Fully abstract denotational semantics for concurrent Prolog

Fully abstract denotational semantics for concurrent PROLOG

by

R. Gerth
M. Codish
Y. Lichtenstein
E. Shapiro

87/21

december 1987

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands

editor: F.A.J. van Neerven

Fully Abstract Denotational Semantics for Concurrent Prolog

Rob Gerth*,
Eindhoven University of Technology[1]

Mike Codish, Yossi Lichtenstein, Ehud Shapiro
Weizmann Institute of Science[2]

extended abstract
Version: December 24, 1987

Abstract. We develop a denotational, hence, compositional semantics for a subset of Concurrent Prolog and relate it to an operational one, that makes divergence and the resultant substitutions of finite computations together with the termination mode — success, failure or deadlock — observable. Relative to this notion of observation we prove that the denotational semantics is fully abstract in the sense that it records the minimal amount of extra information beyond the observables to make it compositional. Full abstraction is an important property because it quantifies the information that one needs in order to reason about individual program-parts independently. It is the first such result in the area of concurrent logic programming.

1. Introduction

Logic programming is based on the idea that first order logic can be used as a programming language. Its origins lie in Robinson's *resolution* principle [Rob65], an inference rule that is eminently suitable for mechanization, and in Kowalski's and Colmerauer's [Kow74, Col73] realization that logical deduction has a procedural interpretation that makes it effective as a programming language.

A typical logic programming clause, $a \leftarrow b_1, \ldots, b_n$, has as logical or *declarative* meaning "a is satisfied whenever all b_i's are satisfied". Such a clause also has a *procedural* reading: "to solve the goal a (i.e., to satisfy a,) solve the goals b_1, ..., b_n." Together with resolution — or rather *unification* — as the computational mechanism, this procedural interpretation yields efficient programming languages, as the various Prolog dialects show.

A third interpretation of a logic clause is possible: the *process* reading. The atom, a is now interpreted as a *process* that may spawn off (instances of) the body-atoms b_1,..., b_n with which it is replaced. Clearly, this interpretation is geared towards parallel and distributed implementations.

* The author is working in and partially supported by ESPRIT project 937,"Debugging and Specification of Real-Time Ada Embedded Systems (DESCARTES)".

[1] Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, the Netherlands.

[2] Department of Applied Mathematics, Weizmann Institute of Science, P.O.Box 26, Rehovot 76100, Israel.

For "pure" logic programs the declarative, procedural and process readings all coincide; see, e.g., [Llo84]. Most concurrent logic programming languages, like Prolog itself, introduce extra-logical constructs for efficiency and expressiveness reasons. Specifically, concurrent logic programming languages all have constructs that allow control information to flow between goals or processes so that the reduction of goals can be synchronized or suspended. Such constructs, on the one hand, are essential for reducing the otherwise intractable computational effort of reducing all processes in parallel, but on the other hand, destroy the correspondence between the declarative meaning of a concurrent logic program and its operational or process meaning. There is an analogy here with the introduction of the *cut*-operator in Prolog [Llo84].

Strictly speaking, such languages no longer belong to the realm of logic programming. They correspond more closely to imperative languages, although the basic computation step still is the resolution of clauses, and a successful computation provides a proof of the goal statement from the clauses of the program.

As with imperative languages, most of the concurrent logic languages have had there meaning defined by an operational semantics. There are good reasons not to be satisfied with this state of affairs and ask for denotational semantics as well. Apart from the theoretician's argument — that they are obviously good to have — we state the following practical reasons

— to localize the debugging [Lic87], analysis and transformation of programs [GCS88], and

— to aid distributed implementations [TSS87].

In either case, it is necessary to consider program parts independently from the other parts and, hence, to determine in what way such parts can be influenced (and can influence). It is precisely the *compositional* nature of a denotational semantics that makes it important in this context.

We develop in this paper a denotational semantics for a subset of Concurrent Prolog [Sha86], *Theoretical Flat Concurrent Prolog, TFCP*. The starting point is a decision on what one wants to *observe* or know about program executions. As usual, we codify this in an operational semantics, O. Our notion of observation is a minimal one: we observe the resultant substitutions of finite computations with their "type", *success*, *failure* or *deadlock*, and whether *divergence* occurs. Moreover, in this paper we limit ourselves to uniprocessor implementations and, hence, allow at most one reduction step at a time.

The next step is to construct a denotational semantics from which not only the operational semantics can be reconstructed, but which also gives independent meaning to the syntactic building blocks of programs. In general this will entail extending the recorded behaviour of programs beyond what we want to observe.

It is not only a theoretical nicety to ask for the *minimum* amount of extra information about programs, upon which a compositional semantics can be based. This is of obvious importance, too, if the semantics forms the theoretical basis for debugging tools and distributed implementations. In other words, we want a denotational semantics that is *fully abstract* with respect to O[3].

The fully abstract semantics developed here is based on the *divergence set semantics* of Brookes

[3] Full abstraction is usually formulated differently. The present characterization as the "smallest morphism above O" is an equivalent one; see e.g., [HGR87].

et al. for TCSP [BHR84]. Hence, the meaning of a program is expressed in terms of the sequences of interactions, i.e., substitutions, that programs participate in, including both the substitutions that a program produces and the substitutions that it assumes its environment produces. So-called *divergence traces*, if any, express at which points a divergent computation can start. Moreover, because at some points, progress in a program-part may depend on the environment, the meaning of a program also quantifies the dependencies on the environment. These take the form of *failure sets* which collect substitutions that will either suspend the program or will not release it from suspension.

In fact, we treat a TFCP program as an ordinary parallel program: substitutions are treated as assignments to variables that are shared between the goal processes and which may cause different processes to synchronize.

The only other work that we know of in this area is a recent paper by Kok [Kok88]. He gives a denotational semantics for a larger set of Concurrent Prolog than we do, using a Banach space of trees as domain. Unlike us, he does not relate his model to any operational semantics and his model is far from being fully abstract relative to our operational semantics.

Section 2 of the paper introduces the operational semantics. The domain and the denotational semantics is the subject of Section 3. Section 4 shows full abstraction and Section 5 contains some final remarks.

For reasons of space, both the operational and the denotational semantics as presented in this abstract ignore divergences of programs. However, we have proven these results for the more general case. The results and proofs will be included in the full paper

2. Concurrent Prolog and its Operational Semantics

2.1 Syntax

The notions of signature, Σ, terms, $Tm(Var, \Sigma)$, atoms, $At(Var, \Sigma^f, \Sigma^p)$, substitutions, renamings, most general unifiers, etcetera, are assumed to be understood. Their definitions and other non-standard notation can be found in the appendix.

Concurrent Prolog extends logic programming with the notion of read-only variables as a synchronization primitive and the commit operator which distinguishes between guard atoms and the proper body atoms of a clause. Concurrent Prolog distinguishes between the *writable* occurrence of a variable, X, and its *read-only* occurrence, $X?$. The intension is that a program that needs $X?$ to be instantiated will suspend until its environment will instantiate a writable occurrence of X.

We restrict ourselves to the so-called flat subset of Concurrent Prolog in which the guard atoms are constructed from a fixed set of test predicates, T.

Definition:

- For any set Var of variables, $Var? = \{X? \mid X \in Var\}$ and $Var_? = Var \cup Var?$
- $\mathcal{S} = \{\sigma \mid \sigma$ is a substitution on $Var_?$ and $dom(\sigma) \subseteq Var\}$

- $\mathcal{B} = \{\sigma \in \mathcal{S} \mid \sigma \text{ is idempotent}\}$
- $\mathcal{B}^1 = \{\sigma \in \mathcal{S} \mid \forall X \in dom(\sigma)\ d(\sigma(X)) \leq 1\}$
- $\mathcal{R} = \{\rho \in \mathcal{B} \mid \rho \text{ is a renaming }\}$
- if $\sigma \in \mathcal{B}$ and $t \in Tm(Var_?, \Sigma)$ then $t\sigma$ is defined as usual except that $(X?)\sigma = \begin{cases} \sigma(X)? & \text{if } \sigma(X) \in Var \\ \sigma(X) & \text{otherwise} \end{cases}$

Definition: *Flat Concurrent Prolog*, $\text{FCP}_{T,\Sigma}$; *initialized programs*, $\text{iFCP}_{T,\Sigma}$

- A (flat) guarded (T,Σ)-clause has the form $a \leftarrow g_1,\ldots,g_m \mid b_1,\ldots,b_n$, $m,n \geq 0$ where $\{a,b_1,\ldots,b_n\} \subseteq At(Var_?,\Sigma)$ and $\{g_1,\ldots,g_m\} \subseteq At(Var_?,\Sigma^f,T)$. We call a the *head*, $g_1,\ldots,g_m$ the *guard* and $b_1,\ldots,b_n$ the *body* of the clause. If $n = 0$, we take the body to be *true*.
- An $\text{FCP}_{T,\Sigma}$ program is a finite set of flat guarded (T,Σ)-clauses.
- A program $P \in \text{FCP}_{T,\Sigma}$ is called *closed* if every predicate in Σ^p that occurs in P is also defined in P, i.e., also occurs in the head of a clause in P.
- $X \in \text{iFCP}_{T,\Sigma}$ iff $\exists P \in \text{FCP}_{T,\Sigma}, a \in At(Var_?,\Sigma)^+ \ \ X \equiv P;a$

Such an initialized program is usually written as $P, \leftarrow a$.

In theoretical FCP, we give syntactic structure to programs and view them as the parallel composition of a number of closed, independent sub-programs. In addition, a goal-list (or *resolvent* as it is usually called), $a_1,\ldots,a_n$, is interpreted in accordance with the process-reading of clauses and, hence, is viewed as the parallel composition of processes. We do not make this explicit in the syntax of the language.

Definition: *Theoretical Flat Concurrent Prolog*, $\text{TFCP}_{T,\Sigma}$; *initialized programs*, $\text{iTFCP}_{T,\Sigma}$

- $\text{TFCP}_{T,\Sigma}$ is the smallest set X such that
  - $P \in X$ if $P \in \text{FCP}_{T,\Sigma}$, P is closed and every guarded clause in P is uniquely determined by its head and guard[4]
  - $P_1 \parallel P_2 \in X$ if $P_1, P_2 \in X$ and P_1 and P_2 have no predicate symbols in common.
- $\text{iTFCP}_{T,\Sigma}$ is the smallest set X such that
  - $P;a \in X$ if $P \in \text{FCP}_{T,\Sigma} \cap \text{TFCP}_{T,\Sigma}$ and $P;a \in \text{iFCP}_{T,\Sigma}$
  - $P_1 \parallel P_2 \in X$ if $P_1, P_2 \in X$ and P_1 and P_2 have no predicate symbols in common.

Observe that any predicate in a TFCP-program, P, inherits its definition from one of the (closed) FCP sub-programs of P. As we will see, this means that parallel components can only influence each other via the variable bindings they produce. Likewise for iTFCP-programs. We feel that at this stage, it is a reasonable assumption to make.

We usually write TFCP instead of $\text{TFCP}_{T,\Sigma}$, write $C \equiv H \leftarrow G \mid B$ for a guarded clause C and often interpret a TFCP-program as just a collection of clauses.

[4] This unique determination is a non-essential assumption that makes the denotational semantics somewhat easier to formulate.

To define the operational semantics we have to define read-only unification. Although $p(X?)$ and $p(f(a))$ will unify in the classical sense, they should not unify here, because $X?$ can only become instantiated through a writable occurrence of X. Such unification attempts will become *suspended*. Moreover, successful unification depends on satisfaction of guards.

Definition: *admissible substitutions, read-only mgu, mgu?*[5] .

- A substitution $\theta \in \mathcal{B}$ is *admissible* for a term $t \in Tm(Var_?, \Sigma)$ if $X? \in vars(t) \Rightarrow X \in vars(t)$ holds for any $X \in dom(\theta)$. A substitution is admissible for an atom $p(t_1, \ldots, t_n)$ if it is admissible for every t_i.
- $\pi : At(Var_?, \Sigma, T)^* \rightarrow \{true, false, suspend\}$ is some fixed interpretation of guard atoms such that $\pi(\varepsilon) = true$.
- For a clause $C \equiv H \leftarrow G \mid B$ and an atom a
  - $mgu^{\rightarrow}(a, H) = \{\ \theta \in \mathcal{B} \mid \theta$ is an mgu for $\{a = H\}$, and $ran(\theta) \cap vars(a) = \emptyset\ \}$

    Our definition of hiding, in section 3, requires the goal variables in mgu's to be in the domain.
  - $\theta \in mgu^{\rightarrow}(a, H)$ is admissible if θ is admissible for a and for H.
  - $mgu_?(a, C) \ni \begin{cases} \theta & \text{if } \theta \in mgu^{\rightarrow}(a, H) \text{ is admissible and } \pi(G\theta) = true \\ fail & \text{if } mgu^{\rightarrow}(a, H) = \emptyset \text{ or} \\ & \text{if for all admissible } \theta \in mgu^{\rightarrow}(a, H),\ \pi(G\theta) = false \\ suspend & \text{otherwise} \end{cases}$

Note that for $t \in \{fail, suspend\}$, if $t \in mgu_?(a, C)$ then $mgu_?(a, C)$ is a singleton set. The results of this paper are actually quite independent of any particular variant of (read-only) unification. The only assumptions that we need to make, is that unification yields a substitution, *fail* or *suspend*, that its yield only depends on π, a clause and a head and that it yields *suspend* if a read-only instance needs to be bound.

2.2 Operational Semantics

The operational semantics gives only minimal information about program executions. For any finite computation it records the resultant substitution θ together with the termination mode — success, (θ, tt), deadlock, (θ, dl), or fail, (θ, ff) — . This seems to be the minimal amount of information that one would like to know about a program. With every program, P, we associate a transition system, Π_P as follows

Definition: *Transition system,* Π_P

Given a program $P \in TFCP$, $\Pi_P = (Q, \rightarrow)$, where

- $Q = \{\langle A; \theta\rangle \mid A$ is a multiset of atoms or $A \in \{tt, dl, ff\},\ \theta \in \mathcal{B}\}$.
- $\rightarrow \subseteq Q \times \mathcal{B} \times Q$ is defined as the smallest relation such that:
  - $\langle\{A_1, \ldots, A_i, \ldots, A_n\}; \theta\rangle \xrightarrow{\sigma} \langle\{(A_1, \ldots, B\rho, \ldots, A_n)\sigma\}; \theta \circ \sigma\rangle$

[5] These definitions differ slightly from the ones used in FCP [Sha86]

for any $C \equiv H \leftarrow G \mid B \in P$ and renaming $\rho \in \mathcal{R}$ such that $vars(C) \subseteq dom(\rho)$, $ran(\rho) \cap (vars(A_1, \ldots, A_n) \cup dom(\theta)) = \emptyset$ and $\sigma \in mgu_?(A_i, C\rho) \in \mathcal{B}$ [6]

- $\langle \{A_1, \ldots, A_i, \ldots, A_n\}; \theta \rangle \xrightarrow{\tau} \langle ff; \theta \rangle$

  if $A_i \not\equiv true$ and for every clause $C \in P$ $fail \in mgu_?(A_i, C)$.

- $\langle A; \theta \rangle \xrightarrow{\tau} \langle dl, \theta \rangle$

  if $\exists A_i \in C$ $A_i \not\equiv true$ and $\forall A_i \in A \; \forall C \in P$, $A_i \not\equiv true \Rightarrow suspend \in mgu_?(A_i, C)$

- $\langle true, ..., true; \theta \rangle \xrightarrow{\tau} \langle tt; \theta \rangle$

- $Seq(\Pi_P, q) = \left\{ (q_i \xrightarrow{\theta_i} q_{i+1})_{i<\alpha} \mid \alpha < \omega,\ q_0 = q,\ q_\alpha = \langle R; \theta \rangle,\ R \in \{tt, ff, dl\} \right\}$

Definition: *Operational Semantics* $\multimap \cdot \multimap$

For $Prog = P_1; a_1 \| \cdots \| P_n; a_n \in \text{iTFCP}_{T,\Sigma}$, let $P = P_1 \cup \cdots \cup P_n$ and $a = a_1, \ldots, a_n$. Then

- $\multimap Prog \multimap = \left\{ (\theta \mid vars(a), R) \;\middle|\; \exists (q_i \xrightarrow{\theta_i} q_{i+1})_{i<\alpha} \in Seq(\Pi_P, \langle a; \tau \rangle), \alpha < \omega,\ q_\alpha = \langle R; \theta \rangle \right\}$

3. Denotational Semantics

A compositional semantics has to give meaning to each individual program part, independent from the environment consisting of the other parts. For TFCP this means that we have to give meaning to the individual clauses of a program and hence to anticipate in these meanings the possible substitutions that any environment may produce. In this sense, there is a close correspondence with ordinary CSP-like, concurrent languages. Here, too, meanings or denotations must be expressed in terms of sequences of interactions — i.e., substitutions — differentiating between *input*-substitutions, θ^I, offered by the environment, and *output*-substitutions, θ^O, offered by the program part. This correspondence goes even further as we, too, have to describe the ways in which a program depends on its environment in order to proceed — e.g., because a read-only variable needs to be bound — .

3.1 The Domain

In fact, our semantics will be based on the *failure set semantics* for TCSP [BHR84]. So, a program denotation is a set, $\mathbf{F}$, of *suspensions*, (c, s). Each (c, s) records a computation sequence, c, of input and output substitutions together with a set of substitutions, s, input of none of which will allow the program to produce any other output substitution.

Definition: *Sequences, Suspensions*

- $\mathcal{B}^\alpha = \{\theta^\alpha \mid \theta \in \mathcal{B}\},\ \alpha \in \{I, O\}$
  for $\alpha = I$ ($\alpha = O$) $\mathcal{B}^\alpha$ is called the set of input (output) bindings.

- Let $\mathbf{A} \subseteq \mathcal{B}^I \cup \mathcal{B}^O$

[6] As $dom(\theta\sigma) \cap ran(\theta\sigma) = \emptyset$, $\theta \circ \sigma$ exists and is equal to $\theta\sigma$; see [LMM86].

- $\mathcal{SEQ}(A) = \left\{ \theta_0^{\alpha_0} \hat{} \theta_1^{\alpha_1} \hat{} \cdots \hat{} \theta_n^{\alpha_n} \in \mathbf{A}^\star \;\middle|\; \begin{array}{l} \forall i,j = 1..n\ i \neq j \Rightarrow dom(\theta_i) \cap dom(\theta_j) = \emptyset, \\ \forall i = 1..n\ \theta_0 \circ \cdots \circ \theta_i \text{ exists} \end{array} \right\}$
- $\mathcal{SEQ} = \mathcal{SEQ}(\mathcal{B}^I \cup \mathcal{B}^O)$

- If $c = \theta_0^{\alpha_0} \hat{} \cdots \hat{} \theta_n^{\alpha_n} \in \mathcal{SEQ}$ then $\theta_c = \theta_0 \circ \theta_1 \circ \cdots \circ \theta_n$
- $\mathcal{SUSP} = \{(c,s) \mid c \in \mathcal{SEQ},\ s \subseteq \mathcal{B}^1 \text{ or } s \in \{ff, tt\}\ \}$

Before we can define the denotational domain, we have to introduce renamings of suspensions.

Definition: *renaming of suspensions*

Let $(c,s) \in \mathcal{SUSP}$, $c = \theta_1^I \hat{} \cdots \hat{} \theta_n^I \theta^O \hat{} \bar{c}$. Let $\rho \in \mathcal{R}$ be such that $vars(\rho) \cap (\bigcup_{i=1}^{n} vars(\theta_i) \cup dom(\theta)) = \emptyset$. Then $(c,s)\rho$ is a *renaming* of (c,s) and defined as $(c.\rho, s\rho)$

Since θ^O is the first substitution produced by a unification of the program that (c,s) is a behaviour of, this definition mirrors the way that variables are renamed during actual computations.

Program denotations will be sets of suspensions. However, not every such set will be a program denotation. In the definition below we will impose a number of closure conditions on these sets. Their purpose is not so much to characterize precisely those sets that do obtain as program denotations, but rather to express the closure conditions that we need in the sequel. E.g., conditions L1 and L2 allow us to use set inclusion as the ordering on the domain.

Definition: *Denotational Domain, $\mathcal{DEN}$*

- $\mathcal{DEN} \subseteq 2^{\mathcal{SUSP}}$ and $\mathbf{F} \in \mathcal{DEN}$ iff:

(L1) $(\varepsilon, \emptyset) \in \mathbf{F}$

(L2) $(c \hat{} \bar{c}, \emptyset) \in \mathbf{F} \Rightarrow (c, \emptyset) \in \mathbf{F}$

(L3) $(c,s) \in \mathbf{F},\ \bar{s} \subseteq s \Rightarrow (c,\bar{s}) \in \mathbf{F}$

(L4) $\forall s' \subseteq_{fin} s\ (c,s') \in \mathbf{F} \Rightarrow (c,s) \in \mathbf{F}$

(L5) Let $(c,s) \in \mathbf{F}$, $(c,tt) \notin \mathbf{F}$ and $\theta \in \mathcal{B}^1$. Then
$(c,s) \in \mathbf{F},\ (c \hat{} \theta^I, \emptyset) \in \mathbf{F},\ \forall \bar{\theta} \in \mathcal{B}\ (c \hat{} \theta^I \hat{} \bar{\theta}^O, \emptyset) \notin \mathbf{F} \Rightarrow (c, s \cup \{\theta\}) \in \mathbf{F}$

(L6) $(c,s) \in \mathbf{F} \Rightarrow (\bar{c},\bar{s}) \in \mathbf{F}$ for any renaming $(\bar{c},\bar{s})$ of (c,s).

(L7) $(c \hat{} \bar{c}, \emptyset) \in \mathbf{F} \iff (c \hat{} (\theta.\rho)^I \hat{} \bar{c}, \emptyset) \in \mathbf{F}$ for any $\theta \in \mathcal{B}$ and some renaming, ρ, such that $vars(c \hat{} \bar{c}) \cap dom(\theta.\rho) = \emptyset$

(L8) $(c \hat{} \theta^I \hat{} \bar{\theta}^I \hat{} \bar{c}, s) \in \mathbf{F} \iff (c \hat{} (\theta \circ \bar{\theta})^I \hat{} \bar{c}, s) \in \mathbf{F}$, provided $\theta, \bar{\theta} \in \mathcal{B}$

(L9) $(c,\emptyset) \in \mathbf{F} \Rightarrow \left| \{\theta \in \mathcal{B} \mid (c \hat{} \theta^O, \emptyset) \in \mathbf{F}\} \backslash \approx \right| < \aleph_0$ or $\mathbf{F} = \mathcal{SUSP}$

The closure conditions L1, ..., L4 are the standard ones, corresponding to the conditions N1, ..., N4 of [HGR84]). L5 adapts condition N5 to the asynchronous nature of "communication" in TFCP. L6 is a compactness condition analogous to L4, and is needed to ensure continuity of the restrict operator. It expresses the arbitrariness of renamings. L7 and L8 function in the full abstraction proof. Bounded nondeterminism, finally, is expressed in condition L9. We take the quotient with respect to $\approx$ so as to ignore the effect of renamings of variables.

Theorem:

$\mathbf{D} = (\mathcal{DEN}, \sqsubseteq, \perp)$ with $\mathbf{F} \sqsubseteq \mathbf{F}'$ iff $\mathbf{F} \supseteq \mathbf{F}'$ and $\perp = \mathcal{SUSP}$ is a complete partial order.

Proof: standard [BHR84].

3.2 The Equations

Technically speaking, the semantic equations pose fairly standard problems. As stated earlier, computations are modelled after the failure set model for TCSP [HGR84]. The "recursion skeleton" of the semantics is analogous to, e.g., the one used by Joost Kok [Kok88] or by Neil Jones for PROLOG in [Jon87]. There are two exceptions.

In the operational semantics, every unification induces a renaming of the spawned-off body-clauses. Since we aim at full abstractness, the denotational semantics must mimic this. Indeed, the equations are parametrized by three additional arguments: two renamings, used to rename the goal and the clause it unifies with, and a set of fresh variables that newly constructed renamings can rename to. An alternative would have been to use in both semantics a fixed renaming scheme. We felt that it was preferable to keep the operational semantics as simple and clean as possible.

The second exception concerns the hiding of variables. The variables in the initial goal are always visible, but a new variable becomes visible because it occurs in the binding to an already visible variable (see the definition of *closure* below). This dynamic character makes the hide-operator more difficult to define.

We need some notation and auxiliary functions.

Definition: Let P be a TFCP program and $\mathbf{F} \subseteq \mathcal{SUSP}$; let $c = \theta^I \hat{} \bar{c} \in \mathcal{SEQ}$ and let $g \in At(Var_?, \Sigma)^*$.

- the *closure* of g with respect to c:

$$closure(g, c) = \bigcup \{\, v' \in vars(v\theta_{c'}) \mid v \in vars(g), c' \preceq c \,\}$$

- input variables: $input(c) = \bigcup \{dom(\theta) \mid \exists c_1, c_2 \;\; c = c_1 \hat{} \theta^I \hat{} c_2\}$
- $c \restriction V$ is defined by a recursion on the length of c:
  $\varepsilon \restriction V = \varepsilon$, $(\theta^I \hat{} \bar{c}) \restriction V = \theta^I \hat{} (\bar{c} \restriction V)$, $(\theta^O \hat{} \bar{c}) \restriction V = \begin{cases} (\theta \restriction V)^O \hat{} (\bar{c} \restriction V) & \text{if } \theta \neq \tau \\ \bar{c} \restriction V & \text{otherwise} \end{cases}$.
- $suspended(g, P) = \{\; s \subseteq \mathcal{B}^1 \mid \forall \sigma \in s \; \forall C \in P \; \forall a \in g \; suspend \in mgu_?(a\sigma, C) \;\}$
- $c \hat{} \mathbf{F} = \{(c \hat{} \bar{c}, s) \mid (\bar{c}, s) \in \mathbf{F}\}$ for $c \in \mathcal{SEQ}$
- PFC($\mathbf{F}$) is the smallest set $X \supseteq \mathbf{F}$ satisfying L1 and L2.

Now, we can define the auxiliary functions, $restrict(V, \mathbf{F})$, respectively, $\mathbf{F}_1 \parallel \mathbf{F}_2$, that hide variables in the suspension, $\mathbf{F}$, respectively, parallelly compose the suspensions $\mathbf{F}_1$ and $\mathbf{F}_2$.

Definition: *restrict* and $\parallel$

- $restrict : 2^{Var} \times \mathcal{DEN} \to \mathcal{DEN}$

$$restrict(V, \mathbf{F}) = \mathrm{PFC}\left(\bigcup \left\{\; hide(V, (c, s)) \;\middle|\; \begin{array}{l} (c, s) \in \mathbf{F} \text{ and } \neg\, \exists \bar{c}, (c_i)_{i<\omega} \in \mathcal{SEQ} \\ \forall i < \omega : c_i \preceq c_{i+1} \;\&\; (c \hat{} \bar{c} \hat{} c_i) \in \mathbf{F} \;\& \\ \quad hide(V, c \hat{} \bar{c} \hat{} c_i) = hide(V, c \hat{} \bar{c}) \end{array} \right\}\right),$$

  where

- $hide(V,(c,s)) = \begin{cases} \{(c \restriction \tilde{V}, s \restriction \tilde{V})\} \\ \text{where } \tilde{V} = closure(V,c) \\ \text{and } \tilde{V} = Var \setminus lvars(V,c) & \text{if } input(c) \cap lvar(V,c) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$
- $lvars(V,c) = \left\{ v \;\middle|\; \begin{array}{l} \exists c_1, \theta, c_2 \;\; c = c_1 \theta^O c_2 \\ v \in ran(\theta) \backslash closure(V,c) \end{array} \right\}$ local variables

- $\| : \mathcal{DEN} \times \mathcal{DEN} \to \mathcal{DEN}$

  $\mathbf{F}_1 \| \mathbf{F}_2 = \{(c_1 \| c_2, s_1 \| s_2) \mid (c_i, s_i) \in \mathbf{F}_i, i = 1,2\}$, where
  - $s_0 \| s_1 = s_1 \| s_0$, for $s \neq ff$ $s \| tt = s$, $s \| ff = ff$, for $s_0, s_1 \subseteq \mathcal{B}^1$ $s_0 \| s_1 = s_0 \cap s_1$
  - $c_0 \| c_1 = c_1 \| c_0$, $\varepsilon \| \varepsilon = \varepsilon$, for $l \in \{I, O\}$ $\theta^l \hat{} c_0 \| \theta^I \hat{} c_1 = \theta^l \hat{} (c_0 \| c_1)$

Lemma:

The functions $restrict(V, \cdot)$ and $\cdot \| \cdot$ are well-defined and continuous. Also, $\cdot \| \cdot$ is commutative and associative.

Proof: See the full paper.

The semantics of a TFCP-program, P, wil be given by $\mathbf{P}(P)$. It will be defined as a fixed point involving auxiliary functions $\mathbf{A}$ and $\mathbf{B}$ that give meaning to atom-lists and individual atoms relative to a program. We make use of program environments, $\mathcal{ENV}$, that record the meaning of the program clauses (remember that they are identified by their head and guard):

$\mathcal{ENV}: \; At(Var_?, \Sigma) \times At(Var_?, \Sigma^I, T)^* \to 2^{Var} \times \mathcal{R} \times \mathcal{R} \to \mathcal{B} \to \mathcal{DEN}$

The types of the functions involved, are as follows:

— $\mathbf{P} : TFCP \to \mathcal{ENV}$ program meaning

— $\mathbf{A} : At(Var_?, \Sigma)^+ \to \mathcal{ENV} \to 2^{Var} \times \mathcal{R} \times \mathcal{R} \to \mathcal{B} \to \mathcal{DEN}$ atom list

— $\mathbf{B} : At(Var_?, \Sigma) \to \mathcal{ENV} \to 2^{Var} \times \mathcal{R} \times \mathcal{R} \to \mathcal{B} \to \mathcal{DEN}$ single atom

We first define the meaning of TFCP-programs without parallel operators.

Definition: *FCP-program meaning*

Take $P \in \mathrm{TFCP}_{T,\Sigma} \cap \mathrm{FCP}_{T,\Sigma}$. Let $P \equiv C_1, \ldots, C_n$ and $PVar = vars(P)$, where $C_i \equiv H_i \mid B_i$ (and $H_i \equiv a_i \leftarrow g_{i1}, \ldots, g_{im_i}$, $B_i \equiv b_{i1}, \ldots, b_{in_i}$).

- $\mathbf{P}(P) = \Omega\{\mathcal{C}_1/H_1, \ldots, \mathcal{C}_n/H_n\}$, where
  $(\mathcal{C}_1, \ldots, \mathcal{C}_n) = \mu \mathcal{C}_1 \ldots \mathcal{C}_n \,.\, [\mathbf{A}(B_1)(\mathbf{P}(P)), \ldots, \mathbf{A}(B_n)(\mathbf{P}(P))]$
- $\mathbf{A}(a_1, \ldots, a_n)\eta V \rho \bar{\rho} = \mathbf{B}(a_1)\eta V_1 \bar{\rho} \rho_1 \| \cdots \| \mathbf{B}(a_n)\eta V_n \bar{\rho} \rho_n$, where

  $ran(\rho_i) \cap ran(\rho_j) \neq \emptyset \Rightarrow i = j$, $\rho_1, \ldots, \rho_n \in \mathcal{R}$, $PVar \cup vars(a_1, \ldots, a_n) \subseteq \bigcap_i dom(\rho_i)$, $\bigcup_i ran(\rho_i) \subseteq V$ and $V_1, \ldots, V_n$ is a partition of $V \setminus \bigcup_i ran(\rho_i)$ such that each V_i is infinite.
- $\mathbf{B}(a)\eta V \rho \bar{\rho} \theta = restrict(vars(a\rho\theta), \mathbf{F})$ where
  - $\mathbf{F} = \left\{ (\epsilon, s) \;\middle|\; \begin{array}{l} s \in suspended(a\rho\theta, dom(\eta)), \\ \forall \bar{\theta} \in s : \; dom(\bar{\theta}) \cap dom(\theta) = \emptyset, \; vars(\bar{\theta}) \cap V = \emptyset \end{array} \right\} \bigcup$
    $\{(\epsilon, tt) \mid a \equiv true\} \bigcup$
    $\{ (\epsilon, ff) \mid a \not\equiv true, \forall H \in dom(\eta) \; fail \in mgu_?(a\rho\theta, H) \} \bigcup$
    $\{ \bar{\theta}^I : \mathbf{B}(a)\eta V \rho \bar{\rho}(\theta \circ \bar{\theta}) \mid \bar{\theta} \in \mathcal{B}, dom(\theta) \cap dom(\bar{\theta}) = \emptyset, vars(\bar{\theta}) \cap V = \emptyset \} \bigcup$

$$\{\ \bar{\theta}^{O}{:}\eta(H)V\rho\bar{\rho}(\theta\circ\bar{\theta})\ \mid \bar{\theta}\in mgu_{?}(a\rho\theta, H\bar{\rho}), \bar{\theta}\in\mathcal{B}, H\in dom(\eta)\ \}$$

Now, the semantics of a TFCP-program, $P_1 \parallel P_2$ is straightforwardly defined:

Definition: TFCP-*program meaning*

- $\mathbf{P}(P_1 \parallel P_2) = \mathbf{P}(P_1) \cup \mathbf{P}(P_2)$.

  (Note that $\mathbf{P}(P_1)$ and $\mathbf{P}(P_2)$ have disjoint domains.)

Definition: *Denotational semantics,* $[\![\cdot]\!] : \mathrm{iTFCP}_{T,\Sigma} \to \mathcal{DEN}$

- $[\![P_1 \parallel P_2]\!] = [\![P_1]\!] \parallel [\![P_2]\!]$
- $[\![P;a]\!] = \mathbf{A}(a)(\mathbf{P}(P))\ Var\ \tau\tau\tau$

Theorem: *well-definedness and continuity*

The functions **P**, **A**, **B** and $[\![\cdot]\!]$ are well-defined and continuous

Proof: Included in the full paper.

Finally, we can relate $\multimap\cdot\circ\!\!-$ and $[\![\cdot]\!]$ as follows

Theorem: $\multimap\cdot\circ\!\!- = \alpha\circ[\![\cdot]\!]$ where $\alpha : \mathcal{DEN} \to \mathcal{B}\times\{tt, dl, ff\}$ is defined by

$$\alpha(\mathbf{F}) = \left\{ (\theta_c, t) \;\middle|\; (c,s)\in\mathbf{F},\ input(c)=\emptyset,\ t = \begin{cases} s & \text{if } s\in\{tt,ff\} \\ dl & \text{if } s\subseteq\mathcal{B} \text{ and } s\neq\emptyset \end{cases} \right\}$$

4. Full Abstraction

The last theorem of the previous section shows that the observables can be retrieved from the program denotations: $\multimap\cdot\circ\!\!- = \alpha\circ[\![\cdot]\!]$. But what is the status of those parts of the denotations that α abstracts away from? We intend to prove that those parts are really necessary. Specifically, any other denotational semantics from which the observables can be retrieved will have at least as much information in its denotations as $[\![\cdot]\!]$ has: $[\![\cdot]\!] = \beta\circ\overline{[\![\cdot]\!]}$ for some β, where $\overline{[\![\cdot]\!]}$ is any other denotational semantics such that $\multimap\cdot\circ\!\!- = \gamma\circ\overline{[\![\cdot]\!]}$ for some γ.

There is a (folk?) theorem that links such an *abstract* semantics to the discriminatory power of *contexts*: whenever $[\![\cdot]\!]$ differentiates between two programs, P and Q, there must be another program with a "hole", $E(x)$, such that plugging in P and Q will result in an observable difference: $\multimap E(P)\circ\!\!- \neq \multimap E(Q)\circ\!\!-$. See, e.g., [HGR87].

In our case, the syntax of iTFCP suggests the following notion of contexts: $E(x)\in Cont_{T,\Sigma}$ iff $\exists P_e \in \mathrm{iTFCP}_{T,\Sigma'}\ E(x) = P_e \parallel x$ and $\Sigma'\cap\Sigma = \emptyset$.

Theorem: *Abstractness of* $[\![\cdot]\!]$ *w.r.t.* $\multimap\cdot\circ\!\!-$

Let $var(\cdot)\in T$ be a unary predicate symbol such that $\pi(var(\cdot))$ is the characteristic predicate of $Var_{?}$. Then, for every $P, Q\in\mathrm{iTFCP}_{T,\Sigma}$:

- $[\![P]\!]\neq[\![Q]\!] \Rightarrow \exists E(x)\in Cont_{T,\Sigma}\ \multimap E(P)\circ\!\!- \neq \multimap E(Q)\circ\!\!-$ (and $E(P), E(Q)\in\mathrm{iTFCP}_{T,\Sigma}$).

Note that we prove abstraction under an assumption about the available test-predicates, T. We have no such result yet for arbitrary T.

Proof: *sketch*

Take some $(c,s) \in [\![P]\!] \setminus [\![Q]\!]$. We only sketch the proof for the most complicated case, where $s \neq \emptyset$. The context, $E(x)$, must somehow make the behaviour (c,s) possible — by producing the necessary substitutions — and must be able to sense and make observable the fact whether (c,s) has occurred or not. Assume for the moment that $E(x)$ can sense when x has behaved like $(c,\emptyset)$. If, after having sensed that, $E(x)$ would bind, say, *true* to the fresh variable *success* and after that produce, non-deterministically, any of the substitutions in s[7], then we would have $(\theta, dl) \in \multimap\!\!\diamond E(P) \diamond\!\!\multimap$ for some $\theta \in \mathcal{B}$ with $success\theta = true$. Strictly speaking, the context will sense whether x behaves like $(\tilde{c},\emptyset)$ or not, with $\tilde{c} \approx c$ ($\approx \subseteq \mathcal{B} \times \mathcal{B}$ is extended pointwise to sequences). This is because a context cannot sense the renamings of variables as, e.g., in $p(x) \leftarrow q(x)$. Condition L6 ensures that we may ignore renamings.

Now consider $E(Q)$. If, after behaving like c, Q can do anything but suspend, we have obtained an observable difference. Obviously, there is no reason why Q could not suspend as well. However, if $(c,\emptyset) \in [\![E(Q)]\!]$ then Q *cannot* suspend immediately after $E(x)$ produces some substitution in s. If it could, then L5 and L3 would imply that $(c,s) \in [\![Q]\!]$, which is untrue. In other words, if Q suspends at all, it can only do so after having produced at least one extra output substitution.

To force an observable difference in this case, consider the set $B = \{\bar{\theta} \mid (c\hat{}\theta^I\hat{}\bar{\theta}^O, \emptyset) \in \mathbf{F}, \theta \in s, \bar{\theta} \in \mathcal{B}\}$. By L9, $B\backslash\approx$ is finite. Now, if $E(x)$ suspends until one of the substitutions in one of the equivalence classes, $[\phi_i]$, in $B\backslash\approx$ is produced and then fails, it will be able to fail Q before Q can suspend: $(\theta,t) \in \multimap\!\!\diamond E(Q) \diamond\!\!\multimap$ for some $\theta \in \mathcal{B}$ only if $success\theta \neq true$ or $t \neq dl$.

The actual construction of the context is quite subtle and we will not be able to give all the details here.

Let $c = \theta_1^{l_1}\hat{}\cdots\hat{}\theta_n^{l_n}$, $s = \{\sigma_1,\ldots,\sigma_k\}$ and $B\backslash\approx = \{[\phi_1],\ldots,[\phi_m]\}$. Let V be the variables in the goals of both P and Q. Define V_1 as V and for $i > 0$, $V_{i+1} = vars(V\theta_{\theta_1^{l_1}\hat{}\ldots\hat{}\theta_i^{l_i}})$. I.e., V_i are the variables that could be affected "in step i". With each $\theta_i^{l_i}$ we associate a set of clauses, $P_i^{l_i}$. With s and B we associate sets of clauses P_s and P_B. Basically, if $l_i = O$ then P_i^O will fail if anything but θ_i is output. If $l_i = I$ then P_i^I will produce the required input substitution. Similarly, P_s will produce any of the σ_j and P_B will await any of the ϕ_j and will then fail.

In the pseudo code below, ";" stands for sequentialization, which can be achieved by the standard *short circuit* technique of A. Takeuchi [Sha86]. Moreover, $await(\theta)$ waits until the variables in $dom(\theta)$ are bound and fails if they are bound differently than in some $\tilde{\theta} \in [\theta]$; $produce(\theta)$ produces some $\tilde{\theta} \in [\theta]$; *fail* is a clause that fails; $test(V)$ checks whether the variables in V are bound to non-variables. The first three pseudo commands are FCP programs, while $test(\{x_1,\ldots,x_l\})$ stands for the guard $var(x_1),\ldots,var(x_l)$.

The environment is $E(x) \equiv \bigcup_{i=1}^{n} P_i^{l_i} \cup P_s \cup P_B; e_1(V_1, success) \parallel x$, where

- $P_i^O \equiv e_i(V_i, success) \leftarrow await(\theta_i); \bar{e}_i(V_{i+1}, success)$
  $\bar{e}_i(V_{i+1}, success) \leftarrow test(V_{i+1}) \mid e_{i+1}(V_{i+1}, success)$
- $P_i^I \equiv e_i(V_i, success) \leftarrow test(V_i) \mid produce(\theta_i); e_{i+1}(V_{i+1}, success)$

[7] By L3 and L4 we may assume that s is finite.

- $P_s \equiv e_{n+1}(V_{n+1}, true) \leftarrow produce(\sigma_1); e_{n+2}$
  $\ldots$
  $e_{n+1}(V_{n+1}, true) \leftarrow produce(\sigma_k); e_{n+2}$
- $P_B \equiv e_{n+2} \leftarrow await(\phi_1); fail$
  $\ldots$
  $e_{n+2} \leftarrow await(\phi_m); fail$

As can be seen from the code, executing any of the fragments of $E(x)$ will result in more than one unification and, hence, will produce substitutions that do not appear in (c, s). The closure conditions L7 and L8 ensure that we can "expand" c to make room for the additional resolutions.

5. Conclusions and Further Work

We have developed, here, the first fully abstract semantics for any concurrent logic programming language. In doing so, we have consciously ignored the logic programming origins of FCP and have treated it as just another concurrent language. It might come as a surprise to see that standard modeling techniques, developed for CSP, apply in this context, too. The asynchronous nature of the interaction of FCP processes does complicate things, as does unification as the basic computation step. This can be seen from the full abstraction proof. The construction of the context here is more cumbersome than for CSP, CCS or for DNP-R [HGR87].

We intend to extend our results to more general computational domains. In particular, we want to have denotational models that correspond to step- semantics and to partial order semantics for FCP; these semantics capture more of the behaviour of distributed implementations of FCP.

Finally, the resulting denotational semantics will be the starting point for proof systems and verification and debugging tools for FCP.

Acknowledgements.

The first two authors thank each other's departments for their hospitality. We thank John Gallagher and Daniel Szoke for their fruitful comments.

6. References

[BHR84] Brookes, S.D., Hoare, C.A.R., Roscoe, W.; "A Theory of Communicating Sequential Processes", JACM 31, pp.499-560, 1984.

[Col73] Colmerauer, A., Kanoui, H., Roussel, P., Pasero, R.; "Un Systeme de Communication Homme-Machine en Français", Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[GCS88] Gallagher, J., Codish, M., Shapiro, E.; "Specialising Prolog and FCP Programs Using Abstract Interpretation", to appear in New Generation Computing, special issue on Partial Evaluation and Mixed Computation, 1988.

[HGR87] Huizing, C., Gerth, R., de Roever, W.P.; "Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language", Proc. 14th ACM POPL, pp.223-238, 1987.

[Jon87] Jones, N.; "A Semantics-Based Framework for the Abstract Interpretation of Prolog", in Abstract Interpretation of Declarative Languages, Ellis-Horwood, 1987.

[Kok88] Kok, J.; "A Compositional Semantics for Concurrent Prolog", Proc. STACS, 1988.

[Kow74] Kowalksi, R.A.; "Predicate Logic as a Programming Language", Proc. IFIP74, pp.569-574, 1974.

[Lic87] Lichtenstein, Y.; "Algorithmic Debugging of Flat Concurrent Prolog", M.Sc.Thesis, Department of Computer Science, Weizmann Institute of Science, 1987.

[Llo84] Lloyd, J.W.; "Foundations of Logic Programming", Springer Verlag, New York, 1984.

[LMM86] Lassez, J.-L., Maher, M.J., Mariott, K.; "Unification Revisited", IBM Thomas J. Watson Research Center, 1986.

[Rob67] Robinson, J.A.; "A Machine-Oriented Logic Based on the Resolution Principle", JACM 12-1, pp.23-41,1967.

[Sha86] Shapiro, E.; "Concurrent Prolog: A Progress Report", in Fundamentals of Artificial Intelligence, Springer Verlag, 1986.

[TSS87] Taylor, S., Safra, S., Shapiro, E.; "Parallel Execution of FCP", International Journal of Parallel Programming, Vol. 15, No. 3, pp.245-275, 1987.

7. Appendix

Let Var denote some (countable) set of variables and Σ some first-order signature; $\Sigma = \Sigma^p \cup \Sigma^f$, where Σ^p collects the predicate-symbols of Σ and Σ^f the function-symbols.

Terms and atoms over Var and Σ are defined as usual and are denoted by $Tm(Var, \Sigma)$ and $At(Var, \Sigma)$. Also, $At(Var, \Sigma_1, \Sigma_2)$ denotes the set of atoms that take their predicate-symbols solely from Σ_2. So, $At(Var, \Sigma) = At(Var, \Sigma^f, \Sigma^p)$.

Definition: *depth of a term, $d(t)$*

- $d(t) = 0$ for $t \in Var \cup \Sigma^f$,
- $d(t) = 1 + max(d(t_1), ..., d(t_n))$ for $t = p(t_1, ..., t_n)$

For any syntactic object, ψ, $vars(\psi)$ stand for the set of variables occurring (free) in ψ.

We often use *sequences* of objects. Then, "$\hat{}$" denotes concatenation of two sequences, "ϵ" stands for the empty sequence and "$c_1 \preceq c_2$" means that c_1 is a prefix of c_2: $\exists \bar{c}_1 \; c_1 \hat{} \bar{c}_1 = c_2$.

Definition: *substitutions*

- A substitution on Var is a total function $\sigma : Var \rightarrow Tm(Var, \Sigma)$ such that the set $dom(\sigma) = \{X \in Var \mid \sigma(X) \neq X\}$ is finite. The set $dom(\sigma)$ is the domain of the substitution σ. We let $ran(\sigma) = \{vars(\sigma(X)) \mid X \in dom(\sigma)\}$. The identity function on Var is called the empty substitution and is denoted by τ.
- A substitution ρ is a *renaming* if $\rho(X) \subseteq Var$, for every $X \in Var$, $dom(\rho) \cap ran(\rho) = \emptyset$, and ρ is injective (on its domain $dom(\rho)$).
- For terms or atoms t, $t\sigma$ is defined inductively by $X\sigma = \sigma(X)$ and $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. This generalizes to sets of terms or atoms in the obvious way.

- The composition of two substitutions σ and φ is defined by $X(\sigma\varphi) = (X\sigma)\varphi$. A substitution σ is *idempotent* if $\sigma\sigma = \sigma$. The *idempotent composition* of two substitutions is defined by $\sigma \circ \varphi = \theta$ if θ is idempotent and $\exists n \geq 0\ (\sigma\varphi)^n = \theta$, where ψ^n is defined by $\psi^0 = \tau$ and $\psi^{n+1} = \psi\,\psi^n$.
- A substitution, α, is *more general* than a substitution, β if there exists a substitution γ such that $\beta = \alpha\gamma$.
- A substitution, σ is a *unifier* for a set of equations, $E = \{a_1 = b_1, ..., a_n = b_n\}$, where the a_i, b_i are atoms or terms, if $a_i\sigma = b_i\sigma$ for $i = 1..n$. We call σ a *most general unifier*, *mgu*, for E, if σ is more general than any other unifier for E.
- If σ is a substitution and V' is a subset of V, then the restriction $\sigma \restriction V'$ of σ to V' is defined by $(\sigma \restriction V')(X) = \sigma(X)$ for $X \in V'$ and $(\sigma \restriction V')(X) = X$ for $X \in V \backslash V'$. For a set of substitutions s the restriction $s \restriction V$ is defined by $s \restriction V = \{\sigma \restriction V \mid \sigma \in s\}$.
- If σ is a substitution and ρ is a renaming then the *ρ-renaming* of σ is the substitution $\sigma.\rho$ defined by $\forall X \in dom(\sigma)\ \ (X\rho)\sigma.\rho = X\sigma\rho$. This generalizes to *sequences* of substitutions in the obvious way.
- The equivalence relation $\approx \subseteq \mathcal{B} \times \mathcal{B}$ is defined by $\theta \approx \phi$ iff $\exists \rho \in \mathcal{R}\ \theta\rho = \phi$. Note that only the variables in the range of θ are renamed.

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arum Kumar	Compositional semantics for real-time distributed computing (Inf. & Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responders commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of a sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J. Paredaens	A formal approach to distributed information systems
87/04	T. Verhoeff	Delay-insensitive codes - An overview
87/05	R. Kuiper	Enforcing non-determinism via linear time temporal logic specification

87/06	R. Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch)
87/07	R. Koymans	Specifying message passing and real-time systems with real-time temporal logic
87/08	H.M.J.L. Schols	The maximum number of states after projection
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics
87/10	T. Verhoeff	Three families of maximally nondeterministic automata
87/11	P. Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model
87/12	K.M. van Hee A. Lapinski	OR and AI approaches to decision support systems
87/13	J. van der Woude	Playing with patterns, searching for strings
87/14	J. Hooman	A compositional proof system for an occam-like real-time language
87/15	G. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.J. Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J. Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2-Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples