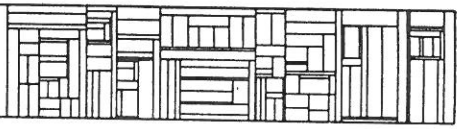
ISSN 0105-8517

A Denotational Semantics for Logic Programming

Gudmund Frandsen

DAIMI PB - 201
November 1985

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet



PB - 201 G. Frandsen: A Denotational Semantics for Logic Programming

TRYK: RECAU (06) 12 83 55

A DENOTATIONAL SEMANTICS FOR LOGIC PROGRAMMING

by

Gudmund Frandsen
Computer Science Department
Aarhus University

Abstract

A fully abstract denotational semantics for logic programming has not been constructed yet. In this paper we present a denotational semantics that is almost fully abstract. We take the meaning of a logic program to be an element in a Plotkin power domain of substitutions. In this way our result shows that standard domain constructions suffice, when giving a semantics for logic programming. Using the well-known fixpoint semantics of logic programming, we have to consider two different fixpoints in order to obtain information about both successful and failed computations. In contrast, our semantics is uniform in that the (single) meaning of a logic program contains information about both successful, failed and infinite computations. Finally, based on the full abstractness result, we argue that the detail level of substitutions is needed in any denotational semantics for logic programming.

1. INTRODUCTION

One may view semantics as the compound problem of defining proper universes of denotations and describe the meaning function that assigns denotations to programs. The universe of denotations should preferably be so fine-grained that it is possible to answer all interesting questions about a program from considering its denotation alone. Conversely, programs that we perceive to be equivalent should be assigned identical denotations. In addition the meaning function must be specified in a short comprehensible way.

Operational semantics may be considered the canonical way of defining semantics. Usually a program gives rise to a computation, which may contain a lot of irrelevant details such as the names of (locations assigned to) temporary variables. Therefore the meaning of a program is taken to be some proper abstraction of the corresponding computation.

Axiomatic and denotational semantics are techniques for specifying the meaning of a program directly without bothering about all the details contained in a computation. Yet, the correctness of an axiomatic or denotational semantics is usually defined with respect to a concrete operational semantics.

Denotational semantics is based on an abstract syntax (defined by a context-free grammar) and the meaning function is homomorphic in that the denotation of a compound syntactic phrase is determined uniquely by the denotations of the component phrases. In the most widely known form of denotational semantics, the Scott-Strachey approach, denotations are taken to be elements of Scott-domains.

Apart from correctness, it is considered a virtue of a denotational semantics to be fully abstract with respect to an operationally given meaning function $\mathcal{O}[\![\cdot]\!]$ [8,11,14], i.e. if two program components s_1, s_2 behave identically in all possible contexts $C[\cdot]$: $\mathcal{O}[\![C[s_1]]\!] = \mathcal{O}[\![C[s_2]]\!]$, then these program components have identical denotations, $\mathcal{D}[\![s_1]\!] = \mathcal{D}[\![s_2]\!]$.

It is the purpose of this paper to give an operational semantics for logic programming and develop a corresponding (almost) fully abstract denotational semantics. Such a task has not been accomplished previously but we sketch some important earlier ideas.

In the case of logic programming the original fixpoint semantics by van Emden and Kowalski [4] specified the meaning function in a very elegant way as the least fixpoint of an inference operator attributed to the program. However, the underlying universe of sets of groundterms was rather coarse-grained. By the time of origin, this was considered exclusively a virtue, since all posed questions regarding successful computations could be answered by means of this semantics. It could not be classified as neither operational, denotational nor axiomatic, but this fact was not (and should not be) considered a defect either.

Later, there was a demand for finding a proper semantics for negation as failure and infinite computations (cf. ex. 1). Negation as failure became understood through a model theoretic characterisation given by Clark [3], although completeness was an open problem until a paper by Jaffar, Lassez and Lloyd [9]. Simultaneously, people tried to extend the nice fixpoint semantics of van Emden and Kowalski to incorporate negation as failure. Apt and van Emden [1] discovered a discontinuity of the underlying inference operator in this connection. Recently van Emden, Lloyd and Nait Abdallah [5,10,12] have introduced different universes containing sets of infinite terms in order to preserve continuity of the underlying operator. This approach has also resulted in a characterization of infinite computations [10,12]. The ideas behind infinite term semantics seem very similar to the topological intuition behind Scott-domains and independently Frandsen [6,7] has established a denotational semantics technically based on the latter. In this case the universe of denotations consists of a double continuous substitution domain.

There are several points to note, when viewing the various semantics above separately and in comparison. Firstly, one must consider two different meanings in order to deal with both success and negation as failure. It seems that no work has yet explored

the possibility of making a uniform approach with only one meaning of a logic program. Secondly, the mathematical bases are all to some degree developed ad hoc. In particular the denotational semantics of Frandsen [6,7] uses a double continuous domain with a technically complicated construction without giving a convincing argument for the insufficiency of ordinary domain-constructions. Thirdly, the semantics vary with respect to detail level in that a universe of substitutions is more fine-grained than one of terms. This fact raises a natural question. Is it possible to construct a result similar to [6,7] by the use of terms alone? Or we may pose the question differently: Is the denotational semantics of [6,7] fully abstract with respect to some operational semantics? To avoid confusion, one should note that the original paper [6] uses the term "full abstraction" in the sense of Winskel [19], which is different from the sense in which the term is used in this paper [11,14].

We shall treat all three points above. Firstly, we build a universe of denotations by means of the usual Plotkin power domain construction [13,18,20], thus demonstrating the sufficiency of using ordinary constructions. Secondly, we give a uniform denotational semantics, i.e. there is just one meaning of a program and from this meaning information concerning both success, negation as failure and infinite computations can be extracted. Thirdly, we prove a weak form of full abstractness with respect to a specific operational semantics. Based on this "weak abstractness" result we provide an argument that the present substitution level of detail is necessary for any denotational semantics that has an expressive power at least equalling the fixpoint semantics of van Emden and Kowalski [4].

This paper may without reference presuppose acquaintance with some of the author's previous work [6,7]. The outline is as follows. We start by defining the possible computations for every possible logic program by means of non-deterministic, synchronic transition rules. Here non-deterministic means that program rules (definite clauses) are chosen non-deterministically and synchronic means that the subgoals contained in the righthandside of a program rule are pursued concurrently. In a later work we hope to investigate alternative choices. Our use of transition rules is

inspired by the structural approach to operational semantics recommended by Plotkin [15,16]. From the non-deterministic computations we abstract denotations in the form of substitutions. We form a modal language of substitution elements by use of Winskel's ideas [20] and a modified version of substitution data objects [6,7]. We associate a denotation to a logic program by considering those words of the modal language, which are true for the corresponding non-deterministic computation. In this way our universe of denotations actually becomes a Plotkin power domain. We continue to give a denotational semantics. It is very similar to the ones reported earlier [6,7] although a few deficiencies have been corrected. We prove the operational and denotational semantics to be equivalent and we prove that all syntactic categories apart from program rules (definite clauses) have a fully abstract denotation.

Example 1

We illustrate the different types of computation that a logic program may give rise to. The program rules used in this example are widespread in the literature [10,12].

program rules, r:

$$\mathrm{sum}(0,X,X) \Leftarrow \varepsilon.$$
$$\mathrm{sum}(s(X),Y,s(Z)) \Leftarrow \mathrm{sum}(X,Y,Z)$$
$$\mathrm{lsum}(A.X,B.Y,C.Z) \Leftarrow \mathrm{sum}(A,B,C),\ \mathrm{lsum}(X,Y,Z)$$

queries:

$$q_1: \Leftarrow \mathrm{sum}(s(0),U,V)$$
$$q_2: \Leftarrow \mathrm{sum}(s(0),W,0)$$
$$q_3: \Leftarrow \mathrm{lsum}(0.0.F,s(0).F,F)$$

computations: We use a simplified version of the syntax defined later for specifying computations such that bindings to temporary variables are not seen.

1) a successfull computation:
$r \vdash (q_1,[\,]) \rightarrow^2 [V \rightarrow s(U)]$

2) a failed computation:
$r \vdash (q_2,[\,]) \rightarrow \mathrm{fail}$

3) an infinite computation

$r \vdash (q_3, [\,]) \rightarrow^{n+1}$ (sum(A',B',C'),lsum(X',Y',Z'),
[F →1.1.2.3.5.... .fib(n)]) → ...,
where we have abbreviated $s^j(0)$ to simply j.

□

2. NON-DETERMINISTIC COMPUTATIONS

We will adopt Plotkin's general method for specifying computations [15,16]. To do this we must fix a set of possible programs by giving an abstract syntax, and we must specify the set of configurations each of which may be thought of as an instantaneous snapshot of a single computation branch. Every program phrase induces a set of possible transitions between configurations. In Plotkin's approach these transitions are defined by structural induction on the program phrases. Given a program and an initial configuration, c_0, we may perform a(n)(in)finite computation by composing transitions: $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow \ldots$. All such possible computation branches can be collected into a non-deterministic computation that constitutes the concrete meaning of a program as opposed to the abstract meaning, which we are going to extract later.

We start by defining the abstract syntax. It is slightly changed compared to [6,7] in that we use binary trees instead of lists. However, we still ignore arities and mix predicate identifiers with function identifiers for reasons of technical simplicity. We use countable sets of variables (V) and identifiers (I). The class of literals or terms over V,I is the basic syntactic unit: Lit = T = IT*|V. The total set of syntactic phrases are given by the following grammar rules:

```
 Prog ::= Rules Query
Rules ::= Rules or Rules | Head Tail
Query ::= Query co Query | Lit
 Head ::= Lit
 Tail ::= Query | ε
```

Our next concern is the structure of a configuration. Basically every configuration consists of a query and a substitution. An exception is made by terminal configurations that consist of a substitution or "fail" exclusively. In order to avoid name confusion by repeated use of the same rule we allow variables to be indexed by a string over {0,1,2}. The number of

zero's in an index represents the number of computation steps performed, when the corresponding indexed variable was created, and the sequence of 1's and 2's in an index represents a posi in the binary co-tree of a query. We define the set of indexed variables $V' = V \times \{0,1,2\}^*$. We will use the notation V_j for all variables with index j. In particular $V_\varepsilon = V$. Correspondingly the notions of literal and query is extended: $Lit' = T' = IT'^* \mid V'$ and $Q' = Q'$ co $Q' \mid Lit' \times \{0,1,2\}^*$. Queries on Q'-form are to be used as components of configurations. Each literal contained in a co-tree of Q' has attached a position index to insure the assignment of unique variable indexes in case of further computation. The following inductive assertion should hold for position indexes: For every co-tree $q \in Q'$ it is possible to find a position index k such that (i) if q is a literal, (l,j), then k = j and otherwise (ii) if q is a compound query, q_1 co q_2, then the position indexes assigned to q_1 and q_2 equals $k \cdot 1$ and $k \cdot 2$ respectively. In order to assure this requirement, we define a special operator that assigns indices to queries: For any queries q_1 co q_2, $l \in Query$ and indices j,k: q_1 co $q_2[j,k] = q_1[j,k \cdot 1]$ co $q_2[j,k \cdot 2]$ and $l[j,k] = (l[j],k)$ where l[j] is l with all variables indexed j.

The second component of a configuration is a substitution, $\Theta = V' \to T'$. A particular substitution has only non-trivial values for finitely many variables and contains no cycles. When applying a substitution to a term, we assume implicitly the substitution to be of idempotent form (e.g. $g(X)[X \to f(Y), Y \to a] = g(X)[X \to f(a), Y \to a] = g(f(a))$). Substitutions arise as a result of unification. We do not bother about the exact unification procedure in use, but we simply assume the existence of an algorithm mgu: $T' \times T' \times \ldots \times T' \to \Theta \cup \{fail\}$, which computes the most general unifier of two or more terms if possible and otherwise returns "fail" [17].

We are going to specify a synchronic transition function, i.e. sub-goals in a query are pursued in parallel. Consequently a need arises to combine independently computed substitutions. We define $\cdot : \Theta \times \Theta \to \Theta \cup \{fail\}$ as follows. Let p be any identifier and let $\{X_1,\ldots,X_n\}$ be all variables for which θ_1 or θ_2

has non-trivial values. Then $\theta_1 \cdot \theta_2 = mgu(p(X_1,\ldots,X_n), p(X_1[\theta_1],\ldots,X_n[\theta_1]), p(X_1[\theta_2],\ldots,X_n[\theta_2]))$. We will use the $\cdot$-operator even if one of the arguments is fail. In such a case the result is always fail.

We now define the set of configurations $C = (Q' \times S) \cup S \cup \{fail\}$. The configurations without a query component are the terminal ones. The transition relation is specified by structural induction [15,16] and we start by considering the transition caused by a single rule from a single-query state. In this case we simply perform one resolution step:

$$1)\quad \frac{0 \cdot j = j',\ h[j'] = h',\ \theta \cdot mgu(l,h') = \theta' \mid fail}{hq \vdash ((l,j)\theta) \to (q[j',j'],\theta') \mid fail,\ h\varepsilon \vdash ((l,j),\theta) \to \theta' \mid fail}$$

We should here make a comment on notation. The vertical bar is a shorthand for writing two different transition rules in one:

$\frac{x = a \mid b}{r \vdash c \to c_1 \mid c_2}$ abbreviates $\frac{x = a}{r \vdash c \to c_1}$ and $\frac{x = b}{r \vdash c \to c_2}$.

For composite program rules, we introduce non-determinism:

$$2)\quad \frac{r_1 \vdash c \to c'}{r_1 \ \underline{or}\ r_2 \vdash c \to c'} \qquad \text{and} \qquad 3)\quad \frac{r_2 \vdash c \to c'}{r_1 \ \underline{or}\ r_2 \vdash c \to c'}$$

For composite query states, we define synchronic transitions:

$$4)\quad \frac{r \vdash (q_1,\theta) \to (q_2',\theta_2) \mid \theta_1 \mid fail,\ r \vdash (q_2,\theta) \to (q_2',\theta_2) \mid \theta_2 \mid fail}{\begin{array}{l} r \vdash (q_1 \text{ co } q_2,\theta) \to (q_1' \text{ co } q_2',\theta_1\cdot\theta_2) \mid (q_1',\theta_1\cdot\theta_2) \mid fail \\ \qquad \mid (q_2',\theta_1\cdot\theta_2) \mid \theta_1\cdot\theta_2 \mid fail \\ \qquad \mid fail \mid fail \mid fail \end{array}}$$

Given a logic program $p = rq$ the corresponding non-deterministic computation is $(T_p,\to)$, where $T_p = \{c \mid r \vdash c_0 \to^* c\}$ given the initial configuration $c_0 = (q[\varepsilon,\varepsilon],[])$.

The next step consists in constructing a suitable universe of denotations. Preferably, it should be coarse-grained. However, a full abstractness result will later indicate that the deno-

tation of a query (in a denotational semantics) must have the detail level of substitutions. In order to avoid constructing both a coarse-grained domain (e.g. based on terms) and a more fine-grained of substitutions, we simply let the universe of denotations be a domain of substitutions.

3. FINITE SUBSTITUTION ELEMENTS

We choose a power domain of substitutions for our universe of denotations. As a basis of the power domain construction, we need a preorder of finite substitution elements. The construction of such preorder is the goal of this section. In a previous paper [7] we have introduced the notion of information content. Actually, this notion was too poor, and led us to demand a double continuous domain. Here we use a richer notion of information content, which distinguishes substitutions occurring in non-terminal state of a configuration from substitutions that are the result of a successful computation. Consider the following example:

Example 2

program rules: r_1: $d(X) \Leftarrow d(f(X))$

r_2: $d(X) \Leftarrow$

query: q: $\Leftarrow d(Y)$

computations: T_1: $r_1 \vdash ((d(Y),\varepsilon),[]) \rightarrow ((d(f(X_0)),0),[Y \rightarrow X_0]) \rightarrow \ldots$

T_2: $r_2 \vdash ((d(Y),\varepsilon),[]) \rightarrow [Y \rightarrow X_0] \not\rightarrow$

We notice that the substitution $[Y \rightarrow X_0]$ occurs in both computations, but in different senses. In T_1 we can only say that $[Y \rightarrow X_0]$ delimits the possible results of successful computation branches (if any exist). In the case of T_2 we know in addition that the substitution itself represents a successful result. □

This example and the discussion of anonymous variables in [7] lead us to choose the following definition of substitution data elements: $\mathcal{D} = (\{\bot, s\} \times P(V') \times \theta) \cup \{fail\}$. The first component x of a data element d expresses whether d denotes a possible result ($x = \bot$) or a certain result ($x = s$). The second component is a set of named (as opposed to anonymous) variables and the third component consists of a finite cyclefree substitution.

The information content of a substitution element is expressed by two functions $h_\bot, h_s: \mathcal{D} \rightarrow 2^{S_0}$, where $S_0 = V' \rightarrow T_0$ is the set of ground substitutions and $T_0 = IT_0^*$ is the set of ground

terms: $h_\perp(\text{fail}) = h_s(\text{fail}) = h_s(\perp,W,\theta) = \emptyset$ and $h_\perp(\perp,W,\theta) = h_\perp(s,W,\theta) = h_s(s,W,\theta) = h(\theta)|_W$, where $h(\theta) = \{s \in S_0 \mid \forall v \in V'.\ \theta(v)[s] = s(v)\}$ and $S|_W = \{s' \in S_0 \mid \exists s \in S.\ \forall v \in W.\ s(v) = s'(v)\}$. $h(\theta)|_W$ selects all the $s \in S_0$ that agree with θ on the named variables W. Hence $h_\perp$ induces a filter ordering on $\mathcal{D}$: $h_\perp(d_1) \supsetneq h_\perp(d_2)$ means that d_1 allows less possibilities than d_2. However, we cannot conclude that d_1 therefore contains more information (i.e. is better determined) than d_2. It may be that d_2 in addition contains information about certainty (success): $h_s(d_2) = h_\perp(d_2)$ and in this case the information content of the two substitution elements are incomparable independently of the value of $h_s(d_1)$, since $h_s(d_1) \subseteq h_\perp(d_1)$.

We define a preorder on $\mathcal{D}$: $d_1 \sqsubseteq d_2$ iff $h_\perp(d_1) \supseteq h_\perp(d_2) \wedge h_s(d_1) \subseteq h_s(d_2)$. It is easily verified that $\perp = (\perp,\emptyset,[\ ])$ is a minimal element in this preorder. Returning to example 2, we can denote the two versions of the substitution $[Y \to X_0]$ occurring in T_1 and T_2 by respectively $d_1 = (\perp,\{Y\},[Y \to X_0])$ and $d_2 = (s,\{Y\},[Y \to X_0])$, which have different information contents: $d_1 \sqsubsetneq d_2$.

For later use we define some operators on the preorder $(\mathcal{D},\sqsubseteq)$. Restriction $\cdot|_W: \mathcal{D} \to \mathcal{D}$ is a monotone idempotent function: $(x,W_1,\theta)|_{W_2} = (x,W_1 \cap W_2,\theta)$ and $\text{fail}|_W = \text{fail}$.

Reindexing $R_{ij}: \mathcal{D} \to \mathcal{D}$ is a monotone function: $R_{ij}(\text{fail}) = \text{fail}$ and if $W \cap (V'_i \cup V'_j) = \emptyset$, $V'_i \subseteq V_i$, $V'_j \subseteq V_j$ then $R_{ij}(x,W \cup V'_i \cup V'_j,\theta) = (x,W' \cup V_i \cup V_j,\theta')$, where θ' is a reindexed version of θ: variables in $V_i \setminus V'_i$ and $V_j \setminus V'_j$ have indices changed to some k respectively l that does not occur in θ or in $W' \cup V_i \cup V_j$. Moreover variables in V'_i and V'_j have indices switched to j and i respectively. W' consists of precisely those variables in W that do occur in θ.

The composition operator that we have defined on θ can be extended to a monotone, associative, commutative and absorptive function $\cdot: \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ (if we define associativity etc. modulo the equivalence $\sqsubseteq \cap \sqsupseteq$): $\text{fail} \cdot d = d \cdot \text{fail} = \text{fail}$ and

$$(x_1,W_1,\theta_1).(x_2,W_2,\theta_2) = \begin{cases} (x_1\cdot x_2, W_1 \cup W_2, \theta_1' \cdot \theta_2'), & \theta_1' \cdot \theta_2' \neq \text{fail} \\ \text{fail} & , \text{otherwise} \end{cases}$$

where

$$x_1 \cdot x_2 = \begin{cases} s & , x_1 = x_2 = s \\ \perp & , \text{otherwise} \end{cases}$$

and where the anonymous variables of θ_1 and θ_2 have been renamed to avoid unwanted common variables: θ_i' is θ_i with all variables in $\mathcal{C}W_i$ renamed away from $W_i \cup W_{3-i}$ (cfr. [6], p. 40). The renaming must preserve indices of variables. One may easily verify that $1 = (s,\emptyset,[\,])$ is a neutral element for the combination function.

Simple distributive rules relate these operators pairwise:

Theorem 1 (The equivalence $\simeq$ abbreviates $\sqsubseteq \cap \sqsupseteq$)

i) if $W_1 \cap W_2 \subseteq W$ then $((x_1,W_1,\theta_1)\cdot(x_2,W_2,\theta_2))|_W \simeq (x_1,W_1,\theta_1)|_W \cdot (x_2,W_2,\theta_2)|_W$

ii) if index $i \notin \{j,k\}$ then $R_{jk}(d)|_{V_i} \simeq d|_{V_i}$, and $R_{jk}(d|_{V_j}) \simeq R_{jk}(d)|_{V_k}$

iii) $R_{ij}(d_1 \cdot d_2) \simeq R_{ij}(d_1) \cdot R_{ij}(d_2)$.

Proof We only consider (i). Here the equivalence $(x_i,W_i,\theta_i) \simeq (x_i,W_i,\theta_i')$ should be used repeatedly (θ_i' refers to the definition of the $\cdot$-operator). □

We will now define an operational meaning of a logic program based on computations, finite substitution elements and modal operators.

4. OPERATIONAL SEMANTICS

We have previously defined the non-deterministic computation induced by a given logic program. The abstract meaning of a logic program is now abstracted from the corresponding non-deterministic computation. So we define the denotation of single programs before we construct the entire universe of denotations (in the next section). However, this sequence of affairs helps us to select the proper ordering of subsets of $\mathcal{D}$, namely the Egli-Milner ordering, which is used in the power domain-construction.

The abstract meaning is defined using Winskel's modal characterisation of non-deterministic computations [20]. We start by formalizing an earlier observation. Every configuration in a computation can naturally be assigned a finite substitution element (cfr. the reference to ex. 2 on page 11). Given a computation $(T_p,\rightarrow)$, define $val:T_p \rightarrow \mathcal{D}$ by

$$val(c) = \begin{cases} (\bot,V_\varepsilon,\theta) & , \quad c = (q,\theta) \\ (s,V_\varepsilon,\theta) & , \quad c = \theta \\ fail & , \quad c = fail \end{cases}$$

Here all named variables in val(c) are index-free and correspond to the variables occurring in the initial query of the computation. We use the term non-deterministic $\mathcal{D}$-computation for the tuple $(T_p,\rightarrow,val)$. It should be noted that $c \rightarrow c'$ implies $val(c) \sqsubseteq val(c')$ (since $h(\theta_1 \cdot \theta_2) \subseteq h(\theta_1)$).

Winskel discusses three different modal languages for talking about non-deterministic computations. Only one of these languages uses both the operator $\Box$ (necessary) and the operator $\Diamond$ (possible). We need both of them in that $\Diamond$ is used to describe success (there exists a computation branch such that) whereas $\Box$ is used to describe failure (for all computation branches it is the case that ...). Consequently we define the following modal language (L):

Syntax: The words of L are defined inductively: L is the least language such that

basis $\mathcal{D}$ is a subset of L

step if $s,s' \in L$ then i) $s \vee c' \in L$,
ii) $\Box s \in L$ and
iii) $\Diamond s \in L$.

Semantics: Given a non-deterministic $\mathcal{D}$-computation $(T,\rightarrow,val)$, we define the satisfaction relation $\models$ on $T \times L$ as the least relation, which satisfies

basis let $d \in \mathcal{D}$, if $d \sqsubseteq val(c)$ then $c \models d$

step let $s,s' \in L$

i) if $c \models s$ or $c \models s'$ then $c \models s \vee s'$

ii) if $c \models s$ or $[\exists c' \in T.c \rightarrow c' \wedge (\forall c' \in T.c \rightarrow c' \Rightarrow c' \models \Box s)]$ then $c \models \Box s$

iii) if $c \models s$ or $[\exists c' \in T.c \rightarrow c' \wedge c' \models \Diamond s]$ then $c \models \Diamond s$

If c_0 is the initial configuration in T, we define $T \models s$ iff $c_0 \models s$. Let us comment on this definition. If d is on the form (s,V_ε,θ) then $T_p \models \Diamond d$ means that it is possible non-deterministically to choose program rules from p such that the initial query of p gives rise to a successful computation resulting in the substitution θ. Conversely, $T_p \models \Box$ fail means that every non-deterministic choice of rules from p results in a failed computation.

The characterisation of an infinite computation is a bit more complicated: If $p = rq_3$ (with rq_3 as defined in example 1) then p computes the Fibonacci-sequence and in fact $T_p \models \Box(\bot,\{F\},[F \rightarrow 1.1.2.3.5.8. \ldots .fib(n)])$ for all n (but $T_p \not\models \Box$ fail).

We now define the operational meaning of a logic program p: $\mathcal{O}[\![p]\!] = \{\{d_1,\ldots,d_n\} \mid T_p \models \Box(d_1 \vee \ldots \vee d_n), T_p \models \Diamond d_1,\ldots,T_p \models \Diamond d_n\}$. We proceed to define the universe of denotations alias the Plotkin-power domain over $\mathcal{D}$. We shall later see that $\mathcal{O}[\![p]\!]$ is an element in this universe.

5. DOMAIN OF DENOTATIONS

We will now construct a power domain of substitutions using ideals as described by Winskel [20]. We choose the Plotkin-power domain that is based on the Egli-Milner ordering, because this power domain has the same expressive power as the modal language we have just considered [20]. By using one of the two weaker (Hoare- or Smyth-) power domains with correspondingly weaker orderings we would exclude ourselves from speaking about either failure or success respectively.

Winskel's construction is actually based on a domain (ω-algebraic complete partial order). However, only finite elements of this domain are used and the construction works for the preorder $(\mathcal{D}, \sqsubseteq)$ as well. We start by forming M[D], the finite subsets of $\mathcal{D}$. M[D] is a preorder, ordered by the Egli-Milner ordering $\leqq$:

$$m_1 \leqq m_2 \quad \text{iff} \quad \forall d_1 \in m_1\ \exists d_2 \in m_2.\ d_1 \sqsubseteq d_2 \wedge \forall d_2 \in m_2\ \exists d_1 \in m_1\ \ d_1 \sqsubseteq d_2.$$

Intuitively, an $m \in M[D]$ represents information about some cross section of a non-deterministic $\mathcal{D}$-computation. We now use completion by ideals to form the power domain P[D] from M[D] :

$$P[D] = \{X \subseteq M[D] \mid X \neq \emptyset,\ \text{(i)}\ \forall m_1, m_2 \in M[D].\ m_1 \leqq m_2 \in X \Rightarrow m_1 \in X,$$
$$\text{(ii)}\ \forall m_1, m_2 \in X.\ \exists r \in X.\ m_1 \leqq r \wedge m_2 \leqq r\},$$

i.e. P[D] consists of leftclosed (i) and directed (ii) non-empty subsets of M[D]; and the power domain is ordered by setinclusion.

There is a natural monotone insertion operator $i: \mathcal{D} \to P[D]$, $i(d) = \{m \mid m \leqq \{d\}\}$. $i(\bot)$, $i(1)$ and $i(\text{fail})$ is simply denoted $\bot$, 1 and fail respectively. In addition there is a natural continuous union operator $\uplus$: $P[D] \times P[D] \to P[D]$, which is associative, commutative and absorptive: $p_1 \uplus p_2 = \{m_1 \cup m_2 \mid m_1 \in p_1,\ m_2 \in p_2\}$.

We would like to define restriction, reindexing and combination operators on P[D], generalising the operators on $\mathcal{D}$. For this purpose we state a general theorem:

Theorem 2 Given a monotone function $f: \mathcal{D}^n \to \mathcal{D}$, there exists a unique extension of f, denoted $\hat{f}: P[D]^n \to P[D]$ such that $\hat{f}$ is continuous and linear, which latter property means that $\hat{f}(f)$ is a homomorphism with respect to $\cup$ and i. Moreover

i) For unary f: The extension preserves idempotency.

ii) For binary f: The extension preserves associativity and commutativity but not necessarily absorptivity.

Proof We only sketch a proof in the case of a unary f: Given $p \in P[D]$, we know that $p = \bigsqcup_n p_n$ for a sequence of finite elements $\{p_n\} \subseteq P[D]$. Each p_n is of the form $p_n = \{m \mid m \leq \{m_n\}\}$ for some $m_n \in M[D]$, i.e. $m_n = \{d_{n_1}, \ldots, d_{nj_n}\} \subseteq \mathcal{D}$. This means that $p_n = i(d_{n_1}) \cup \ldots \cup i(d_{nj_n})$ and because we have required $\hat{f}$ to be continuous and linear, there is only one possible extension of f: $\hat{f}(p) = \bigsqcup_n (i(f(d_{n_1})) \cup \ldots \cup i(f(d_{nj_n})))$. Conversely, this extension does exist as it fulfils $\hat{f}(p) = \overline{\{\{f(d_1), \ldots, f(d_n)\} \mid \{d_1, \ldots, d_n\} \in p\}}$, where the horizontal bar denotes leftclosure, i.e. $\overline{X} = \{m \in M[D] \mid \exists m' \in X.\ m \leq m'\}$. □

Theorem 2 means that we have a continuous linear combination operator, $\cdot: P[D] \times P[D] \to P[D]$, which is associative and commutative. Given $W \subseteq V'$ we have a continuous linear restriction operator $\cdot|_W: P[D] \to P[D]$, which is idempotent and given two indices $i,j \in \{0,1,2\}^*$ we have a continuous linear reindexing operator $R_{i,j}: P[D] \to P[D]$.

Furthermore Theorem 1 extends to P[D] by using continuity and linearity of the involved operators.

The universe of denotations will now be used by a denotational semantics.

6. A DENOTATIONAL SEMANTICS

We regard the synchronic operational semantics ($\mathcal{O}[\![\cdot]\!]$) as the fundamental meaning of a logic program. We will now specify this meaning without talking about computations, but using the denotational method due to Scott and Strachey.

In this section we present a denotational semantics, and we prove its correctness with respect to the operational semantics. In the next section, we discuss to what degree the present denotational semantics is fully abstract. We have already defined an abstract syntax so we simply describe the semantic functions that use the domain K = Lit → P[D] apart from P[D].

$\mathcal{P}$: Prog → P[D]
$\mathcal{R}$: Rules → K → K
$\mathcal{Q}$: Query → K → P[D]
$\mathcal{H}$: Head → Lit → P[D]
$\mathcal{T}$: Tail → K → P[D]

$\mathcal{P}[\![rq]\!] = \mathcal{Q}[\![q]\!]\ (\mathrm{lfp}(\mathcal{R}[\![r]\!]))$
$\mathcal{R}[\![r_1\ \underline{\mathrm{or}}\ r_2]\!]kl = \mathcal{R}[\![r_1]\!]kl \cup \mathcal{R}[\![r_2]\!]kl$
$\mathcal{R}[\![ht]\!]kl = [\mathcal{H}[\![h]\!]l \cdot R_{\varepsilon,0}(\mathcal{T}[\![t]\!]k)]|_{V_\varepsilon}$
$\mathcal{Q}[\![q_1\ \underline{\mathrm{co}}\ q_2]\!]k = \mathcal{Q}[\![q_1]\!]k \cdot \mathcal{Q}[\![q_2]\!]k$
$\mathcal{Q}[\![1]\!]k = k(1)$
$\mathcal{H}[\![h]\!]1 = i(s, V_0 \cup V_\varepsilon, \mathrm{mgu}(h[0], 1))$
$\mathcal{T}[\![q]\!]k = \mathcal{Q}[\![q]\!]k$
$\mathcal{T}[\![\varepsilon]\!]k = 1.$

The rest of this section is devoted to proving

<u>Theorem 3</u> $\quad \forall p \quad \mathcal{O}[\![p]\!] = \mathcal{P}[\![p]\!].$

The main line in the proof is borrowed from Hennessy & Plotkin [8]. They characterize both the operational and the denotational meanings as fixpoints of suitably defined operators. We apply this strategy. Let $r \in$ Rule be a rule and define ψ_r: $(Q' \to P[D]) \to (Q' \to P[D])$ by

$$\psi_r(\xi)(q) = (\cup\{[i(s,V_\theta,\theta)\cdot\xi(q')]|_{V_q} \mid r \vdash (q,[\,]) \to (q',\theta)\})$$
$$\cup(\cup\{i(s,V_\theta,\theta)|_{V_q} \mid r \vdash (q,[\,]) \to \theta\})$$
$$\cup(\cup\{fail \mid r \vdash (q,[\,]) \to fail\}),$$

where V_θ and V_q denote the set of variables occurring in θ and q respectively. Furthermore define φ_r: $(Q' \to \theta \to P[D]) \to (Q' \to \theta \to P[D])$ by

$$\varphi_r(\xi)(q)(\theta) = (\cup\{\xi(q')(\theta') \mid r \vdash (q,\theta) \to (q',\theta')\})$$
$$\cup\ (\cup\{i(s,V_\theta,,\theta') \mid r \vdash (q,\theta) \to \theta'\})$$
$$\cup\ (\cup\{fail \mid r \vdash (q,\theta) \to fail\}).$$

Finally let id: $Q' \to \theta \to P[D]$ be defined by $id(q)(\theta) = i(\bot,V_\theta,\theta)$. Theorem 3 is a simple consequence of the following three lemmas:

Lemma 1: If $p = rq_0$ then $\mathcal{P}[\![p]\!] = lfp(\psi_r)(q_0[\varepsilon,\varepsilon])$.

Lemma 2: If $p = rq_0$ then $\mathcal{O}[\![p]\!] = \bigcup_n \varphi_r^n(id)(q_0[\varepsilon,\varepsilon])([\,])|_{V_\varepsilon}$

Lemma 3: If $r \in Rules$, $q \in Q'$ then $lfp(\psi_r)(q) = \bigcup_n \varphi_r^n(id)(q)([\,])|_{V_q}$

We sketch the proofs of all three lemmas, but leave out details that could obscure the simple basic structure of the proofs.

Proof of lemma 1 Consider the following two assertions:

(i) Given $j,k \in \{0,1,2\}^*$ such that j is a subsequence of k and given $q \in Query$ it is the case that
$\forall n.\, R_{\varepsilon,j}(Q[\![q]\!](R[\![r]\!]^n\bot)) = \psi_r^n(\bot)(q[j,k])$.

(ii) Given q_1 co $q_2 \in Q'$ then $\forall n.\ \psi_r^n(\bot)(q_1) \cdot \psi_r^n(\bot)(q_2) = \psi_r^n(\bot)(q_1$ co $q_2)$.

Observe that lemma 1 follows from (i), which is itself proved by induction on n. The induction step is proved by structural induction on q. Here the basis step ($q \in Lit$) uses Theorem 1

in the P[D]-version and the induction step ($q = q_1$ co q_2) uses (ii) above in addition. For the proof of (ii) we also use induction on n. In this case the induction step uses Theorem 1 in the P[D]-version.

Proof of lemma 2 Define the computation tree $T^n_{r,c}$, i.e. the collection of configurations obtainable from c in at most n steps, inductively as follows: $T^0_{r,c} = \{c\}$ and $T^{n+1}_{r,c} = \{c\} \cup \{c_1 \mid \exists c_2 \in T^n_{r,c}.\ r \vdash c_2 \to c_1\}$, and define the natural meaning correspondingly:

$$M^n_{r,c} = \{\{d_1,\ldots,d_m\} \mid T^n_{r,c} \models \Box(d_1 \vee \ldots \vee d_m),\ T^n_{r,c} \models \Diamond d_1,\ldots,T^n_{r,c} \models \Diamond d_m\}.$$

Now consider the statement

(iii) Given T_p let (q,θ) be a configuration that satisfy $\exists\theta'.\ (q,\theta\theta') \in T_p$, then $\forall n.\ M^n_{r,(q,\theta)} = \varphi^n_r(id)(q)(\theta)|_{V_\varepsilon}$

Observe that lemma 2 is a simple consequence of (iii), which is easily proved by induction on n.

Proof of lemma 3 Consider the following two assertions

(iv) $\forall n.\ \psi^n_r(\bot)(q) = \varphi^n_r(id)(q)([\ ])|_{V_q}$ for $q \in Q'$

(v) Let (q,θ) be given such that θ only refers variables that have indices, which are subsequences of one or more of the position indices occurring in q, then $\forall n.\ i(s,V_\theta,\theta) \cdot \varphi^n_r(id)(q)([\ ]) = \varphi^n_r(id)(q)(\theta)$.

Lemma 3 follows immediately from (iv). Both (iv) and (v) are proved by induction on n. In the case of (iv) the induction step uses (v) and in the case of (v) one must use the induction assumption twice. □

Following this briefing of a complex proof, we proceed to discuss full abstractness.

7. FULL ABSTRACTNESS

It appears that the denotational semantics described in the last section to a large extent is fully abstract in that the semantic functions H, Q and T all are non-redundant in the sense required for full abstractness. In one way it should be no surprise that Q is fully abstract. In the definition of Q we have carefully sought to eliminate all names of temporary variables that have to appear in an actual computation. This means that $Q[\![q_1 \text{ co } q_2]\!] = Q[\![q_2 \text{ co } q_1]\!]$ for arbitrary q_1, q_2 although q_1 co q_2 and q_2 co q_1 in general generate different computations. However, the full abstractness result for Q is non-trivial: One may draw an analogue to Algol-like languages [2]. Consider the two program phrases "new x = 0; new y = 1" and "new y = 1; new x = 0". In a concrete operational semantics, the assignment of location to x and y would differ for the two phrases. However, the abstract behaviour of a program is unaffected by changing one phrase to the other. The construction of fully abstract semantics that reflected the irrelevancy of exact location assignment has appeared difficult [2].

Unfortunately, the R-semantic function is not fully abstract. We later present a counter example and suggest alternative definitions of R. We start by proving H, Q, T to be fully abstract:

Theorem 4 Let S denote one of the semantic functions H, Q or T and let s_1, s_2 be program phrases of proper syntactic category. If $S[\![s_1]\!] \neq S[\![s_2]\!]$ then we may find a context $C[\cdot]$ such that $O[\![C[s_1]]\!] \neq O[\![C[s_2]]\!]$.

Proof $S = H$: Assume $H[\![h_1]\!] \neq H[\![h_2]\!]$, in which case $h_1 \neq h_2$. Let $\{X_1, \ldots, X_n\}$ be the set of all variables occurring in h_1 or h_2 and let $\theta = [X_1 \to c_1, \ldots, X_n \to c_n]$, where none of the identifiers $c_1, \ldots, c_n$ occur in h_1 or h_2. Define $C[\cdot] = (\cdot \Leftarrow p(X_1, \ldots, X_n)$ or $p(c_1, \ldots, c_n) \Leftarrow \varepsilon)$ $(\Leftarrow h_1[\theta])$, where the identifier p does not occur in h_1, then $O[\![C[h_1]]\!] \supseteq 1 \not\subseteq \text{fail} = O[\![C[h_2]]\!]$, where $p_1 \sqsubseteq p_2$ iff $p_1 \uplus p_2 = p_2$.

$\underline{S = Q}$: Assume $\mathcal{Q}[\![q_1]\!] \neq \mathcal{Q}[\![q_2]\!]$, in which case q_1 and q_2 cannot be identical when considered as multisets of literals (by the associativity and commutativity of $\cdot$). Furthermore $\mathcal{Q}[\![l]\!] \in \{\bot, 1, \text{fail}\}$ for a variablefree literal l, which implies absorptivity: $\mathcal{Q}[\![l]\!] \cdot \mathcal{Q}[\![l]\!] = \mathcal{Q}[\![l]\!]$ for such l. Using these observations we need only consider two cases:

i) q_1 and q_2 are not identical, when considered as sets of literals, i.e. $\exists l \in q_2.\ l \notin q_1$.

ii) q_1 and q_2 are identical, when considered as sets of literals, but they are not identical when considered as multisets and their "multiset-difference" contains a literal with variables, i.e. $q_1 \overset{\text{set}}{=\!=} q_2$, $\exists l \in q_1$. l has variable X, $\#l$ in $q_1 = a > b = \#l$ in q_2.

First consider case (i): Let $\{X_1,\ldots,X_n\}$ be all variables occurring in q_1 or q_2 and let $\theta = [X_1 \to c_1,\ldots,X_n \to c_n]$, where none of the identifiers $c_1,\ldots,c_n$ occur in q_1 or q_2. Moreover let $l_1,\ldots,l_k$ be all literals belonging to q_1 viewed as a set. Define the context $C[\cdot] = (p(X_1,..,X_n) \Leftarrow \cdot \ \underline{\text{or}}\ l_1[\theta] \Leftarrow \varepsilon \ \underline{\text{or}}\ \ldots \ \underline{\text{or}}\ l_k[\theta] \Leftarrow \varepsilon)$ $(\Leftarrow p(c_1,\ldots,c_n))$, where the identifier p does not occur in q_1. Then $\mathcal{O}[\![C[q_1]]\!] \supseteq 1 \not\subseteq \text{fail} = \mathcal{O}[\![C[q_2]]\!]$, where $p_1 \subseteq p_2$ iff $p_1 \uplus p_2 = p_2$.

Next consider case (ii): let $\{X,X_1,\ldots,X_n\}$ be all variables occurring in q_1 (or q_2) and let $\theta = [X_1 \to c_1,\ldots,X_n \to c_n]$, where none of the identifiers $c_1,\ldots,c_n$ occur in q_2 (or q_1). Moreover let $\theta_i = [X \to f(Y_1,\ldots,Y_{i-1},c_i,Y_{i+1},\ldots,Y_a)]$, $1 \leq i \leq a$, where neither the identifier f, nor the variables $W = \{Y_1,\ldots,Y_a\}$ occur in q_1 (or q_2) and let $l,l_1,\ldots,l_k$ be all literals occurring in q_1 (or q_2). Define the context $C[\cdot] = (p(X,X_1,\ldots,X_n)) \Leftarrow \cdot \ \underline{\text{or}}$ $l[\theta\theta_1] \Leftarrow \varepsilon \ \underline{\text{or}}\ \ldots \ \underline{\text{or}}\ l[\theta\theta_a] \Leftarrow \varepsilon \ \underline{\text{or}}\ l_1[\theta] \Leftarrow \varepsilon \ \underline{\text{or}}\ \ldots \ \underline{\text{or}}\ l_k[\theta] \Leftarrow \varepsilon)$ $(\Leftarrow p(f(Y_1,\ldots,Y_a),c_1,\ldots,c_n))$, where the identifier p does not occur in q_1 (or q_2). Then $\mathcal{O}[\![C[q_1]]\!] \supseteq i(s,W,[Y_1 \to c_1,\ldots,Y_a \to c_a]) \not\subseteq \mathcal{O}[\![C[q_2]]\!]$.

$\underline{S = T}$: Assume $T[\![t_1]\!] \neq T[\![t_2]\!]$. If $t_1, t_2 \in$ Query then we may use the contexts constructed above. Otherwise we may assume $t_1 = \varepsilon$ and $t_2 \in$ Query. In this case define the context $C[\cdot] = (p \Leftarrow \cdot)(\Leftarrow p)$ where p is an identifier not occurring in t_2. Then $O[\![C[t_1]]\!] = 1 \neq \text{fail} = O[\![C[t_2]]\!]$. □

It would be pleasant to argue for the necessity of using substitutions based on this full abstractness result. However, it is not directly possible to do so, since we have introduced substitutions in the abstract operational meaning. Suppose we had defined a Plotkin-power domain of terms similar to the present one of substitutions (i.e. the finite elements should be $\{\perp, s\} \times T$, and the preordering comes from taking all variables of a term to be anonymous), and suppose moreover that the abstract operational meaning was defined as an element of this power domain of terms in the natural way. In this case, we claim that a denotational semantics would still need a power domain of substitutions in order to give meaning to a query. We base our claim on the proof of Theorem 4 (iii). Consider case (i) of the proof. Only the elements "1" and "fail" of the power domain are referred to. These elements should translate to "$i(s,p(c_1,\ldots,c_n))$" and "fail" respectively in a power domain of terms. In case (ii) the element "$i(s,W,[y_1 \to c_1,\ldots,y_a \to c_a])$ translates to "$i(s,p(f(c_1,\ldots,c_a), c_1,\ldots,c_a))$". So the basic construction in the full abstractness proof for Q seems to work also when the operational meaning is taken to be an element in a Plotkin-power domain of terms.

Let us see, why R is not fully abstract. Take r_1 to be $p(f(X)) \Leftarrow p(X)$ $\underline{\text{or}}$ $p(0) \Leftarrow \varepsilon$ and let $r_2 = r_1$ $\underline{\text{or}}$ $p(f^{100}(0)) \Leftarrow \varepsilon$. It should be clear that $R[\![r_1]\!] \neq R[\![r_2]\!]$, but $O[\![C[r_1]]\!] = O[\![C[r_2]]\!]$ for any context $C[\cdot]$. The impossibility of distinguishing r_1 and r_2 operationally seems to rely on the fact that $\text{lfp}(R[\![r_1]\!]) = \text{lfp}(R[\![r_2]\!])$, i.e. operationally two sets of rules are only distinguished, if their behaviours "in infinity" differ. A natural solution to the full abstractness problem consists in defining a new semantic function R' by $R'[\![r]\!] = \text{lfp}(R[\![r]\!])$ (and $P'[\![rq]\!] = Q[\![q]\!](R'[\![r]\!])$. R' is easily seen to be correct and fully abstract, but a non-trivial problem remains: How do we specify $R'[\![r_1$ $\underline{\text{or}}$ $r_2]\!]$ as a function of

$R'[\![r_1]\!]$ and $R'[\![r_2]\!]$? A more complicated solution may arise from the following semantic function: $R''[\![hq]\!] = G(R[\![hq]\!])$ and $R''[\![r_1 \underline{\text{or}} r_2]\!] = G(R''[\![r_1]\!] \uplus R''[\![r_2]\!])$, where $G = \text{lfp}(G')$, $G'(g)(L)k = L(k) \uplus L(g(L)k)$. R'' certainly is homomorhpic. In return it seems non-trivial to prove correctness and full abstractness. A third method for obtaining full abstractness consists in augmenting our language with a construct that makes use of the detailed information delivered by the present semantic function R. Such a construct could take the form of a flag, which attached to a program rule indicated that the rule concerned could be used no more than once in a single (branch of) computation.

8. FUTURE WORK

We gave a non-deterministic, synchronic operational semantics without arguing for the reasonableness of this choice. We intend to compare the present choice with mixed synchronic/ sequential and pure sequential semantics in the future.

References

[1] Apt. K.R. and van Emden, M.H.: "Contributions to the theory of logic programming", Journal of the ACM, Vol. 29, 1982, pp. 841-862.

[2] Brookes, S.D.: "A fully abstract semantics and a proof system for an Algol-like language with sharing", Carnegie Mellon University, 1985, CMU-CS-84-118A.

[3] Clark, K.L.: "Negation as failure". In "Logic and databases" (eds.: Gallaire, H. and Minker, J., Plenum Press, New York), 1978, pp. 293-322.

[4] van Emden, M.H. and Kowalski, R.A.: "The semantics of predicate logic as a programming language", Journal of the ACM, Vol. 23, 1976, pp. 733-742.

[5] van Emden, M.H. and Nait Abdallah, M.A.: "Top down semantics of fair computations of logic programs", Journal of Logic Programming, Vol. 2, 1985, pp. 67-75.

[6] Frandsen, G.S.: "Logic Programming, Substitutions and Finite Computability", Aarhus University, 1985, DAIMI PB-186.

[7] Frandsen, G.S.: "Logic Programming and Substitutions". In proceedings of FCT '85 (Cottbus, GDR), LNCS 199, 1985, pp. 146-158.

[8] Hennessy, M.C.B. and Plotkin, G.D.: "Full abstraction for a simple parallel programming language". In proceedings of MFCS '79 (Olomouc, Czechoslovakia), LNCS 74, 1979, pp. 108-120.

[9] Jaffar,J., Lassez, J.L. and Lloyd, J.: "Completeness of the negation as failure rule". In proceedings of the 8th IJCAI (Karlsruhe, FRG), 1983, pp. 500-506.

[10] Lloyd, J.W.: "Foundations of logic programming", Springer Verlag, 1984.

[11] Milner, R.: "Fully abstract models of typed λ-calculi", Theoretical Computer Science, Vol. 4, 1977, pp. 1-22.

[12] Nait Abdallah, M.A.: "On the interpretation of infinite computations in logic programming". In proceedings of ICALP '84 (Antwerp, Belgium), LNCS 172, 1984, pp. 358-370.

[13] Plotkin, G.D.: "A power domain construction", SIAM Journal of Computation, Vol. 5, 1976, pp. 452-487.

[14] Plotkin, G.D.: "LCF considered as a programming language", Theoretical Computer Science, Vol. 5, 1977, pp. 223-255.

[15] Plotkin, G.D.: "A structural approach to operational semantics", Aarhus University, 1981, DAIMI FN-19.

[16] Plotkin, G.D.: "Structural view of operational semantics", 1981. Unpublished supplement to [15].

[17] Robinson, J.A.: "A machine oriented logic based on the resolution principle", Journal of the ACM, Vol. 12, 1985, pp. 23-41.

[18] Smyth, M.B.: "Powerdomains", Journal of Computer and System Sciences, Vol. 16., 1978, pp. 23-36.

[19] Winskel, G.: "Modelling with theories - an overview". Unpublished lecture notes, Aarhus University, 1980.

[20] Winskel, G.: "On powerdomains and modality". Theoretical Computer Science, Vol. 36, 1985, pp. 127-137.