LiTH-IDA-R-88-44 ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP

UNIVERSITETET OCH TEKNISKA HÖGSKOLAN LINKÖPING

Towards a Framework for the Abstract Interpretation of Logic Programs

by

Ulf Nilsson

RESEARCH REPORT
LOGPRO, December 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

The Department of Computer and Information Science
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 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.
No 174	**Johan Fagerström:** A Paradigm and System for Design of Distributed Systems. 1988.
No 192	**Dimiter Driankov:** Towards a Many-Valued Logic of Quantified Belief. 1988.

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
No 126	**Dan Strömberg:** Transfer and Distribution of Application Programs. 1987
No 127	**Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems. 1987
No 139	**Christer Bäckström:** Reasoning about Interdependent Actions. 1988.
No 140	**Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing. 1988.
No 146	**Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System. 1988.
No 150	**Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behaviour. 1988.

LiTH-IDA-R-88-44 ISSN-0281-4250

Towards a Framework for the Abstract Interpretation of Logic Programs

by

Ulf Nilsson
E-mail: urn@ida.liu.se

Abstract: A new framework for abstract interpretation of logic programs is presented. The idea is to take as the basis a simplified semantics that approximates the standard operational semantics of logic programs but still makes it possible to derive non-trivial abstract interpretations. The relative simplicity of the basic semantics facilitates systematic derivation of abstract interpretations and static analyses of logic programs. Sufficient conditions for termination and correctness of the derived interpreters are provided. The approach is illustrated by inferring groundness information for an example program.

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

Towards a Framework for the Abstract Interpretation of Logic Programs*

Ulf Nilsson
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden

December 13, 1988

Abstract: *A new framework for abstract interpretation of logic programs is presented. The idea is to take as the basis a simplified semantics that approximates the standard operational semantics of logic programs but still makes it possible to derive non-trivial abstract interpretations. The relative simplicity of the basic semantics facilitates systematic derivation of abstract interpretations and static analyses of logic programs. Sufficient conditions for termination and correctness of the derived interpreters are provided. The approach is illustrated by inferring groundness information for an example program.*

1 Introduction

A logic program and a set of goal-clauses together describe a set of computations in a universe of substitutions. If we restrict our attention to *definite* programs and goals and adopt SLD-resolution as our operational semantics, they denote a set of SLD-derivations. A naïve approach to determine whether the program exhibits some run-time property would be to construct the set of all possible derivations and to explicitly test whether the property holds. Needless to say, the method would not, in general, terminate in finite time.

Abstract interpretation (e.g. [AH87], [Bru87], [CC77], [DW88], [MS88]) is a general method which allows us to effectively approximate the semantics of a program. In abstract interpretation the domain(s) of the concrete interpretation is replaced by some abstract domain(s) and the operations on the concrete domain(s) are replaced by abstract operations on the abstract domain(s).

*An earlier version of this paper appeared in [Nil88].

The aim of the paper is to provide a general and simple framework for abstract interpretation of logic programs, facilitating construction of different abstract interpretations in a simple and systematic way. We intend to exploit the ideas and results as the basis for efficient implementation of abstract interpretation in different abstract domains.

The approaches presented in the literature takes as a starting point some precise semantics of concrete interpretation — e.g. the *static* semantics of [CC77] and the *collecting* semantics of [JS87]. Such a semantics may become quite complicated. Therefore development of correctness proof for an abstract interpretation may be difficult. In this paper we suggest a concrete semantics which is in itself an approximation of the precise static/collecting semantics. Our semantics (which is called a static$^+$ semantics) uses the same domain as the static semantics but is a superset of it. We then demonstrate how further abstractions may be achieved by replacing the concrete domain by an abstract domain and by giving abstract interpretations to the operations of the static$^+$ interpretation. Because of the relative simplicity of the static$^+$ semantics such abstractions can be obtained in a systematic way. We also formulate some simple sufficient conditions which guarantee the termination and correctness of abstract interpretations.

The rest of the paper is organized as follows — section 2 provides the necessary definitions and notation. Section 3 informally introduces the concept of abstract interpretation, assertions and contexts. Section 4 develops a static$^+$ semantics based on the set of possible SLD-derivations. Section 5 presents some properties and requirements necessary to guarantee correctness and termination of abstract domains and operations. Section 6 gives an abstract domain and operations to infer groundness information for an example program. Section 7, finally, contains conclusions and comparisons with related work.

2 Preliminaries

If nothing else is said we adopt the notation and terminology of e.g. [Llo87] and [Apt87]. Below some of the most important concepts are outlined.

Without lack of generality we assume that the standard (leftmost) computation rule of Prolog is employed. Moreover, to avoid having to refer to "the i:th subgoal in the j:th clause" we assume, for notational convenience, that all literals in a program are unique.

The substitutions we consider in this paper are all idempotent. The *composition* $\theta' \circ \theta''$ of substitutions θ' and θ'' will be defined in the standard way (see [Llo87]).

Definition 2.1 Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution and denote by $Var(x)$ the set of variables in a term/formula, x. Then $Dom(\theta) = \{X_1, \ldots, X_n\}$

is called the *domain* of θ and $Range(\theta) = Var(t_1) \cup \ldots \cup Var(t_n)$ is called the *range* of θ. ∎

Definition 2.2 The *restriction* $\theta|_V$ of a substitution θ to a set of variables V is the substitution $\{X/t \mid X/t \in \theta \wedge X \in V\}$. ∎

The notation $\theta' \bullet \theta''$ denotes the substitution $(\theta' \circ \theta'')|_{Dom(\theta')}$. Throughout the paper we assume that $\circ$ binds stronger than $\bullet$, i.e. $\theta \bullet \theta_1 \circ \ldots \circ \theta_n$ is identical to $\theta \bullet (\theta_1 \circ \ldots \circ \theta_n)$ ($\circ$ is associative). Usually $\circ$ is omitted altogether.

Definition 2.3 A *renaming* (substitution) of a term/formula, x, is a substitution $\sigma = \{X_1/Y_1, \ldots, X_n/Y_n\}$ such that $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ are distinct variables and $Dom(\sigma) = Var(x)$. ∎

Definition 2.4 Let P be a definite program and G_0 a definite goal. An *SLD-derivation* of G_0 is a sequence $\langle G_0, C_0\sigma_0\rangle, \ldots, \langle G_{n-1}, C_{n-1}\sigma_{n-1}\rangle, G_n$ where:

- σ_i is a renaming of $C_i \in P$ such that $C_i\sigma_i$ contains no variables used prior in the sequence, and
- if $C_i = (x \leftarrow x_1, \ldots, x_m)$, $G_i = (\leftarrow y_1, y_2, \ldots)$ and $\theta_{i+1} = mgu(x\sigma_i, y_1)$ then $G_{i+1} = (\leftarrow x_1\sigma_i, \ldots, x_m\sigma_i, y_2, \ldots)\theta_{i+1}$.

The *computed substitution* of the derivation above is the composition $\theta_1 \ldots \theta_n$ of all mgu's in the derivation. ∎

Definition 2.5 The (initial) *connection graph*[1] of a program P (including one or more goal clauses) is the set

$$\{x \rightarrow y \mid (\ldots \leftarrow \ldots, x, \ldots) \in P \wedge (y \leftarrow \ldots) \in P \wedge \exists\theta(x\theta = y\sigma\theta)\}$$

where σ is a renaming of y such that $Var(x) \cap Var(y\sigma) = \emptyset$ ∎

Example 2.1 The connection graph of the *reverse*/2-program below is depicted in Figure 1.

[1]The definition given here diverges slightly from the one in e.g. [Kow79]

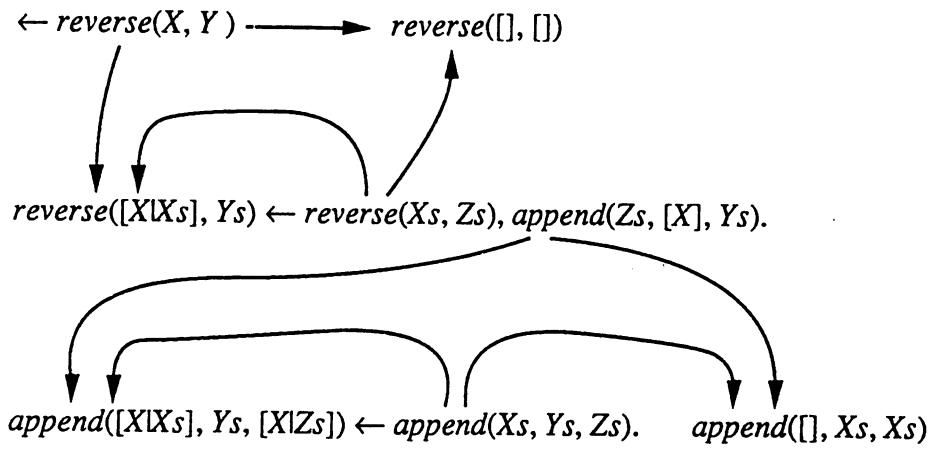

Figure 1: Connection-graph of $reverse/2$-program

$\leftarrow reverse(X, Y).$
$reverse([], []).$
$reverse([X|Xs], Ys) \leftarrow reverse(Xs, Zs), append(Zs, [X], Ys).$
$append([], Xs, Xs).$
$append([X|Xs], Ys, [X|Zs]) \leftarrow append(Xs, Ys, Zs).$

∎

We conclude the section with the following result due to [Tar55] and [Kle52].

Theorem 2.1 Let $\mathcal{D}$ be a complete lattice with bottom element $\perp$, f be a mapping $f : \mathcal{D} \rightarrow \mathcal{D}$ and denote by:

$$\begin{aligned} f \uparrow 0 &= \perp \\ f \uparrow (n+1) &= f(f \uparrow n) \\ f \uparrow \omega &= \bigcup_{i=0}^{\infty} f \uparrow i \end{aligned}$$

If f is continuous then $f \uparrow \omega$ is the least fixed point of f. ∎

3 Abstract Interpretation

Operationally the body of a (definite) clause may be viewed as a sequence of procedure-calls. For each call there are two distinguished program points — the point immediately to the left of a call x will be referred to as the *calling point* of x, and the point to the right will be called the *success point* of x. The leftmost and rightmost points in a clause will be called the *entry-* and *exit points* respectively of the clause.

Let p be the calling point of some literal x. We say that the program point p is *current* (in a derivation d) when (an instance of) x is the selected subgoal in d. For instance, the calling point of B_1 is current in the last goal of the derivation in Figure 2. By analogy the success point of x is current whenever x is immediately solved. Similarly entry- and exit-points may be current. Notice that several points may be current simultaneously — in the last step of Figure 3 the success point of A_i, the calling point of A_{i+1} and the exit point of $(B \leftarrow \ldots)$ are current (among others).

With every program point we want to associate some invariant assertion — i.e. an expression which is always true whenever the program point is current. Roughly speaking, the objective of abstract interpretation is to produce correct assertions for the program points of a program.

This idea is similar to [DM87] except that abstract interpretation generates assertions whereas [DM87] verify assertions. In contrast to [DM87] we are not going to introduce any specific meta-language for assertions. Another difference is that our assertions are associated with the program points — in [DM87] they are associated with the predicates.

Instead of introducing a new meta-language for assertions we are going to use the object-language. This is not necessarily a restriction since assertions usually refer to the bindings of (local) variables of the object language. In logic programming variable bindings correspond to substitutions. Thus, instead of associating an assertion with a program point we may collect a set of substitutions — namely all substitutions, θ, such that there exists a derivation where the program point is current and the computed substitution is θ. However, this leads to three complications.

First of all we are usually not interested in the whole substitutions — only in bindings for variables in the clause where the program point appears. Secondly, assertions refer to the variables in a clause of the program whereas, in derivations, *renamed* clauses are used. Consequently the substitutions contain bindings not for the program clause but for a renamed clause. Finally there are infinitely many ways to rename the variables in the clauses of a derivation. These problems will be handled as follows;

Consider a derivation $\ldots, \langle G_i, C\sigma \rangle, \ldots, G_j$ and assume that one of the pro-

gram points in C is current in G_j. Then $\sigma \bullet \theta_1 \ldots \theta_j$ should be contained in the set of substitutions associated with the program point. This solves the first two problems — it gives us a substitution whose domain is exactly the variables in the program clause C. Such a substitution will be called an *environment* and $\mathcal{E}_i$ designates the set of environments which may be associated with the program point i.

The third problem will not be dealt with — we shall simply not consider assertions which refer to the actual names of variables in the range of an environment. This is not a great restriction since no standard treatment is available how to rename the variables of a clause in a derivation.

Now, with every program point of a program we want to associate the set of all possible environments when the point is current. This set will be called a *context.*

If no restrictions are imposed on the use of a program practically any context may appear at each program point of the program and no useful assertions will be deduced. To limit the set of possible derivations we shall assume that the user provides information about the set of possible goals given to the program. Such a set will be called a *call-pattern.* To simplify the notation we assume that it is given on the form $\leftarrow p(X_1, \ldots, X_n), \{\theta_1, \ldots, \theta_m\}$ where $X_1, \ldots, X_n$ are distinct variables, θ_i is a substitution and $Dom(\theta_i) = \{X_1, \ldots, X_n\}$ for $1 \leq i \leq n$. I.e. as a definite goal with distinct variables as arguments together with a context for the goal. For instance, $\leftarrow reverse(X, Y), \{\{X/[a, b, c], Y/Y_1\}, \{X/[], Y/Y_1\}\}$ may be a possible call pattern of Example 2.1.

Let $\mathcal{C}_i$ $(= \wp(\mathcal{E}_i))$ be the set of all contexts which may be associated with point i of a program. Then clearly $\langle \mathcal{C}_i, \subseteq, \cup, \cap \rangle$ is a complete lattice with $\cup$ and $\cap$ as least upper- and greatest lower bound respectively.

Let P be a program with n program points. The set $\mathcal{C}_0 \times \ldots \times \mathcal{C}_n$ of *context vectors* will be denoted $\mathcal{CV}$. It is easy to see that $\langle \mathcal{CV}, \tilde{\subseteq}, \tilde{\cup}, \tilde{\cap} \rangle$ where

$$\begin{array}{rcl} \langle x_1, \ldots, x_n \rangle \tilde{\subseteq} \langle y_1, \ldots, y_n \rangle & \Leftrightarrow & x_1 \subseteq y_1 \wedge \ldots \wedge x_n \subseteq y_n \\ \langle x_1, \ldots, x_n \rangle \tilde{\cup} \langle y_1, \ldots, y_n \rangle & = & \langle x_1 \cup y_1, \ldots, x_n \cup y_n \rangle \\ \langle x_1, \ldots, x_n \rangle \tilde{\cap} \langle y_1, \ldots, y_n \rangle & = & \langle x_1 \cap y_1, \ldots, x_n \cap y_n \rangle \end{array}$$

is also a complete lattice. We will use the symbol Θ to denote context vectors and refer to its components by subscripts. E.g. $\Theta_{call(x)}$ refers to the value of the calling point of x in the context vector Θ. Sometimes we use integers as indices.

4 A Static$^+$ Semantics

In this section we develop a semantics which associates with every program point of a program (at least) the set of all possible contexts which may occur in

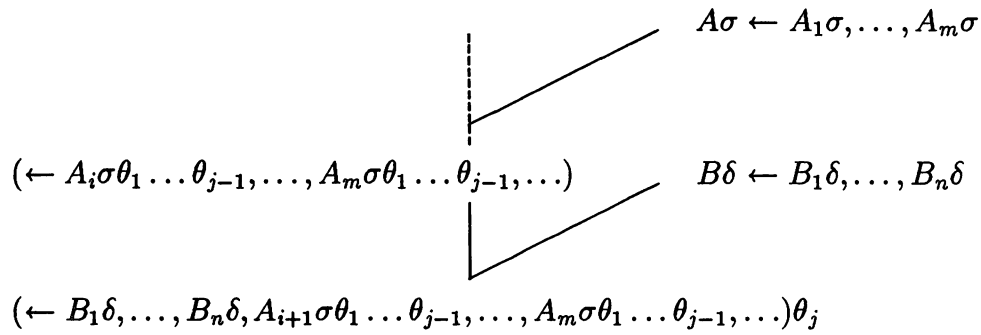

Figure 2: Entry-point of B is current

a derivation whenever the point is current.

Consider the entry-point of a clause $B \leftarrow B_1, \ldots, B_n$. Clearly, in order for this point to become current there has to be a derivation like that depicted in Figure 2. For each such derivation the context $\delta \bullet \theta_1 \ldots \theta_j$ must be contained in $\Theta_{entry(B)}$.

It is straightforward to see that:

$$\delta \bullet \theta_1 \ldots \theta_j = \delta \bullet \theta_j = \delta \bullet mgu(A_i\sigma\theta_1 \ldots \theta_{j-1}, B\delta)$$

and since $A_i\sigma\theta_1 \ldots \theta_{j-1} = A_i\sigma \bullet \theta_1 \ldots \theta_{j-1}$ we get the following equation for each clause, $x \leftarrow x_1, \ldots, x_n$, of the program:

$$\Theta_{entry(x)} = \bigcup_{y \rightarrow x} \{\delta \bullet mgu(y\theta, x\delta) \mid \theta \in \Theta_{call(y)} \wedge \delta = Ren(x, y\theta)\}$$

where $Ren(x, y)$ returns a renaming substitution of $x \leftarrow x_1, \ldots, x_n$ whose range is disjoint from the variables in y. For the sake of brevity we shall use $f(\Theta, y \rightarrow x)$ to the denote the set former.

Next we consider the success-point of a body literal, x. To reach this point there has to be a derivation which first "passes" the calling point of x with some computed substitution θ. Then $x\theta$ must be unified with a clause head and all subgoals in the body of that clause must be satisfied (i.e. we have to reach the exit-point of the clause). In other words there has to be a derivation like the one depicted in Figure 3. For each such derivation $\sigma \bullet \theta_1 \ldots \theta_k$ must be contained in $\Theta_{succ(x)}$.

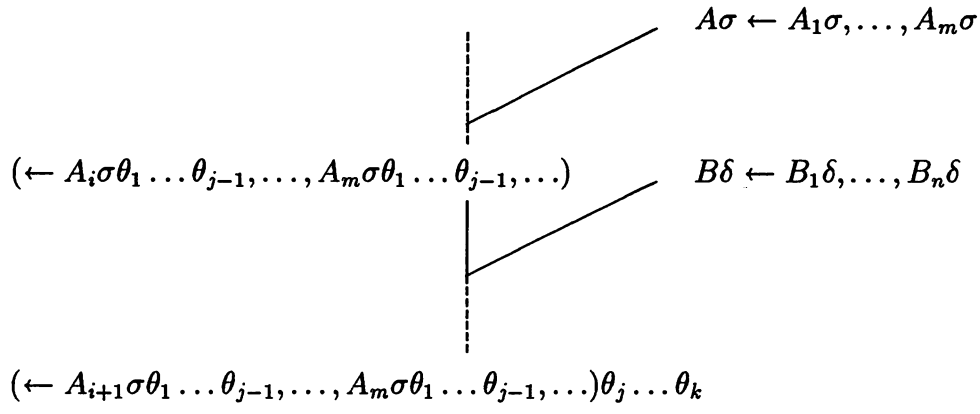

Figure 3: Success-point of A_i is current

Theorem 4.1 For the derivation in Figure 3 it holds that:

$$\sigma \bullet \theta_1 \ldots \theta_k = (\sigma \bullet \theta_1 \ldots \theta_{j-1}) \circ mgu(B\delta \bullet \theta_1 \ldots \theta_k, A_i\sigma \bullet \theta_1 \ldots \theta_{j-1})$$

∎

Proof: Initially we observe that:

$$B\delta\theta_1 \ldots \theta_k = B\delta\theta_j \ldots \theta_k = A_i\sigma\theta_1 \ldots \theta_k$$

Now let

$$B' = B\delta\theta_j \ldots \theta_k = B\delta \bullet \theta_j \ldots \theta_k$$
$$A'_i = A_i\sigma\theta_1 \ldots \theta_{j-1} = A_i\sigma \bullet \theta_1 \ldots \theta_{j-1}$$

Clearly B' is an instance of A'_i and thus:

$$B' = A'_i\alpha \text{ where } \alpha = mgu(A'_i, B')$$

Consequently

$$Dom(\alpha) \subseteq Var(A'_i) \subseteq Range(\sigma \bullet \theta_1 \ldots \theta_{j-1})$$

and

$$A'_i\alpha = A'_i\theta_j \ldots \theta_k$$

Thus, $\alpha \mid_{Var(A'_i)} = (\theta_j \ldots \theta_k) \mid_{Var(A'_i)} = \alpha$.

Now $\theta_j \ldots \theta_k$ is obtained by satisfying the goal $\leftarrow A'_i$. Consequently the domain of $\theta_j \ldots \theta_k$ will contain only variables in $Var(A'_i)$ and some new variables which do not appear in $Range(\sigma \bullet \theta_1 \ldots \theta_{j-1})$. Thus:

$$Dom(\theta_j \ldots \theta_k) \cap Range(\sigma \bullet \theta_1 \ldots \theta_{j-1}) \subseteq Var(A'_i)$$

and finally

$$\begin{aligned} \sigma \bullet \theta_1 \ldots \theta_k &= (\sigma \bullet \theta_1 \ldots \theta_{j-1}) \circ (\theta_j \ldots \theta_k)|_{Range(\sigma \bullet \theta_1 \ldots \theta_{j-1})} \\ &= (\sigma \bullet \theta_1 \ldots \theta_{j-1}) \circ (\theta_j \ldots \theta_k)|_{Var(A'_i)} \\ &= (\sigma \bullet \theta_1 \ldots \theta_{j-1}) \circ \alpha \end{aligned}$$

∎

It is worth noticing that if there is a derivation like the one in Figure 3 then $B\delta \bullet \theta_1 \ldots \theta_k$ has to be an instance of $A_i\sigma \bullet \theta_1 \ldots \theta_{j-1}$. Thus for each body literal, x, we get the equation:

$$\Theta_{succ(x)} = \bigcup_{x \to y} \{\theta \circ mgu(y\delta, x\theta) \mid \theta \in \Theta_{call(x)} \wedge \delta \in \Theta_{exit(y)} \wedge \exists \omega (y\delta = x\theta\omega)\}$$

The set former will be abbreviated $g(\Theta, x \to y)$.

Since all points in a program are either entry- or success-points we may formulate an equation for every point of the program. However, the entry point of the goal given in the call pattern needs special treatment since its context is already provided in the call pattern.

Example 4.1 Denote by A the literal $reverse(X, Y)$ in Example 2.1, by B the literal $reverse([], [])$ etc. We then get the following set of equations:

$$\begin{aligned} \Theta_{call(A)} &= \{\{X/[a,b,c], Y/Y_1\}, \{X/[], Y/Y_1\}\} \\ \Theta_{succ(A)} &= g(\Theta, A \to B) \cup g(\Theta, A \to C) \\ \Theta_{entry(B)} &= f(\Theta, A \to B) \cup f(\Theta, D \to B) \\ \Theta_{entry(C)} &= f(\Theta, A \to C) \cup f(\Theta, D \to C) \\ \Theta_{succ(D)} &= g(\Theta, D \to B) \cup g(\Theta, D \to C) \\ \Theta_{succ(E)} &= g(\Theta, E \to F) \cup g(\Theta, E \to G) \\ \Theta_{entry(F)} &= f(\Theta, E \to F) \cup f(\Theta, H \to F) \\ \Theta_{entry(G)} &= f(\Theta, E \to G) \cup f(\Theta, H \to G) \\ \Theta_{succ(H)} &= g(\Theta, H \to F) \cup g(\Theta, H \to G) \end{aligned}$$

∎

For the general case, we want to find a Θ such that

$$\{\Theta_1 = \Delta_1(\Theta), \ldots, \Theta_n = \Delta_n(\Theta)\}$$

This is equivalent to finding a fixed point of the function $cv : \mathcal{CV} \rightarrow \mathcal{CV}$

$$cv(x) = \langle \Delta_1(x), \ldots, \Delta_n(x) \rangle$$

Obviously cv may have several fixed points (corresponding to several solutions to the set of equations). All solutions contain the set of all possible contexts at each point. Clearly we want to find the the most accurate of the solutions, i.e. the *least* fixed point of cv (assuming the existence of a least fixed point).

It can be shown that cv is continuous and since we know that $\mathcal{CV}$ is a complete lattice we conclude that $cv \uparrow \omega$ is the least fixed point of cv. The fixed point $cv \uparrow \omega$ will be called the static$^+$ semantics of the program. The static$^+$ semantics is similar to the static semantics of [CC77]. However, the latter gives precise information about the set of possible environments whereas the static$^+$ semantics may be seen as an approximation (i.e. a superset) of the static semantics.

5 Abstract Domains

Unfortunately it is usually not possible to compute the static$^+$ semantics of a program in finite time. The natural way to avoid this problem is to find some *abstract domain* $\mathcal{CV}'$ which corresponds to a suitable subset of $\mathcal{CV}$. In this section some general guidelines are outlined how to find such an abstract domain.

Let $\mathcal{C}_1 \times \ldots \times \mathcal{C}_n$ be the set $\mathcal{CV}$ of context vectors of some program. The set of *abstract* context vectors $\mathcal{CV}'$ is the set $\mathcal{C}'_1 \times \ldots \times \mathcal{C}'_n$ where each $\mathcal{C}'_i$ corresponds to a subset of $\mathcal{C}_i$.

The first requirement is that $\mathcal{CV}'$ is a complete lattice. For this to hold, $\langle \mathcal{C}'_i, \sqsubseteq_i, \sqcup_i, \sqcap_i \rangle$ should be a complete lattice and there should be a monotonic mapping

$$\gamma_i : \mathcal{C}'_i \rightarrow \mathcal{C}_i$$

which will be called the *concretization mapping* (of $\mathcal{C}'_i$).

Clearly $\langle \mathcal{CV}', \tilde{\sqsubseteq}, \tilde{\sqcup}, \tilde{\sqcap} \rangle$ is a complete lattice if

$$\begin{aligned} x \,\tilde{\sqsubseteq}\, y &\Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge \ldots \wedge x_n \sqsubseteq_n y_n \\ x \,\tilde{\sqcup}\, y &= \langle x_1 \sqcup_1 y_1, \ldots, x_n \sqcup_n y_n \rangle \\ x \,\tilde{\sqcap}\, y &= \langle x_1 \sqcap_1 y_1, \ldots, x_n \sqcap_n y_n \rangle \end{aligned}$$

Furthermore, there is an injection

$$\gamma : \mathcal{CV}' \rightarrow \mathcal{CV}$$

based on the concretization of the components. It is easy to see that γ is monotonic, i.e.

$$x \tilde{\sqsubseteq} y \Rightarrow \gamma(x) \tilde{\subseteq} \gamma(y)$$

Definition 5.1 Let there be two lattices $\langle \mathcal{A}, \preceq \rangle$, $\langle \mathcal{B}, \sqsubseteq \rangle$ and a monotonic injection $\gamma : \mathcal{B} \rightarrow \mathcal{A}$ (i.e. $x \sqsubseteq y \Rightarrow \gamma(x) \preceq \gamma(y)$). We say that $f' : \mathcal{B}^n \rightarrow \mathcal{B}$ is a *safe γ-approximation* of $f : \mathcal{A}^n \rightarrow \mathcal{A}$ iff

$$\forall x_1, \ldots, x_n \in \mathcal{B}^n \ (f(\gamma(x_1), \ldots, \gamma(x_n)) \preceq \gamma(f'(x_1, \ldots, x_n)))$$

∎

Informally this definition states that $f'(x_1, \ldots, x_n)$ contains at least as much information as f applied to the objects denoted by $x_1, \ldots, x_n$.

For instance, it is easy to show that the restriction $\tilde{\sqcup} : \mathcal{CV}' \times \mathcal{CV}' \rightarrow \mathcal{CV}'$ above is a safe γ-approximation of $\tilde{\cup} : \mathcal{CV} \times \mathcal{CV} \rightarrow \mathcal{CV}$ — because of the monotonicity of γ we have that $\gamma(x) \tilde{\subseteq} \gamma(x \tilde{\sqcup} y)$ and $\gamma(y) \tilde{\subseteq} \gamma(x \tilde{\sqcup} y)$. Hence, $\gamma(x) \tilde{\cup} \gamma(y) \tilde{\subseteq} \gamma(x \tilde{\sqcup} y)$.

It is now possible to prove the following

Theorem 5.1 If the monotonic operation cv' is a safe (γ-) approximation of cv and $\mathcal{CV}'$ is a *finite height* lattice then

- There exists an $n \geq 0$ such that $cv' \uparrow (n+1) \ = \ cv' \uparrow n$, and
- for that n, $cv \uparrow \omega \tilde{\subseteq} \gamma(cv' \uparrow n)$

∎

Proof: The first proposition is proved using the monotonicity of cv' and the finite height of the domain. The second proposition follows from the more general result that $cv \uparrow \alpha \tilde{\subseteq} \gamma(cv' \uparrow \alpha)$ for *any* ordinal α. The latter is shown via transfinite induction. ∎

It is easy to see that $cv'(x) = \langle \Delta'_1(x), \ldots, \Delta'_n(x) \rangle$ is a safe approximation of $cv(x) = \langle \Delta_1(x), \ldots, \Delta_n(x) \rangle$ if Δ'_i is a safe approximation of Δ_i for $1 \leq i \leq n$. Notice that $\Delta_i : \mathcal{CV} \rightarrow \mathcal{C}_i$ and by "safe approximation" we mean that

$$\forall x \in \mathcal{CV} \ (\Delta_i(\gamma(x)) \subseteq \gamma_i(\Delta'_i(x)))$$

A sufficient condition for safeness of Δ'_i (wrt. Δ_i) follows

- $\Delta'_i \in \mathcal{C}'_i$ is a safe approximation of $\Delta_i \in \mathcal{C}_i$ if $\Delta_i \subseteq \gamma_i(\Delta'_i)$.

- $\Delta'_i = f'(x,1) \sqcup_i \ldots \sqcup_i f'(x,m)$ is a safe approximation of $\Delta_i = f(x,1) \cup \ldots \cup f(x,m)$ if $f'(x,y)$ is a safe approximation of $f(x,y)$ for each edge y of the connection graph.

- $\Delta'_i = g'(x,1) \sqcup_i \ldots \sqcup_i g'(x,m)$ is a safe approximation of $\Delta_i = g(x,1) \cup \ldots \cup g(x,m)$ if $g'(x,y)$ is a safe approximation of $g(x,y)$ for each edge y of the connection graph.

The last two points are easy to prove using the fact that $\sqcup_i : \mathcal{C}'_i \times \mathcal{C}'_i \rightarrow \mathcal{C}'_i$ is a safe γ_i-approximation of $\cup : \mathcal{C}_i \times \mathcal{C}_i \rightarrow \mathcal{C}_i$.

6 Groundness Analysis (Example)

In this section we use the results above to derive groundness information for the *reverse*/2-programs. I.e. to determine whether some predicate is always called with some specific argument being ground, or succeeds with some argument always being ground.

Let $\mathcal{V}_i$ denote the set of variables in the clause containing the program point i. Then $\mathcal{C}'_i$ shall be the set $\wp(\mathcal{V}_i)$ and γ_i is defined thus

$$\gamma_i(x) = \{\theta \in \mathcal{E}_i \mid X \in x \Rightarrow ground(X\theta)\}$$

Now, obviously $\langle \wp(\mathcal{V}_i), \supseteq, \cap, \cup \rangle$ is a complete lattice of finite height (with $\mathcal{V}_i$ and $\emptyset$ as bottom and top elements respectively). If $\mathcal{CV}' = \mathcal{C}'_1 \times \ldots \times \mathcal{C}'_n$ then $\langle \mathcal{CV}', \tilde{\supseteq}, \tilde{\cap}, \tilde{\cup} \rangle$ is a complete lattice when the operations apply component-wise.

Next, consider the abstract operations f' and g'. The core of both operations is the unification of the literals in the second argument. Let x and y be two literals and assume that x and y contain no common variables. What can be said about the mgu of $x\sigma$ and $y\sigma$ if our only knowledge about σ is that for some $X/t \in \sigma$, t is ground? Clearly, in order for $x\sigma$ and $y\sigma$ to unify x and y must be unifiable. Let θ be an mgu of x and y and let $v = \{X \mid X/t \in \sigma \wedge ground(t)\}$ be the set of variables which are known to be bound to ground terms in σ.

Now there are two cases to consider — if $X/t \in \theta$ and $X \in v$ then all variables in t are definitely bound to ground terms as a result of unification between $x\sigma$ and $y\sigma$ (obviously this is only true if unification of $x\sigma$ and $y\sigma$ succeeds).

Example 6.1 Let $x = p(X,Y)$ and $y = p(f(Z), g(V,W))$ and assume that $v = \{X,Y\}$ — i.e. we know that X and Y are always bound to ground terms when the two atoms are unified. Then $\theta = \{X/f(Z), Y/g(V,W)\}$. Thus, we conclude that Z, V, W are all ground after unification of $x\sigma$ and $y\sigma$. ∎

Secondly, assume $X/t \in \theta$ then if (i) t is ground or (ii) all variables in t are either in v or are ground due to case 1 then X will be ground as a result of unification between $x\sigma$ and $y\sigma$[2].

Example 6.2 Let $x = p(f(X,Y),g(Y))$ and $y = p(Z,W)$ and assume that $v = \{X,W\}$. Then $\theta = \{Z/f(X,Y), W/g(Y)\}$. Clearly, Y must be ground as a result of case 1 and consequently Z is ground due to case 2. ∎

Using this we may define the abstract operations f' and g' as follows

$$\begin{aligned}
f'(\Theta, x \rightarrow y) &= close(mgu(x\sigma, y), \Theta_{call(x)}\sigma, y) \\
g'(\Theta, x \rightarrow y) &= \Theta_{call(x)} \cup close(mgu(x, y\delta), \Theta_{call(x)} \cup \Theta_{exit(y)}\delta, x) \\
close(\theta, x, y) &= Var(y) \cap (case1(\theta, x) \cup case2(\theta, case1(\theta, x) \cup x)) \\
case1(\theta, x) &= \{X \mid Y/t \in \theta \wedge Y \in x \wedge X \in Var(t)\} \\
case2(\theta, x) &= \{X \mid X/t \in \theta \wedge Var(t) \subseteq x\}
\end{aligned}$$

where σ (δ) is a renaming of x (y) such that $x\sigma$ $(y\delta)$ and y (x) contain no common variables. By $\{X_1, \ldots, X_n\}\sigma$ we denote $\{X_1\sigma, \ldots, X_n\sigma\}$.

In addition to taking the union of the two cases, *close* also restricts the resulting set of "ground" variables to those which appear in the third argument. The definition of f' simply is a call to *close* whereas g' is the union of *close* and the variables which are known to be ground before the call (clearly variables which become bound remain bound while executing the subgoals to the right of the point where they become bound).

The mapping cv' is now defined as $cv'(x) = \langle \Delta'_1(x), \ldots, \Delta'_9(x) \rangle$ where (cf. Examples 2.1 and 4.1)

$$\begin{aligned}
\Delta'_1(x) &= \{X\} \\
\Delta'_2(x) &= g'(x, A \rightarrow B) \cap g'(x, A \rightarrow C) \\
&\vdots \\
\Delta'_9(x) &= g'(x, H \rightarrow F) \cap g'(x, H \rightarrow G)
\end{aligned}$$

Here the first equation says that the program is always called with the first argument being ground. It does not say anything about the second argument — it may be any term.

Clearly, the bottom element in $\mathcal{CV}'$ is

$$\begin{aligned}
cv' \uparrow 0 = \langle &\{X,Y\}, \{X,Y\}, \emptyset, \{X, Xs, Ys, Zs\}, \{X, Xs, Ys, Zs\}, \\
&\{X, Xs, Ys, Zs\}, \{Xs\}, \{X, Xs, Ys, Zs\}, \{X, Xs, Ys, Zs\} \rangle
\end{aligned}$$

[2]This is based on the assumption that θ is an idempotent mgu.

and the fixed point of cv' is reached after four iterations (where " ..." denotes components which do not change)

$$\begin{aligned}
cv' \uparrow 1 &= \langle \{X\}, \ldots, \ldots, \{X, Xs, Ys\}, \ldots, \ldots, \ldots, \ldots, \ldots \rangle \\
cv' \uparrow 2 &= \langle \ldots, \ldots, \ldots, \{X, Xs\}, \ldots, \ldots, \ldots, \ldots, \ldots \rangle \\
cv' \uparrow 3 &= \langle \ldots, \ldots, \ldots, \ldots, \{X, Xs, Zs\}, \ldots, \ldots, \ldots, \ldots \rangle \\
cv' \uparrow 4 &= \langle \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \{X, Xs, Ys\}, \ldots \rangle \\
cv' \uparrow 5 &= cv' \uparrow 4
\end{aligned}$$

As a result of the analysis we may conclude e.g. the following "assertions":

- if $reverse/2$ is called with the first argument being ground, then the second argument is ground on success (second component of $cv' \uparrow 5$).
- whenever $reverse/2$ is called the first argument is ground (components 1 and 4).
- all calls to $append/3$ are made with the first two arguments being ground (components 5 and 8).

7 Conclusions

A theoretical framework for the abstract interpretation of logic programs was presented. We developed a static$^+$ semantics for definite programs to provide a basis for further abstractions. The static$^+$ semantics plays the rôle of the static semantics in [CC77] and the collecting semantics in [JS87]. However, the static$^+$ semantics is an approximation (a superset) of these two.

From the static$^+$ semantics further abstractions are possible. This is achieved by constructing abstract domains and by giving alternative, abstract interpretations to the operations used to define the static$^+$ semantics. The first step towards an abstract interpretation would be to construct an abstract domain. By selecting a complete lattice of finite height as domain, termination can be ensured. Secondly if there exists a monotonic mapping, γ, from the abstract domain to the concrete one and if there is a monotonic mapping, cv', which safely approximates cv wrt. γ, then for some finite n, $\gamma(cv' \uparrow n)$ approximates (contains) $cv \uparrow \omega$ — i.e. the static$^+$ semantics (and of course as a consequence also the static semantics). We also showed how safeness of cv' wrt. cv can be broken down to safeness of more primitive operations (like f' and g' in our case).

Roughly speaking, the consequence of having only a static$^+$ semantics is that we do not keep full track of "where information originates from". More precisely, two different calls to a clause may give different contexts at the exit point of that

clause, however, when we compute the success point of a call we take into account *all* contexts at the exit point — not just the ones which were due to the call. This leads to three consequences

- The result of abstract interpretation may become worse. That is, the approximation *may* be less accurate;
- The efficiency (speed) of the analysis is usually improved;
- The formulation of the concrete semantics becomes less complicated.

To some extent accuracy can be traded against efficiency in our approach by performing "unfolding" in the connection graph. Clauses which are called from several body literals may be copied. Thus, avoiding some imprecision at the expense of extra analysis time.

Some previous attempts to provide general frameworks for abstract interpretation are available (e.g. [Bru87], [JS87], [Mel87], [DW88]). Like [Bru87] we have based our framework on the standard operational semantics (SLD-resolution) of definite programs. This is by no means the only possibility — in [JS87] denotational semantics is used and in [MS88] a bottom-up fixed point semantics.

In one respect [Mel87] and [DW88] distinguish themselves from the other three. Neither of them records the set of contexts associated with (what we call) program points. Instead they record the set of all calls and successes of each predicate. For some important run-time properties this information is not sufficient. It is e.g. not possible to deduce whether two subgoals in the body of a clause are always independent (contain a common variable) during run-time. However, both [Mel87] and [DW88] are primarily interested in, so called, mode analysis of logic programs.

In [JS87] contexts are recorded only at the entry point of each clause.

The aim of this paper was to provide a theoretical framework for efficient implementation of abstract interpretation in different abstract domains. Some preliminary tests indicate that cv' can be implemented efficiently in a language with constant access time to arrays (to represent the context vector). This follows from the observation that $cv' \uparrow (n+1)$ and $cv' \uparrow n$ usually only differ in some of their components. Therefore it is not necessary to compute a completely new context vector in each iteration — some (most) components will remain the same.

This paper covers only static analysis of *definite program.* However, it would not be very hard to include also some of the *extralogical features* of most Prolog implementations. For instance, taking the operational semantics of most Prolog systems into account we know that for a subgoal $t_1 < t_2$ all of the variables in t_1 and t_2 are ground on success. However, some built-in predicates cause additional problems — in particular those which modify the database ($assert/1$

and *retract*/1) and those which are implemented in terms of *call*/1 (*setof*/2, *not*/1 etc).

Acknowledgements

The author is grateful to Jan Małuszyński for his continuous support and advice. S.Bonnier, M.Bruynooghe, W.Drabent and H.Søndergaard provided many useful and constructive comments on previous versions of the paper. Fundings were supplied by the Swedish National Board for Technical Development (STU).

References

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[Apt87] K. Apt. *Introduction to Logic Programming.* Report TR-87-35, Dept. of Computer Science, The University of Texas at Austin, 1987. Revised and Extended Version in 1988.

[Bru87] M. Bruynooghe. *A Framework for the Abstract Interpretation of Logic Programs.* Report CW62, Katholieke Universiteit, Leuven, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, Los Angeles, 1977.

[DM87] W. Drabent and J. Małuszyński. Proving runtime properties of logic programs. In *Proc. of TAPSOFT'87,* Pisa, LNCS 250, Springer-Verlag, 1987.

[DW88] S. Debray and D.S. Warren. Automatic mode inference for logic programs. *J. Logic Programming*, 5(3), 1988.

[JS87] N.D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[Kle52] S.C. Kleene. *Introduction to Metamathematics.* North-Holland Publ. Co, Amsterdam-New York, 1952.

[Kow79] R. Kowalski. *Logic For Problem Solving.* Elsevier, North-Holland, New York, 1979.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Mel87] C. Mellish. Abstract interpretation of prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[MS88] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proc. of 5th International Conf/Symp. on Logic Programming,* Seattle, MIT Press, 1988.

[Nil88] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In *Proc. of PLILP'88,* Orleans, To appear in Springer-Verlag LNCS series, 1988.

[Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5, 1955.

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-88-44

Upplagans storlek:
Number of copies:

Datum:
Date: December 1988

Projekt:
Project:

Titel:
Title: Towards a Framework for the Abstract Interpretation of Logic Programs

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: A new framework for abstract interpretation of logic programs is presented. The idea is to take as the basis a simplified semantics that approximates the standard operational semantics of logic programs but still makes it possible to derive non-trivial abstract interpretations. The relative simplicity of the basic semantics facilitates systematic derivation of abstract interpretations and static analyses of logic programs. Sufficient conditions for termination and correctness of the derived interpreters are provided. The approach is illustrated by inferring groundness information for an example program.

Nyckelord (högst 8):
Keywords (8):

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-88-38 **Zebo Peng:** Let's Design Asynchronous VLSI Systems. Also published in *Microprocessing and Microprogramming, the Euromicro Journal, Vol. 24, 1988.*

LiTH-IDA-R-88-37 **Tony Larsson, Zebo Peng:** A Relational Approach to VLSI Design Databases. Also in *Proc. of the 36th ISMM International Conference on Mini and Microcomputers and Their Applications*, Sant Feliu de Guixols, Spain, June 27-30, 1988.

LiTH-IDA-R-88-36 **Zebo Peng:** Semantics of a Parallel Computation Model and its Applications in Digital Hardware Design. Also in *Proc. of the 1988 International Conference on Parallel Processing,* Pennsylvania State University, August 15-19, 1988.

LiTH-IDA-R-88-35 **Zebo Peng:** A Horizontal Optimization Algorithm for Data Path/Control Synthesis. Also in *Proc. of the 1988 IEEE International Symposium on Circuits and Systems,* Helsinki Univ. of Technology, Finland, June 7-9, 1988.

LiTH-IDA-R-88-34 **Arne Jönsson, Nils Dahlbäck:** Talking to a computer is not like talking to your best friend. Also in *Proc. of the First Scandinavian Conference on Artificial Intelligence*, March 9-10, 1988, Tromsö, Norway

LiTH-IDA-R-88-32 **Erik Sandewall:** An Approach to Non-Monotonic Entailment. To appear in *Methodologies for Intelligent Systems*, vol II, *Proc. of ISMIS 1988 conference*, Torino, Italy, 1988, Z.W. Ras and L. Saitta (eds), *North Holland*, 1988, pp. 391-397.

LiTH-IDA-R-88-31 **Erik Sandewall:** The Semantics of Non-Monotonic Entailment Defined Using Partial Interpretations.

LiTH-IDA-R-88-30 **Erik Sandewall:** Future Developments in Artificial Intelligence. Also in *Proc. of European Conference on Artificial Intelligence (ECAI)*, Munich, August, 1988.

LiTH-IDA-R-88-29 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part III: Decision Procedure.

LiTH-IDA-R-88-28 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part II: Concurrent Actions.

LiTH-IDA-R-88-27 **Erik Sandewall:** Non-Monotonic Entailment for Reasoning about Time and Action. Part I: Sequential Actions.

LiTH-IDA-R-88-26 **Lin Padgham:** NODE: A Database for Use by Intelligen Systems. Also in *Proc. of ISMIS '88,* Torino Italy, October 12-15, 1988.

LiTH-IDA-R-88-23 **Johan Fagerström, Yngve Larsson:** Two Contributions on Debugging Distributed Systems. Also in *Proc. of ACM SIGPLAN/SIGOPS, Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 5-6, 1988.

LiTH-IDA-R-88-22 **Johan Fagerström:** Design and Test of Distributed Applications. Also in *Proc. of the International Conference on Software Engineering, ICSE*, Singapore, April 1988.

LiTH-IDA-R-88-21 **Johan Fagerström, Lars Strömberg:** A Paradigm for Distributed System Design and Test. Also in *Proc. of COMPCON'SPRING 88*, San Fransicso, Febr-March 1988.

LiTH-IDA-R-88-19 **Lin Padgham, Ralph Rönnquist:** From a Technical to a Humane Environment: A Software System Supporting Cooperative Work. Also in *Proc. of the GDI Internationa Conference on USER INTERFACES,* Rüschlikon, Switzerland, October 20-21, 1986.

LiTH-IDA-R-88-18 **Nils Dahlbäck:** Mental Models and Text Understanding - a Commented Review.

LiTH-IDA-R-88-17 **Bengt Lennartsson:** Multi-Level Software Architectures - Industrial Experiences and Research Directions. This report is a revised version of a paper in *Advance Working Papers of The Second International Workshop on Computer-Aided Software Engineering.* Cambridge, Massachusetts. July 12-15, 1988.

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.
- *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 1988

LiTH-IDA-R-88-44 **Ulf Nilsson:** Towards a Framework for the Abstract Interpretation of Logic Programs.

LiTH-IDA-R-88-43 **Göran Rydqvist, Mikael R.K. Patel, Leif Larsson:** Types in Space - Towards Flexible High-speed Object Oriented Data Management Tools of the Future.

LiTH-IDA-R-88-42 **Mikael R.K. Patel:** What should a Programming Environment for Threaded Interpretive Languages provide? Also in *Proc. of the 8th Rochester Forth Conference on Programming Environments*, Rochester, New York, USA, June 1988.

LiTH-IDA-R-88-41 **Tony Larsson:** Hardware Verification based on Algebraic Manipulation and Partial Evaluation. Also in *Proc of IFIP WG 10.2 International Working Conference on "The Fusion of Hardware Design and Verification"*, Glasgow, Scotland, July 3-6, 1988.

LiTH-IDA-R-88-39 **Zebo Peng, Krzysztof Kuchcinski, Bryan Lyles:** CAMAD: A Unified Data Path/Control Synthesis Environment. Also in *Proc. of the* [illegible] *gies for VLSI and Co*[illegible]-23, 1988.