Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 307-319

INDEPENDENT AND-PARALLELIZATION OF LOGIC PROGRAMS USING STATIC SLICING

J. Paakki (Helsinki, Finland)
T. Gyimóthy and T. Horváth (Szeged, Hungary)

Abstract. A technique for independent and-parallelization of logic programs is presented. The method applies static program slicing originally developed for algorithmic debugging of logic programming. The slicing is based on the idea of isolating dependent and independent program parts. By analyzing the (transitive) dependency between positions in the logic program the slicing is able to statically extract such subgoals that can be full executed in parallel.

1. Introduction

Program slicing is a standard technique applied in the dependency analysis of programs. Intuitively, a program slice with respect to a specific variable (at some programe point) contains all those parts of the program that may affect the value of the variable, or that may be affected by the value of the variable. In the first case, the analysis technique is conventionally referred to the as *backward slicing*, and in the second case as *forward slicing*, corresponding to the data flow direction of producing the slice.

Slicing technique can be classified into *static* and *dynamic* ones as well. Static slicing is based on an analysis of the programe without executing it. As the consequence, a static slice is valid for all execution of the programe but it may be imprecise by containing data flow which is actually not manifested during a perticular execution. Dynamic slicing involves the program's execution and hence extracts the precise data flow. On the other hand, a dynamic slice may be different for each execution and shall therefore always be produced separately whenever running the program.

Slicing has usually been applied on imperative languages, such as Pascal [17], [15]. Such languages are amenable to analyzing the dependences between variables, thanks to explicitly expressing the data flow within the program in terms of assignments and input/output parameters. This does not hold for logic programming languages where the notions of data flow and data dependences are rather implicit and thus harder to grasp. Indeed, slicing is often mentioned as one of the modern program analysis techniques still missing from the logic programming paradigm [8].

Slicing in connection with logic programming can be applied in same areas as mentioned above for traditional cases. The problem of slicing logic programs is, however, much more complicated than normally. Before a slice over a logic program can be produced, its implicit data flow has to be approximated. And for approximating the data flow, the implicit input/output data dependences have to be extracted from the program.

The program slicing is based on the *annotation inference* technique introduced in [3]. Given an annotation, *dependence graphs* are constructed for the logic program. Our slicing technique is based on *program dependence graphs*, a concept recently introduced as a unified representation for static program analysis and widely applied as the basis of slicing imperative languages (see e.g. [12]).

So far, the main application area of slicing in general has been algorithmic debugging [9].

Because slicing is based on the idea of isolating dependent and independent program parts, our techique could also be applied in the and-parallelization of logic programs. By analyzing the (transitive) dependency between positions in the program or in the proof tree, slicing is able to statically extract such program parts that can be fully executed in parallel, in the same manner as suggested e.g. in [13] and [19] under an abstract interpretation scheme. The independence information could then be exploited by a compiler to generate parallel code for the program.

2. Preliminaries

Attribute grammars were originally introduced as a definition formalism for context-free languages and compilers [16]. In this section we present the concepts and notations that are needed for establishing a relation from logic (Prolog) programs to attribute grammars [6], [18], [7]. In what follows, P will denote a logic (Prolog) program, and T will denote a *proof tree* in the conventional sense where each node stands for a resolved atom, and each subtree

with $p(...)$ as the root and $p_1(..), ..., p_n(...)$ as its sons coresponds to an instance of a clause $p(...) : -p_1(...), ..., p_n(...)$ in the program P. The root node of a proof tree corresponds to a query atom.

We let the clauses of P be numbered $C_1, c_2, ...$, and the atoms of every clause be numbered 0,1,..., and denote the 0:th atom is the head, the 1st atom is the leftmost atom in the body, and so forth. Let q be a predicate in P. We let the argument positions of q be numbered 1,2,..., and cenote the k:th argument position in q with q_k. Let $\mathcal{A}rgpos(q)$ be the set of argument positions in q, and let $\mathcal{A}rgpos(P) = \bigcup_{q \in P} \mathcal{A}rgpos(q)$.

Let C be a clause in P of the form $a_o : -a_1, ..., a_n$. We can now unambiguously refer to the k :th argument position in a_j with the tuple (C, j, q, k), where q is the predicate symbol of a_j. We call such a tuple a *(program) position*. Let $Pos(P)$ denote the set of positions in P, and $Pos(C)$ the set of positions in C. We will not make a distinction between positions and terms contained in them; e.g. if γ is a position, we write $\gamma\theta$ instead of $t\theta$, where t is the term in γ.

For applying techniques of attribute grammars on logic programs, we must be able to model the data flow within a program. The idea is that some of the prdicatearguments in P are *annotated*, either as *inherited* ($\downarrow$) or *synthesized* ($\uparrow$), that is: there is a function (an *annotation*) $\mu : \mathcal{A}rgpos(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \square\}$, where $\updownarrow$ is read *dual*, and $\square$ is read *unannotated*. The flags $\downarrow, \uparrow, \updownarrow, \square$ are called *modes*. An annotation is *partial* if some positions are dual or unannotated. Intuitively, inherited arguments represent input information for a predicate, synthesized arguments represent output information, dual arguments represent information that may be input or output depending on the context, and unannotated arguments represent auxiliary information whose direction is not relevant for the user. For an n-ary predicate q, the annotation $\mu(q_k) = m_k$ for each $k = 1, ..., n$ is denoted by $q : (m_1 ... m_n)$. Notice the essential difference to attribute grammars: in addition to the inherited and synthesized modes, the special properties of logic programs call for dual and unannotated modes as well.

Another important concept is the notion of *input* and *output* arguments of a *clause* C : If$\mu(\mathrm{q_k}) = \downarrow$ and a_j is the head atom of C, or if $\mu(q_k) = \uparrow$ and a_j is a body atom in C, we call (c, j, q, k) an *input* position. If $\mu(q_k) = \uparrow$ and a_j is the head atom in C, or if $\mu(q_k) = \downarrow$ and a_j is a body atom in C, we call (C, j, q, k) an *output* position. The intuitive explanation for these names is that data is brought *in* to a clause through the input positions, and sent *out* through the output positions. Notice that dual and unannotated positions do not express a definite direction of data flow, and that is why they are not strictly classified as input or output ones. Let $\mathcal{I}(C)$ and $\mathcal{O}(C)$ denote the input

and output positions of C, respectively. We denote $\mathcal{I}(C) = \bigcup_{C \in P} \mathcal{I}(C)$ and $\mathcal{O}(P) = \bigcup_{C \in P} \mathcal{O}(C)$.

We write $C_1 \rightarrow C_2$ if the clause C_2 can be called from C_1, i.e. there is an atom a in the body of C_1 such that unification of a and the head of C_2 (with variables renamed) succeeds. The relation $\rightarrow$ is called the *static call graph*.

3. Dependence analysis and program annotation

A logic program does not explicitly specify the direction of data flow within it. The data dependences within a logic program can, however, be approximated with techniques based on analyzing the *sharing* of logical variables: if two terms of a clause have a common variable, they are dependent and (possibly) represent a data flow.

The static analysis technique in our slicing method has been originally developed for groundness analysis of (functional) logic programs [2], [3].

In a logic program, information can be passed in two ways: either *within* a clause (between two arguments sharing a variable), or *between* two clauses (through unificaton at an SLD-resolution step). Let us assume that there exists an annotation $\mu : \mathcal{A}rgpos(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \square\}$. As for attribute grammars, this information can be used to extract the *direction* of information flow: from input positions to output positions within a clause, and from output positions to input positions at unification. Recall from Chapter 2 that input and output positions are defined in terms of inherited ($\downarrow$) and synthesized ($\uparrow$) positions only and do not consider dual ($\updownarrow$) or unannotated ($\square$) positions whose data-flow directon is statically unknown. To reflect the data flow, we introduce the notions of *local dependence graph* $\sim_C$ for each clause C and *transition graph* $\sim_{C,D}$ for each pair of clauses C and D.

Definition 3.1. [Local dependence graph] *For each clause C, the local dependence graph $\sim_C \subseteq \mathcal{I}(C) \times \mathcal{O}(C)$ is defined as follows:*

$$\beta \sim_C \gamma \quad \textit{if } \beta \textit{ and } \gamma \textit{ have at least one common variable}$$

Definition 3.2. [Transition graph] *Let C and D be two clauses, b_o the head atom in D, and a_j a body atom in C such that a_j and b_o unify. The tradition graph $\sim_{C,D}$ on $\mathcal{O}(C \cup D) \times \mathcal{I}(C \cup D)$ is defined as follows:*

$$\gamma \sim_{C,D} \beta \, iff \begin{cases} \gamma = (C, j, q, k) \\ \beta = (D, 0, q, k) \\ \mu(q_k) = \downarrow \end{cases} \quad or \quad \begin{cases} \gamma = (D, 0, q, k) \\ \beta = (C, j, q, k) \\ \mu(q_k) = \uparrow \end{cases}$$

As for attribute grammars, the transition graph specifies how clauses and their arguments are "pasted" together in a proof tree. If the connected positions are inherites, data flowa from the (calling) body atom of clause C into the (calles) head atom of clause D. The flow is reverse for synthesized positions, from the (called) head atom of D into the (calling) body atom of C.

The total data flow for a logic program can be modeled by combining the local dependence graphs with the transition graphs. This *global dependence graph* is an approximation of the data flow in any proof tree for an operationally complete program.

Definition 3.3. [Global dependence graph] *The global dependence graph $\sim_G$ is defined as follows:*

$$\sim_G = \bigcup_{C \in P} \sim_c \cup \bigcup_{C,D \in P} \sim_{C,D}$$

We let $\sim_G^8$ denote the transitive and reflexive closure of $\sim_g$.

Since it may be hard for the user to provide a proper annotation, it can be *automatically inferred* [2], [3].

After obtaining the initial annotation from the system the user may annotate the program further in terms of its intended use by assigning amode on those positions that are not constrained into a fixed data flow. That is, the user may define one of the modes $\downarrow, \uparrow$, *updownarrow* for each predicate position which is still unannotated ($\square$). Typically the user annotates the top-level predicate, but in general he/she is free to annotate as many (or as few) positions that he/she thinks are relevant for his/her intended model of the program. The degree of annotation, however, may significatly influence the size of the slice: The more inherited ($\downarrow$) and synthesized ($\uparrow$) annotations, the more precise a slice.

4. Static slicing

Semantic analysis of programs requires talking into account both data flow and control flow aspects. One program representation supporting the analysis of these both are *program dependence graphs* that have been suggested for software engineering tasks such as program understanding, program integration, program differencing, debugging, and testing [12], [1]. Our basic backward slicing method also is based on the concept ofprogram dependence graphs that model the data flow within a logic program in terms of local dependence graphs

and variable sharing, and the control flow in terms of transition graphs and unification.

The program dependence graph can be condtucted simultaneously with annotating the program. For slicing it is necessary to know the data dependences on dual and unannotated positions as well, because they also may be the source of incorrect results even though the direction of their influence is not statically known. Therefore, the program dependence graph is a superset of the global dependence graph, by including a conservative two-directional data flow for all the dual and unannotated positions. Recall also that, unlike inherited and synthesized arguments, dual and unannotated arguments need not be ground. Hence, incomplete or undefined information is modeled in our method by those modes.

Definition 4.1. [Program dependence graph] *Let C be a clause, and let $\mathcal{U}(C)$ denote the set of dual and unannotated positions of C. Let $\beta \in \mathcal{U}(C)$ and $\gamma \in \mathcal{P}os(C)$. The graph $\leftrightarrow_C \subseteq \mathcal{P}os(C) \times \mathcal{P}os(C)$ is defined as follows:*

$$\beta \leftrightarrow_C \gamma \text{ and } \gamma \leftrightarrow_C \beta \text{ iff } \beta \text{ and } \gamma \text{ have at least one common variable.}$$

Let C and D be two clauses, b_o the head atom in D, and a_j a body atom in C such that a_j and b_o unify. Let $\beta = (C, j, q, k) \in \mathcal{U}(C), \gamma = (D, 0, q, k) \in \mathcal{U}(D)$. The graph $\leftrightarrow_{C,D} \subseteq \mathcal{U}(C \cup D) \times \mathcal{U}(C \cup D)$ is defined as follows:

$$\beta \leftrightarrow_{c,D} \gamma \text{and} \gamma \leftrightarrow_{C,D} \beta.$$

The program dependence graph $\sim_P$ is defined as follows:

$$\sim_P = \sim_G \cup \bigcup_{C \in P} \leftrightarrow_c \cup \bigcup_{C,D \in P} \sim_{C,D} .$$

We let $\sim_P^*$ denote the transitive and reflexive closure of $\sim_P$.

The program slice is defined over the program dependence graph (PDG) of a logic program. We will define a program slice, called $p - slice$. The idea behind a p-slice (Definition 5.1) is that if the p-slice with respect to a position r does not contain a position q then this holds for each proof tree of the program. More precisely, for any proof tree T holds that if γ and β are the tree positions corresponding to r and q, respectively, then γ does not (transitively) depend on β. So, we can characterize that r is fully independent of q in the p-slice case.

In the following P will denote a logic program and G will denote a PDG of P constructed from a consistent annotation of P. Program positions of P are denoted by letters p, q, r.

Definition 4.2 [p-slice] *A p-slice over G with respect to r is a subgraph of G, such that a note $q \in G$ is in the p-slice for r iff there is a directed path in G from q to r.*

Definition 4.3 [Set of atoms constructed from a p-slice] *Let s be a p-slice of the program P for position r. The set $P'(s)$ constructed from s is a subset of the atoms of P, such that an atom l is in $P'(s)$ iff l contains a position in s.*

Using p-slices a strong independence for the atoms of a program can be introduced. An atom l_1 is strongly independent of an atom l_2 iff no atom set $P'(s)$ contains l_2, where $P'(s)$ is a p-slice for a program position in l_1.

5. Independent and-parallelism

Parallelization of logic programs is an active research area [11]. The intuitive meaning of the parallelization of logic programs is the following: the independent goals in a given resolvent must be determined and then executed parallelly in independent environments. Three main types of parallelization have been investigated so far:

(i) The *independent and-parallelism* is based on the "divide-and-conquer" paradigm.

(ii) The *or – parallelism* is used when more than one clauses from the program can be unified with the same atom from the goal to be executed.

(iii) The *dependent and – parallelism* is applied when two or more atoms from the goal share the same variable.

In this section we present a technique for the independent AND-parallelization of logic programs based on static slicing. The independent AND-parallelization technique has recently attained growing interest, thanks to its obvious soundnesss and conceptual simplicity when compared to other approaches, such as *dependent AND*-parallelism. An extensive general introduction to the technique is given in [11]. In independent AND-parallelism goals may be executed concurrently only if they cannot access common variables. To utilize concurrency to the optimal limit, this nonexistence of shared common variables calls for dynamic checking. However, since such runtime checks would obviously be rather numerous and therefore expensive, the presented approaches usually rely on a *static approximation* of sharing which totally eliminates the dynamic independence checks.

We citate some omportant concepts and notations from [11]. The goals g_1 and g_2 are said to be strictly independent for a given substitution θ if $g_1\theta$ and $g_2\theta$ are variable disjoint (i.e. $Var(g_1\theta) \cap Var(g_2\theta) = \emptyset$). If the goals g_1 and g_2 are strictly independent for the substitution θ then the parallel execution of $g_1\theta$ and $g_2\theta$ gives correct results. An independence condition is correct w.r.t. strict independence for the goals g_1, g_2 and for the set of substitution Θ if for any substitution $\theta \in \Theta$ it holds that if the condition is true for θ then g_1 and g_2 are strictly independent for θ. An independence condition is said to be locally correct if it is correct for the set of all possible substitutions. An independence condition for the goals g_1 and g_2. Let SVI be the set of pairs (u, v) of variables, where u and v do not belong to the set SVG. In [11] it has been shown that the condition

$$i_cond: \ ground(SVG) \cap indep(SVI)$$

is locally correct, where $ground(x)$ is true if x is ground and $indep(x, y)$ is true if x and y do not share variables (the symbols x and y denote terms or literals). Consider the clause of the form $p_o : -p_1, ..., p_n$. In the sequential left-to-right execution the execution of the literal p_i must preceed the execution of p_j if and only if $1 \leq i < j \leq n$. Hence, a precedence relation $<_\pi$ can be defined on the set of literals occurred in the body as follows:

$$p_i <_\pi p_j \quad iff \quad 1 \leq i < j \leq n.$$

Applying the independence condition i_cond defined above, this relation can be relaxed as follows:

$$p_i <_{\pi'} p_j \quad iff \quad p_i <_\pi p_j \ and$$

$$ground(SVG(p_i, p_j)) \ or \ indep(SVI) \ is \ false$$

Clearly, if $p_i <_{\pi'} P_j$ then the execution of p_i must preceed of the execution of p_j, otherwise p_1 and p_2 can be executed in parallel.

Example 5.1. *Consider the clause* $p(x, y) : -q(x), r(y, z), s(x, y)$. *By the definition of* $<_\pi, q(x) <_\pi r(y, z), q(x) <_\pi s(x, y)$ *and* $r(y, z) <_\pi s(x, y)$. *Using the definition of* $<_{\pi'}$, *we can relax* $<_\pi$ *in the examples below as follows:*

$(a) q(x) <_{\pi'} r(y, z)$ *iff* $indep(\{x\}, \{y, z\})$ *is false.*

$(b) q(x) <_{\pi'} s(x, y)$ *iff* $ground(x)$ *is false.*

$(c) r(y, z) <_{\pi'} s(x, y)$ *iff* $ground(y)$ *is false or* $indep(\{x\}, \{z\})$ *is false.*

The most obvious case is when all the statically shared arguments of the two atoms are *ground* whenever the atoms are invoked. This typical case of groundness is modeled in our technique simply by having all the shared positions *inherited* in the annotation. Thus, inherited annotations directly

represent one form of independent AND-parallelism. Another groundness-preserving property is to have the positions as *synthesized* and thus ground at the exit of the atoms' excution. Of course, when computing shared output information of different atoms in parallel, it must be separetely checked that the results unify [11]. It is important to notice that the shared positions in the atom must have the same annotation, i.e. either *inherited* or *synthesized.*

For the other annotation modes (dual, unannotated), unsharing is not directly obvious but must be checked in terms of the relevant slices (and the program dependence graph). If these positions depend on the same position then the corresponding atoms cannot be executed parallelly.

6. Discussion and related work

We have presented a technique for the parallelization of logic programs. We have concentrated especially on the *independent AND* class of the parallelization schemes. The independent AND-parallelization technique has recently attained growing interest, thanks to its obvious soundness and conceptual simplicity when compared to other approaches, such as *dependent AND*-parallelism. An extensive general introduction to the technique is given in [11].

In independent AND-parallelism, goals may be executed concurrently only if they cannot access common variables. To utilize concurrency to the optimal limit, this nonexistence of shared common variables calls for dynamic checking. However, since such runtime checks would obviously be rather numerous and therefore expensive, the presented approaches usually rely on a *static approximation* of sharing which totally eliminates the dynamic independence checks.

The conventional technique for statically inferring the independence information, is *abstrct interpretation*; see e.g. [13] and [19]. Abstract interpretation is close to actual execution by approximating the behavior of a program under an abstract and well-defined domain. Therefore a powerful abstract interpreter can simulate the program's concrete run-time execution closely and provide for quite exact (sharing or alias) information, when selecting a suitable abstract domain. On the other hand, an abstract interpreter is not absolutely static and universal because it has to have a specific goal (query) where to start from.

Our parallelization approach is not based on abstract interpretation but rather on *static slicing* [10]. Moreover, abstract interpretation is absent even from our slicing technique whose roots are in the annotation inference technique originally presented in [3]. The essentiak difference between our scheme and

abstract interpretation is that we do not analyze the program by abstractly executing it but rather by employing a purely static analysis technically based on attribute grammars [16]. Staying solely on a static analysis framework makes our approach more general since there is no need for knowing how the query possibly looks like; the results apply for every excution pattern of the program. On the other hand, abstract interpretation may produce more precise results due to approximating the program's behavior with respest to a specific execution.

The principles of annotation inference and slicing conform well with the fundamentals of independent AND-parallelism. Our notionof slice is based on a *program dependence graph* that includes all the possible dependencies between program positions and the vaariables therein. Hnece, two atoms are certainly independent and can be solved in parallel if they do not expose any mutual data flow within the program dependence graph. Having a total representation of program dependencies is another novel feature which makes our technique different from those based on a conventional, more local abstract interpretation.

Recall that by the definition of independent AND-parallelism, two body atoms can be excuted concurrently if they do not share a varable at run-time. Notice especially that the atoms may have a common variable in the program, as far as the sharing disappears when executing the program and execution reaches the atoms. The most obvious case is when all the statically shared arguments of the two atoms are *ground* whenever the atoms are invoked. This typical case of groundness is modeled in our technique simply by having all the shared positions *inherited* in the annotation. Thus, inhaerited annotations directly represent one from of independent AND-parallelism. Another groundness-preserving property is to have the positions as *synthesized* and thus ground at the exit of the atom's execution. Of course, when computing shared output information of different atoms in parallel, it must be separately checked that the results unify [11]. For the other annotation modes (dual, unannotated), unsharing is not directly obvious but must be checked in terms of the relevant slices (and the program dependence graph).

The groundness properties of our technique also share some ideas with those presented in the context of abstract interpretation. For instance, [4] and [5] suggest abstract interpretation for inferring data-dependence and data-flow characteristics for a logic program. That approach, however, doesnot consider program slicing.

In our method both annotation inference and slicing rely on techniques originally developed for attribute grammars. Yet another area where attribute grammars can be exploited as the model is parallelization. Parallelization of attribute grammars has recently been a rather active area of research that has produced techniques both for analyzing the concurrent properties of attribute grammars as well as for implementing and executing them in parallel. E.g. [14]

gives an extensive introduction to the topic. The parallel attribute evalution methods could be applied for parallelizing the execution of logic programs. This approach would, in a straihtforward setting, need a concrete proof tree as the basis [21].

Acknowledgements. The work of the second an third authors was supported by the LPD COPERNICUS Project CP 93:6638 and MKM grant 435/94.

References

[1] **Bates S. and Horwitz S.,** Incremental Program Testing Using Program Dependence Graph., *Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South California, 1993,* ACM Press, 1993, 384-396.

[2] **Boye J., Paakki J. annd Maluszynski J.,** *Dependency-Based Groundness Analysis of Functional Logic Programs,* Research Report LITH-IDA-R-93-20, Department of Computer and Information Science, Linköping University, 1993.

[3] **Boye J., Paakki J. annd Maluszynski J.,** Synthesis of Directionality Information for Functional Logic Programs, *Proc. 3rd Int. Workshop on Static Analysis, Padova, 1993, LNCS* **724**, Springer, 1993, 165-177.

[4] **Debray S. K.,** Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transactions on Programming Languages and Systems,* **11** (3) (1989), 418-450.

[5] **Debray S. K. and Warren D. S.,** Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems,* **11** (3) (1989), 451-481.

[6] **Deransart P. and Maluszyński J.,** Relating Logic Programs and Attribute Grammars, *Journal of Logic Programming,* **2** (1985), 119-156.

[7] **Deransart P. and Maluszyński J.,** *A Granmmatical View of Logic Programming,* The MIT Press, 1993.

[8] **Ducassé M. and Noyé J.,** Logic Programming Environments: Dynamic Program Analysis and Debugging, *Journal of Logic Programming,* **19-20** (1994), 351-384.

[9] **Fritzson P., Gyimóthy T., Kamkar M. and Shahmehri N.,** Generalized Algorithmic Debugging and Testing, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices* **26** (6) (1991), 317-326.

[10] **Gyimóthy T. and Paakki J.**, Static Slicing of Logic Programs, *Proc. 2nd Int. Workshop on Automated and Algorithmic Debugging (AADEBUG '95), St. Malo, France, 1995.*

[11] **Hermenegilo M. V. and Rossi F.**, Strict and Nonstrict Independent AND-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions, *Journal of Logic Programming,* **22** (1) (1995), 1-45.

[12] **Horwitz S. and Reps T.**, The Use of Program Dependence Graphs in Software Engineering, *Proc. 14th Int. Conference on Software Engineering, Melbourne, 1992*, IEEE Computer Society Press, 1992, 392-410.

[13] **Jacobs L. and Langen A.**, Static Analysis of Logic Programs for Independent And Parallelism, [JLP 92], 291-314.

[14] **Jourdan M.**, A Survey of Parallel Attribute Evaluation Methods, *Proc. Int. Summer School on Attribute Grammars, Applications and Systems (SAGA), Prague, 1991, LNCS* **545**, Springer, 1991, 234-255.

[15] **Kamkar M.**, *Interprocedural Dynamic Slicing with Applications to Debugging and Testing,* Linköping Studies in Science and Technology - Dissertations No. **297**, Department of Computer and Information Science, Linköping University, 1993.

[16] **Knuth D. E.**, Semantics of Context-Free Languages, *Mathematical Systems Theory,* **2** (1968), 127-145.

[17] **Korel B. and Laski J.**, Dynamic Slicing of Computer Programs, *The Journal of Systems and Software,* **13** (3) (1990), 187-195.

[18] **Maluszynski J.**, Attribute Grammars and Logic Programs: A Comparison of Concepts, [AIM 91], 330-357.

[19] **Muthukumar K. and Hermenegilo M.**, Compile-Time Derivation of Variable Dependency Using Abstract Interpretation, [JLP 92], 315-347.

[20] **Paakki J.**, *Multi-Pass Evolution of Functional Logic Programs,* Research Report LITH-IDA-R-93-02, Department of Computer and Information Science, Linköping University, 1993.

[21] **Paakki J.**, Multi-Pass Execution of Functional Logic Programs, *Conf. Record of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, 1994.*, ACM Press, 1994, 361-374.

[22] **Paakki J., Gyimóthy T. and Horváth T.**, Effective Algorithmic Debugging for Inductive Logic Programming, *Proc. 4th Int. Workshop on Inductive Logic Programming (ILP-94), Bad Honnef/Bonn, 1994,* ed. S. Wrobel, GMD-Studien **237**, Gesellschaft für Mathematik und Datenverarbeitung, 1994, 175-194.

[23] **Weiser M.**, Programmers Use Slices When Debugging, *Communications of the ACM,* **25** (7) (1982), 446-452.

[24] **Weiser M.**, Program Slicing, *IEEE Transactions on Software Engineering,* **10** (4) (1984), 352-357.

J. Paakki
Department of Computer Science
University of Helsinki
P.O.B. 26
FIN-00014 Helsinki, Finland
paakki@cs.helsinki.fi

T. Gyimóthy
Res.Group on the Theory of Automata
Hungarian Academy of Sciences
Aradi vértanúk tere 1.
H-6720 Szeged, Hungary
gyimi@inf.u-szeged.hu

T. Horváth
Department of Applied Informatics
József Attila University
P.O.B. 652
H-6701 Szeged, Hungary
thorvath@inf.u-szeged.hu