

Lecture Notes in Computer Science

- Vol. 117: Fundamentals of Computation Theory. Proceedings, 1981. Edited by F. Göcseg. XI, 471 pages. 1981.
- Vol. 118: Mathematical Foundations of Computer Science 1981. Proceedings, 1981. Edited by J. Gruska and M. Chytil. XI, 589 pages. 1981.
- Vol. 119: G. Hirst, Anaphora in Natural Language Understanding: A Survey. XIII, 128 pages. 1981.
- Vol. 120: L. B. Rall, Automatic Differentiation: Techniques and Applications. VIII, 165 pages. 1981.
- Vol. 121: Z. Zlatev, J. Wasniewski, and K. Schaumburg, Y12M Solution of Large and Sparse Systems of Linear Algebraic Equations. IX, 128 pages. 1981.
- Vol. 122: Algorithms in Modern Mathematics and Computer Science. Proceedings, 1979. Edited by A. P. Ershov and D. E. Knuth. XI, 487 pages. 1981.
- Vol. 123: Trends in Information Processing Systems. Proceedings, 1981. Edited by A. J. W. Duijvestijn and P. C. Lockemann. XI, 349 pages. 1981.
- Vol. 124: W. Polak, Compiler Specification and Verification. XIII, 269 pages. 1981.
- Vol. 125: Logic of Programs. Proceedings, 1979. Edited by E. Engeler. V, 245 pages. 1981.
- Vol. 126: Microcomputer System Design. Proceedings, 1981. Edited by M. J. Flynn, N. R. Harris, and D. P. McCarthy. VII, 397 pages. 1982.
- Vol. 127: Y. Wallach, Alternating Sequential/Parallel Processing. X, 329 pages. 1982.
- Vol. 128: P. Branquart, G. Louis, P. Wodon, An Analytical Description of CHILL, the CCITT High Level Language. VI, 277 pages. 1982.
- Vol. 129: B. T. Hailpern, Verifying Concurrent Processes Using Temporal Logic. VIII, 208 pages. 1982.
- Vol. 130: R. Goldblatt, Axiomatising the Logic of Computer Programming. XI, 304 pages. 1982.
- Vol. 131: Logics of Programs. Proceedings, 1981. Edited by D. Kozen. VI, 429 pages. 1982.
- Vol. 132: Data Base Design Techniques I: Requirements and Logical Structures. Proceedings, 1978. Edited by S.B. Yao, S.B. Navathe, J.L. Weldon, and T.L. Kunii. V, 227 pages. 1982.
- Vol. 133: Data Base Design Techniques II: Proceedings, 1979. Edited by S.B. Yao and T.L. Kunii. V, 229-399 pages. 1982.
- Vol. 134: Program Specification. Proceedings, 1981. Edited by J. Staunstrup. IV, 426 pages. 1982.
- Vol. 135: R.L. Constable, S.D. Johnson, and C.D. Eichenlaub, An Introduction to the PL/CV2 Programming Logic. X, 292 pages. 1982.
- Vol. 136: Ch. M. Hoffmann, Group-Theoretic Algorithms and Graph Isomorphism. VIII, 311 pages. 1982.
- Vol. 137: International Symposium on Programming. Proceedings, 1982. Edited by M. Dezani-Ciancaglini and M. Montanari. VI, 406 pages. 1982.
- Vol. 138: 6th Conference on Automated Deduction. Proceedings, 1982. Edited by D.W. Loveland. VII, 389 pages. 1982.
- Vol. 139: J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, W. Kirchgässner, An Attribute Grammar for the Semantic Analysis of Ada. IX, 511 pages. 1982.
- Vol. 140: Automata, Languages and programming. Edited by M. Nielsen and E.M. Schmidt. VII, 614 pages. 1982.
- Vol. 141: U. Kastens, B. Hutt, E. Zimmermann, GAG: A Practical Compiler Generator. IV, 156 pages. 1982.
- Vol. 142: Problems and Methodologies in Mathematical Software Production. Proceedings, 1980. Edited by P.C. Messina and A. Murli. VII, 271 pages. 1982.
- Vol. 143: Operating Systems Engineering. Proceedings, 1980. Edited by M. Maekawa and L.A. Belady. VII, 465 pages. 1982.
- Vol. 144: Computer Algebra. Proceedings, 1982. Edited by J. Calmet. XIV, 301 pages. 1982.
- Vol. 145: Theoretical Computer Science. Proceedings, 1983. Edited by A.B. Cremers and H.P. Kriegel. X, 367 pages. 1982.
- Vol. 146: Research and Development in Information Retrieval. Proceedings, 1982. Edited by G. Salton and H.-J. Schneider. IX, 311 pages. 1983.
- Vol. 147: RIMS Symposia on Software Science and Engineering. Proceedings, 1982. Edited by E. Goto, I. Nakata, K. Furukawa, R. Nakajima, and A. Yonezawa. V, 232 pages. 1983.
- Vol. 148: Logics of Programs and Their Applications. Proceedings, 1980. Edited by A. Salwicki. VI, 324 pages. 1983.
- Vol. 149: Cryptography. Proceedings, 1982. Edited by T. Beth. VIII, 402 pages. 1983.
- Vol. 150: Enduser Systems and Their Human Factors. Proceedings, 1983. Edited by A. Blaser and M. Zoepritz. III, 138 pages. 1983.
- Vol. 151: R. Piloty, M. Barbacci, D. Borriore, D. Dietmeyer, F. Hill, and P. Skelly, CONLAN Report. XII, 174 pages. 1983.
- Vol. 152: Specification and Design of Software Systems. Proceedings, 1982. Edited by E. Knuth and E. J. Neuhold. V, 152 pages. 1983.
- Vol. 153: Graph-Grammars and Their Application to Computer Science. Proceedings, 1982. Edited by H. Ehrig, M. Nagl, and G. Rozenberg. VII, 452 pages. 1983.
- Vol. 154: Automata, Languages and Programming. Proceedings, 1983. Edited by J. Diaz. VIII, 734 pages. 1983.
- Vol. 155: The Programming Language Ada. Reference Manual. Approved 17 February 1983. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983. IX, 331 pages. 1983.
- Vol. 156: M.H. Overmars, The Design of Dynamic Data Structures. VII, 181 pages. 1983.
- Vol. 157: O. Østerby, Z. Zlatev, Direct Methods for Sparse Matrices. VIII, 127 pages. 1983.
- Vol. 158: Foundations of Computation Theory. Proceedings, 1983. Edited by M. Karpinski. XI, 517 pages. 1983.
- Vol. 159: CAAP'83. Proceedings, 1983. Edited by G. Ausiello and M. Protasi. VI, 416 pages. 1983.
- Vol. 160: The IOTA Programming System. Edited by R. Nakajima and T. Yuasa. VII, 217 pages. 1983.
- Vol. 161: DIANA, An Intermediate Language for Ada. Edited by G. Goos, W.A. Wulf, A. Evans, Jr. and K.J. Butler. VII, 201 pages. 1983.
- Vol. 162: Computer Algebra. Proceedings, 1983. Edited by J.A. van Hulzen. XIII, 305 pages. 1983.
- Vol. 163: VLSI Engineering. Proceedings. Edited by T.L. Kunii. VIII, 308 pages. 1984.
- Vol. 164: Logics of Programs. Proceedings, 1983. Edited by E. Clarke and D. Kozen. VI, 528 pages. 1984.
- Vol. 165: T.F. Coleman, Large Sparse Numerical Optimization. V, 105 pages. 1984.
- Vol. 166: STACS 84. Symposium of Theoretical Aspects of Computer Science. Proceedings, 1984. Edited by M. Fontet and K. Mehlhorn. VI, 338 pages. 1984.

Gudmund Kneuberg Franklin

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

199

Fundamentals of Computation Theory

FCT '85

Cottbus, GDR, September 9-13, 1985

Edited by Lothar Budach



Springer-Verlag
Berlin Heidelberg New York Tokyo

Logic Programming and Substitutions

Gudmund Frandsen
 Department of Computer Science
 Aarhus University
 Ny Munkegade
 DK-8000 Aarhus C
 Denmark

1. Introduction

In this paper we construct a domain of substitutions that may form the basis of a semantics for logic programming, since a computation involving a logic program and a query naturally results in a substitution. Our domain differs from usual domains in that ω -continuity is defined with respect to both increasing and decreasing chains. This difference is a consequence of our principle of computational equivalence: The results of two computations should be given the same denotation precisely when they bear the same finitely computable information content.

In the following we discuss the principle of computational equivalence alternated with a stepwise construction of the substitution domain. Finally we present a denotational semantics based on the finished domain.

This paper is abstracted from a research report [2], where further details can be found. Concerning prerequisites in the areas of logic programming and Scott-domains the reader may consult [3] and [4] respectively.

2. Information content

We mentioned above that a computation involving a logic program and a query naturally results in some sort of substitution. In a semantic domain such substitutions should have the same denotation if and only if they are equivalent in an appropriate sense. Consider the following example:

Example 1.

Program: $Q(f(Z), g(Z)) \leftarrow p(Z).$
 $P(a).$

Query: $Q(X, Y)?$

Result of computation in the form of some possible syntactic answers:

- 1): $[X \rightarrow f(\underline{Z}), Y \rightarrow g(\underline{Z}), \underline{Z} \rightarrow a].$
- 2): $[X \rightarrow f(\underline{W}), Y \rightarrow g(\underline{W}), \underline{W} \rightarrow a].$
- 3): $[X \rightarrow f(a), Y \rightarrow g(a)].$

Underscore denotes an anonymous variable. All variables that do not occur in the query are made anonymous in the answer. By giving names to anonymous variables, we obtain the possibility of relating different undefined values. However, the information content of a substitution is independent of a specific choice of names: answers 1) and 2) are equivalent. In the case of ex. 1 we may completely remove anonymous variables and obtain the answer 3). The information contents of the three substitution answers are identical. Consequently the three substitutions should have the same denotation in a domain of substitutions.

We are going to build a domain based on Scott's information system framework [4]. Apparently concepts such as information content and anonymous variable are important, when dealing with computational equivalence. We will demonstrate, how these concepts can form the basis of a definition of substitution dataobject.

3. Substitution dataobject

Initially we provide some basic definitions. The syntactic basis consists of one countable set of identifiers and another of variables. We do not bother whether an identifier denotes a function or a predicate, neither do we deal with arities. On the basis of identifiers and variables we form the recursively defined set of terms and the set of substitutions.

Basic definitions:

I	a countable set of identifiers
V	a countable set of variables
$T = IT^* V$	the set of terms
$S = V \rightarrow T$	the set of substitutions
$T_0 = IT_0^*$	the set of groundterms
$S_0 = V \rightarrow T_0$	the set of groundsubstitutions
$\cdot[\cdot]: T \times S_0 \rightarrow T_0$	the substitution operator:
$t[s] =$	"t, where any variable occurrence V is replaced by $s(v)$."
$h: S \rightarrow 2^{S_0}$	the groundexpansion of substitutions:
$h(s) = \{s' \in S_0 \mid \forall v. s(v)[s'] = s'(v)\}$	
$\cdot _{\cdot}: 2^{S_0} \times 2^V \rightarrow 2^{S_0}$	the restriction operator, which "erases" any information on variables that are not members of the specified set:
$s' _{V'} = \{s \in S_0 \mid \exists s' \in S'. \forall v \in V'. s(v) = s'(v)\}$	

Groundterms and groundsubstitutions are introduced to measure information content, and the restriction operator is defined to formalize the intuitive notion of anonymous variable. For a treatment of the intuition behind groundexpansions the reader may consult [2]. Here we present a simple example:

Example 2.

Let the substitution s be defined as follows:

$$s(v) = \begin{cases} f(Z) & , v = X \\ g(Z) & , v = Y \\ a & , v = Z \\ v & , \text{otherwise} \end{cases}$$

The groundexpansion of s is

$$h(s) = \{s \in S_0 \mid s(X) = f(s(Z)), s(Y) = g(s(Z)), s(Z) = a\}$$

The restriction of this expansion to $\{X, Y\}$ is easily constructed:

$$h(s)|_{\{X, Y\}} = \{s \in S_0 \mid s(X) = f(a), s(Y) = g(a)\},$$

which should be the common groundexpansion of the three equivalent substitution dataobjects occurring in ex. 1.

The set of dataobjects is now formed from the finite, cyclefree substitutions with distinct named (non anonymous) variables.

Substitution dataobjects:

$$\mathcal{D}_s = \{(V', s) \in 2^V \times S \mid \text{i) } V' \text{ and } \text{def}(s) \text{ are finite ii) } h(s) \neq \emptyset \text{ (s is cyclefree)}\},$$

where $\text{def}(s) = \{v \mid s(v) \neq v\}$

$\cdot|_{\cdot}: \mathcal{D}_2 \times 2^V \rightarrow \mathcal{D}_s$ the restriction operator on dataobjects:

$$(V', s)|_{V''} = (V' \cap V'', s)$$

$h: \mathcal{D}_s \rightarrow 2^{S_0}$. the groundexpansion of dataobjects:

$$h((V', s)) = h(s)|_{V'}$$

In a dataobject $(V', s) \in \mathcal{D}_s$, the substitution s may contain information on both named variables (V') and anonymous variables ($\langle V' \rangle$). The set of named variables may be reduced by use of the restriction operator. The groundexpansion h measures the information content of a dataobject, i.e. a small groundexpansion corresponds to a big information content. In this way a dataobject with empty expansion is inconsistent. However, such dataobjects are excluded from \mathcal{D}_s by restriction ii) in the definition. At the opposite end of the information scale reside the uninformative dataobjects, among which one is special: $\Delta_s = (V', s)$, where $V' = \emptyset$ and $\text{def}(s) = \emptyset$. It is easily seen that $h(\Delta_s) = S_0$. Many dataobjects are equally uninformative, since a dataobject (V', s) with trivial substitution ($\text{def}(s) = \emptyset$) or a dataobject (V', s) without named variables ($V' = \emptyset$) both have expansion S_0 .

It may now be easily verified that the three dataobjects of ex.1 have equal information contents by use of the formal definition (cfr. ex.2).

Before continuing the construction of a Scott-information-system we consider nondeterministic computations.

4. Computational equivalence

Consider the following example:

Example 3.

Program: $p(f(X)) \Leftarrow p(X)$.
 $p(a)$.

Query: $p(Y)?$

result of nondeterministic computation in the form of possible syntactic answers:

- 1) $[Y \rightarrow a]$
- 2) $[Y \rightarrow f(a)]$
- 3) $[Y \rightarrow f^2(a)]$
- \vdots

Unlike ex.1, the possible answers above are not equivalent. Here each answer corresponds to a specific sequence of nondeterministic choices made during the computation. We may syntactically describe the result of such a nondeterministic computation as an infinite disjunction of substitution dataobjects. In the case of ex.3:
 $s = [Y \rightarrow a] \vee [Y \rightarrow f(a)] \vee [Y \rightarrow f^2(a)] \vee \dots$. What information content should we associate with such a disjunction? It seems natural to take the groundexpansion of a disjunction to be the union of the individual groundexpansions [2], i.e. the information content of a disjunction is less than the information content of any disjunct. In the above case we get $h(s) = \bigcup_{j=0}^{\infty} h([Y \rightarrow f^j(a)])$. With a partial order based on information content, s may be perceived as the greatest lower bound of all its finite approximations (i.e. the finite subdisjunctions of s). What denotation is "correct" for an (in)finite disjunction of dataobjects? Scott's power-domain of indeterminacy [4] has a natural denotation for each of the finite approximations. However, the greatest lower bound of these denotations is the uninformative bottom element \perp_s , which is not a good approximation for s , since $h(\perp_s) = S_0 \neq h(s)$. As a consequence we must build a new domain capable of representing such infinite disjunctions in a satisfactory way.

There is yet another reason for demanding a new type of domain. In case s had an information content equal to the information content of the bottom element (i.e. none), we should not use \perp_s as a denotation for s anyway, because every finite approximation to s contains some information. We may formulate:

Principle of computational equivalence:

Two substitutions should have the same denotation iff

- 1) They bear equal information content and
- 2) the truth of 1) is established in a finite computation.

We need a domain, which is continuous for decreasing chains such that the glb of a sequence of informative approximations is finitely informative (and possibly infinitely uninformative).

Such a domain is easily constructed if we reverse the information order, so the above demand is on increasing chains. Yet if we want to treat negation by finite failure in addition, we will need ω -continuity of decreasing and increasing chains. Hence we do not solve the problem by reversing the information order.

For a more thorough discussion of negation by finite failure, we refer to [2]. Her should just be mentioned that Apt and van Emden [1] characterize the semantics of negation by finite failure by means of a concept equivalent to the information content of a term, although they do not discuss the necessity of a finite computability restriction. Consequently they have to deal with a discontinuous operator.

We are now going to realize the principle of computational equivalence by building a domain, where ω -continuity is defined with respect to both decreasing and increasing chains.

5. The cd-domain of substitutions

When building Scott domains from information systems, a domain element consists of the conjunction of all those data-objects that contain an amount of information less than or equal to the information content of the element itself. We will now build a domain (the cd-domain), where each element consists of the conjunction of all those disjunctions of dataobjects that contain no more information than the information content of the element itself.

Basic definitions:

$\mathbb{P}_S = 2^{D_S}$ The set of all conjunctions of disjunctions of substitution dataobjects (cds's).

$h: \mathbb{P}_S \rightarrow 2^{S_0}$ The groundexpansion generalized to $\mathbb{P}_S: h(c) = \bigcap_{d \in c} \bigcup_{s \in d} h(s)$.

$\mathbb{F}_S = \{c \in \mathbb{P}_S \mid c \text{ is finite, } \forall d \in c. d \text{ is finite}\}$. The set of finite cds's.

We represent conjunctions of disjunctions by sets of sets (cds's). The ground-expansion is designed in order to measure the information content of a cds according to this representation. If $h(c_1) \subseteq h(c_2)$ then c_1 contains more information than c_2 . The set of finite cds's, \mathbb{F}_S , is all we need to represent the result of a finite computation. The information system of substitutions is now formed by defining consistency and entailment for such finite cds's.

Information system of substitutions: $(D_S, \Delta_S, \text{Cons}_S, \vdash_S)$

D_S, Δ_S

The set of dataobjects with a distinct uninformative element has already been defined.

$\text{Cons} = \{C \in \mathbb{F}_S \mid h(C) \neq \emptyset\}$

The set of finite cds's, which are consistent, i.e. Scott's Con modified to conjunctions of disjunctions.

$\vdash_S = \{(c, s) \in \mathbb{F}_S \times D_S \mid h(c) \subseteq h(s)\}$ The entailment relation, i.e. Scott's \vdash modified to conjunctions of disjunctions.

If a cds is consistent then there exists a groundsubstitution that is included in at least one alternative (disjunct) of every conjunct of this cds. Unlike Scott we define entailment from an inconsistent cds. A finite cds c entails every dataobject, which contains no more information than c contains. In particular, an inconsistent cds entails everything. The cd-domain of substitutions is defined as the set of deductively closed cds's ordered by information content, i.e. setinclusion:

$$\vdash_s = \{(c,d) \in \mathbb{F}_s \times 2^{\mathcal{D}_s} \mid \exists \{s_1, \dots, s_n\} \in \mathcal{D}_s. h(c) \subseteq \bigcup_{i=1}^n h(s_i)\}.$$

Entailment is generalized to disjunctions of dataobjects.

$\bar{\cdot} : \mathcal{P}\mathcal{P}_s \rightarrow \mathcal{P}\mathcal{P}_s$. The deductive closure of a cds:

$$\bar{u} = \{d \mid \exists \{d_1, \dots, d_n\} \subseteq u \forall f_1, \dots, f_n. (\forall i. f_i \subseteq d_i, f_i \text{ is finite}) \Rightarrow \{f_1, \dots, f_n\} \vdash_s d\}$$

$|S| = \{s \in \mathcal{P}\mathcal{P}_s \mid s = \bar{s}\}$ The cd-domain of substitutions, ordered by set-inclusion.

$\cdot|_V : |S| \times 2^{\mathcal{V}} \rightarrow |S|$ The restriction operator generalized to $|S|$:

$$s|_V \uparrow \{d|_V \mid d \in s\}, \text{ where the restriction of a disjunction is } \{s_1, \dots, s_n, \dots\}|_V = \{s_1|_V, \dots, s_n|_V, \dots\}.$$

The closure of a cds u consists of those disjunctions, which are finitely entailed by all finite subdisjunctions of some finite subconjunction of u . Note this repeated occurrence of the notion of finiteness.

The finite elements of $|S|$ are those which may be represented by a finite cds: $\{\bar{c} \mid c \in \mathbb{F}_s\}$. Every finite element s may be represented by a singleton, i.e. we can find $s_1, \dots, s_n \in \mathcal{D}_s$ such that $s = \overline{\{s_1, \dots, s_n\}}$ (for a proof, see [2]).

The bottom element $\perp_s = \bar{\emptyset} = \{\{\Delta_s\}\}$ is the only finitely uninformative element of $|S|$. Every conjunct of \perp_s offers at least one alternative (disjunct), which is uninformative. The ground expansion is $h(\perp_s) = S_0$.

The top element $\top_s = \{\bar{\emptyset}\} = 2^{\mathcal{D}_s}$ is the only finitely inconsistent element of $|S|$: \top_s includes the inconsistent empty disjunction, which entails everything. The ground expansion is $h(\top_s) = \emptyset$.

We have now established a domain basis for a semantics of logic programming: In [2] an operational semantics is constructed founded on a suitable inference operator. Here we present a denotational semantics.

6. A denotational semantics

We start by defining an abstract syntax, where literals are the basic syntactic unit.

Abstract syntax:

L	The set of literals is identical to the set of terms defined previously
N=L*	Negative clauses
C=LN	Positive (definite) clauses
P=C*	Programs

As mentioned priorly we do not distinguish predicate and function identifiers (L=T). Our motivation is merely to obtain technical simplicity. So the result should not be perceived as a proposal to extend logic programming beyond first order logic.

When specifying semantics, we shall see that the list ordering (*-notation) of negative clauses (N=L*) is irrelevant. A negative clause may be regarded as a finite set of literals. The same is true for programs with respect to clauses (P=C*).

Apart from the domain of substitutions we need a domain of program meanings, i.e. functions from questions (literals) to answers (substitutions).

Semantic domains:

$|S|$ The domain of substitutions.
 $M = L \rightarrow |S|$ The domain of program meanings.

Semantic functions:

$L: L \rightarrow M \rightarrow |S|$
 $N: N \rightarrow M \rightarrow |S|$
 $C: C \rightarrow M \rightarrow M$
 $P: P \rightarrow M$
 $P_\neg: P \rightarrow M$ (The meaning of a program in connection with negation by finite failure).

$$L[[l]]m = m(l)$$

$$N[[n]]m = \bigcup_{l \in n} L[[l]]m$$

$$C[[l_1 \cdot n]]m = \lambda l_2. (mgu(l_1 \theta, l_2)) \cup (N[[n]]m) \theta \Big|_{\text{var}(l_2)} \quad (*^1)$$

$$P[[p]] = \text{gfp}(\lambda m. \bigwedge C[[c]]m) \quad (*^2)$$

$$P_\neg[[p]] = \text{lfp}(\lambda m. \bigwedge_{\substack{c \in p \\ c \in p}} C[[c]]m) \quad (*^3)$$

(*¹): θ renames the variables in l_1, n in order to avoid common names of l_1, n and l_2 . The specific choice of new names is unimportant, since the variables of $(l_1, n)\theta$ are made anonymous by the restriction " $\Big|_{\text{var}(l_2)}$ ".

mgu denotes the most general unifier of two terms (literals)

(*²): The meaning of a question $(l?)$ in relation to a program p is the substitution $P[[p]]l$.

(*³): The meaning of a negated question $(\text{not}(l)?)$ in relation to a program (p) is the complemented substitution $\neg P_\neg[[p]]l$.

The semantics contains no detailed specification of the most general unifier (mgu). There exist recursive definitions of mgu [2], from which we may obtain a fixed point characterization on request.

The above renaming θ could be handled by an explicit introduction of environments if wanted.

Logic programming has a procedural interpretation, in which a positive clause may be regarded as a procedure declaration. This view leads us to perceive program meanings (M) as continuations. The meaning of a positive clause (C), becomes a continuation transformation. The above semantics is equivalent to the model theoretic semantics:

Theorem:

Let a program p and a question l be given and form the completion* p' of p . Then the following holds:

- i) The denotation of a program is correct:
 $\{s \in S_0 \mid p \models l[s]\} = h(P[[p]]l)$.
- ii) The same is true for negation by finite failure:
 $\{s \in S_0 \mid p' \models \neg l[s]\} = \neg h(P_\neg[[p]]l)$.

(*): The completion of a logic program is liberally speaking formed by replacing "if" by "if and only if", e.g. the completion of the program in ex.1 is:
 $\forall X, Y. (q(X, Y) \Leftrightarrow \exists Z. X=f(Z) \wedge Y=g(Z) \wedge p(Z)), \forall X. (p(X) \Leftrightarrow X=a)$ and some axioms about equality.

Proof: see [2].

Apart from the correctness stated above, it should be noted that the present semantics are more detailed about the notion of substitution than well-known semantics [1,3]. Nevertheless our semantics are independent of specific resolution strategies. Besides we have managed to characterize negation by finite failure in terms of the least fixed point of a continuous operator.

7. References:

- [1]: Apt, van Emden: "Contributions to the theory of logic programming". JACM, Vol. 29, 1982, p.p. 841-862.
- [2]: Frandsen: "Logic programming, substitutions and finite computability". DAIMI PB-186, January 1985. Computer Science Department, Aarhus University. (42 pages).
- [3]: Lloyd: "Foundations of Logic programming". Springer Verlag 1984.
- [4]: Scott: "Domains for denotational semantics". A corrected and expanded version of a paper prepared for ICALP'82, Aarhus, Denmark, July 1982.