Speeding up parallelism detection for attribute grammars

Matthijs F. Kuiper

RUU-CS-92-18
April 1992

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Speeding up parallelism detection for attribute grammars

Matthijs F. Kuiper

Technical Report RUU-CS-92-18
April 1992

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Speeding up parallelism detection for attribute grammars

Matthijs F. Kuiper
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: kuiper@cs.ruu.nl

Abstract

This report presents methods for speeding up parallelism detection for attribute grammars. For large existing attribute grammars these methods speed up parallelism detection with factors ranging from 8 to 160. The optimizations exploit the ability to efficiently predict whether a graph introduces useful dependencies and skipping all computations with useless graphs. These predictions are based on an abstract interpretation of dependency graphs.

1 Introduction

This report presents an efficient parallelism detector for attribute grammars that analyses existing attribute grammars 8 to 160 times faster than a non-optimized version. It achieves these improvements by skipping useless computations. It predicts whether a computation is useless by using an abstract interpretation of dependency graphs. When analysing attribute grammars for Pascal and occam, more than 99% of all useless computations are skipped.

This research shows that parallelism detection for attribute grammars, even for large grammars, is practical. We present the results of various detailed measurements that illustrate the effectiveness of our optimizations.

The remainder of this report is structured as follows. Section 2 defines some notions related to attribute grammars. Then, sections 3 through 5 introduce the non-optimized algorithm for parallelism detection. Section 6 defines our optimizations and sections 7 and 8 discuss the effects of these optimizations. Section 9 discusses the possibility of further improvements. Section 10 discusses related work on attribute grammar analysis.

2 Attribute grammars

We assume that the reader is familiar with attribute grammar terminology. We use the notation of attribute grammars from [12], with the following peculiarities.

The graph $D(pr)$ contains the attribute dependencies from the attribute equations associated with production pr.

A structure tree is a tree derived from a nonterminal (not necessarily the root) of the underlying context-free grammar. Nodes in a *structure tree* are labelled with the production applied at them. If u is a node then *prod* u is the production applied at u, and *nont* u is the left-hand side nonterminal of *prod* u. The *dependency graph* of a structure tree S is denoted by $DG(S) = (XG(S), E(S))$; $XG(S)$ is the set of all attribute instances attached to S and $E(S)$ contains an edge from α to β if β depends on α. An edge from α to β is depicted as $\alpha \rightarrow \beta$.

To shorten the explanations, a tree is sometimes associated with its root and vice versa. For example, the dependency graph of a node u, $DG(u)$, is the dependency graph of the tree rooted at u. Similarly, the label of a tree is the label of the root of the tree. We use the convention that nodes of structure trees are named with lower case letters. The nonterminal labelling a node is the corresponding upper case letter. Thus node x is labelled with X.

3 Parallelism detection

To be able to build parallel attribute evaluators we must know which attribute instances are independent. We can, however, not compute *in*-dependencies. To conclude that the instances of one or more attributes are independent requires the computation of all possible dependencies between attributes. The absence of a possible dependency is proof of independence. For this reason, our parallelism detectors compute all possible dependencies between attribute instances attached to *different* and possibly *remote* nodes.

As an illustration of our parallelism detectors consider a structure tree S with two interior nodes u and v, labelled with nonterminals U and V, respectively, as illustrated in figure 1. Neither of the two nodes u and v is an ancestor of the other. We call such nodes *cousins*.

From this example tree and its dependency graph we conclude the following fact:

> there exists a tree S labelled with X and with two interior cousin nodes u and v so that $u.\beta$ depends on $v.\alpha$. In particular, the path from $v.\alpha$ to $u.\beta$ in $DG(S)$ does not contain a part that runs through $DG(u)$ or $DG(v)$.

The dependency of $u.\beta$ upon $v.\alpha$ is called a *remote dependency*. Remote dependencies are represented with so called *dependency patterns*. The above dependency is represented as $([X, U, V], \{V.\alpha \rightarrow U.\beta\})$. This dependency pattern, called a *triple*, consists of a list of three nonterminals and a set of attribute dependencies. It must be read as a shorthand notation for the above fact. The list of three nonterminal $[X, U, V]$ says that there exists a structure tree S whose root is labelled with X and that contains two interior cousin

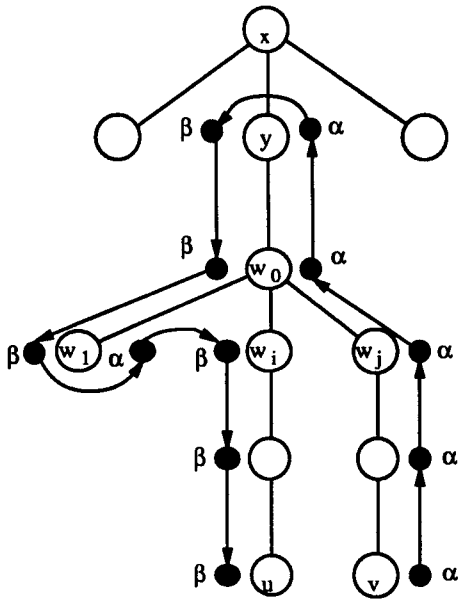

Figure 1: A tree fragment illustrating a remote dependency from an instance of $V.\alpha$ to an instance of $U.\beta$. White dots represent tree nodes; black dots represent attribute instances.

nodes that are labelled with U and V, respectively. The set of dependencies says that the dependency graph of S contains a path from $v.\alpha$ to $u.\beta$. Dependency patterns, as introduced above, can be either true or false. Parallelism detection for attribute grammars amounts to the computation of all true dependency patterns, without computing a false one.

The above dependency pattern is computed from simpler dependency patterns in which the lists of nonterminals contain one or two elements. A dependency pattern with one nonterminal, a *singleton*, has the form $([X], r)$, where r is a set of dependencies among the attributes of X. It states that there exists a tree whose root is labelled with X and that all dependencies in r occur among the attribute instances attached to the root of S. A dependency pattern with two nonterminals, a *pair*, has the form $([X, U], r)$. Again r contains dependencies among attributes of X and U. This dependency pattern must be read as:

> there exists a tree S with a root labelled X and an interior u node labelled with U. Furthermore all dependencies in r occur between the attribute instances attached to u and the root of S, and these dependencies do not run through $DG(u)$.

The rules for computing dependency patterns mimic the bottom up construction of dependency graphs. We illustrate how triples are computed from singletons and pairs by considering the computation of triple $([X, U, V], \{V.\alpha \rightarrow U.\beta\})$. We assume that all true singletons and all true pairs have already been computed.

The example tree in figure 1 shows the truth of the singleton $([W_1], \{W_1.\beta \rightarrow W_1.\alpha\})$, since in the tree rooted at w_1, $w_1.\alpha$ depends on $w_1.\beta$. The singleton $([W], \emptyset)$ is, of course, vacuously true. In the tree we can further verify the truth of the following pairs: $([W_j, V], \{V.\alpha \rightarrow W_j.\alpha\})$; $([X, W], \{W.\alpha \rightarrow W.\beta\})$; and $([W_i, U], \{W_i.\alpha \rightarrow U.\beta\})$.

From these true singletons and pairs and the dependencies in $D(prod\ w)$ the triple $([X, U, V], \{V.\alpha \rightarrow U.\beta\})$ is computed. Figure 2 suggest how this is done. The dependency patterns are considered as if they represent a fragment of a structure tree with associated dependencies and the same holds for production pr and the dependencies in $D(pr)$. The tree fragments are pasted together and the dependencies are combined. To obtain the triple the transitive closure of the dependencies is projected on the attributes of X, U and V.

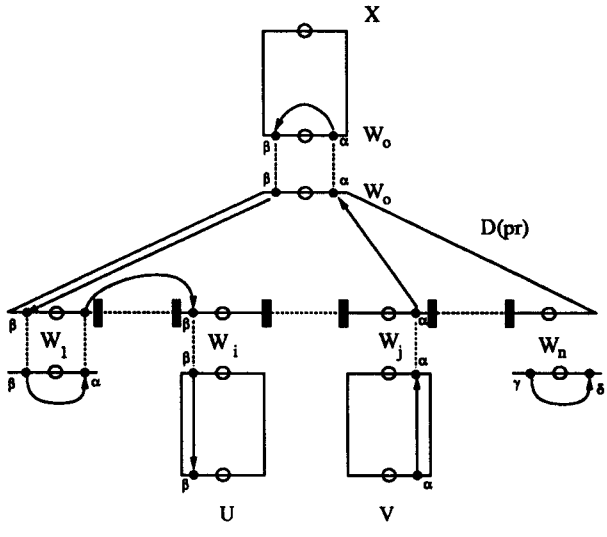

Figure 2: Dependencies from singletons, pairs and $D(pr)$ are combined. Black dots representing the same attribute are connected by a dotted line. $([W_1], \{W_1.\beta \rightarrow W_1.\alpha\})$ and $([W_n], \{W_n.\gamma \rightarrow W_n.\delta\})$ are singletons. The dependencies in the latter singleton do not occur on the path from $V.\alpha$ to $U.\beta$. The result is the triple $([X, U, V], \{V.\alpha \rightarrow U.\beta\})$.

The step illustrated in figure 2 is the basic step in the computation of triples.

4 Algorithm for computing dependency patterns

The algorithm for computing dependency patterns consists of three steps. The first step computes the set *SINGLE* that contains all true singletons. This step is basically Knuth's singularity test [4, 5]. The second step computes the set *PAIR* that contains all true pairs. The third step computes the set *TRIPLE* that contains all triples. Finally the dependencies from *TRIPLE* are used to determine which nonterminals are independent from each other.

The algorithms for computing *SINGLE* and *PAIR* can be found in [6]. Here we only discuss the computation of *TRIPLE*, because it is the most expensive of the four steps.

4.1 Computing triples

The algorithm to compute triples combines singletons, pairs and dependencies from productions as illustrated in figure 2. The algorithm below computes all triples with first nonterminal X, i.e. the set $Triple(X) = \{([X,G,H],r)|([X,G,H],r) \in TRIPLE\}$. If one wants to compute all triples then the algorithm must be applied for all nonterminals.

The algorithm uses the notion of a selection: a selection for a production *pr* consists of a singleton for each right-hand side nonterminal occurrence in *pr*. The set $Pairs(X)$ contains all pairs in *PAIRS* with first nonterminal X: $Pairs(X) = \{([X,G],r)|([X,G],r) \in PAIRS\}$.

algorithm: computation of triples
input: the set *SINGLE*, containing all true singletons
the set *PAIR*, containing all true pairs
a nonterminal X
output: the set $Triple(X)$
var *TRIPLE*: set of triples

(1) **for all** *upper_pair* $\in Pairs(X)$ **do**
(2) **for all** productions *pr* with *lhs* $pr{=}X$, $pr : W_0 \rightarrow W_1 \ldots W_n$ **do**
(3) **for all** selections *sel* $\in$ *selections*(*pr*) **do**
(4) **for all** (*left*, *right*) $\in Pairs(W_i) \times Pairs(W_j)$, $i \neq j$ **do**
(5) compute triple from $D(pr)$, *sel*, *upper_pair*, *left* and *right*
(6) insert triple in *TRIPLE*
(7) **od**
(8) **od**
(9) **od**
(10) **od**

Line (4) of the above algorithm consists of four nested loops:

(4.a) **for all** *ntocc0* $\in$ right-hand side nonterminals *pr* **do**
(4.b) **for all** *ntocc1* $\in$ right-hand side nonterminals *pr*, *ntocc1* $\neq$ *ntocc0* **do**
(4.c) **for all** *left* $\in Pairs(ntocc0)$ **do**
(4.d) **for all** *right* $\in Pairs(ntocc1)$ **do**

5 The non-optimized parallelism detector

Our basic parallelism detector implements the algorithm from section 4.1 with three trivial improvements.

The first improvement comes from realizing that it suffices to compute, for each X, U and $V \in N$, either all triples of the form $[X, U, V]$ or all triples of the form $[X, V, U]$ since both kinds of triples contain the same dependencies. This improvement is achieved by first defining an arbitrary order $\leq$ on nonterminals and then computing only triples $[X, U, V]$ where $U \leq V$.

The second improvement is to compute only *maximal* relations. A dependency pattern (p, r) is then not considered if there is a dependency pattern (p, t) with $r \subseteq t$. This improvement might have as effect that not all true dependency patterns are computed. If the above (p, r) is a true pattern then it will not be computed. This optimization can, however, be safely applied if one is only interested in the *existence* of remote dependencies.

The third improvement comes from not considering productions pr in line (2) that have at most one nonterminal on their right-hand side. Such productions do not introduce a remote dependencies.

We call the basic algorithm with the above three improvements NOOPTS.

6 Optimizing triple computation

This section presents our optimizations to algorithm NOOPTS.

6.1 Skipping productions

Computations with production pr can be skipped if $D(pr)$ has no dependencies among the attributes of the right-hand side nonterminals. This is, for example, the case if all dependencies in the production run only from a right-hand side attribute to a left-hand side attribute. This can be tested at line (2) of the algorithm.

6.2 Skipping nonterminal occurrences

Computations with the nonterminal occurrence *ntocc0*, see line (4.a), can be skipped if its attributes are isolated from attributes of all other nonterminal occurrences in the production. For example, this is the case if the nonterminal occurrence has only dependencies with attributes of the left-hand side nonterminal.

Computations with the second nonterminal occurrence, *ntocc1* at line (4.b), can be skipped if its attributes are not connected with those of *ntocc0*.

6.3 Skipping pairs

Algorithm NOOPTS computes many triples at line (5) that contain no remote dependencies. It turns out that we can skip many of these computations by inspecting the graphs in the pairs and the dependencies in $D(pr)$. The tests that detect these non-productive combinations are performed at lines (4.a) and (4.d) of the algorithm.

To predict whether a combination does not introduce a remote dependency we maintain the ***kind*** of a pair-graph (a graph in a pair). The kind of a pair-graph is either *unconnected*, *up*, *down* or *general*. The kind of a pair-graph $([X,U]),r)$ is defined as follows. The kind of graph r is *unconnected* if r does not contain a dependency among the attributes of X and U; its kind is *up* if all dependencies among attributes of X and U run from U to X; its kind is *down* if all dependencies among attributes X and U run from an attribute of X to an attribute of U; all other graphs have as kind *general*.

These graph kinds are ordered as follows. The kind *unconnected* is smaller than all other kinds and the kind *general* is greater than all other kinds. An equivalent definition of the order among graph kinds is: $k < l$ iff a graph g of kind k can be turned into a graph of kind l by adding edges to g.

For each combination of *upper_pair* (line (1) of the algorithm) and production *pr* (line (2)) we define the kind of dependencies among two different right-hand side nonterminal occurrences. Let production *pr* be $pr : W_0 \rightarrow W_1, \ldots, W_i, \ldots, W_j, \ldots, W_n$ and let $([X, W_0], r)$ be the upper pair.

The kind of dependencies between W_i and W_j is either *unconnected*, *LtoR*, *RtoL* or *general*. The kind is *unconnected* if there are no dependencies from an attribute occurrence of W_i to one of W_j and if there are no no dependencies from an attribute occurrence of W_j to one of W_i; the kind is *LtoR* if all dependencies among attribute occurrences of W_i and W_j run from W_i to W_j; the kind is *RtoL* if all dependencies among attribute occurrences of W_i and W_j run from W_j to W_i; all other dependencies have kind *general*.

The kinds of dependencies among right-hand side nonterminal occurrences are ordered like those of pair-graphs. The kind *unconnected* is smaller than all other kinds and *general* is larger than all others.

The parallelism detector is optimized by skipping combinations of graphs that do not introduce a remote dependency. The decision to skip a combination is based solely on the kinds of the graphs involved. Table 1 lists all combinations of pair-graph kinds that do not introduce a remote dependency. The column labelled "line" indicates where in the algorithm the optimization can be applied. As an example, consider the case where the pairs *left* (line 4.c) and *right* (line 4.d) both have kind *up*. Combining these pairs with those from a production introduces no remote dependencies, because at least one of the pairs must have an edge that runs down. Stated formally one of the pairs must have the form $([G,H],r)$, where r contains an edge $G.\alpha \rightarrow H.\beta$.

The overhead of computing the kinds of pair-graphs is negligible. First, the cost of computing the kind of a pair-graph is linear in the number of edges in the graph. Second, the kind can be stored with the graph, thereby making it unnecessary to recompute it each time the graph is used.

6.4 Skipping dependent nonterminals

The last optimization is applicable if one only wants to compute remote dependencies among nonterminals instead of computing all triples. In that case combinations of pairs *left* $= ([F,G],r)$ and *right* $= ([K,L],s)$ can be skipped once it has been determined that

left	right	deps	line
$\leq$ uncon	-	-	4.c
-	$\leq$ uncon	-	4.d
down	down	-	4.d
up	up	-	4.d
$\leq$ down	-	$\leq$ LtoR	4.c
$\leq$ up	-	$\leq$ RtoL	4.c
-	$\leq$ down	$\leq$RtoL	4.d
-	$\leq$ up	$\leq$LtoR	4.d

Table 1: Table with useless combinations, based on the kinds of graphs. Columns "left" and "right" are the kinds of the pair-graphs in the algorithm; column "deps" is the kind of dependency between *ntocc0* and *ntocc1*. Column "line" gives the line-number in the algorithm where this optimization can be applied.

there exist a remote dependency among G and L. This optimization can be applied at line (4.d).

7 Effects of optimizations

We implemented three parallelism detectors, which we call NOOPTS, HALFOPTS and ALLOPTS. NOOPTS is the parallelism detector as defined in section 5. HALFOPTS is the parallelism detector that applies all optimizations from section 6, but that does not skip dependent nonterminals, i.e. it does not apply the optimization from section 6.4. ALLOPTS applies all optimizations, including the one from section 6.4.

We applied the three parallelism detectors to three existing attribute grammars. The grammars are an attribute grammar for the semantic analysis of Pascal, an attribute grammar for the semantic analysis of occam [8] and one for the semantic analysis of LOTOS [11]. All three grammars have a respectable size. The following table lists, for each grammar, the number of nonterminals and productions in the abstract syntax [1] and the number of lines of the ssl sources.

grammar	$\|N\|$	$\|P\|$	#lines
Pascal	79	203	7025
LOTOS	96	201	22125
occam	83	292	6947

We measured the effects of the optimizations in three ways. For each combination of grammar and parallelism detector we measured the execution time of the triple computations and we counted the number of computed triples. We also counted the number of computed triples without a remote dependency, since the optimization try to reduce this number.

[1] A nonterminal is part of the abstract syntax if it is derivable from the root of the attribute grammar. A production is part of the abstract syntax if its left-hand side nonterminal occurs in the abstract syntax.

We measured the time for computing triples on an otherwise unloaded Hewlett Packard 9000/750. Table 2 gives the execution times for the computation of triples.

	time			speedup	
grammar	NOOPTS	HALFOPTS	ALLOPTS	HALFOPTS	ALLOPTS
Pascal	154.5s	35.6s	3.2s	4.3	48.3
LOTOS	346.7s	256.5s	42.6s	1.3	8.1
occam	418.3s	44.4s	2.6s	9.4	160

Table 2: Times in seconds for the computation of triples. Speedups are with respect to NOOPTS

We counted the number of triples computed by each of the parallelism detectors, i.e. the number of times that line (5) of the algorithm was executed. Table 3 gives the number of computed triples and the relative improvement of HALFOPTS and ALLOPTS over NOOPTS.

	#computed triples			relative improvement	
grammar	NOOPTS	HALFOPTS	ALLOPTS	HALFOPTS	ALLOPTS
Pascal	306570	39926	4154	7.7	73.8
LOTOS	131310	85802	16889	1.5	7.8
occam	1293032	120207	4051	10.8	319.2

Table 3: Number of computed triples

We also counted the number of computed triples that do not contain a remote dependency. The aim of the optimizations is to detect such triples without executing line (5) of the algorithm. Table 4 gives the number of these nonproductive combinations. It can be seen

	#nonproductive computations			as percentage of NOOPTS	
	NOOPTS	HALFOPTS	ALLOPTS	HALFOPTS	ALLOPTS
Pascal	275243	8599	2183	3.1%	0.8%
LOTOS	90965	45457	13915	49.9%	15.3%
occam	1229717	56892	2497	4.6%	0.2%

Table 4: The number of nonproductive combinations computed by the three algorithms. For HALFOPTS and ALLOPTS the table also lists the number of nonproductive combinations as a percentage of the same number of NOOPTS.

that a large percentage of nonproductive combinations is skipped by the optimized algorithms. For the Pascal and occam grammar algorithm HALFOPTS computes less than 5% of the useless combinations computed by NOOPTS; algorithm ALLOPTS even computes less than 1% of NOOPTS' combinations. Even for the LOTOS grammar, ALLOPTS skips almost 85% of the non-productive combinations of NOOPTS.

The effect of our optimizations is further illustrated by counting the number of productive combinations, i.e. computations of triples with a remote dependency. Table 5 gives these

Grammar	Algorithm	#computed	#productive	%productive
Pascal	NOOPTS	306570	31327	10.2
Pascal	HALFOPTS	39926	31327	78.5
Pascal	ALLOPTS	4154	1971	47.4
LOTOS	NOOPTS	131310	40345	30.7
LOTOS	HALFOPTS	85802	40345	47.0
LOTOS	ALLOPTS	16889	2974	17.6
occam	NOOPTS	1293032	63315	4.9
occam	HALFOPTS	120207	63315	52.7
occam	ALLOPTS	4051	1554	38.4

Table 5: Number of computed and number of productive triples.

numbers. Algorithm HALFOPTS illustrates the effects of using the kinds of pair-graphs. Not only does HALFOPTS compute much less triples than NOOPTS, it is also much more effective. Analysing Pascal, for example, only 22.5% of the triples computed by HALFOPT does not contain a remote dependency.

8 Differences among the grammars

Our optimizations work remarkably well for the Pascal and the occam grammar. They work reasonably for the LOTOS grammar. The LOTOS grammar is probably one of the largest attribute grammars that were ever written. Its nonterminals contain much more attributes than the grammars for Pascal and occam. Table 6 lists the average size of the parameter to the transitive closure operation. These sizes influence the number of triples

Grammar	NOOPTS	HALFOPTS	ALLOPTS
Pascal	36.5	48.5	31.4
LOTOS	92.1	98.1	79.1
occam	23.8	24.6	13.7

Table 6: Average size of relations whose transitive closure is computed. Size is given in the number of elements in the underlying set.

computed per second. Parallelism detector ALLOPTS, for example, computes more then 1500 triples per second when analysing the occam grammar and less than 400 triples per second when analysing the LOTOS grammar.

9 Further improvements?

It seems unlikely that we can reduce the number of computed triples in a significant way without using costly operations. More clever ways of predicting whether a triple will contain a remote dependency probably requires computing a transitive closure, the very operation we want to avoid.

Improving the time for parallelism detection with another order of magnitude is not possible for all three grammars. When analysing the Pascal grammar, ALLOPTS spends only one third of her time on computing triples, and the computation of triples for occam takes only one sixth of the analysis time. The remainder of the time is used for parsing, semantic analysis and the computation of singletons and pairs. The LOTOS grammar is different in this respect because triple computation takes 84% of total analysis time. As can be seen in table 5, only 17% of the computed triples contain a remote dependency.

We have implemented versions of the parallelism detectors that compute less precise singletons, pairs and triples. These versions compute at most one singleton for each nonterminal. The singleton $([X], r)$ contains the union of all dependencies that would be computed by the precise versions of the parallelism detectors. Likewise they also compute at most one pair and one triple for each combination of two and three nonterminals. Surprisingly, for all three example grammars, the imprecise version of ALLOPTS computes the same set of remote dependencies among nonterminals. Table 7 lists the number of triples computed by the imprecise version of ALLOPTS.

	#computed triples		relative improvement
grammar	NOOPTS	ALLOPTS imprecise	ALLOPTS imprecise
Pascal	306570	2942	104.2
LOTOS	131310	6754	19.4
occam	1293032	2091	618.4

Table 7: Number of computed triples by an imprecise version of ALLOPTS

10 Related work

Efficient implementations of the circularity test for attribute grammars have been widely studied [1, 2]. Our experience is that a relative straightforward implementation of the circularity test is fast enough. Our implementation, that uses the weak stability optimization from [1], takes from 0.2 seconds for occam to 0.5 seconds for LOTOS.

Möncke and Wilhelm [9] have developed a general framework for grammar analysis. Their framework, however, is concerned with computing properties for a *single* nonterminal. They do not discuss cases, like our algorithms, where one computes relations for pairs and triples of nonterminals.

11 Acknowledgements

Henk Penning implemented a library of fast graph operations that is used in the parallelism detectors. I came up with the optimizations when I explained to him the need for efficient graph operations.

References

[1] Pierre Deransart, Martin Jourdan, and Bernard Lorho. Speeding up circularity tests for attribute grammars. *Acta Informatica*, 21:375–391, 1984.

[2] Martin Jourdan and Didier Parigot. More on speeding up circularity tests for attribute grammars. Rapports de Recherche 828, INRIA, April 1988.

[3] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[4] D.E. Knuth. Semantics of context-free languages (correction). *Math. Syst. Theory*, 5(1):95–96, 1971.

[5] Matthijs F. Kuiper. *Parallel Attribute Evaluation.* PhD thesis, University of Utrecht, November 1989.

[6] INMOS Limited. occam *2.* Prentice Hall, 1988.

[7] Ulrich Möncke and Reinhard Wilhelm. Grammar flow analysis. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, LNCS 545, pages 151–186, 1991.

[8] Peter van Eijk. Attribute grammar applications in prototyping LOTOS tools. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Application*, LNCS 461, pages 91–100, 1990.

[9] W.M. Waite and G.Goos. *Compiler Construction.* Springer, 1984.