 LiTH-IDA-R-88-02 ISSN-0281-4250

# INSTITUTIONEN FÖR DATAVETENSKAP

## UNIVERSITETET OCH TEKNISKA HÖGSKOLAN LINKÖPING

---

# Inferring Restricted AND-Parallelism in Logic Programs Using Abstract Interpretation

by

Ulf Nilsson



---

RESEARCH REPORT
LOGPRO, January 1988

*Postadress:*
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

*Mailing address:*
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

# Computer and Information Science Department
# Linköping University

## PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

( *Linköping Studies in Science and Technology. Dissertations.* )

*No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981.

*No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981.

*No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982.

*No 94 **Hans Lunell:** Code Generator Writing Systems, 1983.

*No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983.

*No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984.

*No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987.

## Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs. 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986.

No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-based Editing. 1987

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987

LiTH-IDA-R-88-02 ISSN-0281-4250

# Inferring Restricted AND-Parallelism in Logic Programs Using Abstract Interpretation

by

Ulf Nilsson

**Abstract: The paper addresses the problem of inferring restricted AND-parallelism in logic programs. We generalize previous work by Chang to allow automatic compilation of definite clauses into DeGroot's execution expressions. The approach is based on abstract interpretation. A new scheme for abstract interpretation is given which provides a general framework for static analysis of logic programs. The result of the analysis relies on the particular abstract domain. In this paper the choice of domain is used to influence the number of run-time tests needed in the execution expressions. An abstract domain is presented which is a trade-off between precision and time of analysis.**

*Postadress:*
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

*Mailing address:*
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

# Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation

Ulf Nilsson
Department of Computer & Information Science, Linköping University
S-581 83 Linköping, SWEDEN

**ARPA:** urn@ida.liu.se **UUCP:** urn@liuida

**Abstract:** *The paper addresses the problem of inferring restricted AND-parallelism in logic programs. We generalize previous work by Chang [Ch] to allow automatic compilation of definite clauses into DeGroot's execution expressions [DG1]. The approach is based on abstract interpretation. A new scheme for abstract interpretation is given which provides a general framework for static analysis of logic programs. The result of the analysis relies on the particular abstract domain. In this paper the choice of domain is used to influence the number of run-time tests needed in the execution expressions. An abstract domain is presented which is a trade-off between precision and time of analysis.*

## 1 Introduction

One of the main problems encountered in restricted AND-parallelism [Co],[DG1] is to determine if two literal are independent (i.e. contain no common variables) in run-time. This can be done either during run-time or by a compile time analysis. The former approach is suggested by e.g. Conery [Co] and is able to exploit most of the AND-parallelism in a program, but is likely to suffer from inefficiency since such run-time tests are fairly expensive. Chang [Ch], on the other hand, suggested an approach which was entirely based on static compile time analysis that completely eliminates the need for run-time tests. Since such an analysis, by nature, has to be a "worst case" analysis it may happen that some parallelism is not detected.

In [DG1] and in [Ci] execution models were suggested that combine compile-time analyses with *simple* run-time tests for literals which might be independent in run-time but cannot be determined to be so in compile-time. DeGroot suggested an intermediate language of so called *execution expressions* for this.

The contribution of this paper is twofold -- the main objective is to propose a method which enables compilation of logic programs into execution expressions. But we do more than this -- by using *abstract interpretation* run-time properties of the program are inferred that help us to reduce the number of run-time tests which would otherwise be necessary with a naive analysis of individual clauses (as shown below). Secondly, a general framework for abstract interpretation is proposed. We give a scheme which, when augmented with an application-specific domain and operations over the domain, can be used to infer many

different types of run-time properties. Specifically, we give a domain and operations to infer *groundness* and *dependence* (sometimes called *sharing*) information.

To simplify the presentation we restrict our attention to pure Prolog programs. In [DG2] additional problems induced by e.g. side-effects are addressed. The remaining problem of compiling execution expressions into an instruction set of some abstract machine model is discussed in [H].

The paper is organized as follows -- section 2 outlines the concept of DeGroot's execution expression to be used in the sequel. Section 3 presents an abstract interpretation scheme independent of any particular application. We discuss correctness and give a sufficient condition for termination of the algorithm. Section 4 defines an abstract domain for the interpreter and operations over this domain. Section 5 gives an algorithm which uses the information obtained during abstract interpretation to compile the clause-bodies into execution expressions. Section 6 contains conclusions and suggestions for future work.

Throughout the paper we follow essentially the notation and terminology of [L].

## 2 Execution Expressions

Let `L` be a literal, `E1...En` execution expressions and `t1...tm` terms. The language of *execution expressions* proposed by DeGroot [DG1] is defined by the rules[1]:

(1) `L`

(2) `(SEQ E1 ... En)`

(3) `(PAR E1 ... En)`

(4) `(GPAR (t1 ... tm) E1 ... En)`

(5) `(IPAR (t1 ... tm) E1 ... En)`

Operationally, the expression `(SEQ E1 ... En)` reads "first solve `E1` then ... and finally `En`". The third expression reads "solve `E1` and ... and `En` in parallel". In (4) `E1...En` are solved in parallel only if `t1...tn` are all ground at run-time. Finally, in expression (5) `E1...En` are solved in parallel only if `t1...tn` are all mutually independent at run-time.

One aim when introducing the execution expression was to compile bodies of logic programs into *safe* execution expressions. Safe here means that if, in run-time, two literals are solved in parallel, then they are definitely *independent* (i.e. contain no common variable). It turns out that for very simple clauses safeness can easily be achieved. However, for nontrivial clauses it is not so easy to obtain a safe expression, and in any case, if the expression is safe the involved run-time tests are fairly awkward. To be able to execute e.g. the body literals of the clause `p(X,Y,Z):-q(X,Y),r(X,Z)` in parallel we have to test either whether `X,Y` or `X,Z` or `X,Y,Z` are ground in order to guarantee safeness. Such tests could certainly be simplified if we knew something about the run-time behaviour of the program. If we e.g. know in advance that `p/3` is always called with its first argument being ground we could compile the clause into `p(X,Y,Z):-(IPAR (Y Z) q(X,Y) r(X,Z))` and if we know that it is always called with the first two arguments being ground `p(X,Y,Z):-(PAR q(X,Y) r(X,Z))` is safe.

---

1) In [DG1] a sixth expression was introduced but we do not use it herein.

# 3 An Abstract Interpretation Scheme

The objective of this section is to give a general framework for *abstract interpretation* of logic programs. Such schemes have previously been suggested in e.g. [M], [JS] and recently [B] (refer to these and [CC] for definitions of the concepts). Like these, a scheme is given which is independent of any particular application and in the following section we "plug in" a suitable abstract domain and primitive operations over this domain to complete the scheme. Although previous frameworks (including this one) seem to be similar they differ considerably in the way of presentation. Like Bruynooghe we give herein a framework based on "trees" (or rather graphs) which is probably more natural for logic programmers than e.g. the denotational approch in [JS]. However, instead of explaining abstract interpretation as the construction of a finite AND/OR-tree it is explained in terms of computations over a *connection graph.* Since this (finite) graph is given a priori there is no need for the complicated and expensive fixpoint mechanism of [B] to keep the graph finite. Some technical details like variable renaming are not discussed.

Under the procedural interpretation of logic programs a clause-body may be viewed as a procedure definition consisting of a (possibly empty) sequence of procedure calls. With every call we associate two *program points.* The point before a body literal L is called the *calling point of L* and the point after L is called the *success point of L.* Most program points are both calling points and success points, but there is one point which is not a success point -- the leftmost program point in a clause -- and there is one point which is not a calling point -- the rightmost program point in the clause. Since these points are of special importance they are called the *entry* and *exit points of the clause.*

We now would like to associate with every program point an expression which characterizes the state of the computation whenever the "program pointer" passes that particular program point. This is similar to the concept of *assertions* [DM] though we are not specific about the general syntax of our expressions. Usually the expressions will be used for describing specific properties of programs (e.g. modes) and will have different syntax for different properties. Another difference is that assertions are associated with the predicates of a program while our expressions are associated with the program points.

Our expressions will be based on the bindings for the variables in a clause invocation, i.e. the *environment* of the clause. In the case of logic programs, the environment can be seen as a substitution, namely the composition of all mgu's obtained up to that point in an SLD-derivation but restricted to the variables in the clause. To make the substitution total it is extended with identity mappings for unbound variables. Now, obviously control may pass a program point more than once. Not only in different derivations but also in the same derivation. Therefore the expression associated with each program point should "contain" all such environments. Such an expression a called a *summarizing substitution* and we think of it as a *set of substitutions* for the variables of the clause. We use $\Sigma_{\theta,C}$ to denote the domain of summarizing substitutions for the clause C.

In order for the abstract interpreter to yield a useful result it is assumed (but not necessary) that the user supplies some information about the possible set of calls he/she intends to make to the program. To simplify the presentation it is assumed that this is given as *a set of substitutions* and *a single literal call* of

the most general form (i.e. with distinct variables as arguments). Or in other words, a call plus a summarizing substitution. Such a "set of calls" is called a *call pattern.*

We say that a summarizing substitution associated with a program point is *correct* if, in any derivation starting from a goal in the call pattern it holds that whenever control passes that point the composition of the mgu's up to the point and restricted to the variables of the clause is contained in the summarizing substitution associated with that point.

Now refering to figure 1, if we know the value of the summarizing substitution associated with the calling point of a literal L, $\Theta_{call,L}$, its value directly affects the summarizing substitution at all entry points of clauses, C, called by L (i.e. with heads unifiable with L). Informally, an entry point, $\Theta_{entry,C}$, can be computed by *APPLYing* $\Theta_{call,L}$ to L, denoted $L\Theta_{call,L}$. Intuitively, this gives us a set of procedure calls (since $\Theta_{call,L}$ is a set of substitutions). Then by *UNIFYing* the head of the clause with $L\Theta_{call,L}$ we get a set of mgu's and by restricting the mgu's to the variables of the clause we get $\Theta_{call,L}$'s contribution to $\Theta_{entry,C}$. But the clause may of course be invoked by other literals as well and the total value of $\Theta_{entry,C}$ is obtained by taking the *union* of all such contributions.

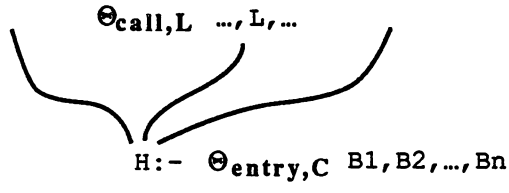


**Figure 1**

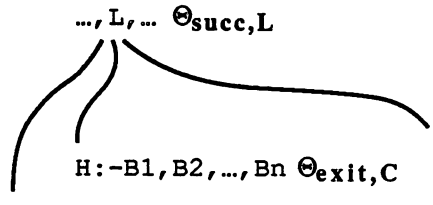


**Figure 2**

Next, consider figure 2. We see that the success point of a literal L depends on the exit points of all clauses which are used to satisfy L. Hence, in order to compute $\Theta_{succ,L}$ we should first *APPLY* $\Theta_{exit,C}$ to the head of the clause and then *UNIFY* this with L. This gives us a set of mgu's which we restrict to the variables of L. However, this only gives us information about the bindings for some variables of the clause where L occurs. Thus, to compute $\Theta_{succ,L}$ we must *combine* the "set of mgu's" with $\Theta_{call,L}$. We call this operation *EXTEND*. Again, L may be satisfied using several clauses and $\Theta_{succ,L}$ is therefore obtained by taking the *union* of all contributions.

The discussion above is the foundation of our abstract interpretation scheme. We now adopt the concept of connection graph (introduced in [K]) to simplify the construction and presentation of the scheme. A *connection graph* is a finite set of clauses and a set of arcs such that there is an arc from a body literal L to a clause head H iff L unifies with H[1].

Since the connection graph is finite it also contains a finite number of program points. With each point we associate two attributes -- a *flag* and a *summarizing substitution.*

1) In [K], each arc also has associated a set of equations obtained when unifying the atoms.

Now before calling our *abstract interpreter* outlined below some preprocessing of this graph is needed. Hence, given a program P and a call pattern (:- L plus a set of substitutions) -- first construct the connection graph for $P \cup \{:- L\}$ and then instantiate the attributes of the graph in the following way:

(1) Set the flag associated with the calling point of L to indicate that it is "updated" and unset all the remaining flags in the graph.

(2) The calling point of L is instantiated to the set of substitutions in the call pattern. The summarizing substitutions of all other points are instantiated to the empty set of substitutions.

---

```
while there exist a Θ marked as updated do
    unmark Θ;
    if Θ is the calling point of L then
        for each C such that L → C do
            Θ_entry,C := Θ_entry,C ⊔ UNIFY(LΘ_call,L,H);
            if Θ_entry,C changed then mark it as updated;
        od
    fi
    if Θ is the exit point of C then
        for each L such that L → C do
            Θ_succ,L := Θ_succ,L ⊔ EXTEND(UNIFY(HΘ_exit,C,L),Θ_call,L);
            if Θ_succ,L changed then mark it as updated;
        od
    fi
od
```

**Figure 3**

---

Figure 3 contains the skeleton of an abstract interpreter which operates on this decorated graph ($L \rightarrow C$ denotes the existence of an arc from literal L to the head of clause C in the connection graph). The interpreter is incomplete in the sense that the primitive operations (unification, application etc) need to be defined. Depending on what run-time property we want to infer we only need to define an appropriate abstract domain and the primitive operations over this domain -- the skeleton itself is generic.

As concerns termination of the algorithm. It is easy to prove that if the domain of the summarizing substitution at each program point is a *lattice of finite height* and $\sqcup$ is defined to be the *least upper bound*-operation, then the execution terminates after a finite number of iterations. In our case summarizing substitutions are sets which are partially ordered under set inclusion with $\sqcup$ being the set join-operation. Alas, this lattice is not (in general) of finite height and hence, the algorithm does not necessarily terminate. However, by a careful construction of domains of *abstract* summarizing substitutions termination can be guaranteed.

# 4 The Abstract Domain

In this section we introduce domains of *abstract summarizing substitutions* which are isomorphic to subsets of summarizing substitutions. We also provide correct definitions of the primitive operations outlined in the previous section. The domain supports analysis of *groundness* and *dependence* (sometimes called *sharing*) in a clause body. Similar domains are outlined e.g. in [JS] and [M] but not in a very uniform way -- they contain one domain for groundness analysis and another for sharing analysis. Of course it is always possible to do what is done in [JS] -- to construct a domain of pairs from the previous two domains. However we feel that a single uniform domain which supports both groundness and independence analysis is more natural.

First we define the set, AT, of *abstract terms* to be the smallest set such that:

- $\{\} \in AT$
- Any variable $X \in AT$.
- Any set of variables $\{X1, \ldots, Xn\} \in AT$.

Let C be a clause with variables $(X1, \ldots, Xn)$, ordered e.g. alphabetically. The set $\Sigma_{AT,C}$ of *abstract summarizing substitutions* over C is equal to the set $AT^n \cup \{\perp\}^n$. An abstract summarizing substitution denotes a set of concrete substitutions. The relationship is expressed in terms of a *concretization* mapping, $\gamma : \Sigma_{AT,C} \rightarrow \Sigma_{\theta,C}$, defined as follows. Let $(t1, \ldots, tn) \in AT^n$ and let $(s1, \ldots, sn)$ denote a substitution which assigns the term $si$ to the variable $Xi$. Let $H_P$ be the set of all ground terms and let $T$ be the set of all (possibly nonground) terms, then:

- $\gamma(\perp, \ldots, \perp) = \emptyset$
- $\gamma(t1, \ldots, tn) = \{ (s1, \ldots, sn) \mid$
  (1) if $ti = \{\}$ then $si \in H_P$, and
  (2) if $ti = X$ then $si = X$, and
  (3) if $ti = \{Y1, \ldots, Ym\}$ then $si \in T$, and
  (4) if $ti$ and $tj$ contain no common variables then $si$ and $sj$ contain no common variables $\}$

NOTE: The case (1) actually is a special case of (3).

**EXAMPLE:** Let C be a clause with variables $(X, Y, Z)$ and consider the following abstract summarizing substitutions associated with a program point in C:

- $\gamma(\{X1, X2\}, \{X1\}, \{X2\}) = \{ (s1, s2, s3) \in T^3 \mid s2 \text{ and } s3 \text{ are independent} \}$. That is, if the expression is correct, then $Y$ and $Z$ will always be independent at this program point. However, $X$ and $Y$ *may* contain a common variable and so may $X$ and $Z$.
- $\gamma(\{X1\}, \{\}, \{X1\}) = \{ (s1, s2, s3) \mid (s1, s3) \in T^2 \text{ and } s2 \in H_P \}$. If the expression is correct then $Y$ is always bound to a ground term and is thus not dependent on any other variable. However, $X$ and $Z$ may be dependent.
- $\gamma(X, \{X1\}, \{X, X1\}) = \{ (X, s2, s3) \in T^3 \mid (s2, s3) \in T^2 \text{ and } s2 \text{ does not contain } X \}$. Here $X$ is always unbound. We also see that $Z$ may be bound, while $Y$ never is bound to a term which contains $X$. Moreover, $Y$ and $Z$ may be bound to mutually dependent terms. ❑

A natural ordering, $\sqsubseteq$, may now be imposed on the elements of $\Sigma_{AT,C}$. Let `(s1,...,sn)` and `(t1,...,tn)` $\in \Sigma_{AT,C}$ and let $\gamma$(`s1,...,sn`) = $\Theta_1$ and $\gamma$(`t1,...,tn`) = $\Theta_2$, then `(s1,...,sn)` $\sqsubseteq$ `(t1,...,tn)` iff $\Theta_1 \subseteq \Theta_2$.

It is straightforward to prove that $\sqsubseteq$ is both reflexive and transitive but unfortunately not anti-symmetric since e.g. $\gamma$(`{X1},{X1}`) = $\gamma$(`{X1,X2},{X1,X2}`) = {`(s1,s2)` $\in T^2$}. On the other hand we can easily construct a partial order by introducing an equivalence relation. Let `t1, t2` $\in \Sigma_{AT,C}$ then $\equiv$ defined by `t1` $\equiv$ `t2` iff $\gamma$(`t1`) = $\gamma$(`t2`) is an equivalence relation and the quotient set of $\Sigma_{AT,C}$ relative to $\equiv$ is a partial order. Formally $\Sigma_{AT,C}/\equiv$ is our abstract domain. However, to have a set of equivalence classes as domain is not very convenient and in practice we pick some *canonical* element in each equivalence class to simplify the computations. Although formally operating in the domain $\Sigma_{AT,C}/\equiv$ , the computations below are carried out over the canonical elements of $\Sigma_{AT,C}$ where a canonical element is obtained as follows.

Let `(t1,...,tn)` $\in \Sigma_{AT,C}$. The corresponding canonical element is obtained by a mapping $C$ which replaces two variables `X1,X2` by a new variable `X` never used before iff `X1,X2` always occur together in the same `ti`'s. For instance, (up to renaming of variables):

$C$(`{X1,X2},{X1,X2,X3}`) = (`{X4},{X4,X3}`)

$C$(`{X1,X2},{X1}`) = (`{X1,X2},{X1}`)

$C$(`{X1,X2,X3},X3,{X1,X2}`) = (`{X4,X3},X3,{X4}`)

It is not very hard to see that $\Sigma_{AT,C}/\equiv$ is a partial order of finite height with (`{X},...,{X}`) being the top element. In fact, it can be shown that it is a *lattice* of finite height. That is, the requirement needed in section 3 to guarantee termination of the abstract interpretation scheme (of course under the assumption that the domain-specific operations are terminating).

Again the user has to provide a call-pattern. It can of course be given in many different forms. Here we assume that it is given as a single (most general) literal, `:-p(X1,...,Xn)`, and an abstract summarizing substitution, `(t1,...,tn)`. However, to simplify the definition below it is assumed that if `ti = X` then `X = Xi`. We now conclude this section with definitions of the application specific operations missing in figure 3. I.e. $\sqcup$, APPLY, UNIFY and EXTEND (in the following we use a function var(`t`) which returns the set of variables occuring in `t`).

**LUB:** Let `(s1,...,sn)` and `(t1,...,tn)` be canonical abstract summarizing substitutions. The least upper bound is defined componentwise by $C$(`s1`$\sqcup$`t1,...,sn`$\sqcup$`tn`) where:

- $\perp \sqcup$ `t` = `t` — for any term `t`
- {} $\sqcup$ `t` = var(`t`) — for any `t` $\neq \perp$
- `X` $\sqcup$ `t` = {`X`} $\cup$ var(`t`) — for any `t` $\neq \perp$ and `t` $\neq$ `X`
- {`X1,...,Xm`} $\sqcup$ `t` = {`X1,...,Xm`} $\cup$ var(`t`) — for any `t` $\neq \perp$

It is also assumed that `t` $\sqcup$ `t` = `t` and `s` $\sqcup$ `t` = `t` $\sqcup$ `s`. NOTE: before carrying out the operation it is recommendable to rename the variable in one of the operands to avoid unnecessary name clashes.

**APPLY:** Let `p(t1,...,tn)` be a literal in a clause C and let $\Theta$ be canonical, then `p(t1,...,tn)`$\Theta$ = `p(t1`$\Theta$`,...,tn`$\Theta$`)` where the application of $\Theta$ to a term is defined by:

- `X`$\Theta$ = var(`t`) if `X/t` $\in \Theta$,

- $c\Theta = \{\}$,
- $f(s1,\ldots,sm)\Theta = var(s1\Theta) \cup \ldots \cup var(sm\Theta)$.

The first and third case actually introduce more "noise" than necessary but, for the sake of clarity and length of presentation, we accept this rather crude approximation.

**UNIFY**: Let `p(t1,...,tn)` and `p(s1,...,sn)` be an abstract and a standard literal respectively. From the definition of application we see that we only have to consider `ti`'s ($1 \le i \le n$) of the form `{X1,...,Xj}` where $j \ge 0$. To avoid unnecessary dependencies due to name clashes we first rename the variables in `p(t1,...,tn)` by new variables never used before. We then construct a set of pairs `{s1/t1,...,sn/tn}` and iterate while one of the following rules match any of the pairs:

- `f(r1,...,rm)/t` replace the pair by `r1/t,...,rm/t`.
- `c/{X1,...,Xi}` where $i \ge 0$. Remove the pair.
- `X/{X1,...,Xi}` and there is another pair `X/{Y1,...,Yj}` with $i \ge j > 0$. Rename, in all pairs, any occurrence of `X1,...,Yj` by a single variable never used before.
- `X/{}` and there is another pair `X/t`. Remove `X/t`.

When no longer any of the rules above are applicable we have an abstract summarizing substitution for all variables in `p(s1,...,sn)`. Actually, when computing the entry point it is also necessary to add pairs of the form `X/X` for all variables, `X`, which do not occur in the head of the clause.

**EXTEND**: The extension-operator was needed to combine the state before a call with the changes in the state arising during the call. Let L be a literal with calling point $\Theta_{call,L}$ and let $\Theta_{mgu}$ be the state change, i.e. the "set of mgu's" obtained in the unification (c.f. figure 3). To compute $\Theta$ = EXTEND($\Theta_{mgu}$,$\Theta_{call,L}$) we first compute $\Theta = \Theta_{call,L} \cup \Theta_{mgu}$ and proceed as follows:

- if `X/{}` $\in \Theta$ remove any other `X/t` from $\Theta$.
- if `X/{X1,...,Xn}` $\in \Theta$ ($n > 0$) and there is another pair `X/{Y1,...,Ym}` $\in \Theta$ ($m > 0$) then rename any occurrence of the variables `X1,...,Ym` in $\Theta$ by a variable never used before.
- if `X/X` $\in \Theta$ and there is another pair `X/t` $\in \Theta$ then remove `X/X` from $\Theta$.

NOTE: Here we assume that $\Theta_{call,L}$ does not contain components of the type `X/Y` where `X` $\neq$ `Y`. If the original call pattern contains no such components then it can be shown (from the definitions above) that no program point will have components of this kind.

**EXAMPLE**: Let L be a body literal `p(f(X,Y),Y,Z)` and C be a clause with head H=`p(f(X,Z),X,Z)`. Let the current values of relevant program points be:

- $\Theta_{exit,C}$ = `{X/{X1}, Y/{}, Z/{X1}}`
- $\Theta_{call,L}$ = `{X/{X1}, Y/{X2}, Z/{X3}, W/{X1}}`
- $\Theta_{succ,L}$ = `{X/{X1}, Y/{X1}, Z/{}, W/{}}`

Let the exit point of C be marked as updated. We now must update the value of the success point of L. First we apply $\Theta_{exit,C}$ to H:

(1) H$\Theta_{exit,C}$ = `p({X1},{},{X1})`

This is renamed into `p({X4},{},{X4})` and unified with L

(2) UNIFY((1), `p(f(X,Y),Y,Z)`) = `{X/{X4}, Y/{}, Z/{X4}}`

We now extend this with the calling point of L:

(3) EXTEND((2),$\Theta_{call,L}$) = {X/{X6}, Y/{}, Z/{X6}, W/{X6}}

Finally we take the least upper bound with $\Theta_{succ,L}$:

(4) $\Theta_{succ,L} \sqcup$ (3) = {X/{X1,X6}, Y/{X1}, Z/{X6}, W/{X6}}

Hence, X may be dependent on any other variable, but Y may only depend on X. We mark the point as updated and continue the analysis according to figure 3. ❑

# 5 Compiling Clauses Into Execution Expressions

After having carried out the analysis of a program a correct abstract summarizing substitution is associated with each program point of the program. Below an algorithm is presented that exploits this information to compile the body of a clause into a *safe* execution expression. Safeness can of course only be guaranteed if the program is invoked in accordance with the given call pattern. First some definitions:

- A set of variables, $\Psi$, is *definitely ground* in $\Theta$ if for all X $\in \Psi$ it holds that X/{} $\in \Theta$. If not, $\Psi$ is *possibly nonground* in $\Theta$.
- A set of variables, $\Psi$, is *definitely nonground* in $\Theta$ if there exists an X $\in \Psi$ such that X/X1 $\in \Theta$.
- Two sets of variables, $\Omega$ and $\Xi$, are *definitely independent* in $\Theta$ if for all Y $\in \Omega$ and all Z $\in \Xi$ it holds that Y/t1$\in \Theta$, Z/t2$\in \Theta$ and var(t1) $\cap$ var(t2) = $\emptyset$. If not the case, we say $\Omega$ and $\Xi$ are *possibly dependent* in $\Theta$.
- Two sets of variables, $\Omega$ and $\Xi$, are *definitely dependent* in $\Theta$ if there exists a Y $\in \Omega$ and a Z $\in \Xi$ such that Y/X1$\in \Theta$ and Z/X1$\in \Theta$.

In the following, let $P_i, \ldots, P_k$ be a sequence of body literals and let $\Theta_i$ denote the abstract summarizing substitution associated with the calling point of $P_i$. Given a clause body $P_1, \ldots, P_n$ and $\Theta_1, \ldots, \Theta_n$ the algorithm in figure 4 compiles the body into a safe execution expression.

In this process two nondeterministic choices are made -- first when partitioning a conjunction. Then given a partition several safe execution expressions can sometimes be produced.

**EXAMPLE:** Consider the clause p(X,Y,Z):-q(X,Y),r(Y,Z) and the abstract summarizing substitution $\Theta_1$ = {X/{X1},Y/{}, Z/{X2}}. Clearly case 2e is applicable but case 2b may also be used since $\Omega$={X} and $\Psi$={Y} are definitely independent and {Y} definitely ground.

If $\Theta_1$ = {X/{X1},Y/{},Z/{X1}} it is no longer safe to execute the goals in parallel without a runtime check (since X and Z are possibly dependent). Therefore the expression (SEQ q(X,Y) r(Y,Z)) or (IPAR (X Z) q(X,Y) r(Y,Z)) is returned.

Finally, if $\Theta_1$ = {X/X,Y/{X1},Z/Z} the variable Y may be nonground (X and Z are always unbound) and we return either (SEQ q(X,Y) r(Y,Z)) or (GPAR (Y) q(X,Y) r(Y,Z)). ❑

In general cases 2a and 2b should be used whenever possible since they do not involve any run-time tests. However, there may be other aspects which suggest that literals should not be satisfied in parallel (e.g. shortage of processing elements). Such considerations are not taken into account here.

**INPUTS:** $P_i, \ldots, P_k$ and $\Theta_i, \ldots, \Theta_k$.

**OUTPUT:** An execution expression

*1)* **If** `i=k` **then** return $P_i$.

*2)* **else** divide $P_i, \ldots, P_k$ into $P_i, \ldots, P_{j-1}$ and $P_j, \ldots, P_k$. Let `E1` and `E2` be the result of applying the algorithm recursively to $P_i, \ldots, P_{j-1}$ and $P_j, \ldots, P_k$ respectively. Let $\Omega$, $\Xi$ and $\Psi$ be the sets of variables occurring only in $P_i, \ldots, P_{j-1}$, only in $P_j, \ldots, P_k$ and only in both respectively. Then apply one of the following rules:

*2a)* **If** $\Psi$ is definitely nonground in $\Theta_i$ **or**
$\Omega$ and $\Xi$ are definitely dependent in $\Theta_i$
**then** return `(SEQ E1 E2)`.

*2b)* **If** $\Psi$ is definitely ground in $\Theta_i$ **and**
$\Omega$ and $\Xi$ are definitely independent in $\Theta_i$
**then** return `(PAR E1 E2)`.

*2c)* **If** $\Psi$ is definitely ground in $\Theta_i$ **and**
$\Omega$ and $\Xi$ are possibly dependent in $\Theta_i$
**then** return `(IPAR (X1...Xm) E1 E2)` where $\{X1...Xm\} \subseteq \Omega \cup \Xi$ such that:
$X \notin \{X1...Xm\}$ iff $X \in \Omega$ ($\Xi$) and X is definitely independent from $\Xi$ ($\Omega$).

*2d)* **If** $\Psi$ is possibly nonground in $\Theta_i$ **and**
$\Omega$ and $\Xi$ are definitely independent in $\Theta_i$
**then** return `(GPAR (X1...Xn) E1 E2)` where $\{X1...Xm\} \subseteq \Psi$ such that:
$X \notin \{X1...Xm\}$ iff X is definitely ground.

*2e)* Return `(SEQ E1 E2)`.

**Figure 4**

# 6 Discussion and Future Work

We proposed an approach to the automatic inference of restricted AND-parallelism in logic programs. Through abstract interpretation, information such as definite groundness and independence between the variables of a clause is obtained. With this information we are able to compile the bodies of clauses into safe execution expressions. Such expressions were introduced in [DG1] but we have never seen any method to compile non-annotated logic programs into such expressions. The approach described above not only compiles logic programs into execution expressions but also, through abstract interpretation, (potentially) reduces the number of run-time tests which would be needed with a naive inspection of each clause.

Our approach is a generalization of what was informally described in [Ch]. Although not formulated as abstract interpretation, Chang performed a static data dependency analysis resulting in *one* so called

*execution graph* for each clause. Roughly speaking, such an execution graph corresponds to an execution expression built from the first three of DeGroot's expressions. We extended this by allowing several "graphs" for each clause. The final decision what graph to pick can in our approach be delayed until run-time.

Our abstract interpretation scheme is based on the concept of connection graph which in a natural way relates to the concept of SLD-derivation, in contrast to e.g. [JS]. This should make our scheme intuitive for people in the logic programming community -- something which is also true for Bruynooghe's AND/OR-trees [B]. However, since our finite graph is given a priori, we do not have to rely on the very complicated (and expensive) fixpoint mechanism needed in [B] to keep the AND/OR-tree finite. This should result in faster analysis but it may also happen that our approach yields a cruder approximation because of this. However, some improvements in precision can be made in our scheme at the expense of execution time -- By copying/unfolding certain parts of the connection graph (in particular non-recursive clauses which are called from several places in the program) the precision of the analysis can be improved. However, this will increase the number of program points and therefore slow down the analysis. Obviously, the choice of abstract domain and primitive operations also affects the precision of the result. We proposed a single domain which supports both groundness and dependency analysis and which is a trade-off between precision and time of analysis.

The full version of this paper will include proofs concerning correctness of the presented abstract interpretation scheme and the specific operations presented in section 4. The implementation of the scheme and algorithm of section 5 is under development. The future work includes experiments with various abstract domains and various applications of abstract interpretation.

## ACKNOWLEDGEMENTS

I would like to thank professor Jan Maluszynski for his continuous support and encouragement. Discussions with Dr. Maurice Bruynooghe greatly influenced the appearence of the abstract interpretation scheme presented herein. The work was sponsored by the National Swedish Board for Technical Development (STU).

## REFERENCES

[B] Bruynooghe, M - *A Framework for the Abstract Interpretation of Logic Programs*. Report CW 62, Katholieke Universiteit, Leuven (1987)

[Ci] Ciepielewski, A - *Parallel SICStus Prolog: Preliminary Definition*. Working Paper, SICS (1986).

[Ch] Chang, J-H - *High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis*. Ph.D. Thesis, Univ of California, Berkely (1985).

[Co] Conery, J - *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. Ph.D. Thesis, University of California, Irvine (1983)

[CC] Cousot, P and R Cousot - *Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs By Construction or Approximation of Fixpoints*. Proc POPL (1977)

[DG1] DeGroot, D - *Restricted AND-Parallelism*. Proc of the Int'l Conf on FGCS, Tokyo (1984) and *Alternate Graph Expressions for Restricted AND-Parallelism*, COMPCON Spring (1985)

[DG2] DeGroot, D - *Restricted AND-Parallelism and Side Effects*. Proc of the 4:th Symposium on Logic Programming, San Francisco (1987).

[DM] Drabent, W and J Maluszynski - *Proving Runtime Properties of Logic Programs*. Proc of TAPSOFT87, Pisa (1987) also to appear in Theoretical Computer Science.

[H] Hermenegildo, M - *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. Ph.D. Thesis, University of Texas at Austin (1986).

[JS] Jones, N and H Søndergaard - *A Semantics-Based Framework for the Abstract Interpretation of Prolog*. In *Abstract Interpretation of Declarative Languages*, ed. Abramsky and Hankin, Ellis Horwood (1987)

[K] Kowalski, R - *Logic For Problem Solving*. North Holland (1979) and *A Proof Procedure Using Connection Graphs*. JACM, Vol 22, Nº 4 (1975).

[L] Lloyd, J - *Foundations of Logic Programming, 2e*. Springer Verlag (1987).

[M] Mellish, C - *Abstract Interpretation of Prolog Programs*. In *Abstract Interpretation of Declarative Languages*, ed. Abramsky and Hankin, Ellis Horwood (1987)

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr:
Report no: LiTH-IDA-R-88-02

Upplagans storlek:
Number of copies:

Datum:
Date: January 1988

Projekt:
Project:

Titel:
Title: Inferring Restricted AND-Parallelism in Logic Programs Using Abstract Interpretation

Författare:
Author Ulf Nilsson

Uppdragsgivare:
Commissioned by:

Dnr.:
Call no:

Rapporttyp:
Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk:
Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord):
Abstract (150 words):

Abstract: The paper addresses the problem of inferring restricted AND-parallelism in logic programs. We generalize previous work by Chang to allow automatic compilation of definite clauses into DeGroot's execution expressions. The approach is based on abstract interpretation. A new scheme for abstract interpretation is given which provides a general framework for static analysis of logic programs. The result of the analysis relies on the particular abstract domain. In this paper the choice of domain is used to influence the number of run-time tests needed in the execution expressions. An abstract domain is presented which is a trade-off between precision and time of analysis.

Nyckelord (högst 8):
Keywords (8):

Logic Programminge, Restricted AND-Pallelism, Abstract, Interpretation

Bibliotekets anteckningar

# A Selection of Previous Research Reports.

LiTH-IDA-R-87-19 **Zebo Peng:** Construction of Asynchronous Concurrent Systems from their Behavioral Specifications. Also in *Proc. of the 10th World Computer Congress*, Dublin, Ireland, Sept. 1-5, 1986.

LiTH-IDA-R-87-18 **Henrik Nordin:** Reuse and Maintenance Techniques in Knowledge-Based Systems.

LiTH-IDA-R-87-17 **Tony Larsson:** Specification and Verification of VLSI Systems Actional Behaviour This is a close version of a paper presented at *the 8th international conference on Computer Hardware Description Languages, CHDL,* 87.

LiTH-IDA-R-87-16 **Shamsul I. Chowdhury:** State of the Art in Statistical Expert Systems. Also in *Proc. of the Conference on Expert Systems and their Applications*, Avignon, May 1987.

LiTH-IDA-R-87-15 **Nils Dahlbäck:** Kommunikation med datorer i naturligt språk - vad är det och vem behöver det?

LiTH-IDA-R-87-14 **Peter Fritzson:** Window System Architectures - an Overview. Also presented at *A Tutorial on Software Development Environments,* Linköping University, Nov 24-25 1986.

LiTH-IDA-R-87-13 **Mats Wirén:** A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. Also in *Proc. of the Third Conference of the European Chapter of the Association for Computational Linguistics,* Copenhagen, April 1—3, 1987.

LiTH-IDA-R-87-11 **Sture Hägglund, Christer Hansson, Tomas Sokolnicki:** Knowledge-Based Training of Case Management Routines and Emergency Procedures. Also in *Proc. of the 3rd Int. Conf. on Expert Systems*, London, 1987.

LiTH-IDA-R-87-10 **Andrzej Lingas:** On Parallel Complexity of the Subgraph Isomorphism Problem.

LiTH-IDA-R-87-09 **Andrzej Lingas, Marek Karpinski:** Subtree Isomorphism and Bipartite Perfect Matching are Mutually NC Reducible.

LiTH-IDA-R-87-08 **Christos Levcopoulos, Andrzej Lingas, Jörg-R. Sack:** Nearly Optimal Heuristics for Binary Search Trees with Geometric Generalizations. Also in *Proc of ICALP'87*, Karlsruhe, July 1987.

LiTH-IDA-R-87-07 **Peter Haneclou:** A Formal Approach to Reason-maintenance Based on Abstract Domains.

LiTH-IDA-R-87-06 **Johan Hultman:** COPPS - A Software System for Defining and Controlling Actions in a Mechanical System.

LiTH-IDA-R-87-05 **Christer Bäckström:** Logical Modelling of Simplified Geometrical Objects and Mechanical Assembly Processes.

LiTH-IDA-R-87-03 **Mikael R.K. Patel:** A Threaded Interpretive Language Supporting Programming in the Large. Also in *Proc. of The sixth Rochester Forth Conference*, University of Rochester, Rochester, New York, June 11-14, 1986.

LiTH-IDA-R-87-02 **Rober Bilos:** A Token-Based Syntax Sensitive Editor. Also presented at the *Workshop on Programming Environments - Programming Paradigms*, Roskilde, Denmark, October 22-24, 1986.

LiTH-IDA-R-87-01 **Wlodzimierz Drabent:** Do Logic Programs Resemble Programs in Conventional Languages? Also in *Proc of the Fourth IEEE Symposium on Logic Programming*, San Francisco, Aug. 31- Sept. 4, 1987.

LiTH-IDA-R-86-44 **Erik Sandewall et al.:** Department of Computer and Information Science Annual Research Report 1986.

LiTH-IDA-R-86-43 **Ola Strömfors:** Editing Large Programs Using a Structure-Oriented Text Editor. Also presented at *the International Workshop on Advanced Programming Environments*, Trondheim, Norway,June 16-18, 1986.

# The Department of Computer and Information Science
# Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *AIELAB - Artificial Intelligence Environments Laboratory,* which conducts research on representation and manipulation of knowledge, problem solving techniques and expert systems with mechanical engineering applications.
- *ASLAB - Application Systems Laboratory,* which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science,* which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratory*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

## Research Reports 1987 and 1988

LiTH-IDA-R-88-02 **Ulf Nilsson:** Inferring Restricted AND-Parallelism in Logic Programs Using Abstract Interpretation.

LiTH-IDA-R-88-01 **Jacek Leszczylowski, Staffan Bonnier, Jan Maluszynski:** Logic Programming with External Procedures: Introducing S-Unification. A shorter version is to appear in *Information Processing Letters*, North Holland.

LiTH-IDA-R-87-26 **Jonas Löwgren:** Applying a Rapid Prototyping System to Control Panel Dialgues.

LiTH-IDA-R-87-24 **Sven Moen:** Drawing Dynamic Trees.

LiTH-IDA-R-87-22 **Lars Ahrenberg, Arne Jönsson:** An Interactive System for Tagging Dialogues. To appear in *Proc. of 14th Conference of the Association for Literary and Linguistic Computing*, Göteborg 1-5 June, 1987.

LiTH-IDA-R-87-21 **Ha**[illegible] tions in Computers an[illegible]

LiTH-IDA-R-87-20 **Ki**[illegible] m Extraction from Se[illegible] paper is to appear in *M*[illegible] *omicro Journal*, 1988.