manna, Z., "Translating Program Schemas to While-Schemas", SIAM J. Comput., 4:2, June 1975, pp. 125-146.

[ASU] Aho, A.V., Sethi, R. and Ullman, J.D., "Code Optimization and Finite Church-Rosser Systems", TR-92, Computer Sci. Lab., Dept. of E. E., Princeton Univ., N.J., 1971.

[AU1] Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Prentice Hall, Englewood Cliffs, N.J., 1973.

[AU2] Aho, A.V. and Ullman, J.D., "Node Listings for Reducible Graphs", Proc. 7th Annl. ACM Symposium on Theory of Computing, May 1975, pp. 177-185.

[BJ] Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines, and Languages with Only Two Formulation Rules", CACM, 19:5, May 1966, pp. 366-371.

[C] Cocke, J., "Global Common Subexpression Elimination", ACM SIGPLAN Notices (newsletter), 5:7, July 1970, pp. 20-24.

[CS] Cocke, J. and Schwartz, J.T., Programming Languages and Their Compilers, Courant Institute, N.Y. Univ., New York, 1970.

[FKZ] Farrow, R., Kennedy, K. and Zucconi, L., "Graph Grammars and Global Program Data Flow Analysis", Proc. 17th IEEE Symposium on Foundations of Computer Science, Houston, Texas, Oct. 1976, pp. 42-56.

[FKU] Fong, A., Kam, J. and Ullman, J.D., "Applications of Lattice Algebra to Loop Optimization", Conference Record of the 2nd ACM Symposium on Principles of Programming Languages, Jan. 1975, pp. 1-9.

[GW] Graham, S.L. and Wegman, M., "A Fast and Usually Linear Algorithm for Global Flow Analysis", J. ACM, 23:1, Jan. 1976, pp. 172-202.

[HU1] Hecht, M.S. and Ullman, J.D., "Flow Graph Reducibility", SIAM J. Comput., 1:2, June 1972, pp. 188-202.

[HU2] Hecht, M.S. and Ullman, J.D., "Analysis of a Simple Algorithm for Global Flow Analysis", Proc. ACM Conf. on Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 207-217.

[HU3] Hecht, M.S. and Ullman, J.D., "Characterizations of Reducible Graphs", J. ACM, 21:3, July 1974, pp. 367-375.

[HU] Hopcroft, J.E. and Ullman, J.D., "An nlogn Algorithm for Reduction of Flow Graphs", Proc. 6th Annl. Princeton Conf. on Information Sciences and Systems, 1972.

[KU1] Kam, J.B. and Ullman, J.D., "Global Optimization and Iterative Algorithms", TR-146, Computer Sci. Lab., Dept. of E.E., Princeton Univ., Princeton, N.J., Jan. 1974.

[KU2] Kam, J.B. and Ullman, J.D., "Monotone Data Flow Analysis Frameworks", TR-169, Computer Sci. Lab., Dept. of E.E., Princeton Univ., Princeton, N.J., Jan. 1975.

[Ke] Kennedy, K., "Node Listings Applied to Data Flow Analysis", Conf. Record of 2nd ACM Symposium on Principles of Programming Languages, Jan. 1975, pp. 10-21.

[Ki] Kildall, G.A., "A Unified Approach to Global Program Optimization", Proc. ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, Mass., Oct, 1973, pp. 194-206.

[T] Tarjan, R.E., "Depth-First Search and Linear Graph Algorithms", SIAM J. Comput., 1:2, Sept. 1972, pp. 146-160.

[Se] Sethi, R., "Testing for Church-Rosser Property", JACM, 21,4, Oct. 1974.

[U] Ullman, J.D., "Fast Algorithm for the Elimination of Common Subexpressions", Acta Informatica, 2:3, Dec. 1973, pp. 191-123.

[Wi] Wirth, N., "On the Composition of Well-structured Programs", Computing Surveys, 6:4, Dec. 1974.