2.

Continuation Semantics for Prolog with Cut

A. de Bruin

Faculty of Economics,
Erasmus Universiteit, P.O.Box 1738, NL-3000 DR Rotterdam

E.P. de Vink

Department of Mathematics and Computer Science,
Vrije Universiteit, De Boelelaan 1081, NL-1081 HV Amsterdam

ABSTRACT We present a denotational continuation semantics for Prolog with cut. First a uniform language $\mathcal{B}$ is studied, which captures the control flow aspects of Prolog. The denotational semantics for $\mathcal{B}$ is proven equivalent to a transition system based operational semantics. The congruence proof relies on the representation of the operational semantics as a chain of approximations and on a convenient induction principle. Finally, we interpret the abstract language $\mathcal{B}$ such that we obtain equivalent denotational and operational models for Prolog itself.

Section 1 Introduction

In the nice textbook of Lloyd [Ll] the cut, available in all Prolog-systems, is described as a controversial control facility. The cut, added to the Horn clause logic for efficiency reasons, affects the completeness of the refutation procedure. Therefore the standard declarative semantics using Herbrand models does not adequately capture the computational aspects of the Prolog-language. In the present paper we study the Prolog-cut operator in a sequential environment augmented with backtracking. Our aim is to provide a denotational semantics for Prolog with cut and to prove this semantics equivalent to an operational one.

First of all we separate the "logic programming" details (such as most general unifiers and renaming indices) in Prolog from the specification of the flow of control, (e.g. backtracking, the cut operator). This is achieved by extracting the uniform language $\mathcal{B}$ from Prolog - uniform in the sense of [BKMOZ] - which contains only the latter issues. Fitting within the "Logic Programming without Logic" approach, ([Ba2]), our denotational model developed for the abstract backtracking language has enough flexibility for further elaboration to a non-uniform denotational model of Prolog itself. Moreover, the equivalence of this denotational semantics and an operational semantics for Prolog is a straightforward generalization for the congruence proof of $\mathcal{B}$.

Secondly, our denotational semantics uses continuations. This has several advantages over earlier semantics which (essentially) are based on a direct approach. (See [Br] for a discussion on the relative merits of continuations vs. direct semantics.) We arrive at a concise set of semantical equations in which there is no need for coding up the states using cut flags

or special tokens (as in [JM], [DM], [Vi]). Moreover, since operational semantics must contain (syntactical) continuations, congruence of the two semantics can be established much more elegantly.

Our final contribution can be found in the equivalence proof itself. The equivalence proof does not split - as usual - into $\mathcal{O} \subseteq \mathcal{D}$ and $\mathcal{D} \subseteq \mathcal{O}$. Rather, both the operational and denotational semantics are represented as least upperbounds of chains and we prove equality of the approximating elements. (See also [KR], [BM] where - although not made explicit - in the setting of complete metric spaces operational and denotational semantics can be represented as limits of Cauchy sequences.)

The denotational semantics makes use of a fixed point construction with respect to environments. The environment transformation is a continuous operator on a cpo and as such it possesses a least fixed point. Alternatively, iterating this transformation from the bottom-environment yields a chain having the denotational semantics as its least upperbound. The operational semantics is based on a transition system. We shall define an ordering on transition systems such that the transition system underlying the operational semantics can also be obtained as a least upperbound. These transition systems are induced by subsets of configurations with a bound on the nesting of procedure calls. By allowing a deeper nesting of calls we obtain a better approximation of the operational semantics. Moreover, the k-th operational approximation will correspond with the k-th denotational one.

At the level of the approximating transition systems the principle of Noetherian induction holds, providing us with a convenient tool for comparing the two semantics. In fact we prove equivalence of an intermediate semantics (having both denotational and operational ingredients) on the one hand and the approximations of the denotational and operational semantics on the other by induction on the (finite) length of maximal transition sequences.

Related work on the denotational semantics of Prolog with cut includes [JM]a, [DM]a, [Vi]a. Jones and Mycroft present a direct Scott-Strachey style denotational semantics. They do not compare this semantics with an operational one. Instead, correctness of their semantics comes from its systematic construction. In [Vi]a also a direct denotational model is developed and additionally proven correct with respect to a transition based operational meaning. The proof is rather involved, since the cut is modeled by a special token (as in [JM]a). The semantics of Debray & Mishra is a mixture of a direct and continuation semantics. They (need to) have sequences of answers substitutions together with cut flags in their semantics. The denotational semantics is related to an operational one. However, it is not clear to us what makes their equivalence proof work. (In particular we do not understand the proof of theorem 4.1, case 5 in [DM]a.) The semantics mentioned above all denote a program by a sequence of substitutions. In the present paper we only deliver the first one. This does not give rise to loss of generality, since our semantics allows extension to streams of substitutions, (as in [Vi]a). We have chosen not to do so for reasons of space and clarity of the presentation.

The remainder of this paper, in which we present a continuation semantics for an abstract backtracking language $\mathcal{B}$ and for Prolog with cut, is organized as follows. Section 2 introduces the notion of transition system, which is used in section 3 to formulate the operational semantics for $\mathcal{B}$. Section 4 is devoted to the denotational semantics for $\mathcal{B}$. The correctness of this denotational semantics with respect to the operational one is established in section 5. In section 6 we change our point of view from imperative to logic programming. The denotational and operational semantics of the previous sections are interpreted and extended to handle Prolog. Some concluding remarks are made in section 7.

Section 2 Deterministic Transition Systems

In this section we introduce the notion of transition system, ([Pl], [BMOZ]). For reasons of space we restrict ourselves to deterministic transition systems, which already suit our purposes. Collections of transition systems are turned into a cpo such that associating a valuation to a transition system becomes a continuous operation.

(2.1) DEFINITION A deterministic transition system T is a seven tuple $\langle C, I, F, \Omega, D, \alpha, S \rangle$ where the set of configurations C is the disjoint union of I, F and $\{\Omega\}$, I is a set of internal configurations, F is a set of final configurations, Ω is the undefined configuration, D is a domain of values, α: $F \to D$ is a valuation assigning a value to each final configuration and S is a deterministic step- or transition-relation, i.e. a partial function S: $C \to_{part} C$ with $dom(S) \subseteq I$.

Next we show how to extend the valuation α on final configurations to a valuation α_T on arbitrary configurations of a transition system T.

(2.2) DEFINITION Let $T = \langle C, I, F, \Omega, D, \alpha, S \rangle$ be a deterministic transition system. Denote by $D_\perp$ the flat cpo generated by D with least element $\perp$. We associate with T a mapping α_T: $C \to D_\perp$ defined as the least function in $C \to D_\perp$ such that $\alpha_T(\Omega) = \perp$, $\alpha_T(c) = \alpha(c)$ if $c \in F$, $\alpha_T(c) = \alpha_T(c')$ if $(c,c') \in S$ and $\alpha_T(c) = \perp$ otherwise.

Fix sets I and F of internal and final configurations, respectively. Fix an undefined configuration Ω, a domain of values D, a valuation function $\alpha : F \to D$ and put $C = I \cup F \cup \{\Omega\}$. Let $TS = \{ \langle C, I, F, \Omega, D, \alpha, S \rangle \mid S : C \to_{part} C \text{ with } dom(S) \subseteq I \}$ denote the collection of all deterministic transition systems with configurations in C, internal configurations in I, final configurations in F, undefined configuration Ω, domain of values D and valuation function α. In TS we identify a transition system with its transition-relation. (In particular we may write $T(c)$ and $c \to_T c'$ rather than $S(c)$ or $(c, c') \in S$ for a transition system T with step-relation S.)

We consider the set of configurations as a flat cpo with ordering $\leq_C$ and least element Ω. This induces an ordering $\leq_{TS}$ on *TS* as follows: $T_1 \leq_{TS} T_2 \iff dom(T_1) \subseteq dom(T_2)$ & $\forall c \in dom(T_1)$: $T_1(c) \leq_C T_2(c)$. We have that *TS* is a cpo when ordered by $\leq_{TS}$. (The nowhere defined transition system $\varnothing$ is the least element of *TS*; for a chain $\langle T_k \rangle_k$ in *TS* the transition system *T* with $dom(T) = \cup_k dom(T_k)$ and $T(c) = lub_k\ T_k(c)$ acts as least upperbound.) Moreover, the operation $\lambda T.\alpha_T$: $TS \to C \to D_\perp$ that assigns to a transition system the valuation it induces, is continuous with respect to $\leq_{TS}$. (See [Vi]a.)

REMARK Let $I_0 \subseteq I_1 \subseteq \cdots$ be an infinite sequence of subsets of internal configurations such that $I = \cup_k I_k$. Put $C_k = I_k \cup F \cup \{\Omega\}$. Then we can construct for each $T \in TS$ a chain of approximations $\langle T_k \rangle_k$ of *T* in *TS*, where T_k is defined as the least deterministic transition system such that $T_k(c) = T(c)$ if $c \in I_k$, $T(c) \in C_k$, and $T_k(c) = \Omega$ if $c \in I_k$, $T(c)$ is defined but $T(c) \notin C_k$. Then it follows from the above that $T = lub_k\ T_k$ in *TS*. T_k is called the restriction of *T* to I_k since only configurations in I_k act as left-hand side. Note also that only configurations in C_k act as right-hand side.

We shall use this observation in the congruence proof of the operational and denotational semantics.

Section 3 Operational Semantics of $\mathcal{B}$

In this section we introduce the abstract backtracking language $\mathcal{B}$ and present an operational semantics based on a deterministic transition system. $\mathcal{B}$ can be regarded as a uniform version of Prolog with cut. For a program $d \mid s$ in $\mathcal{B}$, the declaration *d* will induce a transition system $\to_d$ while the statement *s* induces (given a state) an initial configuration. The operational semantics then is the value of the final configuration (if it exists) of the maximal transition sequence with respect to $\to_d$ starting from the initial configuration with respect to *s*.

(3.1) DEFINITION Fix a set of actions *Action* and a set of procedure names *Proc*. We define the set of elementary statements $EStat = \{ a, \textbf{\textit{fail}}, !, s_1 \textbf{\textit{ or }} s_2, x \mid a \in Action, s_i \in Stat, x \in Proc \}$, the set of statements $Stat = \{ e_1 ; .. ; e_r \mid r \in \mathbb{N}, e_i \in EStat \}$ and the set of declarations $Decl = \{ x_1 \leftarrow s_1 : .. : x_r \leftarrow s_r \mid r \in \mathbb{N}, x_i \in Proc, s_i \in Stat, i \neq j \Rightarrow x_i \neq x_j \}$. The backtracking language $\mathcal{B}$ is defined by $\mathcal{B} = \{ d \mid s \mid d \in Decl, s \in Stat \}$.

So an elementary statement is either an action in *Action*, the failure statement ***fail***, a PROLOG-like cut **!**, an alternative composition s_1 ***or*** s_2 or a procedure call *x*. A statement is a - possibly empty - sequential composition of elementary statements. The empty statement is denoted by ε. A declaration is a list of procedure definitions for different procedures

names. Programs are made up from a declaration and a program body, i.e. a statement.

We let a range over *Action*, x over *Proc*, e over *EStat*, s over *Stat* and d over *Decl*. We write $x \leftarrow s \in d$ if $x \leftarrow s = x_i \leftarrow s_i$ (for some i) or if $s =$ ***fail*** otherwise.

EXAMPLE Consider the context-free language $\mathcal{L}$ generated by the grammar $X \to YZ$, $Y \to aYa \mid bYb \mid a$, $Z \to cZ \mid c$. $\mathcal{L}$ consists of palindromes over $\{a, b\}$ with a in the middle, followed by an arbitrary but positive number of c's. A parser for $\mathcal{L}$ is implemented by the $\mathcal{B}$-program $d \mid s$ where $d = x \leftarrow y; z : y \leftarrow a; y; a$ ***or*** $(b; y; b$ ***or*** $a) : z \leftarrow c; z$ ***or*** c and $s = x;$ ***eoi***. Intuitively, the actions a, b and c succeed if the corresponding symbol is currently read. Otherwise the actions fail and cause a backtrack to possible (stacked) alternatives with their own local states (i.e. their own tape head positions). Analogously, the action ***eoi*** succeeds if the whole input is scanned yet and fails otherwise.

It is clear that once the palindrome part is recognized the alternative rules concerning the nonterminal Y do not have to be stacked any more. The cut **!** gives a mechanism to discard of these alternatives dynamically, in that it throws away all alternatives that have been generated since the body of the procedure have been entered containing this **!**-operator. Note that we have been careful to attempt to match the longest palindrome over a and b. Thus, here we can speed up the rejection of certain input if we map $X \to YZ$ on $x \leftarrow y; \,!\, ; z$ rather than on $x \leftarrow y; z$. We return to this example later.

Next we give an operational semantics to our backtracking language $\mathcal{B}$. Let $d \in$ *Decl*. The internal configurations of the transition system $\to_d$ associated with d are stacks. Each frame on a stack represents an alternative for the execution of some initial goal, i.e. statement. As such a frame consists of a generalized statement and a local state. The state can be thought of holding the values of the variables for a particular alternative. The generalized statement is composed from ordinary statements supplied with additional information concerning the cut: Each component in a generalized statement corresponds with a (nested) procedure call. The left-most component is the body being evaluated at the moment, i.e. the most deeply nested one. Since executing a cut amounts to restoring the backtrack stack as it was at the moment of procedure entry, we attach to a statement a stack (or pointer), that constitutes (points to) the substack of the alternatives that should remain open after a cut in the statement is executed. We call this stack the dump stack of the statement, cf. [JM]a. (The requirement for dump stacks being substack of (point into) the backtrack stack below the frame is not only for implementation reasons, but also of technical (mathematical) convenience later. See the proof of lemma 5.5.)

(3.2) DEFINITION Fix a set Σ of states. Define the set of generalized statements by *GStat* $= \{ \langle s_1, D_1 \rangle : .. : \langle s_r, D_r \rangle \mid r \in \mathbb{N}, s_i \in Stat, D_i \in Stack, i < j \Rightarrow D_i \geq_{ss} D_j \}$, γ denotes the empty generalized statement, the set of frames by *Frame* $= \{ [g, \sigma] \mid g \in GStat, \sigma \in \Sigma \}$ and the set of

stacks by $Stack = \{ F_1 : .. : F_r \mid r \in \mathbb{N},\ F_i = [\ \langle s_1 , D_1 \rangle : .. : \langle s_q , D_q \rangle, \sigma\ ] \in Frame$ such that $F_{i+1} : .. : F_r \geq_{ss} D_j \}$ (with $S \geq_{ss} S' \Leftrightarrow S'$ is a substack of S). Let $Conf = Stack \cup \Sigma \cup \{\Omega\}$ be the set of configurations.

Fix an action interpretation $I : Action \to \Sigma \to_{part} \Sigma$, that reflects the effect of the execution of an action on a state. (The language $\mathcal{B}$ gains flexibility if actions are allowed to succeed in the one state, while failing in another as is illustrated in the example. Hence we model failure as partiality.) Let *TS* be the collection of all deterministic transition system with configurations in *Conf*, internal configurations in *Stack*, final configurations in Σ, undefined configuration Ω, domain of values Σ, valuation $\alpha : \Sigma \to \Sigma$ with $\alpha(\sigma) = \sigma$. We distinguish $\delta \in \Sigma$ that will denote unsuccessful termination.

(3.3) DEFINITION Let $d \in Decl$. d induces a deterministic transition system in *TS* with as step-relation the smallest subset of $Conf \times Conf$ such that

(i) $\mathrm{E} \to_d \delta$

(ii) $[\ \gamma, \sigma\ ]{:}S \to_d \sigma$

(iii) $[\ \langle \varepsilon , D \rangle{:}g , \sigma\ ]{:}S \to_d [\ g , \sigma\ ]{:}S$

(iv) $[\ \langle a ; s , D \rangle{:}g , \sigma\ ]{:}S \to_d [\ \langle s , D \rangle{:}g , \sigma'\ ]{:}S$ if $\sigma' = I(a)(\sigma)$ exists
$[\ \langle a ; s , D \rangle{:}g , \sigma\ ]{:}S \to_d S$ otherwise

(v) $[\ \langle \textbf{\textit{fail}} ; s , D \rangle{:}g , \sigma\ ]{:}S \to_d S$

(vi) $[\ \langle \textbf{!} ; s , D \rangle{:}g , \sigma\ ]{:}S \to_d [\ \langle s , D \rangle{:}g , \sigma\ ]{:}D$

(vii) $[\ \langle x' ; s , D \rangle{:}g , \sigma\ ]{:}S \to_d [\ \langle s', S \rangle{:}\langle s , D \rangle{:}g , \sigma\ ]{:}S$ if $x \leftarrow s \in d$

(viii) $[\ \langle (s_1 \ \textbf{\textit{or}}\ s_2) ; s , D \rangle{:}g , \sigma\ ]{:}S \to F_1 : F_2 : S$ where $F_i = [\ \langle s_i ; s , D \rangle{:}g , \sigma\ ]$ $(i = 1, 2)$

We comment briefly on each of the above transitions (more precisely transition schemes).

(i) The empty stack, denoted by E, has no alternatives left to be tried. Hence the computation terminates unsuccessfully yielding δ.

(ii) If the top frame contains the empty generalized statement, denoted by γ, the computation terminates successfully. The local state σ of the frame is delivered as result.

(iii) If the left-most component of a generalized statement has become empty (the procedure call or initial statement has terminated), i.e. has format $\langle \varepsilon , D \rangle$, the statement-dump stack pair is deleted from the frame. The computation continues with the remaining generalized statement.

(iv) In case an action a in the top frame has become active, the action interpretation I is consulted for the effect of a in σ. If $I(a)(\sigma)$ is defined the state is transformed accordingly. If $I(a)(\sigma)$ is not defined the frame fails and is popped of the stack.

(v) Execution of ***fail*** amounts to failure of the current alternative. Hence the top frame is popped of the backtrack stack. Control is transferred to the new top frame.

(vi) The transition concerning the cut represents removal of alternatives; the top frame continues its execution. Since the dump stack *D* is a substack of the backtrack stack *S*, replacing the backtrack stack by the current dump stack indeed amounts - in general - to deletion of frames, i.e. of alternatives. (Note that the right-hand stack is well-formed by definition of *GStat*.)

(vii) A call initiates body replacement. The body is looked up in the declaration *d* and becomes the active component of the generalized statement in the top frame. This component has its own dump stack, which is (a pointer to) the backtrack stack at call time.

(viii) Execution of an alternative composition yield two new frames: an active frame corresponding to the left component of the or-construct and a suspended frame corresponding to the right component.

(3.4) DEFINITION The operational semantics $\mathcal{O}: \mathcal{B} \to \Sigma \to \Sigma_\perp$ for the backtrack language $\mathcal{B}$ is defined by $\mathcal{O}(d \mid s)(\sigma) = \alpha_d([\langle s, E \rangle, \sigma])$ where $\alpha_d : \mathit{Conf} \to \Sigma_\perp$ is the valuation associated with the deterministic transition system induced by *d*.

EXAMPLE (Continued) Consider the declaration $d = x \leftarrow y\,;\,!\,;z \;:\; y \leftarrow a;y;a$ ***or*** $(b;y;b$ ***or*** $a) : z \leftarrow c;z$ ***or*** c. Let us calculate as illustration of the definitions parts of the transition sequence for the statement $x;$ ***eoi*** in state *ababad*$. (Here the state reflects the input buffer. $ represents acceptance; all other states represent rejection.) The interpretation of the actions *a*, *b*, *c* and ***eoi*** is as described previously: for $\alpha \in \{ a, b, c \}$ we assume $I(\alpha)(\alpha w) = w$, $I(\alpha)$ is undefined otherwise and $I(\mathbf{eoi})(\$) = \$$, $I(\mathbf{eoi})$ fails, i.e. is undefined, otherwise. Note *ababad* $\notin \mathcal{L}$.

[< x ; eoi , E > , *ababad$*]

(1) →

[< y ; ! ; z , E >:< eoi , E > , *ababad$*]

(2) →

[< a ; y ; a or (b ; y ; b or a) , E >:< ! ; z , E >:< eoi , E > , *ababad$*]

(3) →

[< a ; y ; a , E >:< ! ; z , E >:< eoi , E > , *ababad$*]

[< b ; y ; b or a , E >:< ! ; z , E >:< eoi , E > , *ababad$*]

(4) →

[< y ; a , E >:< ! ; z , E >:< eoi , E > , *babad$*]

[< b ; y ; b or a , E >:< ! ; z , E >:< eoi , E > , *ababad$*]

→

. . .

→

[< ε , E >:< ! ; z , E >:< *eoi* , E > , *d$*]
[< a , #1 >:< a, E >:< ! ; z , E >:< *eoi* , E > , *babad$*]
[< b ; y ; b *or* a , E >:< ! ; z , E >:< *eoi* , E > , *ababad$*]
(5) →
[< ! ; z , E >:< *eoi* , E > , *d$*]
[< a , #1 >:< a, E >:< ! ; z , E >:< *eoi* , E > , *babad$*]
[< b ; y ; b *or* a , E >:< ! ; z , E >:< *eoi* , E > , *ababad$*]
(6) →
[< z , E >:< *eoi* , E > , *d$*]
→
. . .
→
δ

where #1 denotes a pointer into the appropriate substack. Transitions (1) and (2) follow the scheme of 3.3(vii), and create new components in the generalized statements. Transition (3) shows how the alternatives are distributed according to 3.3.(viii). Since the action *a* succeeds in state *ababad$*, yielding *babad$*, the first clause of 3.3(iv) is applicable at transition (4). At transition (5) the procedure call for *y* has terminated. The corresponding component is deleted. At transition (6) one can see the effect of evaluation of the cut: execution of the cut amounts to removal of the two lowest frames.

Section 4 Denotational Semantics for $\mathcal{B}$

One of the claims of this paper is that a more natural denotational semantics for Prolog can be defined when using continuations instead of direct semantics. This section and the next one will provide a justification of this claim. In this section we shall establish a concise set of semantic equations in which there will be no need for "alien" components in our states like the cut indicators in [JM]a, [DM]a and [Vi]a. The next section will feature a straightforward equivalence proof, to be contrasted with [DM]a, [Vi]a.

By now a standard approach has been established for defining a denotational semantics of a sequential procedural language. Cf. [MS], [St], [Ba1], [Te2]. We show that a semantics of $\mathcal{B}$ in this section and Prolog in section 6 can also be given along these lines. Standard semantics uses environments and continuations.

Environments are needed because the denotation $[\![s]\!]_\eta$ of a statement *s* depends amongst others on the meaning of the procedure names occurring in *s*. Therefore the function $[\![\cdot]\!]_\eta$ takes an environment $\eta \in Env$ as a parameter which defines the meaning of all

procedure names.

The flow of control will be described using continuations. For languages like PASCAL, where flow of control is not very intricate, a denotation $[\![s]\!]_s$ needs only one continuation as a parameter. Languages containing backtrack constructs, like SNOBOL4, are best described using two continuations, cf. [Te1], [Te2]a. In order to capture the effects of the cut operator yet another continuation will be needed. (As is observed independently by M. Felleisen in [Wi], p. 273.) In order to explain how these continuations will be used we introduce them one after another. First we shall discuss the PASCAL-subset of $\mathcal{B}$, i.e. $\mathcal{B}$ without ***or*** , ***fail*** and cut **!**. Thereafter we shall examine the SNOBOL4-subset of $\mathcal{B}$, introducing the ***or*** and ***fail*** constructs, and finally we shall explain how all three continuations are used in describing full $\mathcal{B}$. (In section 6 we interpret the language $\mathcal{B}$ and its semantics to arrive at a denotational and operational semantics for Prolog with cut.)

In order to understand the essence of continuation semantics, consider a substatement s that is part of a statement s' (in the PASCAL-fragment of $\mathcal{B}$). The denotation $[\![s]\!]_s$ will be a function that will, in the end, deliver an answer in $\Sigma_\perp$. This answer is not the result of executing s alone, but the result of evaluating the whole statement s' of which s is a substatement. Therefore the result does not only depend on an environment η and an initial state σ, it also depends on a denotation ξ of the remainder of the statement, to be executed once evaluation of s has terminated. This leads to the following functionality of $[\![\cdot]\!]_s : Env \to Cont \to \Sigma \to \Sigma_\perp$. Here $Cont = \Sigma \to \Sigma_\perp$ since the future ξ of a computation will in the end yield an answer, but this answer depends on an intermediate state, viz. the result of evaluating s alone. A typical clause in our semantics up till now, describing the composition operator ";", will be $[\![e ; s]\!]_s \eta\xi\sigma = [\![e]\!]_e \eta\{[\![s]\!]_s \eta\xi\}\sigma$, which says that the answer obtained by executing $e ; s$ before ξ will be equal to the answer resulting from execution of e before { execution of s before ξ }.

The next stage is to introduce backtracking in the language by adding the constructs ***or*** and ***fail*** (and by allowing actions to fail). Describing the flow of control is more complicated now. The problem is that the notion "future of the computation" is not that obvious any more. Evaluation of a statement s can terminate for two reasons now. The first one, successful termination, is similar to the situation we had before. In this case the future of the computation is realized by executing the remainder of the statement textually following s. But now it is also possible that evaluation of s terminates in failure, e.g. by executing a ***fail*** statement. Now the rest of the computation is determined by backtracking to the alternatives built up through execution of ***or*** -statements in earlier stages of the computation. Such a doubly edged future can best be captured by two continuations, a success continuation $\xi \in SCont$ and a failure continuation $\phi \in FCont$. So now $[\![\cdot]\!]_s$ has a new functionality: $[\![\cdot]\!]_s : Env \to SCont \to FCont \to \Sigma \to \Sigma_\perp$. The meaning $[\![s]\!]_s \eta\xi\phi\sigma$ of s will depend on ξ, denoting the rest of the statement following s, and on ϕ, which is a denotation of the stack of alternatives built up

in the past. *FCont* is best understood by investigating the meaning of the ***or*** construct: $[\![s_1 \textbf{ or } s_2]\!]_s \eta\xi\phi\sigma = [\![s_1]\!]_s \eta\xi\phi'\sigma$. This says that executing s_1 ***or*** s_2 amounts to executing s_1 with a new failure continuation ϕ' describing what will happen if s_1 terminates in failure. In that case s_2 should be executed, and only if this also ends in failure the computation should proceed as indicated by the original failure continuation ϕ. Hence we have that ϕ' equals $[\![s_2]\!]_s \eta\xi\phi\sigma$. Combining all this we obtain that $[\![s_1 \textbf{ or } s_2]\!]_s \eta\xi\phi\sigma = [\![s_1]\!]_s \eta\xi\{[\![s_2]\!]_s \eta\xi\phi\}\sigma$. Apparently we have $FCont = \Sigma_\perp$. As far as the structure of *SCont* is concerned, it must be realized that the answer obtained from evaluation of the rest of the statement s' does not only depend on the intermediate state resulting from the evaluation of s but also on the alternatives built up by executing s' up to and including s. For it can very well happen that evaluation of the rest of the statement will terminate in failure. We therefore have $SCont = FCont \to \Sigma \to \Sigma_\perp$. We notice that the meaning of the ***fail*** statement is straightforward. The answer is the one provided by the failure continuation: $[\![\textbf{fail}]\!]_e \eta\xi\phi\sigma = \phi$. This is also the case if an action a does not succeed in a state σ, i.e. $[\![a]\!]_e \eta\xi\phi\sigma = \phi$ if $I(a)(\sigma)$ is undefined, (where I is the fixed action interpretation). If a does succeed the state is transformed according to I and the failure continuation and new state are passed to the success continuation ξ. So $[\![a]\!]_e \eta\xi\phi\sigma = \xi\phi\sigma'$ if $\sigma' = I(a)(\sigma)$ exists.

The only construct of full $\mathcal{B}$ that we did not take into account up to now is the cut operator **!**. This statement resembles the dummy statement because it does not affect the state. There is a side effect however, since a number of alternatives is thrown away. To be more precise, evaluation of **!** discards the alternatives which have been generated since the procedure body in which the **!** occurs has been entered. For our semantics this means that evaluation of **!** amounts to applying the success continuation to the original state (this is the dummy statement aspect), but also to a new failure continuation. This new failure continuation ϕ' is in fact an old one, namely the failure continuation which was in effect on entry of the procedure body in which the **!** occurs. A natural way to obtain this old continuation, which we will call the cut continuation $\kappa \in CCont$ in the sequel, is to provide it as an argument of the meaning function $[\![\cdot]\!]_s$. We finally arrive at the functionality $[\![\cdot]\!]_s : Stat \to Env \to SCont \to FCont \to CCont \to \Sigma \to \Sigma_\perp$, with $FCont = CCont = \Sigma_\perp$. The denotation of **!** can now be given by $[\![\textbf{!}]\!]_e \eta\xi\phi\kappa\sigma = \xi\kappa\kappa\sigma$. On entry of a procedure body a new cut continuation has been established. The meaning of a procedure call is straightforward. We have $[\![x]\!]_e \eta\xi\phi\kappa\sigma = \eta x\xi\phi\kappa\sigma$, i.e. the arguments ξ, ϕ, κ and σ are passed to the meaning ηx of x in the environment η. The real work is performed in the definition of the environment η, which should be derived from the declaration d in the program. We want η to be a fixed point such that ηx, the meaning of the procedure name x is given by $\eta x\xi\phi\kappa\sigma = [\![s]\!]_s \eta\{\lambda\bar{\phi}\bar{\kappa}.\xi\bar{\phi}\kappa\}\phi\phi\sigma$ if $x \leftarrow s \in d$. Two effects can be noticed here. First of all a new cut continuation, viz. the failure continuation ϕ, is "passed", secondly on (successful) termination of s the old cut continuation should be restored and this is captured by passing $\{\lambda\bar{\phi}\bar{\kappa}.\xi\bar{\phi}\kappa\}$ instead of ξ to the body s.

We now give the semantics of $\mathcal{B}$. We first give the domains: the set of failure continuations $FCont = \Sigma_\perp$, the set of cut continuations $CCont = \Sigma_\perp$, the set of success continuations $SCont = FCont \to CCont \to \Sigma \to \Sigma_\perp$ and the set of environments $Env = Proc \to SCont \to FCont \to CCont \to \Sigma \to \Sigma_\perp$. We denote by σ, ϕ, κ, ξ and η typical elements of Σ, *FCont*, *CCont*, *SCont* and *Env*, respectively.

(4.1) DEFINITION

(i) $[\![\cdot]\!]_e : EStat \to Env \to SCont \to FCont \to CCont \to \Sigma \to \Sigma_\perp$

$[\![a]\!]_e \eta\xi\phi\kappa\sigma = \xi\phi\kappa\sigma'$ if $\sigma' = I(a)(\sigma)$ exists

$[\![a]\!]_e \eta\xi\phi\kappa\sigma = \phi$ otherwise

$[\![\textbf{\textit{fail}}]\!]_e \eta\xi\phi\kappa\sigma = \phi$

$[\![!]\!]_e \eta\xi\phi\kappa\sigma = \xi\kappa\kappa\sigma$

$[\![s_1 \textbf{ or } s_2]\!]_e \eta\xi\phi\kappa\sigma = [\![s_1]\!]_s \eta\xi\{[\![s_2]\!]_s \eta\xi\phi\kappa\sigma\}\kappa\sigma$

$[\![x]\!]_e \eta\xi\phi\kappa\sigma = \eta\, x\xi\phi\kappa\sigma$

(ii) $[\![\cdot]\!]_s : Stat \to Env \to SCont \to FCont \to CCont \to \Sigma \to \Sigma_\perp$

$[\![\varepsilon]\!]_s \eta\xi\phi\kappa\sigma = \xi\phi\kappa\sigma$

$[\![e\,; s]\!]_s \eta\xi\phi\kappa\sigma = [\![e]\!]_e \eta\{[\![s]\!]_s \eta\xi\}\phi\kappa\sigma$

(iii) $\Phi : Decl \to Env \to Env$

$\Phi\, d\eta\, x\xi\phi\kappa\sigma = [\![s]\!]_s \eta\{\lambda\bar{\phi}\bar{\kappa}.\xi\bar{\phi}\kappa\}\phi\phi\sigma$ if $x \leftarrow s \in d$

(iv) $[\![\cdot]\!]_{\mathcal{B}} : \mathcal{B} \to \Sigma \to \Sigma_\perp$

$[\![d \mid s]\!]_{\mathcal{B}} \sigma = [\![s]\!]_s \eta_d \xi_0\, \phi_0\, \kappa_0\, \sigma$

where η_d is the least fixed point of $\Phi(d)$, $\xi_0 = \lambda\phi\kappa\sigma.\sigma$ and $\phi_0 = \kappa_0 = \delta$.

The above semantics, using a fixed point construction, is well defined according to the following lemma which can be established by simultaneous induction.

(4.2) LEMMA $[\![\cdot]\!]_e$, $[\![\cdot]\!]_s$ *and* Φ *are continuous in* η. □

REMARK The least fixed point η_d defined in 4.1(iv) can be obtained as the least upper-bound of a chain of iterations $\langle \eta_{d,i} \rangle_i$, with $\eta_{d,i}$ defined by $\eta_{d,0} = \lambda x\xi\phi\kappa\sigma.\perp$ and $\eta_{d,i+1} = \Phi(d)(\eta_{d,i})$. From the continuity of $[\![\cdot]\!]_s$ we have $[\![s]\!]_s \eta_d = lub_i\, [\![s]\!]_s \eta_{d,i}$.

We conclude this section with a few remarks on the similarity of the operational semantics from the previous section and the denotational semantics of this one. There is a natural correspondence between components of a configuration and the parameters of the denotation of a statement. We compare the answer resulting from evaluation of an elementary statement e and the value obtained from a configuration in which e is about to be executed: $[\![e]\!]_e \eta\xi\phi\kappa\sigma$ vs. $[\langle e\,; s, D \rangle : g, \sigma] : S$. Here ξ is a denotation of the statements to be executed once e has terminated successfully. So ξ corresponds to the statement s followed by the

statements in the generalized statement *g*. The failure continuation ϕ is the denotational counterpart of the backtrack stack *S*, the cut continuation κ corresponds to the dump stack *D*. It is to be expected that if the correspondence is set up as above, the resulting answers should be the same. This will be formalized in the next section and is pivotal to the equivalence proof given there.

Section 5 Equivalence of $\mathcal{O}$ and $\mathcal{D}$

In this section we prove the equivalence of the operational and denotational semantics, thus justifying the definition of the latter one.

(5.1) THEOREM *For all* $d \mid s \in \mathcal{B}$: $[\![d \mid s]\!]_{\mathcal{B}} = \mathcal{O}(d \mid s)$.

In order to prove theorem 5.1 we use the cpo-structure on the collection of transition systems *TS* and the continuity of the statement evaluator $[\![\cdot]\!]_{\mathcal{S}}$. According to the remark at the end of section 2 and the remark following lemma 4.2 we have that both the operational and denotational semantics can be represented as the limit of a chain. Hence, equivalence of the two semantics is proven if we can establish that the approximations in the chains are pairwise equal.

To be more specific: Let $d \mid s \in \mathcal{B}$ and $\sigma \in \Sigma$. Let $\rightarrow_d$ and α_d be the deterministic transition system in *TS* and associated valuation induced by *d*, respectively. Let $Stack_0 \subseteq Stack_1 \subseteq \cdots$ be a certain sequence of subsets of *Stack* such that $Stack = \cup_i Stack_i$ (to be defined later). Each subset $Stack_i$ defines an approximation $\rightarrow_{d,i}$ of $\rightarrow_d$, viz. the restriction of $\rightarrow_d$ to $Stack_i$. Let $\alpha_{d,i}$ be the valuation associated with $\rightarrow_{d,i}$. Then we have on the one hand $\mathcal{O}(d \mid s)(\sigma) = lub_i\ \alpha_{d,i}([\langle s, E\rangle, \sigma])$. On the other hand we have that the least fixed point η_d of $\Phi(d)$ can be written as $\eta_d = lub_i\ \eta_{d,i}$ where $\eta_{d,i}$ is the *i*-th iteration of $\lambda x \xi \phi \kappa \sigma.\bot$ by $\Phi(d)$. So $[\![d \mid s]\!]_{\mathcal{B}}\ \sigma = lub_i\ [\![s]\!]_{\mathcal{S}}\ \eta_{d,i} \xi_0\ \phi_0\ \kappa_0\ \sigma$ (with ξ_0, ϕ_0, κ_0 as defined in definition 4.1(iv)). Hence we are done if $\alpha_{d,i}([\langle s, E\rangle, \sigma]) = [\![s]\!]_{\mathcal{S}}\ \eta_{d,i} \xi_0\ \phi_0\ \kappa_0\ \sigma$ for all *i*.

However, in order to prove equality of $\alpha_{d,i}([\langle s, E\rangle, \sigma])$ and $[\![s]\!]_{\mathcal{S}}\ \eta_{d,i} \xi_0\ \phi_0\ \kappa_0\ \sigma$ we need a stronger result. To this end we construct an intercedent between $\alpha_{d,i}$ and $[\![\cdot]\!]_{\mathcal{S}}$: We define a (denotational) function $[\![\cdot]\!]_{\mathcal{C}}$ on (operational) configurations with parameters *d* and *i* and show that for all configurations *C* we have that the value $\alpha_{d,i}(C)$ equals $[\![C]\!]_{\mathcal{C}}\ di$. The desired result then follows from $[\![s]\!]_{\mathcal{S}}\ \eta_{d,i} \xi_0\ \phi_0\ \kappa_0\ \sigma = [\![[\langle s, E\rangle, \sigma]]\!]_{\mathcal{C}}\ di$ which can be checked by routine. Also the equality of $[\![\cdot]\!]_{\mathcal{C}}$ and $\alpha_{d,i}$ will be easy to check once the appropriate tool is available.

Having outlined the strategy for the equivalence proof we continue with the definition of the intermediate function $[\![\cdot]\!]_{\mathcal{C}}$. First we have to specify the subsets of configurations

$Stack_i$. The environment $\eta_{d,i}$, being the i-th iteration of the bottom-environment, yields the right answer in x provided the call of x leads to at most $i-1$ nested inner calls. This depth of nesting can be controlled in our operational semantics as well. Each component $\langle s_i , D_i\rangle$ in a generalized statement g corresponds to a (nested) procedure call. The depth of nesting in $g = \langle s_1 , D_1\rangle : .. : \langle s_r , D_r\rangle$ therefore equals r.

(5.2) DEFINITION Let $GStat_i = \{ g \in GStat \mid \| g \| \leq i \}$ where $\| g \| = r$ for $g = \langle s_1 , D_1\rangle : .. : \langle s_r , D_r\rangle$, $Frame_i = \{ [g, \sigma] \in Frame \mid g \in GStat_i \}$ and $Stack_i = \{ F_1 : .. : F_r \in Stack \mid F_j \in Frame_i \}$.

Although for $g = \langle s_1 , D_1\rangle : .. : \langle s_r , D_r\rangle$ we do not require $D_j \in Stack_i$, this is the case if $[g, \sigma]:S \in Stack_i$ since $[g, \sigma]:S \in Stack$ implies that D_j is a substack of S.

Next we define the intermediate function $[\![\cdot]\!]_{\mathscr{C}}$. Given a stack S the definition of $[\![\cdot]\!]_{\mathscr{C}}$ can only be elaborated further if $S \in Stack_i$. Otherwise the value $\perp$ is returned. Intuitively $\perp$ expresses uncertainty about the value of the configuration. So $\perp$ will be delivered if the elaboration asks for a chain of nested calls of length exceeding i.

(5.3) DEFINITION

(i) $[\![\cdot]\!]_{\mathscr{C}} : Conf \to Decl \to \mathbb{N} \to \Sigma_\perp$
$[\![S]\!]_{\mathscr{C}}\, di = [\![S]\!]_{\mathscr{S}}\, di$, $[\![\sigma]\!]_{\mathscr{C}}\, di = \sigma$, $[\![\Omega]\!]_{\mathscr{C}}\, di = \perp$

(ii) $[\![\cdot]\!]_{\mathscr{S}} : Stack \to Decl \to \mathbb{N} \to \Sigma_\perp$
$[\![E]\!]_{\mathscr{S}}\, di = \delta$, $[\![F:S]\!]_{\mathscr{S}}\, di = [\![F]\!]_{\mathscr{F}}\, di\{[\![S]\!]_{\mathscr{S}}\, di\}$ if $F:S \in Stack_i$,
$[\![S]\!]_{\mathscr{S}}\, di = \perp$ if $S \notin Stack_i$

(iii) $[\![\cdot]\!]_{\mathscr{F}} : Frame \to Decl \to \mathbb{N} \to FCont \to \Sigma_\perp$
$[\![[g, \sigma]]\!]_{\mathscr{F}}\, di\phi = [\![g]\!]_{\mathscr{G}}\, di\phi\sigma$

(iv) $[\![\cdot]\!]_{\mathscr{G}} : GStat \to Decl \to \mathbb{N} \to FCont \to \Sigma \to \Sigma_\perp$
$[\![\gamma]\!]_{\mathscr{G}}\, di\phi\sigma = \sigma$, $[\![\langle s , D\rangle:g]\!]_{\mathscr{G}}\, di\phi\sigma = [\![s]\!]_{\mathscr{S}}\, \eta_{d,i,g}\{\lambda\bar{\phi}\bar{\kappa}.[\![g]\!]_{\mathscr{G}}\, di\bar{\phi}\}\phi\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma$
where $\eta_{d,i,g} = \eta_{d,j}$ with $j = i \dot{-} (\| g \| + 1)$ where $\| \langle s_1 , D_1\rangle : .. : \langle s_r , D_r\rangle \| = r$

$[\![\langle s , D\rangle:g]\!]_{\mathscr{G}}\, di$ should yield the right answer only if this can be obtained with less than i nested calls. Now g is responsible for a nesting depth $\| g \|$. So the whole generalized statement $\langle s , D\rangle:g$ has a chain of $\| g \| + 1$ nested calls already. This means that $\eta_{d,i,g}$ should allow less then $i \dot{-} (\| g \| + 1)$ calls. (Here $\dot{-}$ denotes the monus, i.e. subtraction in $\mathbb{N}$.)

The desired property of the function $[\![\cdot]\!]_{\mathscr{C}}$ is stated in the next lemma. Recall that $\alpha_{d,i}$ is the valuation of the transition system $\rightarrow_{d,i}$ which is the restriction of $\rightarrow_d$ to $Stack_i$. So no transition is defined for stacks S not in $Stack_i$, hence $\alpha_{d,i}(S) = \perp$.

(5.4) LEMMA *For all $d \in Decl$ and $i \in \mathbb{N}$ we have* $[\![\cdot]\!]_{\mathscr{C}}\, di = \alpha_{d,i}$.

Anticipating to the proof of lemma 5.4 at the end of this section, we can obtain from it the equivalence of the operational and denotational semantics. Note the decomposition of both $\mathcal{O}$ and $\mathcal{D}$, i.e. of α_d and $[\![s]\!]_{\mathcal{S}} \eta_d$ into $lub_i\ \alpha_{d,i}$ and $lub_i\ [\![s]\!]_{\mathcal{S}} \eta_{d,i}$, respectively.

PROOF (of theorem 5.1) Let $\sigma \in \Sigma$. $[\![d \mid s]\!]_{\mathcal{B}} \sigma = [\![s]\!]_{\mathcal{S}} \eta_d \xi_0 \phi_0 \kappa_0 \sigma$ (by definition) = $lub_i\ [\![s]\!]_{\mathcal{S}} \eta_{d,i} \xi_0 \phi_0 \kappa_0 \sigma$ (by continuity of $[\![\cdot]\!]_{\mathcal{S}}$) = $lub_i\ [\![[\langle s, \mathrm{E}\rangle, \sigma]]\!]_{\mathcal{S}'} di$ (straightforward) = $lub_i\ \alpha_{d,i}([\langle s, \mathrm{E}\rangle, \sigma])$ (by the lemma) = $\alpha_d([\langle s, \mathrm{E}\rangle, \sigma])$ (by continuity of $\lambda T.\alpha_T$) = $\mathcal{O}(d \mid s)(\sigma)$ (by definition). □

The sequel of this section is devoted to the proof of 5.4. First we establish Noetherianity of the restrictions of $\rightarrow_d$ to $Stack_i$, i.e. the absence of infinite transition sequences with respect to $\rightarrow_{d,i}$. This supplies us with an induction principle that we shall use in the proof of the lemma.

(5.5) LEMMA *Let $d \in Decl$, $i \in \mathbb{N}$ and $\rightarrow_{d,i}$ be the restriction of $\rightarrow_d$ to $Stack_i$. Then we have that $\rightarrow_{d,i}$ is Noetherian.*

PROOF We construct a suitable weight-function for the internal configurations in $dom(\rightarrow_{d,i})$, i.e. for stacks in $Stack_i$. Define a complexity measure on statements as follows: $c(a) = c(\textbf{\textit{fail}}) = c(\,!\,) = c(x) = 1$, $c(s_1\ \textbf{\textit{or}}\ s_2) = 1+c(s_1)+c(s_2)$ and $c(e_1;..;e_r) = 1+c(e_1)+..+c(e_r)$. To $g = \langle s_1, D_1\rangle : .. : \langle s_r, D_r\rangle$ in $GStat_i$ and $[g, \sigma]$ in $Frame_i$ we assign as weight the ordinal $w(g) = w([g, \sigma]) = \omega^i \cdot c(s_r)+..+\omega^{i-r+1} \cdot c(s_1)$. (This defines in fact a reversed lexicographical ordering on $GStat_i$ and $Frame_i$, where we have a bounded number of components in a generalized statement, induced by the complexity measure c.)

Consider the stack $S = F_1 : .. : F_r$ in $Stack_i$. Choose ordinals $\alpha_1 < .. < \alpha_k$ and natural numbers $n_1, .., n_k$ such that there are exactly n_i frames among $F_1, .., F_r$ of weight α_i and $n_1+..+n_k = r$. Assign to S the ordinal $w(S) = \omega^{\alpha_k} \cdot n_k+..+\omega^{\alpha_1} \cdot n_1$. We leave it to the reader to verify that for $S, S' \in Stack_i$ such that $S \rightarrow_{d,i} S'$, it holds that $w(S) > w(S')$. (Note the requirement for dump stacks with respect to the cut-rule. See the remark following definition 5.2.) Since ω_1 is well-ordered we derive immediately that there is no infinite transition sequence for the system $\rightarrow_{d,i}$. That is $\rightarrow_{d,i}$ is Noetherian. □

We have taken the idea of using ordinals, viz. Cantor normal forms, in the context of Noetherianity of transition (reduction) systems from [Kl]. We appreciate the flexibility of this method as opposed to coding within (sequences of) natural numbers.

We proceed with the proof of the equality $[\![\cdot]\!]_{\mathcal{C}} di = \alpha_{d,i}$ (*). First we notice that this holds for final configurations $\sigma \in \Sigma$, for the undefined configurations Ω, and for internal configurations that admit no transition, i.e. stacks not in $Stack_i$.

We shall prove that (*) is also satisfied by internal configurations that do admit a

transition, i.e. stacks in $Stack_i$. For this we observe that given the above it suffices to prove: if $C \to_{d,i} C'$ and (*) holds for C' then (*) holds for C too, by virtue of the Noetherianity of the transition system $\to_{d,i}$. (This is the principle of Noetherian induction, although in our - deterministic - case it specializes to induction on the length of the maximal transition sequence (which is finite) out of a configuration. See e.g. [Hu].) By definition of the valuation $\alpha_{d,i}$ we have $\alpha_{d,i}(C) = \alpha_{d,i}(C')$ provided $C \to_{d,i} C'$. So we only need to show: if $C \to_{d,i} C'$ then $[\![C]\!]_{\mathscr{C}}\, di = [\![C']\!]_{\mathscr{C}}\, di$.

PROOF (of lemma 5.4) Let $d \in Decl$, $i \in \mathbb{N}$ and $C, C' \in Conf$ such that $C \to_{d,i} C'$. Note $C \in Stack_i$. It suffices to show by structural induction on C: $[\![C]\!]_{\mathscr{S}}\, di = [\![C']\!]_{\mathscr{C}}\, di$.

We only treat case (vi) of definition 3.3: $C = [\langle x'; s, D\rangle{:}g,\sigma]{:}S$. Say $x' \leftarrow s' \in d$. We distinguish two subcases: Subcase (a): $\|\langle x'; s, D\rangle{:}g\| = i$. Then we have $C' = \Omega$ and $\|g\| = i-1$.

$$
\begin{aligned}
&[\![[\langle x'; s, D\rangle{:}g,\sigma]{:}S]\!]_{\mathscr{S}}\, di \\
&= [\![x']\!]_{\mathscr{e}}\, \eta_{d,i,g}\{ [\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma \\
&= \eta_{d,i,g}\, x'\{ [\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma \\
&= \eta_{d,0}\, x'\{ [\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma \\
&= \bot \\
&= [\![\Omega]\!]_{\mathscr{C}}\, di.
\end{aligned}
$$

Subcase (b): $\|\langle x'; s, D\rangle{:}g\| < i$. Then we have $C' = [\langle s', S\rangle{:}\langle s, D\rangle{:}g,\sigma]{:}S$ and $\|g\| < i-1$.

$$
\begin{aligned}
&[\![[\langle x'; s, D\rangle{:}g,\sigma]{:}S]\!]_{\mathscr{S}}\, di \\
&= \eta_{d,i,g}\, x'\{ [\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma \\
&= \Phi\, d\, \eta_{d,i-1,g}\, x'\{ [\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![D]\!]_{\mathscr{S}}\, di\}\sigma \\
&= [\![s']\!]_{\mathscr{s}}\, \eta_{d,i-1,g}\bar{\xi}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![S]\!]_{\mathscr{S}}\, di\}\sigma \\
&\quad \text{where } \bar{\xi} = \lambda\bar{\phi}\bar{\kappa}.[\![s]\!]_{\mathscr{s}}\, \eta_{d,i,g}\{\lambda\phi\kappa.[\![g]\!]_{\mathscr{g}}\, di\phi\}\bar{\phi}\{[\![D]\!]_{\mathscr{C}}\, di\} \\
&= [\![s']\!]_{\mathscr{s}}\, \eta_{d,i,\langle s, D\rangle:g}\{\lambda\bar{\phi}\bar{\kappa}.[\![\langle s, D\rangle{:}g]\!]_{\mathscr{g}}\, di\bar{\phi}\}\{[\![S]\!]_{\mathscr{S}}\, di\}\{[\![S]\!]_{\mathscr{S}}\, di\}\sigma \\
&= [\![[\langle s', S\rangle{:}\langle s, D\rangle{:}g,\sigma]{:}S]\!]_{\mathscr{S}}\, di
\end{aligned}
$$

The other cases are similar, (easier) and omitted here. □

Having proved lemma 5.4 the congruence proof of the operational and denotational semantics for $\mathscr{B}$ is completed. In the next section we modify both this operational and denotational semantics in order to give meaning to Prolog with cut.

Section 6 Interpretation of $\mathscr{B}$ into Prolog

At the moment Prolog ([CKRP]) is probably the most important programming language featuring backtracking. It can be viewed as Horn clause logic with a left-most depth-first computation rule. Nevertheless Prolog contains execution oriented constructs, e.g. the cut, that makes the standard declarative semantics, that associates to a set of clauses its least

Herbrand model ([AE], [EK]), less satisfactorily. Although dating from the early seventies, it has lasted until 1984 before a denotational semantics for Prolog was presented, viz. [JM]a, that gave account to the behavioral aspects of the language. More recently other (denotational) semantics based on several approaches have appeared, e.g. [DM]a, [Vi]a. (See also [Fi], [Fr], [AB], [BW].)

Our work on the backtracking language $\mathcal{B}$ in the previous sections makes yet another semantics easily available: we can interpret the abstract or uniform statements, declarations and states such that: a set of Prolog-clauses can be regarded as a declaration, a Prolog-goal corresponds with a statement in the abstract language, while a substitution can be viewed as a state. (After all this is not surprising since we designed $\mathcal{B}$ as an abstraction of Prolog.)

This can be done similarly for the operational semantics. Moreover, the interpretation or de-uniformization is done in such a way that the equivalence proof remains valid (after adaptation to minor technicalities). Having factorized the work for a Prolog-semantics in a control flow component (the abstract language $\mathcal{B}$) and a logical component (the interpretation of $\mathcal{B}$ towards Prolog) we obtain presently a congruence proof for the denotational and operational semantics almost for free. Stated otherwise, we have an instance of the "Algorithm = Logic + Control" paradigm ([Kw]) at the meta level. (In fact, several semantics of logic programming languages can be considered as generalizations of established models for imperative languages with respect to the control; the extensions made are concerned with the particular logic component. Cf. [GCLS], [Kk], [Ba2]a. See in particular [BK] for a related approach in the setting of Concurrent Prolog.)

Unfortunately there is a price to pay for our two pass approach, albeit just a syntactical one. Since we restrict procedure names in $\mathcal{B}$ to have just one procedure body, we can consider clauses with pairwise different head predicates only. We feel free to do so, because this is by no means a computational restriction in the presence of the explicit or-construct and actions interpreted as unifications. (One can use a so called homogeneous form for clauses, as in [EY], and "or" together clauses with the same head predicate.)

Next we define our variant of the Prolog-language. (Note the similarity with the definition of the language $\mathcal{B}$ in section 3.)

(6.1) DEFINITION Let $\mathcal{F}$ be a collection of function symbols, $\mathcal{V}$ a collection of variables and $\mathcal{R}$ a collection of predicate letters. Let *Term* denote the collection of terms generated by $\mathcal{F}$ over $\mathcal{V}$. Define the set of atomic goals $AGoal = \{ t_1 = t_2,\ \textbf{\textit{fail}},\ !,\ G_1\ \textbf{\textit{or}}\ G_2,\ R(t_1,..,t_k) \mid\ t_i \in Term,\ G_i \in Goal,\ R \in \mathcal{R}$ of arity $k \}$, the set of goals $Goal = \{ A_1\&..\&A_r \mid\ r \in \mathbb{N},\ A_i \in AGoal \}$, ***true*** is the empty goal, the set of Prolog-programs $Prog = \{ A_1 \leftarrow G_1 :..: A_r \leftarrow G_r \mid\ r \in \mathbb{N},\ A_i = R_i(\vec{t_i}) \in AGoal,\ i \neq j \Rightarrow R_i \neq R_j,\ G_i \in Goal \}$. Define Prolog $= \{ P \mid G \mid\ P \in Prog,\ G \in Goal \}$.

We next develop an operational semantics for Prolog along the lines of section 3. In

order to obtain a most general answer substitution (i.e. to avoid clashes of logical variables) one is only allowed to resolve an atom against a program clause provided that the variables of the clause are fresh with respect to the computation so far. We can achieve this by having infinite supply of copies of the class of variables and tagging every goal with an index that it should be renamed with. (This is in fact structure sharing.) In a global counter we keep track of the number of the first class of variables not used yet.

(6.2) DEFINITION Let $Term'$ be the set of terms generated by $\mathcal{F}$ over $\mathcal{V} \times \mathbb{N}$ and Σ be the collection of substitutions over $Term'$, i.e. $\Sigma = \{ \sigma : Term' \to Term' \mid \sigma \text{ homomorphic} \}$. The set $GGoal$ of generalized goals is defined by $GGoal = \{ \langle G_1, D_1, m_1 \rangle : .. : \langle G_r, D_r, m_r \rangle \mid r \in \mathbb{N}, G_i \in Goal, D_i \in Stack, i \leq j \Rightarrow D_i \geq_{ss} D_j, m_i \in \mathbb{N} \}$, the set of frames $Frame = \{ [g, \sigma, n] \mid g \in GGoal, \sigma \in \Sigma, n \in \mathbb{N} \}$, the set of stacks $Stack = \{ F_1 : .. : F_r \mid r \in \mathbb{N}, F_i = [\langle G_1, D_1, m_1 \rangle : .. : \langle G_r, D_r, m_r \rangle, \sigma, n] \in Frame$ such that $F_{i+1} : .. : F_r \geq_{ss} D_j \}$ and the set of configurations $Conf = Stack \cup \Sigma \cup \{\Omega\}$.

The transition system underlying the operational semantics is a straightforward modification of definition 3.3.

Execution of actions $t_1 = t_2$ and procedure calls $R(t_1,..,t_k)$ involve unification. We use a black box unification algorithm *mgu* that yields a most general unifier for two atoms or terms if one exists, and is undefined otherwise. (Cf. [JM]a, [Fr]a.) So the effect of the execution of an action $t_1 = t_2$ in state σ is the update $\sigma\theta$, i.e. composition of substitutions, of σ with respect to the most general unifier θ of t_1 and t_2 in state σ (and appropriately renamed).

Slightly more deviating is procedure handling, since one has to unify first the call and the head of the particular clause successfully before body replacement can take place. (Stretching a point one may consider Prolog as a form of conditional rewriting. See also [BW]a, [EY]a.) A call is operationally described as follows. Consider a call, i.e. atom, $R(t_1,..,t_k)$. First the concerning procedure definition, i.e. clause, is looked up in the declaration, i.e. Prolog-program. Say this is $R(\bar{t}_1,..,\bar{t}_k) \leftarrow \bar{G}$. Next we try to unify $R(t_1,..,t_k)$ and $R(\bar{t}_1,..,\bar{t}_k)$ (considering renaming and the current substitution). If this is possible, i.e. a most general unifier exists, we replace the call by the procedure body, i.e. body of the program clause, extended with dump stack and renaming index, and change the state and global counter according to the side effect, i.e. the result of *mgu*, initiated by the call. We refer the reader to the nice tutorial of [Le] for a discussion of unification in logic programming vs. parameter passing and value return in imperative languages.

(6.3) DEFINITION Let $P \in Prog$. P induces a deterministic transition system $\to_P$ with as transition-relation the smallest subset of $Conf \times Conf$ such that

(i) $E \to_P \delta$

(ii) $[\gamma, \sigma, n] : S \to_P \sigma$

(iii) $[\,\langle \textbf{\textit{true}}\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P [\,g,\sigma,n\,]{:}S$

(iv) $[\,\langle t_1{=}t_2\ \&\ G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P [\,\langle G\,, D\,, m\rangle{:}g,\sigma\theta,n\,]{:}S$
if $\theta = mgu(\,t_1^{(m)}\sigma\,,\, t_2^{(m)}\sigma\,)$ exists
$[\,\langle t_1{=}t_2\ \&\ G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P S$ otherwise

(v) $[\,\langle \textbf{\textit{fail}}\ \&\ G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P S$

(vi) $[\,\langle !\ \&\ G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P [\,\langle G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}D$

(vii) $[\,\langle R(t_1,..,t_k)\&G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P [\,\langle \overline{G},S,n\rangle{:}\langle G\,, D\,, m\rangle{:}g,\,\sigma\theta,\,n{+}1\,]{:}S$
if $R(\overline{t}_1,..,\overline{t}_k) \leftarrow \overline{G} \in P$ and $\theta = mgu(\,R(t_1^{(m)},..,t_k^{(m)})\sigma\,,\, R(\overline{t}_1^{(n)},..,\overline{t}_k^{(n)})\,)$ exists
$[\,\langle R(t_1,..,t_k)\&G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to_P S$ otherwise

(viii) $[\,\langle (G_1\ \textbf{\textit{or}}\ G_2)\ \&\ G\,, D\,, m\rangle{:}g,\sigma,n\,]{:}S \to F_1{:}F_2{:}S$
where $F_i = [\,\langle G_i\&G\,, D\,, m\rangle{:}g,\,\sigma,\,n\,]$

In the above definition we denote by $t^{(m)}$ the term in *Term′* obtained by renaming in t variables in $\mathcal{V}$ into the corresponding variables in $\mathcal{V}\times\{m\}$. We use suffix notation for the application and composition of substitutions.

The operational semantics is defined similar to definition 3.4. Here, in the context of logic programming, we choose to fix the start state, viz. the identity substitution σ_{id}. The renaming index is set to 1 having used 0 for the top-level goal already.

(6.4) DEFINITION The operational Prolog-semantics $\mathcal{O}$: Prolog $\to \Sigma_\perp$ is defined by $\mathcal{O}(P|G) = \alpha_P([\,\langle G,\mathrm{E},0\,\rangle,\sigma_{id},1\,])$ where $\alpha_P : \mathit{Conf} \to \Sigma_\perp$ is the valuation associated with the transition system induced by P.

Having discussed already the idiosyncrasies of Prolog with respect to unification-action and call, it is clear how to adapt the denotational semantics of $\mathcal{B}$ in order to obtain a denotational semantics for Prolog.

First we redefine the functionality of environments and success continuations. Define $\mathit{Atom} = \{\, R(t_1,..,t_k) \mid R \in \mathcal{R}$ *of arity* k, $t_i \in \mathit{Term}\,\}$. (*Atom* is the Prolog-counterpart of *Proc*.) Let $\mathit{Env} = \mathit{Atom} \to \mathbb{N} \to \mathit{SCont} \to \mathit{FCont} \to \mathit{CCont} \to \Sigma \to \mathbb{N} \to \Sigma_\perp$ and $\mathit{SCont} = \mathit{FCont} \to \mathit{CCont} \to \Sigma \to \mathbb{N} \to \Sigma_\perp$. We take *FCont* and *CCont* as defined previously (with $\Sigma_\perp$ implicitly changed).

(6.5) DEFINITION

(i) $[\![\ \cdot\ ]\!]_{\mathcal{A}} : \mathit{AGoal} \to \mathit{Env} \to \mathbb{N} \to \mathit{SCont} \to \mathit{FCont} \to \mathit{CCont} \to \Sigma \to \mathbb{N} \to \Sigma_\perp$
$[\![\ t_1{=}t_2\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = \xi\phi\kappa\{\sigma\theta\}n$ if $\theta = mgu(\,t_1^{(m)}\sigma\,,\, t_2^{(m)}\sigma\,)$ exists
$[\![\ t_1{=}t_2\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = \phi$ otherwise
$[\![\ \textbf{\textit{fail}}\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = \phi$
$[\![\ !\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = \xi\kappa\kappa\sigma n$
$[\![\ G_1\ \textbf{\textit{or}}\ G_2\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = [\![\ G_1\ ]\!]_{\mathcal{G}}\ \eta m\xi\{[\![\ G_2\ ]\!]_{\mathcal{G}}\ \eta m\xi\phi\kappa\sigma n\}\kappa\sigma n$
$[\![\ R(\vec{t})\ ]\!]_{\mathcal{A}}\ \eta m\xi\phi\kappa\sigma n = \eta\{R(\vec{t})\}m\xi\phi\kappa\sigma n$

(ii) $[\![\cdot]\!]_{\mathcal{G}} : Goal \to Env \to \mathbb{N} \to SCont \to FCont \to CCont \to \Sigma \to \mathbb{N} \to \Sigma_\perp$

$[\![\textbf{\textit{true}}]\!]_{\mathcal{G}}\ \eta m \xi \phi \kappa \sigma n = \xi \phi \kappa \sigma n$

$[\![A \,\&\, G]\!]_{\mathcal{G}}\ \eta m \xi \phi \kappa \sigma n = [\![A]\!]_{\mathcal{A}}\ \eta m \{[\![G]\!]_{\mathcal{G}}\ \eta m \xi\} \phi \kappa \sigma n$

(iii) $\Phi : Prog \to Env \to Env$

$\Phi P \eta \{R(\vec{t})\} m \xi \phi \kappa \sigma n = [\![G_0]\!]_{\mathcal{G}}\ \eta n \{\lambda \bar{\phi} \bar{\kappa}.\xi \bar{\phi} \kappa\} \phi \phi \{\sigma\theta\} \{n+1\}$

if $R(\vec{t}_0) \leftarrow G_0 \in P$ and $\theta = mgu(R(\vec{t}^{(m)})\sigma, R(\vec{t}_0^{(n)}))$ exists

$\Phi P \eta \{R(\vec{t})\} m \xi \phi \kappa \sigma n = \phi$ otherwise

(iv) $[\![\cdot]\!]_{\mathcal{P}_{rolog}} : \text{Prolog} \to \Sigma_\perp$

$[\![P \mid G]\!]_{\mathcal{P}_{rolog}} = [\![G]\!]_{\mathcal{G}}\ \eta_P\, 0\, \xi_0\ \phi_0\ \kappa_0\ \sigma_{id}\, 1$

where η_P is the least fixed point of $\Phi(P)$, $\xi_0 = \lambda \phi \kappa \sigma n.\sigma$ and $\phi_0 = \kappa_0 = \delta$

It is a matter of routine to obtain the equivalence of the operational and denotational semantics for Prolog along the lines of section 5.

Section 7 Concluding Remarks

In this paper we have established a denotational continuation semantics for Prolog and we have related it to an operational one. Three ideas have contributed to a clean equivalence proof. First we have focused on the control flow aspects in the uniform language $\mathcal{B}$, obtained from Prolog by leaving out the logic programming issues, such as most general unifiers and renaming indices. (Hence illustrating the "Algorithm = Logic + Control" paradigm.) Secondly, the representation of the operational semantics as the least upperbound of a chain of approximations (and the use of continuations) enabled us to establish equivalence on the level of approximations of both the denotational and operational semantics already. Thirdly, this could be done via an intermediate operator on operational configurations in a denotational manner by virtue of an appropriate induction principle. Having obtained the congruence of the operational and denotational semantics of $\mathcal{B}$ we have generalized these semantics to handle Prolog. Also the equivalence proof for $\mathcal{B}$ could be extended straightforwardly to the Prolog situation.

It is subject of future research to establish a denotational semantics for Prolog closer to the declarative semantics of logic programming based on Herbrand models. Another interesting topic under current research is to exploit the idea of using continuations and approximations in order to compare operational and denotational semantics for other programming language concepts.

Acknowledgments. Our appreciation is due to Jaco de Bakker, Frank de Boer, Joost Kok, John-Jules Meyer and Jan Rutten, members of the Amsterdam Concurrency Group, who offered us a stimulating forum. We thank Aart Middeldorp for reading the manuscript. We are indebted to M279 for the hospitality the authors received during the preparation of this

paper.

References

[Ll] J.W. Lloyd, *Foundations of Logic Programming,* Springer (1984).

[BKMOZ] J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog, and J.I. Zucker, ‘‘Contrasting Themes in the Semantics of Imperative Concurrency,’’ pp. 51-121 in *Current Trends in Concurrency: Overviews and Tutorials*, J.W. de Bakker, W.P de Roever & G. Rozenberg (eds.), LNCS **224** (1986).

[Ba2] J.W. de Bakker, ‘‘Comparative Semantics for Flow of Control in Logic Programming without Logic,’’ Report CS-R8840, Centrum voor Wiskunde en Informatica, Amsterdam (1988). To appear in Information and Computation.

[Br] A. de Bruin, *Experiments with Continuation Semantics: Jumps, Backtracking, Dynamic Networks,* Dissertation, Vrije Universiteit, Amsterdam (1986).

[JM] N.D. Jones and A. Mycroft, ‘‘Stepwise Development of Operational and Denotational Semantics for Prolog,’’ pp. 281-288 in *Proc. Symposium on Logic Programming*, Atlantic City (1984).

[DM] S.K. Debray and P. Mishra, ‘‘Denotational and Operational Semantics for Prolog,’’ *Journal of Logic Programming* **5**, pp. 61-91 (1988).

[Vi] E.P. de Vink, ‘‘Equivalence of an Operational and a Denotational Semantics for a Prolog-like Language with Cut,’’ Report IR-151, Vrije Universiteit, Amsterdam (1988).

[KR] J.N. Kok and J.J.M.M. Rutten, ‘‘Contractions in Comparing Concurrency Semantics,’’ pp. 317-332 in *Proc. ICALP’88*, T. Lepistö & A. Salomaa (eds.), LNCS **317** (1988).

[BM] J.W. de Bakker and J.-J.Ch. Meyer, ‘‘Metric Semantics for Concurrency,’’ *BIT* **28**, pp. 504-529 (1988).

[Pl] G.D. Plotkin, ‘‘A Structural Approach to Operational Semantics,’’ DAIMI FN-19, Aarhus Universitet (1981).

[BMOZ] J.W. de Bakker, J.-J.Ch. Meyer, E.-R. Olderog, and J.I. Zucker, ‘‘Transition Systems, Metric Spaces and Ready Sets in the Semantics for Uniform Concurrency,’’ *Journal of Computer and System Sciences* **36**, pp. 158-224 (1988).

[MS] R. Milne and C. Strachey, *A Theory of Programming Language Semantics,* Chapman & Hall and Wiley, 2 Volumes (1976).

[St] J.E. Stoy, *Denotational Semantics - The Scott-Strachey Approach to Programming Language Theory*, MIT Press (1977).

[Ba1] J.W. de Bakker, *Mathematical Theory of Program Correctness,* Prentice Hall International (1980).

[Te2] R.D. Tennent, *Principles of Programming Languages,* Prentice Hall International

(1981).

[Te1] R.D. Tennent, ‘‘Mathematical Semantics of SNOBOL4,’’ pp. 95-107 in *Proc. POPL’73*, Boston (1973).

[Wi] M. Wirsing (ed.), *Formal Description of Programming Concepts - III*, North-Holland (1987).

[Kl] J.W. Klop, *Combinatory Reduction Systems*, Mathematical Centre Tracts 127, Mathematisch Centrum, Amsterdam (1980).

[Hu] G. Huet, ‘‘Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems,’’ *Journal of the ACM* **27**(4), pp. 797-821 (1980).

[CKRP] A. Colmerauer, H. Kanoui, P Roussel, and R. Pasero, ‘‘Une Système de Communication Homme-Machine en Francais,’’ Groupe de Recherche en Intelligence Artificielle, Université d’Aix-Marseille (1973).

[AE] K.R. Apt and M.H. van Emden, ‘‘Contributions to the Theory of Logic Programming,’’ *Journal of the ACM* **29**, pp. 841-862 (1982).

[EK] M.H. van Emden and R.A. Kowalski, ‘‘The Semantics of Predicate Logic as a Programming Language,’’ *Journal of the ACM* **23**(4), pp. 773-742 (1976).

[Fi] M. Fitting, ‘‘A Deterministic PROLOG Fixpoint Semantics,’’ *Journal of Logic Programming* **2**, pp. 111-118 (1985).

[Fr] G. Frandsen, ‘‘A Denotational Semantics for Logic Programming,’’ DAIMI PB-201, Aarhus Universitet (1985).

[AB] B. Arbab and D.M. Berry, ‘‘Operational and Denotational Semantics of Prolog,’’ *Journal of Logic Programming* **4**, pp. 309-329 (1987).

[BW] J.C.M. Baeten and W.P. Weijland, ‘‘Semantics for Prolog via Term Rewrite Systems,’’ pp. 3-14 in *Proc. Conditional Term Rewriting Systems*, S. Kaplan & J.-P. Jouannaud (eds.), LNCS **308** (1987).

[Kw] R. Kowalski, ‘‘Algorithm = Logic + Control,’’ *Communications of the ACM* **22**(7), pp. 424-436 (1979).

[GCLS] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro, ‘‘Fully Abstract Denotational Semantics for Concurrent Prolog,’’ pp. 320-335 in *Proc. LICS’88*, Edinburgh (1988).

[Kk] J.N. Kok, ‘‘A Compositional Semantics for Concurrent Prolog,’’ pp. 373-388 in *Proc. STACS’88*, R. Cori & M. Wirsing (eds.), LNCS **294** (1988).

[BK] J.W. de Bakker and J.N. Kok, ‘‘Uniform Abstraction, Atomicity and Contractions in the Comparative Semantics of Concurrent Prolog,’’ pp. 347-355 in *Proc. International Conference on Fifth Generation Conputer Systems 1988*, Tokyo (1988).

[EY] M.H. van Emden and K. Yukawa, ‘‘Logic Programming with Equations,’’ *Journal of Logic Programming* **4**, pp. 265-288 (1987).

[Le] G. Levi, ‘‘Logic Programming: the Foundations, the Approach and the Role of Concurrency,’’ pp. 396-441 in *Current Trends in Concurrency: Overviews and Tutorials*, J.W. de Bakker, W.P de Roever & G. Rozenberg (eds.), LNCS **224** (1986).