Speeding up circularity tests for attribute grammars

Pierre Deransart, Martin Jourdan, Bernard Lorho

INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports de Recherche

N° 211

SPEEDING UP CIRCULARITY TESTS FOR ATTRIBUTE GRAMMARS

Pierre DERANSART
Martin JOURDAN
Bernard LORHO

Mai 1983

SPEEDING UP CIRCULARITY TESTS

FOR ATTRIBUTE GRAMMARS

Pierre DERANSART

Martin JOURDAN

Bernard LORHO

INRIA

Domaine de Voluceau - Rocquencourt

BP 105

78153 LE CHESNAY

Recherche menée dans le cadre du projet du GRECO de PROGRAMMATION "ATTRISEM" .

Résumé :

Nous présentons deux améliorations à appliquer aux algorithmes testant la circularité des grammaires attribuées. La première, introduite originellement par Chebotar [Che81], établit un ordre optimal pour le choix des productions à traiter et élimine à chaque pas les graphes qui ne sont plus nécessaires pour les étapes suivantes de l'algorithme ; il demande donc moins de temps de calcul et d'espace mémoire. La seconde permet d'éviter des recalculs sur les arbres terminaux, économisant ainsi du temps. Ces deux méthodes peuvent être utilisées séparément ou ensemble pour accélérer les tests de circularité. Nous discutons aussi la complexité pratique de ces tests.

Abstract :

We present two improvements to be applied to algorithms testing the circularity of attribute grammars. The first one, originally introduced by Chebotar [Che81], establishes an optimal order for selection of productions and eliminates at each step those graphs that are unnecessary for subsequent stages of the algorithm, thus requiring less time and space. The second one skips recomputations on terminal trees, thus saving time. These two methods can be used alone or together to speed up circularity tests. We also discuss the practical complexity of circularity tests.

Mots clés : Grammaires attribuées - circularité - tests de circularité - graphes - algorithmes - complexité - arbres de dérivation.

Keywords : Attribute grammars - circularity - circularity tests - graphs - algorithms - complexity - derivation trees.

1 - INTRODUCTION

Since Knuth's original paper [Knu68] about attribute grammars, one of the most debated topics was testing attribute grammars for circularity : indeed, using a circular attribute grammar leads no useful results and even denotes a flaw in its conception. One of the main problems is that the complexity of this test is intrisically exponential [Jaz81], and this might prevent a practical use of attribute grammars, in spite of their attractive advantages to describe the semantics and/or translations of programming languages.

In this paper we present two improvements (plus one originally introduced in [LP75]) to be applied to algorithms testing the circularity of attribute grammars, and test their real efficiency. The first one, called "covering", discards from the set of graphs which is attached to each non-terminal during the test, those graphs that are covered by others from the same set. The second one, first presented by K.S. Chebotar [Che81], uses the syntactic dependencies between non-terminals to split the original grammar into sub-grammars, and applies the test to each of them subsequently. The last one skips recomputations of relations on productions and/or non-terminals which generate only finite trees. These three methods can be used separately or together.

Since these methods do not alter the exponential complexity of the resulting algorithms, we then discuss the practical complexity of these tests, showing, along with Räihä and Saarinen [RS82], that for practical examples the exponential factor seems to be bounded up.

In section 2, we state basic notations and definitions to be used throughout the paper. In section 3, we present the general algorithm and the covering. In section 4 and 5 we introduce Chebotar's algorithm and the notion of weak stability. In section 6 we combine all the methods. In section 7 we discuss the practical complexity of these tests, and in section 8, we state some practical results.

2 - NOTATIONS AND DEFINITIONS

Let $G = \{N,T,P,Z\}$ be a context-free grammar, as usual. Each production $p \in P$ has the form :

$$p : X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$$

To get an attribute grammar AG, we associate to each non-terminal symbol $X \in N$ two finite sets of symbols called attributes, $I(X)$ and $S(X)$ (for inherited and synthesized attributes) such that $I(X) \cap S(X) = \emptyset$. We note $A(X) = I(X) \cup S(X)$, $I = \bigcup_{X \in N} I(X)$ and similarly for S and A.

We denote by (a,i) the attribute a attached to the symbol X_i (attribute occurence).

The terminal symbols have no attributes. The start symbol Z has no inherited attributes.

To each production p, is associated a set of semantic rules defining the computation of the elements of $S(X_0)$ and $I(X_j)$ for $1 \leq j \leq n_p$, in term of the elements of $A(X_i)$, for $0 \leq i \leq n_p$:

$$(a,i) = f_{p,a,i}\ ((a_1,i_1),\ldots,(a_k,i_k)) \qquad \begin{array}{r} i = 0 \text{ and } a \in S(X_0) \\ \text{or } 1 \leq i \leq n_p \text{ and } a \in I(X_0) \end{array}$$

The grammar G augmented with these attributes and semantic rules constitutes an attribute grammar AG.

Let us denote by $AS(p) = \{(a,k) \ / \ a \in A(X_i),\ 0 \leq i \leq n_p\}$ the set of all attribute occurences of production p.

The set $DS(p,a,i) \subset AS(p)$ gathers all the attributes occurences of the production p which are arguments of the function computing (a,i).

Let us define on AS(p) a binary relation DO(p) exhibiting the dependencies between the attribute occurences of the production p :

$$(a,k)\ DO(p)\ (b,\ell) \Leftrightarrow (b,\ell) \in DS(p,a,k)$$

The dependency graph of the production p is the graph of the relation DO(p). $DO^*(p)$ denotes the transitive closure of the relation DO(p).

Let t be a derivation tree for G. The compound dependency graph of t, DG(t), is the graph obtained by pasting together the graphs DO(p) according to the productions labelling the nodes of t. This graph may be circular.

The attribute grammar AG is non-circular iff DG(t) is not circular for any derivation tree t of G.

If DG(t) is not circular, then the attribute instances of t can be evaluated, for instance in the order inverse of the one induced by DG(t) (which is a partial order) (see e.g. [CF82]).

Knuth [Knu71] shows that this dynamic property can be tested statically.

3 - ALGORITHMS FOR TESTING CIRCULARITY

3.1 - General algorithm [Knu71] :

For each non-terminal symbol X, we build a set of binary relations D(X) on A(X). The algorithm is the following :

```
1     for each X in N do D(X) = ∅ ;
2     repeat
        convergence = true
3       for each X_0 in N do
4         for each p in P such that p : X_0 → X_1 X_2...X_np do
5           for each (d_1,d_2,...d_np) ∈ D(X_1) × D(X_2) × ... × D(X_np) do
6                 let d = DO(p) ∪ (j=1 to np) j . d_j ;
7                 compute d* ;
8                 if d* is circular then AG is circular ;
                                        exit ;
9                 let d_0 be the restriction of d* to A(X_0) ;
10                if d_0 ∉ D(X_0) then
                          let D(X_0) = D(X_0) ∪ {d_0} ;
                          let convergence = false
11          endfor
12        endfor
13      endfor
14    until convergence ;
```

In line 6 we mean :

$$a\ d_j\ b \Leftrightarrow (a,j)\ j\ .\ d_j\ (b,j)$$

The inner loop (line 5) must be repeated for each combination of elements of the sets $D(X_i)$, and this makes the whole process intrisically exponential [Jaz81].

The outermost loop must be repeated until we reach the stability of the sets D(X), unless a circularity is detected earlier.

The correctness proof of this algorithm may be found in e.g. [LP75].

3.2 - Discarding redundant relations

[LP75] presented an important improvement to the performance of this algorithm, by modifying the stage nr 10. Indeed, we discard from the sets $D(X_i)$ the redundant relations which carry no information for the circularity test.

Let B be a set of binary relations and $A \subset B$. We see that A covers B iff :

(i) $\forall R \in B - A \quad \exists R' \in A \quad R \subset R'$

(ii) $\forall R, R' \in A \quad R \not\subset R'$.

In the case of binary relations on a finite set, each set B has a covering built from B by discarding any relation contained in another element of B. The decreasing sequence thus obtained, which at each step satisfies (i), ends in the subset satisfying (ii).

Proposition :

If a relation in B *is circular, there exists a relation in the covering of* B *which is circular.*

Proof : See [LP75]. □

The stage nr 10 may be modified in the following way :

10' if d_0 is not covered by $D(X_0)$ then

$D(X_0)$ = covering $(D(X_0) \cup \{D_0\})$;

convergence = false ;

In section 8 it is shown that this improvement decreases the number of relations in each D(X) in a very subsantial nammer.

4 - CHEBOTAR'S ALGORITHM

4.1 - General overview

Another important improvement to reduce the time needed for circularity test was presented by K.S. Chebotar [Che81]. We think that his paper did not receive the full attention it deserves, so we present again here his ideas.

The basic point is to find an ordering of the non-terminals of the grammar such that the above algorithm can be applied onto a sequence of sub-grammars, keeping for the next stage only the results of the current stage. We take thus an exponential problem for a big grammar to the sum of exponential problems for smaller grammars.

Because of the locality principle -all the dependencies are local to a production- it is natural to take into account the "syntactic dependencies" between non-terminals to try to define this ordering. This is done as follows.

4.2 - Syntactic dependency graph

We construct on the non-terminals a relation Γ defined for each production

$$p : X_0 \rightarrow X_1\ X_2 \ \ldots\ X_{n_p} \text{ by}$$

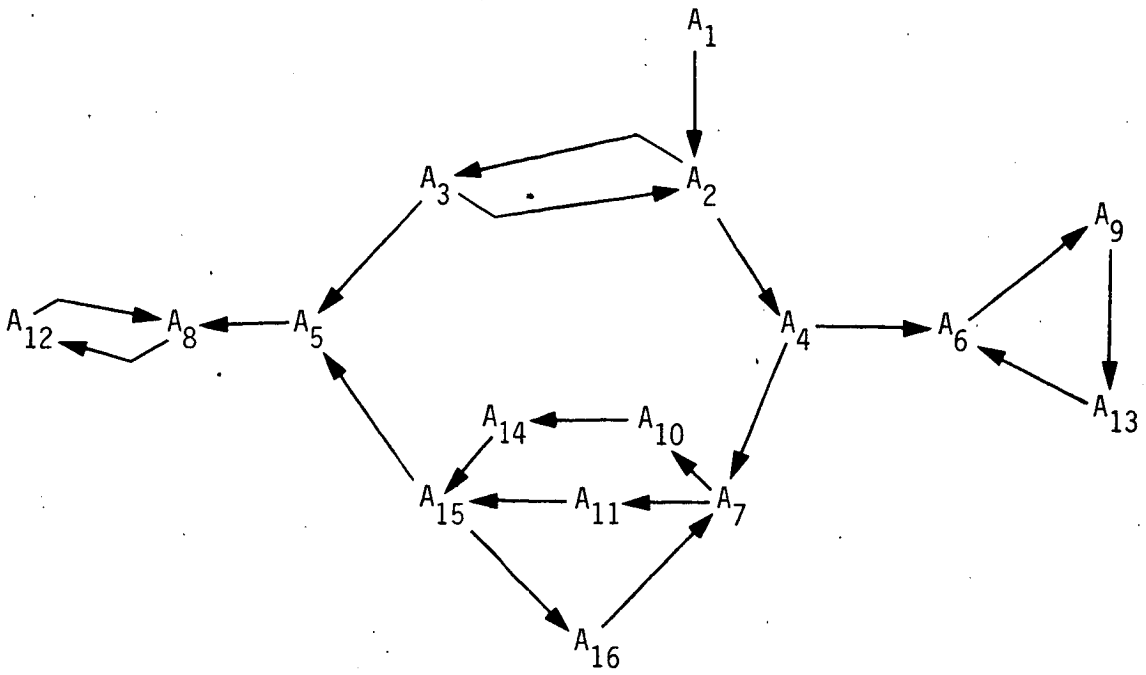
$$X_0\ \Gamma\ X_i \qquad 1 \leq i \leq n_p$$

This relation associates the left-part symbol of a production with the right-part ones.

Example [Che81] (the terminal symbols have been discarded) :

$A_1 \rightarrow A_2$	$A_2 \rightarrow A_3 \mid A_4$	$A_3 \rightarrow A_2\ A_5$	$A_4 \rightarrow A_6\ A_7$
$A_5 \rightarrow A_8$	$A_6 \rightarrow A_9$	$A_7 \rightarrow A_{10} \mid A_{11}$	$A_8 \rightarrow A_{12}$
$A_9 \rightarrow A_3$	$A_{10} \rightarrow A_{14}$	$A_{11} \rightarrow A_{15}$	$A_{12} \rightarrow A_8$
$A_{13} \rightarrow A_6$	$A_{14} \rightarrow A_{15}$	$A_{15} \rightarrow A_{16}\ A_9$	$A_{16} \rightarrow A_7$

Graph of the relation Γ :



4.3 Equivalence classes

From the graph of relation Γ we build the graph of relation Γ_0 of connected components. These connected components are the sets of mutually recursive non-terminals (with respect to the grammar G).

For the above example, we get the following classes :

$B_1 = \{A_1\}$

$B_2 = \{A_2, A_3\}$

$B_3 = \{A_4\}$

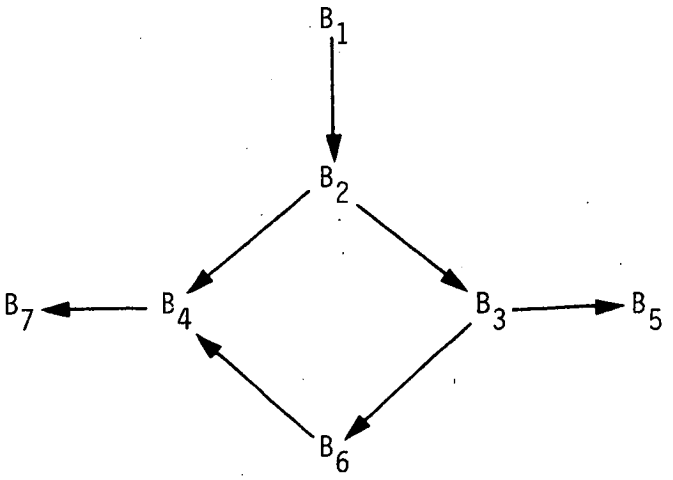
$B_4 = \{A_5\}$

$B_5 = \{A_6, A_9, A_{13}\}$

$B_6 = \{A_7, A_{10}, A_{14}, A_{15}, A_{11}, A_{16}\}$

$B_7 = \{A_8, A_{12}\}$

and the graph Γ_0 :



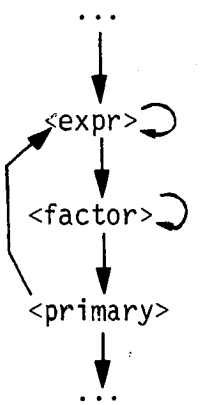
For a more familiar example, let us consider (a part of) a grammar describing arithmetic expressions, with operator precedence treated syntactically :

<expr> = <expr> addop <factor> | <factor>

<factor> = <factor> mulop <primary> | <primary>

<primary> = (<expr>) | constant | identifier

The Γ relation is thus :



and the three non-terminals belong to the same equivalence class.

4.4 - Ordering the connected components graph

The graph Γ_0 exhibit the dependencies between the equivalence classes. It is used to compute, through a topological sort, an order in which these classes are to be treated.

Sorting algorithm :

```
1   let i = 1 ;
2   repeat
3       choose a node in Γ0 such that no edge exits from it ;
4       give it the rank i ;
5       discard it from Γ0, along with the edge(s) that reach it ;
6       i = i + 1 ;
7   until Γ0 = ∅
```

For our example, we get :

Rank	1	2	3	4	5	6	7
nodes	B_5	B_7	B_4	B_6	B_3	B_2	B_1

We can notice that the order of the two first nodes does not matter.

4.5 - Ordering the non-terminals

The above ordering is the order in which we have to process the equivalence classes, that is, the order to use to select the sub-grammars on which to apply the circularity test. At each step of the test, we will have to keep the sets of relations for some of the symbols of the current class, and forget (discard) those for symbols that will not ever be referenced.

The following algorithm gives a priority to non-terminal symbols : it uses the above ordering and, in each class, distinguishes between the non-terminals that are entry points in the class, which will be kept for subsequent stages, and others, which will be forgotten.

Algorithm to determine the entry points of each class.

Notations : ℓ is the total number of equivalence classes.
B^m denotes the class B_i with rank m.

1 let i = ℓ ;

2 repeat

3 choose among nodes A_k of graph Γ not yet dealt with those such that

$$\exists j,\ A_k \in B^j \text{ and } B^i \xrightarrow{\Gamma_0} B^j$$

$$\text{and } \exists A_m \in B^i,\ A_m \xrightarrow{\Gamma} A_k$$

{A_k is an entry point of class B^j} ;

4 give them priority i ;

5 let i = i - 1 ;

6 until i = 0

This first part gives the priority i to the entry point(s) of the class(es) which is(are) directly connected in Γ_0 to the class B^i.

For our example, we get :

Priority	7	6	5	3
Nodes	A_2	A_4, A_5	A_6, A_7	A_8

To the symbols that are not entry points of this x class, we give priority equal to the rank of the class :

for each i, $1 \leq i \leq \ell$ do
give priority i to still unmarked nodes A_k of class B^i
endfor

We get :

Priority	1	2	4	6	7
Nodes	A_9, A_{13}	A_{12}	$A_{10}, A_{14}, A_{15}, A_{11}, A_{16}$	A_3	A_1

And the whole ordering is :

Priority	1	2	3	4	5	6	7
Nodes	A_9, A_{13}	A_{12}	A_8	$A_{10}, A_{14}, A_{15}, A_{11}, A_{16}$	A_6, A_7	A_3, A_4, A_5	A_2, A_1

4.6 - Testing the circularity

The algorithm is now very simple :

```
1    let i = 1 ;
2    repeat
3       apply the algorithm of section 3.1 or 3.2 to the sub-grammar B^i ;
4       discard the relations of non-terminals of priority i ;
5       i = i - 1
6    until i = 0
```

The correctness proofs can be found in [Che81], along with some practical results.

We must notice that :

- the algorithms 4.3 to 4.5 are polynomial, so the whole method can be applied safely to polynomial algorithm testing other properties than non-circularity, e.g. the strong non-circularity ([CF82], or alg. A21 in [LP75]) or the absolute non-circularity ([KW76]).
- we build relations on non-terminals, going upward in the graph Γ (Γ_0) and in the derivation tree.

5 - WEAK STABILITY

As stated before, a circularity test computes relations on a non-terminal, going upwards in the tree. It must loop until these relations are stable. After the i-th iteration, it has thus computed relations on trees, the height of which is less than or equal to i. If some of these trees are terminal, i.e. their leaves are all labelled with terminal symbols (including the empty string), it is unnecessary to recompute the relations for these trees.

Then let us state the following definitions : After each iteration of the algorithm :

- mark as weakly stable each production, the right part of which is composed only by terminals and weakly stable non-terminals ;

- mark as weakly stable each non-terminal such that each production having this non-terminal as left part is weakly stable.

At the beginning of the algorithm, no production and no non-terminal is weakly stable.

Then we have :

Lemma : *After the i-th iteration, the weakly stable non-terminals or productions can generate only terminal trees, of height less than or equal to* i.

Proof : easy by induction on i, noticing that if the right-hand side of a production is made only of terminals or non-terminals which can generate only terminal trees of height $\leq i$, then this production can generate only terminal trees of height $\leq i + 1$. □

Now we can include this notion in any circularity algorithm, skipping the computation for weakly stable productions or non-terminals, and updating the markers after each iteration. Thus we can avoid useless recomputations.

This method is rather limited : if a production is recursive (directly or not), then it (and its left-hand side non-terminal) will never be marked as weakly stable. But it can be useful for grammars having many terminals, like asm (see section 8 and [Jou 82]) (a cross-assembler : each instruction is a distinct terminal) : for that one, the gain is roughly up to 35%. For more

classical grammars we have observed gains ranging between 2 and 10% (cpu time).

Like Chebotar's method, weak stability is independent on the algorithm used, provided it computes relations "upwards the tree". As a special case, we applied it to algorithms testing strong non-circularity (see [CF 82] or algorithm A 21 in [LP 75]).

6 - COMBINING BOTH METHODS

Let us come back to the partitioning method : when computing relations on a given class, the classes which have already been dealt with may be considered as terminal. So we can apply the weak stability to each class. It has then the same limitations as in section 5, but also the same safeness. When convergence of the algorithm is reached for the current class, then we can mark all its elements (non-terminals) as weakly stable. It is useless to do it for productions having these non-terminals as left hand side because they will not be referenced any more.

The main gain achieved by weak stability here is for non-terminals which are alone in their class and not recursive. This means that in the graph Γ they look like :

In one iteration the associated relations are computed. But with the original algorithm, even optimized using partitioning or covering, you need a second iteration to check that you have reached convergence. This second iteration is eliminated by the weak stability.

The practical results of section 8 will show the complementarity and synergy of these two methods.

7 - EVALUATING THE PRACTICAL COMPLEXITY OF CIRCULARITY TESTS

It is well-known that a large difference may exist between the theoretical complexity of an algorithm (either average or worst-case) and its practical behaviour. Attribute grammars are a perfectly good illustration of this fact : although the theoretical worst-case complexity of circularity tests is exponential in the "size" of the grammar, the practical efficiency observed for "useful" examples is quite acceptable. We try to show the reasons for that in this section, evaluating the average complexity of these tests. We show also on which factors of this complexity our improvements act.

We use the following notations : for each grammar AG :

pr is the number of productions,

ant is the average length of the right-hand side of any production, measured only with non-terminals,

ana is the average number of attributes attached to any non-terminal,

it is the number of iterations of the (original) circularity test,

α is the average number of relations in any D(X) during the circularity test.

Let us now compute the average complexity of the circularity test.

The basic step (lines 6 to 10) is of complexity $(\underline{ant} \times \underline{ana})^3 + \alpha \times ant \times ana$ since it is mostly the computation of the transitive closure of a graph with $(\underline{ant} + 1) \times \underline{ana}$ edges and a comparison with α such graphs. This basic step is executed (loop lines 5 - 11) for each combination of relations of the non-terminals an the right hand side : clearly there are $\alpha^{\underline{ant}}$ such combinations. Then the embodying loop (the two loops 3/4 - 12/13) is executed once for each production. At last the whole process (1 - 14) is repeated until convergence, that is it times.

The whole complexity is then :

$$O\ (\underline{it} \times \underline{pr} \times \underline{ant}^3 \times \underline{ana}^3 \times \alpha^{\underline{ant}})$$

So the exponential factor seems to be driven either by the length of right hand sides of productions, or by the number of relations in each D(X), or both.

Räihä and Saarinen [RS 82] find the same complexity (using worst-case

figures) - apart from an additional factor (the number of non-terminals) which seems out of place - and show that the first factor is not responsible : they use a (polynomial) transformation of the grammar into a chomsky normal-form grammar, with at most 2 non-terminals on right hand side of any production.

There stays to evaluate α.

Let A be a set of n elements. Then a binary relation on A is a subset of $A \times A$. So the total number of different binary relations which we can construct on A is 2^{n^2}. The number of acyclic relations is substantially smaller, but however it is still rapidly exponential. In the sequel of this section, n will be identified with ana.

But practical results ([RS 82], [LP 75] and section 8) show that in fact, for real examples, the number of relations in any D(X) during the test does not exceed a reasonable value. Where is the explanation ?

Let us try to isolate some "philosophical" concepts involved in programming with attribute grammars. BNF and attribute grammars (its semantic extension) is a perfect example of top-down structured programming : each non-terminal represents a sub-problem of the global problem of attaching a "value" to a program, represented by the start-symbol of the grammar. These sub-problems may be viewed as "black boxes" with input information - the inherited attributes-and output information - the synthesized attributes-and a transfer function, symbolized by the semantic rules. This stems out of a result of [CF 82], which states that the value of a synthesized attribute at a node depends only on :

- the subtree issued from that node,
- the inherited attributes at that node.

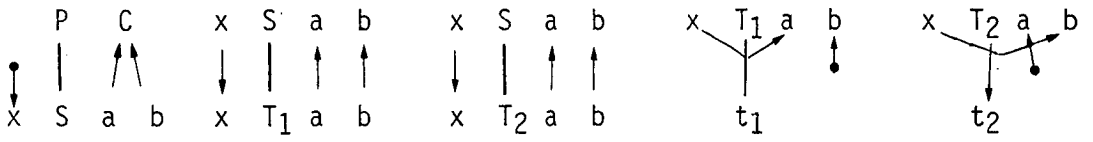
If agreeing with this point of vue, it seems clear that the dependencies between input and output must have some semantic meaning, and so cannot be "anything". Thus, around a given non-terminal, a given synthesized attribute can have only a "reasonable" number of different dependencies. So the number of relations in each D(X) remains "reasonable".

To state yet another argument in this sense, let us recall that nearly all the real examples are strongly/absolutely non-circular : all the dependencies of a synthesized attribute can be gathered in only one graph without being circular. This means that the power of contextual grammars is not employed fully in

these examples, and this is in accordance with the structured design of the attribute grammar.

We now borrow from [RS 82] a useful definition. We say that an attribute grammar is G(k) iff the circularity test for that grammar can be carried out with no more than k relations in any D(X) et any time. The classes G(k) thus defined depend on the algorithm used. As a special case, the covering broadens them, and a grammar in G(k) for the standard algorithm is in G(k') with $k' \leq k$ (and usually $k' << k$) with the covering algorithm.

Talking about [RS 82], we would like to point out that, for the algorithm they use (the standard one), the class G(1) is included in, but not the same as, the class of strongly/absolutely non-circular grammars ([CF 82], [KW 76]). Here is an example, with only the dependencies shown :



This grammar is SNC : x S a b

It is not in G(1), since for S we have two relations, unmergeable : x S a b and x S a b.

We now state a conjecture (and give no hints to solve it) : find an algorithm such that G(1) = SNC (G(1) for this algorithm). Solving this conjecture would supply an algorithm which would be polynomial for SNC attribute grammars and exponential for the others, but would be different from the trivial juxtaposition of the two algorithms.

The covering acts evidently on factor α, the number of graphs in each D(X). To try to estimate the gain, we will use, as also mentionned in [Che 81], Sperner's lemma, which states that, for a set of m elements, we cannot find more than $C_m^{[m/2]}$ subsets such that no one is included in another, where [m] is the integral part of m. Thus the covering reduces the maximal number of relations in any D(X) from 2^{n^2} to $C_{n^2}^{[n^2/2]}$. For small n's the gain seems to be $O(n)$. We did not try to compute the (theorical) maximum number of acyclic relations with covering. For practical cases, the observed gain is also of the order of n(see section 8).

Let us turn to Chebotar's method. Let us assume that we can find k classes of non-terminals in the grammar, each having on the average $\underline{pr}/k$ productions. The complexity becomes ;

$$\sum_{j=1}^{k} (\underline{it}_j \times \underline{pr}/k \times \ldots)$$

where $\underline{it}\ j$ is the number of iterations of the algorithm on the j-th class (subgrammar). Let $\underline{it}_{av}$ be the average of the $\underline{it}_j$'s. Then the complexity is : $(\underline{it}_{av} \times \underline{pr} \times \ldots)$

It seems that the gain is null. But actually we do gain much because the problem on the subgrammar is much easier than the problem on the whole grammar, and so each $\underline{it}_j$ is less than or equal to the original $\underline{it}$. In fact we have $\underline{it} \geq \max_{j=1}^{k} \underline{it}_j$. And generally most of the $\underline{it}_j$'s an significantly smaller than $\underline{it}$, so that $\underline{it}_{av} << \underline{it}$. In section 8 we use a weighted average $\underline{it}_{av}$ to do the comparisons.

The weak stability used alone acts mainly on the product $\underline{it} \times \underline{pr}$, by "forgetting" to process some productions during some iterations. Used together with the partitioning method, it can further decrease some $\underline{it}_j$'s, and so $\underline{it}_{av}$.

The other factors, which are constant of the grammar, are of course not affected.

8 - PRACTICAL RESULTS

We have implemented all these algorithms, and tried them on some real examples. In this section we will state some measures we conducted on them.

We will use the following notations :

A1 = standard algorithm ;
A2 = A1 with covering ;
A3 = A2 with weak stability ;
A4 = A2 with Chebotar's partitioning ;
A5 = A2 with weak stability and Chebotar's partitioning ;

For each attribute grammar :

nt = number of non-terminals ;
pr = number of productions ;
ant = average number of non-terminals in right sides of productions ;
a = number of attributes ;
ana = average number of attributes per non-terminals.

For algorithms A1, A2 and A3 :

it = number of iterations ;
d_{max} = maximum number of relations in any D(X) at any time ;
d_{av} = average number of relations in any D(X) at any time ;
R_{max} = maximum number of co-resident relations, i.e. $\underset{\text{time}}{\text{Max}} \left(\sum_{X} |D(X)| \right)$;
tcl = number of transitive closures computed ;
cpu = cpu time, measured in seconds on the Multics system at INRIA.

For algorithms A4 and A5 :

k = number of non-terminals classes ;
it_{max} = maximum number of iterations on each class ;
it_{av} = weighted average number of iterations on each class, defined by $\left(\sum_{j=1}^{k} it_j \times pr_j \right) / pr$, with obvious notations ; we think it is a far best figure to compare

with the number of iterations of the preceeding algorithms than the plain average, which would be $\left(\sum_{j=1}^{k} it_j \right) /k$;

d_{max}, d_{av}, R_{max}, tcl and cpu as before.

As a comparison, we also measured an algorithm that checks the strong (absolute) non-circularity. It is designated as SNC. It includes Chebotar's partitioning and weak stability.

Now a word on our examples :

- simproc is a simple semantic checker and translator for a toy language with integers, arrays and recursive procedures ;
- asm is a cross-assembler generating hexa-decimal code for a MC6809 microprocessor ;
- simula is a compiler for the language Simula, borrowed from Fang [Fan 72] ; we did not succeed in making it non-circular, but this does not change in fact the complexity of the test ;
- lom is a translator for a FORTRAN-like application language.

The six algorithms have been implemented using the same basic data structures, in order not to bias the results by implementation details.

Here are the rough results :

		simproc	asm	simula	lom	pascal
	nt	11	59	126	134	114
	pr	21	262	244	443	214
	ant	1,62	0,47	1,127	0,97	1,17
	a	14	22	37	27	51
	ana	3,91	3,56	7,28	2,86	6,25
A1	it	5	11		≥ 9	
	d_{max}	14	756			
	d_{av}	1,87	9,15			
	R_{max}	27	887		≥ 451	
	tcl	503	135 268		≥ 750 000	
	cpu	13,46	7 890		≥ 36 000	
A2	it	5	8	16	10	13
	d_{max}	3	4	5	3	5
	d_{av}	0,91	0,66	1,05	0,92	1,06
	R_{max}	12	45	184	132	147
	tcl	142	2 485	5 533	4 999	4 887
	cpu	2,43	25,69	360,5	91,36	357
A3	tcl	116	1 117	4 501	3 232	4 090
	cpu	2,26	20,01	347,4	73,75	344
A4	k	7	58	58	119	57
	d_{max}	3	3	5	3	5
	d_{av}	0,89	0,68	1,1	0,98	1,1
	R_{max}	7	20	95	43	67
	it_{max}	4	3	14	7	13
	it_{av}	2,95	1,36	6,73	3,06	5,79
	tcl	96	672	3 130	1 883	3 109
	cpu	2,53	11,83	221,7	52,64	233,7
A5	it_{max}	4	3	14	7	13
	it_{av}	2,86	1,107	6,51	2,58	5,53
	tcl	80	580	2 899	1 530	2 917
	cpu	1,88	8,43	181,7	47,12	193,4
SNC	tcl	67	542	1 539	1 449	1 120
	cpu	1,43	9,64	110	53,4	115

Except for simproc and asm, our smallest examples, we have only partial results for the standard algorithm, because running it on the other examples led to exceed the maximum cpu time limit for batch processing on the Multics system, which is 36 000 seconds (ten hours !). We will develop this point in the sequel.

Here are condensed tables showing the gains achieved.

Gain on computing time

We show the gain on the number of transitive closures computed (number of basic steps of the algorithms) and the gain on cpu time, which includes all the machinery to achieve the improvements (the gains are expressed in terms of ratios).

		simproc	asm	simula	lom	pascal
tcl	2/1	3,54	54,4		≥ 150	
	3/2	1,22	2,22	1,23	1,55	1,19
	4/2	1,48	3,70	1,77	2,65	1,57
	5/2	1,77	4,28	1,91	3,27	1,68
	5/1	6,29	233		≥ 490	
cpu	2/1	5,54	307		≥ 394	
	3/2	1,075	1,28	1,038	1,24	1,037
	4/2	0,96	2,17	1,626	1,74	1,53
	5/2	1,29	3,05	1,984	1,94	1,846
	5/1	7,16	936		> 764	

Gain on storage

We use therefore R_{max}, which measures the space needed to store the relations.

		simproc	asm	simula	lom	pascal
R_{max}	2 or 3/1	2,25	19,7		≥ 3,42	
	4 or 5/2	1,71	2,25	1,94	3,07	2,19
	4 or 5/1	3,86	44,3		≥ 10,5	

Comments

The first that comes to mind is that the standard algorithm is quite impracticable. You need at least the covering to get realistic algorithms.

Each grammar belongs to the class G (d_{max}) for each algorithm. As shown by the results, the covering acts strongly on d_{max}, since it changes the way the relations are managed. For some examples (asm), the partitioning method also reduces d_{max}, by allowing a covering relation to be detected earlier. The covering also reduces the space needed to store the relations.

The gains achieved by the weak stability depends on the shape of the grammar. If many non-terminals are non-recursive, then it can be important (asm, lom). It is also important if the grammar has many terminals (asm : about half of the productions are of the form nt → t). But the weak stability does not change the number of relations (R_{max}, d_{max}, d_{av}).

The gains achieved by Chebotar's partitioning method also depend on the shape of the grammar. If the grammar is linear (k/nt important, i.e. many non-recursive terminals), the gains can be quite appreciable (asm, lom). Conversely, it the grammar is very recursive, the gains tend to be low (for example, bloc-structured languages like simproc and simula), but never neglectable !!!

The combination of both methods can be quite powerful in reducing the number of iterations : all examples are signifiant on this point.

About storage, the gain achieved by Chebotar's method is due to the fact that we can forget useless relations.

We can conclude that the three improvements, covering, Chebotar's partitioning, and weak stability, are really efficient.

Now we have to present a very interesting example. It is a big grammar describing pure Lisp, for semantic checking and translation into code for a stack machine : nt = 34, pr = 117, a = 27. It is well-known that Lisp is a strongly recursive language : Chebotar's partitioning leads to only 16 classes, and one of these classes contains 15 non-terminals !! About attributes, the grammar is quite complicated, since we find d_{max} = 15. In fact two non-terminals have d_{max} = 15, the others less than 6. The surprising result is that Chebotar's method leads to an increase of the cpu time and of the number of transitive closures computed. Although slight (for A2, tcl = 7863, for A4, tcl = 8378), it is bothering. Here is an explanation : the semantic complexity is such that, on that famous big class, A4 has to do the same number of iterations to reach convergence as A2. But A2 starts with empty relations, and A4 starts with the non-empty relations computed on the non-terminals which are output points of this class. So, during the first iterations, A4 computes more closures than A2. The point here is the semantic complexity, measured by d_{max}. Compare with the algorithm checking strong non-circularity, which computes only one relation per non-terminal : this grammar is strongly non-circular, with only 335 tcl computed... This is confirmed by the very small difference on R_{max} (R_{max} (A2) = 58, R_{max} (A4) = 52) which shows that all the job is done on this big unique class. But this example is rather pathological.

9 - CONCLUSION

We have presented three methods to improve circularity tests for attribute grammars : the covering, Chebotar's partitioning method and the weak stability. All three allow significant savings on time and/or space, so that cirularity tests, a priori exponential, are still practicable, even for large grammars. Their implementations are easy.

Chebotar's partitioning and weak stability can also be applied to many other algorithms than circularity tests.

We have also analyzed the (theoretical) average complexity of these tests. The problem is still opened to discover why real examples do not cause the combinatorial explosion predicted by the theory. It would be interesting to search in the direction of what we called the "methodology for programming with attribute grammars".

10 - REFERENCES

[Che 81] K.S. Chebotar : "Some modifications of Knuth's algorithm for verifying cyclicity of attribute grammars", Programming and Computer Science, 7, 1 (1981), pp 58-61.

[CF 82] B. Courcelle, P. Franchi-Zannettacci : "Attribute grammars and recursive program schemes", Theoretical Computer Science, 17 (1982), pp 163-191 and 235-257.

[Fan 72] I. Fang :"FOLDS, a declarative formal language definition system", Report STAN-CS-72-329, Department of Computer Science, Stanford University (1972).

[Jaz 81] M. Jazayeri : "A simpler construction for showing the intrisically exponential complexity of the circularity problem for attribute grammars", JACM, 28 (1981), pp 715-720.

[Jou 82] M. Jourdan : "Un evaluateur efficace pour les grammaires attribuées fortement non-circulaires", DEA Report,University Paris 7 (1982) (in French). Also Rapport n°82-39, Laboratoire d'Informatique Théorique et de Programmation, Paris, France.

[KW 76] K. Kennedy, S.K. Warren : "Automatic generation of efficient evaluators for attribute grammars", in Conf. Record of 3rd ACM Symp. on Principles of Programmings Languages, Atlanta, Georgia, USA (Jan. 76) pp 32-49.

[Knu 68] D.E. Knuth : "Semantics of context-free languages", Math. Systems Theory, 2 (1968), pp 127-145.

[Knu 71] D.E. Knuth : "Semantics of context-free languages : correction", Math. Systems Theory, 5 (1971), pp 95-96.

[LP 75] B. Lorho, C. Pair : "Algorithms for checking consistency of attribute grammars", in Proving and Improving Programs, (G. Huet and G. Kahn, eds), Colloque IRIA, Arc-et-Senans, France, 1-3 July 1975, pp 29-54.

[RS 82] K. J. Räihä, M. Saarinen : "Testing attribute grammars for circularity", Acta Informatica, 17 (1982), pp. 185-192.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique