Techniques for Improving Grammar Flow Analysis

(*Extended Abstract*)

Martin JOURDAN & Didier PARIGOT
INRIA*

Abstract

Grammar Flow Analysis (GFA) is a computation framework that can be applied to a large number of problems expressed on context-free grammars. In this framework, as was done on programs with Data Flow Analysis, those problems are split into a general resolution procedure and a set of specific propagation functions. This paper presents a number of improvement techniques that act on the resolution procedure, and hence apply to every GFA problem: grammar partitioning, non-terminals static ordering, weak stability and semantic stability. Practical experiments using circularity tests for attribute grammars will show the benefit of these improvements. This paper is a shortened version of [**JoP**].

1 Introduction

In optimizing compilers, we have to statically infer run-time properties of programs, so that we can take advantage of this knowledge to generate better code. For instance, we may want to know whether the value of a given variable in an expression is statically predictable, so that we can use this constant value to generate better code (this is called *constant folding*). It turns out that many similar problems, when expressed formally, all reduce to solving a set of equations on the program graph; these equations relate pieces of information attached to immediately neighboring nodes, and edges are used to propagate this information. The basis of Data Flow Analysis (DFA) is that the method to solve these equations is independent from the semantics of the equations themselves, so that it is possible to devise a generic resolution procedure [**CoC**]. This procedure is parametrized by the specific equations of the problem at hand. Any improvement of this generic resolution procedure will hence benefit to every DFA problem. The DFA framework has been universally acknowledged, and most of the problems dealing with static analysis of programs are expressed in terms of DFA [**ASU**, **JoM**].

*Authors' address: INRIA, Domaine de Voluceau, Rocquencourt, BP 105, F-78153 LE CHESNAY Cedex, France. E-mail: {jourdan,parigot}@minos.inria.fr.

helm and Ulrich Möncke [**Mön 87**, **MW 82**] that transports that theory to the computation of properties of context-free grammars. GFA is performed on the grammar graph, whose nodes correspond either to non-terminals or productions, and whose edges are drawn according to the productions. Propagation functions are defined on productions, and carry some information that is attached to the non-terminals. According to the direction of these functions, we distinguish bottom-up and top-down GFA problems. In a bottom-up problem, the information attached to the LHS non-terminal of a production depends on the information attached to the non-terminals in the RHS. Examples of bottom-up problems are:

- computing the $First_k$ sets on a context-free grammar [**ASU**, **MW 82**];
- computing the synthesized dependency graphs in an attribute grammar (AG) and testing it for circularity [**DJL 88**, **DJL 84**, **JP 88a**, **Knu 68**, **LP 75**];
- pre-computing the sets of matching patterns in the construction of a bottom-up tree pattern matcher [**HoO**, **Mön 85**].

In a top-down GFA problem, the information attached to a non-terminal in the RHS of a production depends on the information attached to the LHS non-terminal, but generally also on the previously computed results of an associated bottom-up problem, which are attached to the non-terminals in the RHS [**Mön 87**]. Examples of these are:

- computing the $Follow_k$ sets on a context-free grammar, given the $First_k$ sets;
- computing the inherited dependency graphs in an AG, which depend on the synthesized ones;
- finding the sets of totally-ordered partitions in the construction of an l-ordered AG equivalent to a given non-circular AG [**EF 82**]; this computation also uses the synthesized dependency graphs.

Although bottom-up and top-down GFA problems are not symmetric in the same way as e.g. forward and backward DFA problems, they are solved by similar generic procedures.

In some cases, the complexity of a GFA problem can be reduced by computing only approximations of the exact solution [**Mön 87**]. This subject will not be addressed in this abstract, but it is in the full report [**JoP**].

As can be seen from the list of examples given above, GFA is a very general and useful technique for every kind of language processors. Thus, every improvement of the general resolution procedure will benefit to the whole broad domain of GFA. The issue of how to efficiently implement the general GFA resolution procedure was only briefly touched in the original papers [**Mön 87**]. Conversely, much work has been done to efficiently solve a particular GFA problem, namely testing an AG for non-circularity. The purpose of this paper is to exhibit a number of more or less well-known techniques, originally devised for that specific purpose, that actually apply to the whole domain of GFA and can improve the resolution of *every* GFA problem. These techniques are:

mars according to the "derives from" relation, and the subgrammars are processed in an order derived from the quotient relation;

- *non-terminals static ordering* [**JP 88a**, **Par 85**], in which the same "derives from" relation is used to derive a near-optimal order for processing the non-terminals;
- *weak stability* [**DJL 84**], which allows to skip the processing of non-terminals and productions that are known to derive only terminal trees;
- *semantic stability* [**Chea**, **JP 88a**, **Par 85**], which takes into account the "age" of each piece of information to avoid redundant computations.

The rest of the paper is organized as follows. Section 2 will present in an informal way the theory of Grammar Flow Analysis; in particular, a naive resolution algorithm will be given. Sections 3 to 6 will each be devoted to one of the four improvement techniques listed above, and the subsequent section discusses their combination. Section 8 briefly presents the results of a practical experiment using non-circularity tests for AGs as the test case, together with a short discussion thereof. The paper ends with some concluding remarks and the list of references. More details, examples and results can be found in [**JoP**].

The improvements will be described in such a way that they are readily applicable to bottom-up GFA problems. Their transposition for top-down problems will generally not be detailed, but it is an easy exercise left to the interested reader (see also [**JoP**]).

2 Bases of Grammar Flow Analysis

This section presents only briefly and informally the basic notions, data structures and algorithms involved in GFA. For a more comprehensive discussion see Möncke [**Mön 87**]. The following formulation is borrowed and adapted from Möncke & Wilhelm [**MW 82**].

2.1 The grammar graph

Let $G = \langle N, T, P, Z \rangle$ be a context-free grammar, with N the set of non-terminals, T the set of terminals—which are irrelevant to GFA[1]—, P the set of productions and Z the start symbol. Each production will be of the form $p : X_0 \to X_1 X_2 \cdots X_{n_p}$, where n_p is the number of non-terminals in the RHS of p and the terminals are omitted. The occurrences of non-terminals in a production are numbered from left to right, $p[0] = X_0$ being the LHS one, and $p[i] = X_i$, $0 < i \leq n_p$, being the RHS ones.

The grammar graph is a directed graph $G = \langle V, E \rangle$ with set of vertices $V = N \cup P$ and set of edges E defined as follows:

$$\begin{aligned} \forall p \in P, \forall X \in N, (p, X) \in E &\iff X = p[0] \\ \forall p \in P, \forall X \in N, (X, p) \in E &\iff \exists i, 0 < i \leq n_p, X = p[i] \end{aligned}$$

[1] They are indeed irrelevant to the GFA framework and resolution procedure, even if they are relevant to a particular problem, e.g. for $First_k$ and $Follow_k$.

A GFA problem is a triple $\langle \mathbf{L}, (\phi_p)_{p\in P}, (\psi_X)_{X\in N}\rangle$. The first component, $\mathbf{L}$, is the space of flow information; pieces of information are attached to non-terminals as $L_X \in \mathbf{L}$.[2]

The second component, $(\phi_p)_{p\in P}$, is a family of *information propagation functions*. For a bottom-up problem, and for each production p, there exists one propagation function $\phi_{p,0} : \mathbf{L}^{n_p} \mapsto \mathbf{L}$, which maps a tuple of elements of flow information $(L_{p[i]})_{0<i\leq n_p}$ on the non-terminals in the RHS of production p into an element of information to be attached to the LHS non-terminal. For a top-down problem, and for each production p, there exists a list of propagation functions $\phi_{p,i} : \mathbf{L} \mapsto \mathbf{L}$ $(0 < i \leq n_p)$, which map an element of flow information on the LHS non-terminal of p into an element of information for $p[i]$, $0 < i \leq n_p$. Each of the $\phi_{p,0}$ and $\phi_{p,i}$ may use other information defined on the grammar, but assuming not to depend on the problem. In addition, in a top-down problem, the $\phi_{p,i}$ may use information on all the non-terminals of p that was previously computed for an associated bottom-up problem.

Lastly, $(\psi_X)_{X\in N}$ is a family of *information combination functions*, $\psi_X : \mathbf{L}^{|P_X|} \mapsto \mathbf{L}$ where

$$P_X = \begin{cases} \{p \in P \mid p[0] = X\} & \text{for a bottom-up problem} \\ \{p \in P \mid \exists i, 0 < i \leq n_p, p[i] = X\} & \text{for a top-down problem}[3] \end{cases}$$

P_X is thus the set of productions in which the information attached to X is to be computed, according to the direction of the problem at hand; then, ψ_X combines the elements of information computed on each of these productions into a single element to be attached to X. Note that the ψ_X must be commutative for the problem to be well-defined.

2.3 The solution of a GFA problem

The solution on grammar G of the DFA problem $\langle \mathbf{L}, (\phi_p)_{p\in P}, (\psi_X)_{X\in N}\rangle$ is a set $\{L_X\}_{X\in N}$, where $L_X \in \mathbf{L}$ for each X in N, of elements of information attached to each non-terminal, verifying one of the following equations:[4]

- for a bottom-up problem:

$$\forall X \in N, L_X = \psi_X[\phi_{p,0}[L_{p[i]}]_{0<i\leq n_p}]_{p\in P_X} \tag{1}$$

- for a top-down problem:

$$\forall X \in N, L_X = \psi_X[\phi_{p,i}(L_{p[0]})]_{p\in P_X, p[i]=X} \tag{2}$$

These equations express that the solution is a combined fixed point of a set of functions, and can thus be computed by iteration over the grammar graph.

---

[2] For some problems there is actually one information space $\mathbf{L}_X$ per non-terminal X, but this does not change much the formulation.

[3] In that case, P_X is really a multiset since a same non-terminal may appear more than once in the RHS of some production.

[4] In these two equations, the notation $f[x_j]_{j\in S}$ stands for $f(x_{j_1}, x_{j_2}, \ldots, x_{j_k})$ if $S = \langle j_1, j_2, \ldots, j_k\rangle$.

```
foreach X ∈ N do L_X ← ⊥ endfor;
repeat convergence ← true;
    foreach p ∈ P do
        let X = p[0];
        old ← L_X;
        L_X ← ψ'_X(φ_{p,0}(L_{p[1]}, L_{p[2]}, ..., L_{p[n_p]}), L_X)
        if L_X ≠ old then
            convergence ← false
        endif
    endfor
until convergence.
```

Figure 1: Naive GFA resolution algorithm

2.4 A naive resolution algorithm

In this section we present an algorithm to solve bottom-up DFA problems (the adaptation to top-down ones is obvious; it can be found in [**JoP**]). We assume that the information space **L** is partially ordered with a bottom element $\perp$ and that a least fixed point is sought. We also assume that the combination functions ψ_X are *incremental*, i.e.,

$$\psi_X(\gamma_1, \gamma_2, \ldots, \gamma_{|P_X|}) = \psi'_X(\gamma_1, \psi'_X(\gamma_2, \cdots \psi'_X(\gamma_{|P_X|}, \perp)\cdots))$$

for some function ψ'_X. This condition is not absolutely necessary but we assume it holds to make the algorithm simpler. Also, all of these assumptions are verified by each of the GFA problems listed in section 1, because in those cases the information space is a set and the combination functions are the union function, which is incremental. The naive resolution algorithm is presented in Fig. 1.[5] It is naive in the sense that it is a simple derivation of the fixed point equation (1) and there exists no special order to process the non-terminals and the productions, i.e., the grammar graph is visited in a totally random order. The purpose of the improvements to be presented in the following sections is to determine a near-optimal order and to eliminate redundant computations, such that information is propagated faster along the graph and convergence is reached faster.

As a special but quite common case[6] we derive a version of this algorithm for problems in which the information space is structured as a set of sets, i.e., each L_X is itself a set, and in which the combination functions are the set-theoretic union. In this case, the propagation functions are more easily expressed in terms of individual elements of these sets. We hence assume that, for each production p, there exists an auxiliary function $\phi'_{p,0} : L_{p[1]} \times L_{p[2]} \times \cdots \times L_{p[n_p]} \mapsto L_{p[0]}$, which maps a tuple of information elements on the RHS non-terminals into an element of information for the LHS one. The whole propagation function is then defined as the set-theoretic generalization of $\phi'_{p,0}$, that is:

$$\phi_{p,0}(L_{p[1]}, L_{p[2]}, \ldots, L_{p[n_p]}) = \{\phi'_{p,0}(\gamma_1, \gamma_2, \ldots, \gamma_{n_p}) \mid \gamma_i \in L_{p[i]}, 0 < i \leq n_p\}$$

[5] The names of the algorithms are chosen to recall those of the practical experiment of section 8, which are themselves a subset of those used in the full report [**JoP**].

[6] As an evidence, Möncke considers only this case in his reference paper on GFA [**Mön 87**]. This is the case for instance of the $First_k$ and synthesized dependency graphs problems.

```
init:          foreach X ∈ N do L_X ← ∅ endfor;
iterate:       repeat convergence ← true;
select-nt:         foreach X ∈ N do
select-prod:           foreach p ∈ P_X do
combine:                   foreach γ_1 ∈ L_{p[1]}, ..., γ_{n_p} ∈ L_{p[n_p]} do
compute:                       γ_0 ← φ'_{p,0}(γ_1, ..., γ_{n_p});
test:                          if γ_0 ∉ L_X then
increase:                          L_X ← L_X ∪ {γ_0};
                                   convergence ← false
                               endif
                           endfor
                       endfor
                   endfor
               until convergence.
```

Figure 2: Naive algorithm for set-based problems

The resulting algorithm is presented in Fig. 2.[7] Our improvements will be expressed on this latter algorithm, with the help of the labels attached to some of the statements. The transposition to the more general algorithm will not be given. The purpose of the improvements will be to reduce as much as possible the number of "compute" and "test" steps, which are assumed to be expensive[8] and hence dominate the running time.

3 Grammar Partitioning

As can be seen from the definition in section 2.2, the information flows exclusively along the edges of the grammar graph. It is thus natural, in order to have this information propagate faster, to take into account the structure of this graph, rather than picking non-terminals and productions at random. The first idea that comes to the mind is hence to partition the grammar graph into strongly connected components and process those components in the order defined by the quotient relation. Note that this technique was used right from the beginning in DFA, where the program is decomposed into a control flow graph of basic blocks, and blocks are processed from inner to outer [**CoC**, **JoM**].

The order in which productions are processed depends on the order in which non-terminals are processed; more precisely, each time a non-terminal is processed, all the productions of which it is the LHS symbol are processed (line labeled "select-prod" in Algorithm A_3). This must be so in order to ensure that when a non-terminal is referenced in the RHS of a production, the information attached to it is as complete as possible. We hence require, when computing the strongly connected components of the grammar graph, that every production vertex be in the same component as the vertex corresponding to

[7] Note that we already have decreased the random factor by subordinating the choice of the production to process to the choice of its LHS non-terminal.

[8] In non-circularity tests for AGs for instance, these steps involve the computation of a transitive closure and testing the membership of a (sometimes big) graph in a set of graphs.

build the graphs Γ and Γ_0 from G;
apply algorithms for Sorting, Entry Points Priority, Other Symbols Priority;[a]
for $i \leftarrow 1$ **to** K **do**
    apply Algorithm A_0 or A_3 to subgrammar B_i;
    discard or store away the information attached to non-terminals of priority i
endfor.

[a] If necessary, more details on these algorithms can be found in the original works [**Cheb**, **DJL 84**] and in [**JoP**].

Figure 3: GFA resolution algorithm using partitioning

its LHS non-terminal. We thus define the following auxiliary relation:

$$\forall X, Y \in N, X \ \Gamma \ Y \iff \exists p \in P, \exists i, 0 < i \leq n_p, X = p[i] \wedge Y = p[0]$$

Γ is the "derives from" relation on non-terminals, and its graph is the same as the grammar graph in which the production vertices are merged with their LHS non-terminal vertices.

The strongly connected components of Γ define the subgrammars of G we are interested in. Moreover, if Γ_0 is the quotient relation associated with Γ, Γ_0 defines a partial order to be used for processing these subgrammars. Since by construction Γ_0 is acyclic, a simple topological sort will derive from it a total order. We hence denote the subgrammars as $B_1, B_2, \ldots, B_K$, where K is the total number of strongly connected components of G, so that B_i will be processed before B_{i+1} (note: $Z \in B_K$).

For reasons to be explained later, it is interesting to distinguish *entry points* and *output points*[9] of these subgrammars B:

$$\begin{aligned} \forall X \in B, X \in EP(B) &\iff \exists Y \notin B, X \ \Gamma \ Y \\ \forall X \in B, X \in OP(B) &\iff \exists Y \notin B, Y \ \Gamma \ X \vee \nexists Y \in N, Y \ \Gamma \ X \end{aligned}$$

Entry points of a subgrammar are thus those non-terminals that derive from non-terminals in subgrammars to be processed later, and output points are those non-terminals that derive into non-terminals of subgrammars that have already been processed or derive only terminal productions. Output points will not be used before next section.

The order for processing subgrammars is used to assign a *priority* to each non-terminal, that is the rank of the stage after which that non-terminal will no longer be referenced. An entry point of a subgrammar will be assigned the priority (rank) of the last subgrammar that references it; other non-terminals will be assigned the priority of their own subgrammar. This notion of priority is useful to reduce the space consumption of the algorithm, because after stage i we can either store in secondary memory or discard completely—if the problem at hand allows it—the information attached to non-terminals of priority i. As will be shown in section 8, this technique is very effective. The resulting algorithm is presented in Fig. 3.

[9] As can be seen from the definitions, the names of "entry" and "output" points are hence rather misleading, because they are inconsistent with the direction of the edges in the grammar graph. We however keep them for the sake of compatibility with previous works [**DJL 84**, **JP 88a**].

entry and output points. The strongly connected components will then be the same, but the processing order will be different; in that case, Z belongs to B_1.

4 Non-Terminals Static Ordering

Grammar partitioning uses the "derives from" relation to exhibit subgrammars and derive an optimal order for their processing; it however leaves unspecified the order for processing non-terminals and productions *inside* each subgrammar. The purpose of this section is to establish a near-optimal order for processing non-terminals. Since by definition each subgrammar is a strongly connected component for the "derives from" relation, the latter is not sufficient to define the processing order, i.e., it is impossible to find a total order compatible with Γ. Heuristics will help make the choice.

The first heuristic will be to order the non-terminals of a given subgrammar from output points to entry points; this order is indeed the "closest" to relation Γ. The core of the ordering algorithm is thus a topological sort based on relation Γ and starting with output points. The sort is however modified to solve three difficulties:

1. Since we want output points to be the starting point of the topological sort, we must first delete from $\Gamma|_B$ every edge whose tail (sink) is an output point, except those whose head (source) is also an output point. Note that each output point is the tail of at least one such "back edge", otherwise it would not belong to the subgrammar.

2. We then consider only output points and edges connecting two output points, and topologically order this subgraph. If this is not possible because of a cycle, we make an arbitrary choice in the cycle.

3. We then proceed to topologically sort the rest of the subgrammar. If a yet unbroken cycle shows up, we make an arbitrary choice in the cycle.

The resulting algorithm is completely detailed in Jourdan & Parigot [**JP 88a**]. To integrate it in the complete GFA algorithm, we need to modify A_3 as follows:

- add a call to the non-terminals static ordering algorithm in line "init";

- use this order in line "select-nt".

As for the order in which to process productions in each P_X (line "select-prod"), none is better than any other because all the non-terminals in the RHS have been processed earlier in the current iteration (apart from back edges, but this is taken into account by the global convergence flag). We can only note that directly recursive productions should be processed after the others. The modified algorithm is not shown and left to the reader.

For top-down GFA, the basic idea is the same except that we use the inverse of Γ and that the roles of entry and output points are reversed. In this case the resulting order is not necessarily the inverse of the bottom-up order.

The advantages of this ordering is that, as expected, information propagates faster, as shown by the important reduction of the number of iterations (line "iterate") needed to reach convergence (see section 8).

Note that this technique is similar to an improvement of DFA called "reasonable node listing", however its efficiency is particularly important for GFA.

Weak stability allows to skip recomputations of relations for non-terminals and/or productions that are known to generate only finite trees of height $\leq i$ and hence become "stable" after iteration i (line "iterate" in A_3). This is done as follows:

1. at the beginning of the algorithm (line "init"), no non-terminal or production is weakly stable;
2. after each iteration (end of loop "iterate"):
   (a) mark as weakly stable those productions that have only weakly stable non-terminals in their RHS;[10]
   (b) mark as weakly stable those non-terminals that derive only weakly stable productions;
3. in the course of an iteration, skip weakly stable non-terminals (line "select-nt") and productions (line "select-prod").

When used alone, this technique is not very efficient because it fails on every recursive non-terminal. However it is not totally useless, as shown in sections 7 and 8. An incremental variant of this technique is described in the full report [**JoP**].

6 Semantic Stability

This improvement, called semantic stability,[11] aims at saving execution time by drastically reducing the number of redundant computations. It is based on the observation that, if the information used in the basic step of the algorithm (inner loop of A_0 or line labeled "compute" in A_3) is "old" enough to have been processed during an earlier iteration, then it is useless to process it again in the current iteration, which means that we may skip the basic step. Of course, doing so is correct only if the information space is partially ordered and if the propagation and combination functions are monotonic, i.e., if skipping a basic step with "old" parameters does not lose any information because that information is already present in the "old" value of L_X, where X is the non-terminal at hand. Note that the formulation of algorithm A_0 should be slightly modified to make this apparent. Note also that these conditions are not very constraining; in particular they are verified in every set-based GFA problem (see section 2.4).

Thus the basic idea of this improvement is to associate a "time-stamp" with each piece of information and run the basic step only if at least one such piece of information appearing in the RHS of the computation step is not "old" enough to have already been processed. In the special case of set-based GFA problems, this time-stamp can even be associated with each element of the sets attached to non-terminals, and tested in the combinations in which this element appears (line labeled "combine" in A_3). In that case, the new algorithm is as presented in Fig. 4.

[10] If partitioning is also used, non-terminals in already processed subgrammars are considered as weakly stable.

[11] because, as opposed to the previous ones, it does not take into account the syntactic information represented by the grammar graph.

```
init:          foreach X ∈ N do L_X ← ∅ endfor;
               it ← 0;
iterate:       repeat convergence ← true;
                   it ← it + 1;
select-nt:         foreach X ∈ N do
select-prod:           foreach p ∈ P_X do
combine:                   foreach γ_1 ∈ L_{p[1]}, ..., γ_{n_p} ∈ L_{p[n_p]} do
check-time:                    if ∃i, 0 < i ≤ n_p, time(γ_i) ≥ max-time(it, p) then
compute:                           γ_0 ← φ'_{p,0}(γ_1, ..., γ_{n_p});
test:                              if γ_0 ∉ L_X then
increase:                              time(γ_0) ← ⟨it, p⟩;
                                       L_X ← L_X ∪ {γ_0};
                                       convergence ← false
                                   endif
                               endif
                           endfor
                       endfor
                   endfor
               until convergence.
```

Figure 4: GFA resolution algorithm using semantic stability

The correctness of this algorithm strongly depends on the choice of the function "max-time". Assuming that the productions are numbered, we define the time-stamp of an element of information as the time when it was created, that is the ordered pair $\langle it, p \rangle$ where it is the number of the iteration and p the number of the production being processed. These pairs are lexicographically ordered.

A weak version of "max-time" is

$$\text{max-time}(it, p) = \langle it - 2, 0 \rangle$$

in which we forget about the production. Indeed, any element of information created during iteration $it - 2$ has been processed at latest during iteration $it - 1$ and processing it again during iteration it or later is redundant.

To achieve a finer condition, we must require that non-terminals and productions are always processed in the same order during each iteration (lines "select-nt" and "select-prod"). In this case, we can assume that the number of a production is the rank at which it is processed during an iteration, and the definition of "max-time" can be refined as:

$$\text{max-time}(it, p) = \begin{cases} \langle it - 1, p - 1 \rangle & \text{if } p \neq 1 \\ \langle it - 2, |P| \rangle & \text{if } p = 1 \end{cases}$$

because any element of information created at $\langle it, p \rangle$ can be used immediately during the rest of the iteration (i.e., at $\langle it, p' \rangle$ with $p' > p$) or during the first part of the next one (i.e., at $\langle it + 1, p' \rangle$ with $p' \leq p$) and thus becomes redundant after $\langle it + 1, p \rangle$.

The figures of section 8 will show that this simple idea is very effective.

All of the four previously presented improvements can be used alone or in combination.

The most effective technique is grammar partitioning. This is because it allows to tailor the number of iterations in each subgrammar to the "semantic complexity" of the GFA problem for this subgrammar, rather than having to process the whole grammar during a number of iterations that is in any case at least equal to the maximum number of iterations in each subgrammar. Furthermore it is the only improvement that reduces space consumption, because the results are produced and can be disposed of incrementally rather than in a single burst at the end.

Non-terminals static ordering reduces the number of iterations by ensuring as much as possible that all uses of some information occur after the time when it is computed. It can be used independently from partitioning by applying it to the whole grammar, the only entry point being the start symbol and the output points being the non-terminal deriving only terminal productions (if the grammar is assumed to be reduced). However its combination with partitioning is more effective since the advantages of the latter are fully retained: time is reduced because partitioning ensures that information on a subgrammar is completely computed before being used in other subgrammars, and space is reduced because we can forget about information attached to non-terminals that will no longer be referenced.

When used alone, weak stability is not very effective because it cannot act on recursive non-terminals. However, grammar partitioning offers more opportunities for weak stability, because non-terminals outside the subgrammar at hand are considered as weakly stable, even if they are recursive in their own subgrammar.

Syntax-based techniques reduce the number of basic computation steps by using static information (the shape of the grammar graph), whereas semantic stability aims at the same goal by using dynamic time-stamping techniques. One could thus think that the latter is subsumed by the former because, since information propagates faster, most of the basic steps are actually useful and cannot be eliminated by semantic stability. As the figures of section 8 will show, this is partly true, but it appears that the combination of both techniques is more effective than each of them taken separately; this is especially true when the semantic complexity of some (sub-) grammar is high, i.e., when the sets L_X contain many different elements.

8 Practical Results

To illustrate in practice the effects of our GFA improvement techniques, we have chosen the non-circularity test for AGs, because they fully illustrate the power of the GFA improvements.

8.1 Rough results

We have implemented several versions of the non-circularity test as part of the FNC-2 AG processing system [**JP 89**]. The various algorithms differ in which improvement technique(s) they use; they are detailed in Table 1. All of them use *covering* [**DJL 84**, **LP**

A_3				
A_4	•			
A_5		•		
A_6	•	•		
A_7			•	
A_8	•	•	•	
A_9				•
A_{10}	•	•	•	•

Table 1: Features of the various algorithms

	simproc	*asm*	*pl1_c*	*pascal*	*simula*
nt	10	58	139	115	126
pr	21	216	376	216	244
$rhs_{\max}$	4	2	4	6	6
rhs_{ave}	1.14	0.46	1.07	1.18	1.13
$att_{\max}$	8	10	16	15	17
att_{ave}	4.60	3.62	8.90	7.39	7.28
K	5	57	92	58	58
$d_{\max}$	1	2	3	4	4
d_{ave}	1	1.07	1.04	1.20	1.10
tcl_{circ}	21	267	389	457	268

Table 2: Characteristics of the example AGs

75], an approximation that strongly reduces the practical complexity of the non-circularity test.

These algorithms were tried on five practical AGs of increasing complexity. All of these AGs are non-circular. We will use the following notations:

- For each AG:

  nt number of non-terminals

  pr number of productions

  $rhs_{\max}$, rhs_{ave} maximum and average number of non-terminals in the RHS of any production

  $att_{\max}$, att_{ave} maximum and average number of attributes per non-terminal

  K number of syntactic equivalence classes (subgrammars)

  The characteristics of our five AGs are presented in Table 2.

- For each algorithm:

  $d_{\max}$, d_{ave} maximum and average number of graphs in any L_X at the end of the computation[12]

[12]Because of covering, some L_X's might be temporarily larger than that in the course of the computation, if one newly created graph happens to cover two or more old graphs.

tcl number of transitive closures (basic steps) computed

tcl_{circ} number of transitive closures computed for actually testing the non-circularity (see below)

Since the three figures $d_{\max}$, d_{ave} and tcl_{circ} characterize the results of the problem at hand, they have been "factored out" and also appear in Table 2.

- For algorithms not using partitioning:

  it number of iterations over the whole grammar

- For algorithms using partitioning:

  $it_{\max}$ maximum number of iterations over any subgrammar

  it_{ave} same in weighted average, i.e., with obvious notations, $(\sum_{j=1}^{K} it_j \times pr_j)/pr$

Statistics gathered from the execution of the various algorithms are presented in Table 3.

The tcl_{circ} figure is related to the way we actually test the non-circularity. Briefly said, we do it globally in a final pass—over either the whole grammar, when partitioning is not used, or each subgrammar—after all the synthesized dependency graphs have been computed, rather than incrementally on every graph. More details can be found in the full report [**JoP**]. Let us only note that tcl_{circ} is interesting on its own since, being the number of final graph combinations, it is a very good measure of the "semantic complexity" of the non-circularity problem for a given AG.

In the naive versions of the algorithm in which no special order is computed, the processing order (lines "select-nt" and "select-prod") is determined by the textual appearance in the grammar source file. Since practical grammars are generally presented in a top-down manner, using textual order is close to the worst case for bottom-up GFA problems!

When partitioning is used, the "useless" graphs (see section 3) are actually discarded rather than stored in secondary memory, and their space is reclaimed by a specialized garbage collector. The $R_{\max}$ figure is computed by this garbage collector.

8.2 Discussion

Algorithm A_4 uses only partitioning. The gains in time (i.e., in the number of basic steps executed) are already quite good, especially on AGs with simple syntactic structure (*asm*, *pl1_c*). This is because information propagates faster, as shown by the decrease of the number of iterations ($it_{\max}(A_4) \leq it(A_3)$ and $it_{\text{ave}}(A_4) \ll it(A_3)$). The most important advantage of grammar partitioning is however the gains in space it allows to achieve: compare the $R_{\max}$ figures for A_4 w.r.t. A_3.

When non-terminals static ordering is used alone (A_5), it also makes information propagate faster; this is proved by the strong decrease of the number of iterations w.r.t. A_3. However it has no influence on space consumption. Comparing the results for A_4 and A_5 gives interesting insights on the way grammar partitioning and non-terminals static ordering act when used separately. Grammar partitioning is more effective on

		simproc	*asm*	*pl1_c*	*pascal*	*simula*
A_3	$R_{\max}$	11	67	147	149	154
	it	7	8	15	21	19
	tcl	145	2,260	5,125	9,768	5,099
A_4	$R_{\max}$	5	21	30	57	80
	$it_{\max}$	4	3	13	21	14
	it_{ave}	3.10	1.12	4.31	7.16	8.29
	tcl	78	561	1,924	6,031	2,632
A_5	$R_{\max}$	11	64	146	147	182
	it	4	3	9	9	7
	tcl	101	1,064	3,645	4,288	2,376
A_6	$R_{\max}$	5	21	30	58	99
	$it_{\max}$	4	3	6	8	7
	it_{ave}	2.76	1.08	2.37	3.41	4.30
	tcl	73	556	1,304	2,746	1,756
A_7	$R_{\max}$	11	67	147	149	154
	it	7	8	15	21	19
	tcl	109	900	3,555	8,446	3,865
A_8	$R_{\max}$	5	21	30	58	99
	$it_{\max}$	4	3	6	8	7
	it_{ave}	2.76	1.08	2.37	3.41	4.30
	tcl	56	545	1,160	2,679	1,655
A_9	$R_{\max}$	11	67	147	149	154
	it	7	8	15	21	19
	tcl	57	588	1,165	1,942	1,628
A_{10}	$R_{\max}$	5	21	30	58	99
	$it_{\max}$	4	3	6	8	7
	it_{ave}	2.76	1.08	2.37	3.41	4.30
	tcl	49	540	915	1,661	1,246

Table 3: Execution statistics

iteration a large share of simple subgrammars. However, for more complicated examples with "big" subgrammars (*pascal*, *simula*), the bad influence of randomly processing the non-terminals inside each subgrammar shows up again, which explains that non-terminals static ordering is more effective in these cases. Anyway, the real winner is the combination of both techniques (A_6), which gives better (and generally much better) results than each of them taken separately: there is a real synergy between these two improvements. The gains in space achieved by grammar partitioning alone are retained in its combination with non-terminals static ordering.

Weak stability used alone (A_7) has a perceptible influence only because the bare algorithm (A_3) is really bad and needs a lot of iterations to reach convergence; this is especially true when the grammar has many terminal productions (*asm*). When weak stability is used in combination with grammar partitioning and non-terminals static ordering (A_8), this influence is however less important, but nevertheless not negligible.

When comparing A_8 with A_3, we can see that the combination of all syntax-based improvements reaches its goal, which is to make information propagate faster, in a quite effective way. Furthermore space consumption is reduced by virtue of grammar partitioning.

Semantic stability used alone (A_9) is also a real winner: it performs roughly as well as the combination of syntax-based improvements (a little better for AGs with a high semantic complexity such as *pascal*, a little worse for AGs with a low syntactic complexity such as *asm*). However we must note that this is achieved at higher bookkeeping costs, and that semantic stability has no influence on space consumption (the $R_{\max}$ figures are the same for A_9 as for A_3).

Combining semantic stability with the syntax-based techniques is always profitable (compare A_{10} with A_8 on one hand and A_9 on the other hand). This means that neither the dynamic approach (semantic stability) nor the static one (syntax-based techniques) subsumes the other and, rather, that they complement each other.

All in all, the four techniques we have presented are quite effective in reducing the practical complexity of GFA problems. Comparing the results for A_{10} w.r.t. A_3 suffices to be convinced of that. To conclude the discussion on the benefits of our improvements, let us examine the processing of a "simple" subgrammar: we need at least one iteration to construct the graphs, another one to check that convergence is reached (this one may be suppressed by weak stability) and, in our scheme, a final one for actually testing the non-circularity. Thus, for a "simple" grammar, the total number of transitive closures should be of the order of three times the tcl_{circ} figure. With all our improvements we reach this behavior for our three simplest AGs, and we reach close to it for the two other ones. This is, in our opinion, an indication (but *not* a formal proof) that our improvements reach close to optimality.

9 Conclusion

This paper showed that Grammar Flow Analysis is an interesting computation framework for many classical problems dealing with grammars. It also showed that simple techniques could drastically improve the resolution of those problems and were able to reach close

that were believed to be intractable because of their computational cost—e.g. circularity tests for AGs—now become quite feasible in practice by using the improvement techniques we described. Furthermore, these techniques are so effective that we have used them in many stages in the FNC-2 system (SNC test, DNC test, OAG test, ...), which allows to cascade these stages while keeping a reasonable execution time (less than 80 seconds to build a complete optimized evaluator for our *pascal* AG).

While the basic techniques for implementing GFA are now well understood, there stays to discover new application fields. We believe that GFA and its efficient implementation will give a new boost to the use of grammars as a basic tool of computer science.

References

[**ASU**] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986.

[**Chea**] K. S. Chebotar, private communication, 1985.

[**Cheb**] K. S. Chebotar, "Some Modifications of Knuth's Algorithm for Verifying Cyclicity of Attribute Grammars," *PCS* **7** (Jan. 1981), 58–61.

[**CoC**] Patrick Cousot & Rahdia Cousot, "Systematic Design of Program Analysis Frameworks," in *6th POPL*, Jan. 1979, 269–282.

[**DJL 88**] Pierre Deransart, Martin Jourdan & Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. #**323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[**DJL 84**] Pierre Deransart, Martin Jourdan & Bernard Lorho, "Speeding up Circularity Tests for Attribute Grammars," *Acta Inform.* **21** (Dec. 1984), 375–391.

[**EF 82**] Joost Engelfriet & Gilberto Filè, "Simple Multi-Visit Attribute Grammars," *J. Comput. System Sci.* **24** (June 1982), 283–314.

[**HoO**] Christoph M. Hoffmann & Michael J. O'Donnell, "Pattern Matching in Trees," *J. Assoc. Comput. Mach.* **29** (Jan. 1982), 68–95.

[**JoM**] Neil D. Jones & Steve S. Muchnick, eds., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[**JoP**] Martin Jourdan & Didier Parigot, "Techniques for Improving Grammar Flow Analysis," INRIA, report to appear, 1990.

[**JP 88a**] Martin Jourdan & Didier Parigot, "More on Speeding up Circularity Tests for Attribute Grammars," INRIA, rapport RR-828, Apr. 1988.

[**JP 89**] Martin Jourdan & Didier Parigot, *The FNC-2 System User's Guide and Reference Manual* release 0.4, INRIA, Feb. 1989, This manual is periodically updated.

[**Knu 68**] Donald E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2** (June 1968), 127–145, Correction: *Math. Systems Theory* **5**, 1 (Mar. 1971), 95–96.

[**LP 75**] Bernard Lorho & Claude Pair, "Algorithms for Checking Consistency of Attribute Grammars," in *Proving and Improving Programs*, Gérard Huet & Gilles Kahn, eds., INRIA, July 1975, 29–54.

bäume: Komponenten des Systems und Mechanismen der Generierung," Univ. des Saarlandes, Diplomarbeit, Saarbrücken, 1985.

[**Mön 87**] Ulrich Möncke, "Grammar Flow Analysis," Univ. des Saarlandes, ESPRIT PROSPECTRA Project report S.1.3.–R-2.2, Saarbrücken, Mar. 1986, revised Jan. 1987, To appear in *TOPLAS*.

[**MW 82**] Ulrich Möncke & Reinhard Wilhelm, "Iterative Algorithms on Grammar Graphs," in *Conf. on Graphtheoretic Concepts in Computer Science (WG'82)*, H. J. Schneider & Herbert Göttler, eds., Hanser Verlag, München, June 1982, 177–194.

[**Par 85**] Didier Parigot, "Un système interactif de trace des circularités dans une grammaire attribuée et optimisation du test de circularité," Univ. de Paris-Sud, rapport de DEA, Orsay, Sept. 1985.